Lehr- und Forschungsgebiet Diskrete Optimierung

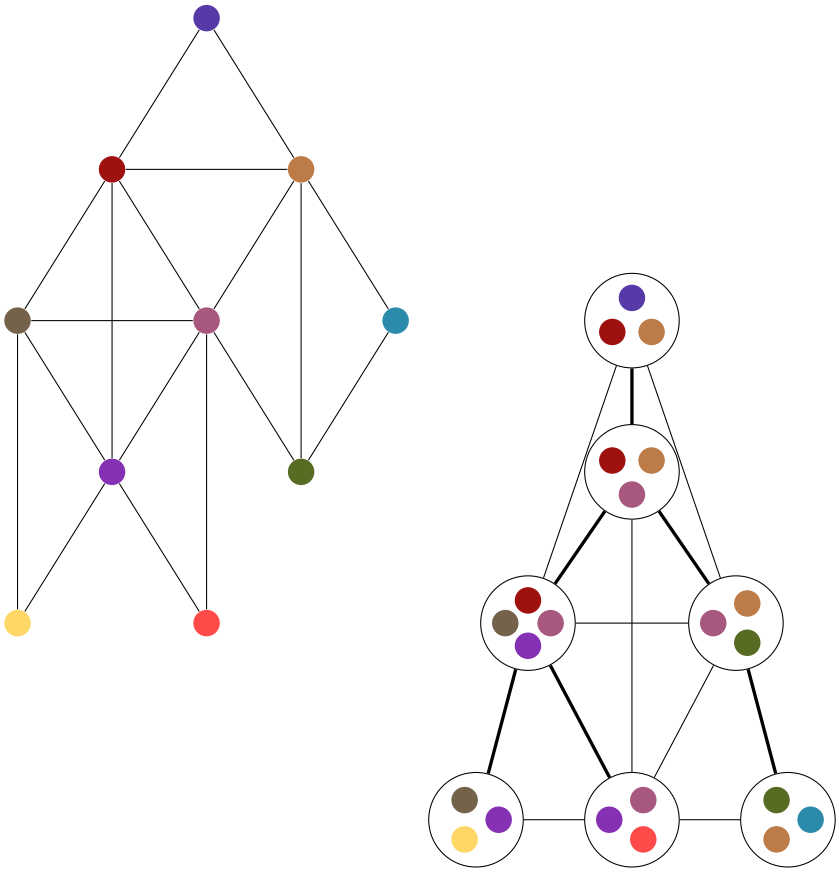
RWTH AACHEN UNIVERSITY

The present work was submitted to the
Lehr- und Forschungsgebiet Diskrete Optimierung

Bachelor Thesis

New Upper Bounds for Treewidth using the Clique Operator and Spanning Trees

20th of August 2024



Student:	Raoul Schutzeichel Y Luqué
Student-ID:	431333
First Examiner/Supervisor:	Prof. Arie M.C.A. Koster
Second Examiner:	Prof. Christina Büsing

Contents

1 Introduction

One of the most prominent research questions in recent mathematical and computer science discussion is the $\mathcal{P}$ versus $\mathcal{NP}$ question. It asks whether problems that are decidable in polynomial time by a non-deterministic turing machine, are also decidable in polynomial time by deterministic turing machines. Of course, we will not answer that question in this thesis, but we will look at treewidth, a concept that can be linked to the question of $\mathcal{P}$ versus $\mathcal{NP}$. Many graph and network problems that are $\mathcal{NP}$-hard on general graphs become efficiently solvable (e.g., with polynomial-time or linear-time algorithms) when the treewidth of the input graph is bounded by a constant, see [AP89; BK07]. In these cases, the running time is polynomial in the given graph, but exponential in the constant bound of the treewidth. A dynamic programming algorithm for solving the $\mathcal{NP}$-hard weighted independent set problem on graphs of bounded treewidth with linear running time is provided in [BK07, p. 258].

The notion of treewidth was originally introduced by Robertson and Seymour in the 1980s, see [RS86], in the context of their graph minor theory. Since then, there has been a great deal of research into the computation and application of treewidth. An overview is given in [Bod98; Bod07].

Computing the treewidth of a given graph is generally $\mathcal{NP}$-hard, which is why much research has been done on optimizing existing algorithms that compute the exact treewidth, or on heuristics that provide lower or upper bounds on the treewidth, see for example [Kor22; DK07; SG97]. Heuristics usually have significantly better running times than exact algorithms, but do not always have guarantees of the quality of their returned bounds.

In this thesis, we want to develop such a heuristic that computes an upper bound on the treewidth of a given graph. For this, we will use a novel approach that combines the concept of treewidth and tree decompositions with the clique graph operator, an operator originally from intersection graph theory. This leads us to the following structure of this thesis. We will start by defining the treewidth of a graph and discussing some basic results in order to lay the foundation of the later developed heuristic. Then, we will define the clique graph operator and give an overview of the research surrounding it in order to then describe the heuristic developed in this thesis. After discussing the different variants of the heuristic, we will present a computational evaluation that was done to find the best variant of the heuristic and compare it with an existing heuristic.

1.1 Notation

The notation for basic graph theoretic concepts used in this thesis is based on the book *Graph Theory* by Diestel, see [Die05]. We use G to denote an undirected graph and $V(G)$ and $E(G)$ to denote the vertex and edge sets, respectively. Unless otherwise specified, we use n and m for the number of vertices and edges, respectively. An edge e is a set of two vertices $e = \{v, w\} \in V \times V$, which we also write as $e = vw$. The set of neighbors of a vertex v (or a subset of vertices $U \subseteq V$) is denoted by $N_G(v)$ (or $N_G(U)$), where $N_G(v)$ is the open neighborhood not containing v and $N_G[v]$ the closed neighborhood containing v. We will use $\delta(G)$ and $\Delta(G)$ to refer to the minimum and maximum degree of G, respectively. To refer to an induced subgraph we will write $G[U]$ and $\overline{G}$ refers to the complement graph (the graph with vertex set $V(G)$ and edge set $(V \times V) \setminus E(G)$). If there is a bijection between the vertex sets of two graphs G and H that preserves the adjacencies such that the same holds for its inverse, we call G and H isomorphic and write $G \simeq H$.

With K_n we want to denote the complete graph on n vertices. A clique is a subset $C \subseteq V(G)$ of vertices such that the induced subgraph is isomorphic to a complete graph $G[C] \simeq K_{|C|}$. We call a clique C maximal if there is no other clique C' such that $C \subsetneq C'$. At last we want to denote the size of the clique with maximum cardinality by $\omega(G)$, which we will also call the clique number, whereas the number of maximal cliques of a graph is denoted by $\kappa(G)$.

In this thesis we will only consider simple, finite graphs. Furthermore, the indices i, j, k, l, m we will use shall be from the set of natural numbers. Finally, we assume that the reader has a basic understanding of graph theory and combinatorial optimization.

2 Treewidth

The introduction highlighted the importance of the concept of treewidth. This chapter serves two purposes. First, to present basic results on treewidth. Second, to establish a basic understanding of the concept upon which we will build our heuristic. At the end of this chapter, we want to present an existing heuristic, with which we will later compare our heuristic.

2.1 Definition

We will start by simply looking at the definition of treewidth, on which we will build our understanding as we progress through this chapter. It cannot be overstated how important this definition is to the rest of this thesis as it will be relevant throughout its entirety. The treewidth of a graph is typically defined in terms of tree decompositions. Therefore, the following definition introduces both concepts at once.

Definition 2.1 (Treewidth [RS86, p. 309])**.** A tree decomposition of a graph $G = (V, E)$ is a pair $(\{X_i \mid i \in I\}, T = (I, F))$, where T is a tree and each vertex $i \in I$ has associated with it a subset of vertices $X_i \subseteq V$, called the bag of i, such that:

(i) Each vertex belongs to at least one bag: $\bigcup_{i \in I} X_i = V$.
(ii) For all edges there is a bag containing both its endpoints, i.e. for all $\{v, w\} \in E$: there is an $i \in I$ with $v, w \in X_i$.
(iii) For all vertices $v \in V$, the set $\{i \in I \mid v \in X_i\}$ induces a subtree of T.

The width of a tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ is $\max_{i \in I} |X_i| - 1$. The treewidth of a graph G is the minimum width over all tree decompositions of G. We denote the treewidth of a graph G by $\mathrm{tw}(G)$.

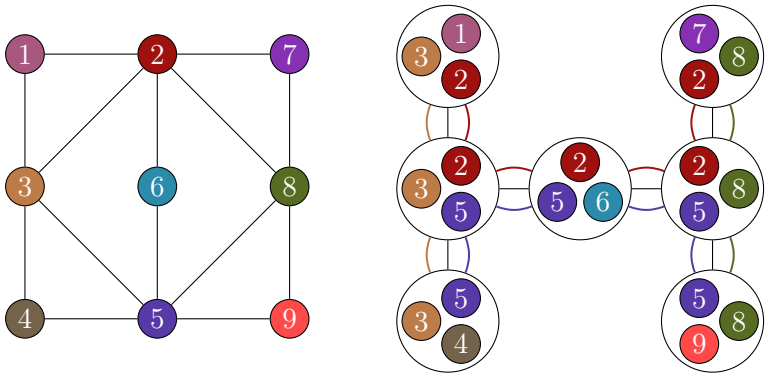
Note that an equivalent definition [see Bod98] is obtained by replacing the third condition by:

(iii) For all $i, j, k \in I$, if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

Furthermore throughout this thesis we will use the term 'vertices' to refer to the vertices $v \in V(G)$ of a graph G and the term 'bags' to refer to the vertices $i \in I$ of the decomposition tree T.

Since the above definition might seem a bit obfuscated at first, we want to look at a tree decomposition of a graph in the following example.

Example 2.2. The figure 2.1 shows a graph G with 9 vertices and a tree decomposition of G onto a tree with seven bags. Each vertex of G is contained in some bag in the decomposition and each adjacent pair of vertices is found together in at least one bag. The bags containing a fixed vertex induce a subtree as visualized with the different colored edges. Thus, all the properties of a tree decomposition are fulfilled. Note that the parallel edges are just for visualization purposes.



(a) G

(b) Tree decomposition of G with width 2

Figure 2.1: Tree decomposition with induced subtrees highlighted by colored edges.

Of course, every graph has a trivial tree decomposition consisting of only one bag containing all vertices. However, as in the definition of treewidth, see 2.1, one is interested in the tree decomposition with minimum width, i.e. minimum maximum bag size.

There are a variety of equivalent characterizations of the notion of treewidth but for our purposes the above definition and one we will present in section 2.3 are sufficient. See [Bod98] for an overview of other characterizations.

One interpretation that is often used for the treewidth of a graph is that it is a measure of how far a graph is from being a tree. We will return to this idea in the course of this chapter and prove at the end of the next section that trees or more precisely forests, are indeed the graphs with the lowest treewidth, i.e., the graphs that are closest to being trees.

2.2 Basic properties

The following basic results on the properties of treewidth will be useful in developing our heuristic and can be easily derived. At the end of this section, the results will culminate in a theorem characterizing graphs with treewidth 1.

Given a subgraph it is clear that one can reuse a tree decomposition of the original graph, which leads to the following inequality.

Lemma 2.3 ([Sch89, p. 9]). *Let $H \subseteq G$ be a subgraph of G. Then*

$$\mathrm{tw}(H) \leq \mathrm{tw}(G).$$

Proof. Let $(\{X_i \mid i \in I\}, (I, F))$ be a tree decomposition of G. Then $(\{X_i \cap V(H) \mid i \in I\}, (I, F))$ is clearly a tree decomposition of H with the same or a lower width than $(\{X_i \mid i \in I\}, (I, F))$. □

This result can be continued for minors as done in the following lemma.

Lemma 2.4 ([Sch89, p. 9]). *Let H be a minor of G. Then*

$$\mathrm{tw}(H) \leq \mathrm{tw}(G).$$

Proof. As in the proof of lemma 2.3, vertex and edge deletions can be handled by reusing the decomposition of G. For contractions of edges $\{v, w\} \in E(G)$ to a vertex x, replace occurrences of v and w in bags by x. The resulting decomposition clearly consists of a tree and satisfies conditions (i) and (ii) from definition 2.1. Note that (iii) is also satisfied, since at least one bag from the decomposition of G contains both v and w and any other bags containing v or w are reachable via the induced subtrees of v or w (in the decomposition of G). □

This leads to the following corollary about the treewidth of disconnected graphs.

Corollary 2.5 ([Sch89, p. 10]). *Let G be a graph and C its connected components. Then*

$$\mathrm{tw}(G) = \max_{H \in C} \mathrm{tw}(G[H]).$$

Proof. Given tree decompositions $(\{X_{i,1} \mid i \in I_1\}, (I_1, F_1)), \ldots, (\{X_{i,|C|} \mid i \in I_{|C|}\}, (I_{|C|}, F_{|C|}))$ of the $|C|$ components, a tree decomposition of G is obtained by adding $|C| - 1$ connecting edges to the forest $(\bigcup_{i=1}^{|C|} I_i, \bigcup_{i=1}^{|C|} F_i)$ and using $\bigcup_{j=1}^{|C|} \{X_{i,j} \mid i \in I_j\}$ as bags. Therefore $\mathrm{tw}(G) \leq \max_{H \in C} \mathrm{tw}(G[H])$. On the other hand we get that for every $H \in C$, $\mathrm{tw}(G) \geq \mathrm{tw}(H)$ by lemma 2.3. □

As discussed already, the treewidth of a graph can be interpreted as a measure of how close a graph is to being a tree. For example, complete graphs are very strongly connected and thus, in a sense, far away from being trees since trees have no cycles by definition. The incorporation of this idea into the definition of treewidth is reflected in the following lemma. An easy proof, as suggested in [BM93, p. 304], is based on lemma 3.17, which will be proved later.

Lemma 2.6 ([BM93, p. 304]). *Let $(\{X_i \mid i \in I\}, T = (I, F))$ be a tree decomposition of G and let $C \subseteq V(G)$ be a clique in G. Then there exists an i such that $C \subseteq X_i$.*

Proof. Let $T_v = \{i \in I \mid v \in X_i\}$. Then $\mathcal{F} = \{T_v \mid v \in C\}$ is a family of subtrees of T. Furthermore, the trees in $\mathcal{F}$ are pairwise intersecting nontrivially since C is a clique. Therefore by lemma 3.17, a vertex $i_0 \in \bigcap_{T_v \in \mathcal{F}} V(T_v)$ exists for which $C \subseteq X_{i_0}$ as desired. □

This implies that a complete graph, or more generally any graph with a clique number of $\omega(G)$, has treewidth of at least $\omega(G) - 1$. Therefore, as the intuition might suggest, complete graphs have a relatively high treewidth, i.e. they are far from being trees. This result is summarized in the following corollary.

Corollary 2.7. *For a graph G it holds that*

$$\mathrm{tw}(G) \geq \omega(G) - 1.$$

Proof. Let $(\{X_i \mid i \in I\}, T = (I, F))$ be a tree decomposition of G and let C be a maximum clique. Then, by lemma 2.6, there exists an i such that $C \subseteq X_i$, therefore $|X_i| \geq C$, i.e. the width of the decomposition is at least $\omega(G) - 1$. □

Finally, we want to characterize those graphs which have a treewidth of 1 as trees, or more precisely as forests, since the treewidth of a disconnected graph is the maximum of the treewidths of its components, as seen in corollary 2.5.

Theorem 2.8. *A graph has treewidth one if and only if it is a forest.*

Proof. As a consequence of corollary 2.5, it suffices to consider connected graphs. Given a tree G, choose an arbitrary vertex $v_0 \in V(G)$ as the root of the tree. Now a decomposition is given by $(\{X_v \mid v \in V(G)\}, G)$ with X_v consisting of v and its parent for $v \neq v_0$ and $X_{v_0} = \{v_0\}$. The properties of a tree decomposition can easily be verified.

On the other hand, let G have treewidth 1. Suppose G is not a tree. Since G is connected, it must hold that G is not acyclic and therefore has K_3 as a minor. By lemma 2.4 and corollary 2.7 this implies that G has treewidth ≥ 2, a contradiction. □

2.3 Chordal graphs and known heuristics for upper bounds on treewidth

In this section, we want to look at known upper bound heuristics, i.e., algorithms that, given a graph G provide an upper bound on the treewidth, e.g., by finding a tree decomposition of G whose width may not be optimal, but which gives us an upper bound on the treewidth of G.

To do this, we first want to look at chordal graphs, a class of graphs that often comes up in treewidth research, as we will see. As the name suggests, chordal graphs are related to the concept of chords, edges between two vertices of a cycle. Chordal graphs will give us a relatively easy way to compute tree decompositions of graphs and thus upper bounds on the treewidth of graphs. To achieve this, we want to look at the following definition of chordal graphs and then look at some basic results on them.

Definition 2.9 (Chordal graph [BK10, p. 261])**.** G is chordal if every cycle in G of length at least four has a chord, i.e., an edge connecting two non-successive vertices in the cycle.

The figure 2.2 shows a chordal graph whose chordality is immediately apparent. The graph consists of triangles and thus every cycle of length at least four has a chord. For this reason, chordal graphs are also called triangulated graphs.

Note that any induced subgraph of a chordal graph is chordal, since cycles in induced subgraphs also exist in the original graph and thus the corresponding chords are also in the induced subgraph, making it chordal.

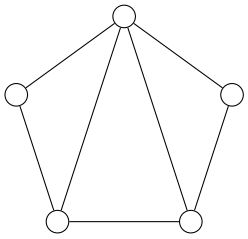


Figure 2.2: Chordal graph

The definition of chordal graphs is in a sense an algorithmic one, since one can transform any given graph into a chordal graph by adding the necessary edges. This could be achieved by adding chords to those cycles of length at least four that do not contain chords yet. The resulting graph has triangles (cycles of length three) where the original graph had cycles of length at least four, which leads back to the name triangulated graphs. Thus, this process of chordalizing a given graph is called triangulating or triangulation, as formalized in the following definition.

Definition 2.10 (Triangulation [BK10, p. 261])**.** A graph H is a triangulation of a graph G if H is a chordal graph obtained by adding zero or more edges to G.

Of course, triangulations are not necessarily unique, as shown in figure 2.3.

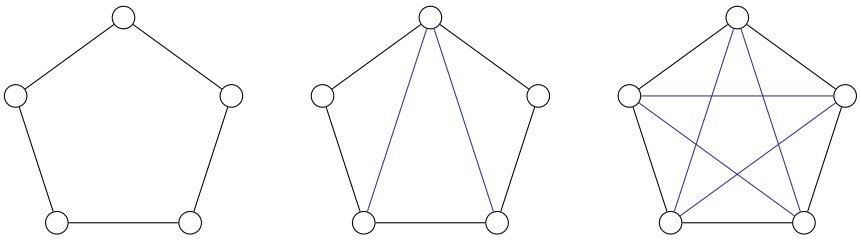


Figure 2.3: Non-chordal graph on the left and two possible triangulations to the right of it

We want to briefly look at k-trees, a subclass of chordal graphs. K-trees will come up again in the computational evaluation of our heuristic. For our purposes we shall define them as follows.

Definition 2.11 (k-tree [Arn85, p. 5])**.** A graph G is a k-tree if either of the following holds.

(i) G is the complete graph on k vertices.
(ii) G can be constructed from an existing k-tree H by adding a vertex v to H and attaching it to k vertices in H that form a clique.

Note that by the above definition every k-tree can be constructed recursively from the complete graph on k vertices. More specifically, there exists a construction sequence of

k-trees to construct a given k-tree starting at K_k and ending at the desired k-tree. Trees can be defined similarly by starting with the graph with one vertex and constructing a new tree from an existing one by attaching a vertex to a single vertex. Therefore, trees are just 1-trees and k-trees can be seen as a generalization of the idea of 1-trees, hence the name k-trees.

From the above definition it is immediately clear that k-trees are indeed chordal graphs. Let $v_1, \ldots, v_l$ be a cycle of length greater than 3 in a k-tree T. Since T is a k-tree, we can assume that there exists a sequence of k-trees $K_k, T_0, \ldots, T$ to construct T from the complete graph on k vertices. We now assume, without loss of generality, that among $v_1, \ldots, v_l$, v_l is added last in the procedure. Therefore, v_1 and v_{l-1} form a clique, which implies that there is a chord in the cycle.

We now want to look at one of many alternative characterizations of chordal graphs, which is based on the same idea we just saw in the definition of k-trees. By definition, k-trees have a vertex v whose closed neighborhood $N[v]$ forms a maximal clique. The following characterization is based on the idea that every chordal graph has such a simplicial vertex, i.e. a vertex whose neighborhood forms a clique that is maximal if the vertex itself is included. To prove this, and thus obtain an alternative characterization of chordal graphs, we need the following result. It is an implication of another characterization of chordal graphs using minimal separators and was first shown in 1961, see [Dir61, p. 71]. Before looking at the result, let us first properly define what simplicial vertices and separators are.

Definition 2.12 (Simplicial vertex [Gol80, p. 81])**.** A vertex $x \in V(G)$ is called simplicial if its neighborhood $N_G(x)$ induces a complete subgraph, i.e. forms a clique.

Definition 2.13 (Separator [Gol80, p. 82])**.** A subset $S \subseteq V$ is a (vertex) separator of G for non-adjacent vertices $a, b \in V(G)$ (or an a-b separator) if removing S from G separates a and b into distinct connected components. If no proper subset of S is an a-b separator, we call S minimal.

The proofs of the following two lemmas and the alternative characterization of chordal graphs are based on the proofs given in [Gol80, p. 83].

Lemma 2.14 ([Gol80, p. 83])**.** *Let G be a chordal graph. Then every minimal a-b separator induces a complete subgraph of G.*

Proof. Let S be a minimal a-b separator, where A and B are the connected components of $G[V \setminus S]$ containing a and b, respectively. Note that since S is minimal, every $v \in S$ is adjacent to some vertex in A and some vertex in B. Otherwise $S \setminus \{v\}$ would be a smaller separator, a contradiction to the minimality of S. Thus, for any pair $v, w \in S$, there exists a path $v, a_1, \ldots, a_k, w$ and $w, b_1, \ldots, b_l, v$ such that $a_i \in A$ and $b_i \in B$ and the paths are chosen such that their lengths are minimal. It follows that $v, a_1, \ldots, a_k, w, b_1, \ldots, b_l, v$ is a cycle of length at least four, implying that it must have a chord. Since $a_i b_j \notin E(G)$ by

the definition of separator and $a_i a_j \notin E(G)$ as well as $b_i b_j \notin E(G)$ by the minimality of k and l, it holds that $vw \in E(G)$. Since v and w were chosen arbitrarily in S, $G[S]$ is a complete subgraph. □

This allows us to prove the following lemma about the existence of simplicial vertices in chordal graphs.

Lemma 2.15 ([Gol80, p. 83])**.** *Let G be a chordal graph. Then G has a simplicial vertex. Furthermore, if G is not a clique, then it has two non-adjacent simplicial vertices.*

Proof. If G is complete, the statement is trivial. We want to prove the desired by induction over the number of vertices n. In the case $n = 1$, G is complete. Let us therefore assume that G has at least two non-adjacent vertices a and b and the statement holds for all graphs with fewer vertices than G. Furthermore let S be a minimal a-b separator, where A and B are the connected components containing a and b, respectively. Observe that since $G[A \cup S]$ is an induced subgraph of G, it is chordal. Therefore, by induction, either $G[A \cup S]$ is complete and thus every vertex $v \in A$ is simplicial, or $G[A \cup S]$ contains two non-adjacent simplicial vertices, one of which must be in A, since $G[S]$ is complete by lemma 2.14. Furthermore, since $N_G(A) \subseteq A \cup S$, a simplicial vertex of $G[A \cup S]$ in A is also simplicial in G. Analogously, B contains a simplicial vertex of G. □

This now motivates the alternative characterization of chordal graphs. If one removes a simplicial vertex, the resulting graph, being an induced subgraph, is also chordal and thus another simplicial vertex can be removed. Repeating this, one obtains an ordering of the vertices where the neighbors of a given vertex that occur after the vertex form a clique. This is formalized in the following definition of (perfect) elimination orderings.

Definition 2.16 (Elimination ordering [BK10, p. 261])**.** An elimination ordering of G is a bijection $\pi : V(G) \to \{1, 2, \ldots, n\}$. An elimination ordering π is perfect, if for all $v \in V$, the set of its higher numbered neighbors $\{w \mid \{v, w\} \in E \wedge \pi(w) > \pi(v)\}$ forms a clique.

Finally, we obtain the alternative characterization using perfect elimination orderings.

Theorem 2.17 ([Gol80, p. 83])**.** *A graph G is a chordal graph if and only if there exists a perfect elimination ordering of G.*

Proof. Let G be a chordal graph. We can prove the implication by induction on the number of vertices. If G has a single vertex, the implication is trivial. Otherwise G contains a simplicial vertex $v \in V(G)$ and removing this vertex yields a chordal graph $H := G[V \setminus \{v\}]$. By induction, there is a perfect elimination ordering for H and prepending v to this ordering yields a perfect elimination ordering of G by construction.

On the other hand, if G permits a perfect elimination ordering π, let C be a cycle of length greater than 3 in G. Furthermore, let v be the vertex in C that first appears in

π. Since the set of its higher numbered neighbours forms a clique and $|N_G(v) \cap C| \geq 2$, there must be a chord in C which proves the chordality of G. □

Since the proof of the first implication was constructive, we can use it to obtain a perfect elimination ordering for the chordal graph we saw in figure 2.2. The graph is shown with the addition of vertex labels in figure 2.4.

Example 2.18. By successively removing simplicial vertices from the graph shown in figure 2.4 as described in the proof of theorem 2.17, one can obtain the following perfect elimination ordering:

$$\pi = (2\ 5\ 3\ 1\ 4).$$

Here we denote the bijection by ordering the vertex labels. For example, in this case $\pi(2) = 1$, i.e. the vertex with label 2 is the first in the ordering, and $\pi(1) = 4$. We also write $v_1 = 2$ or $v_4 = 1$. Note however, that the above theorem 2.17 does not state that all elimination orderings of chordal graphs are perfect. The following is an example of such a non-perfect elimination ordering.

$$\pi_0 = (1\ 2\ 5\ 3\ 4)$$

1 is the first vertex in the ordering, but $\{2, 3, 4, 5\}$ does not induce a clique because vertices 2 and 5 are not adjacent.

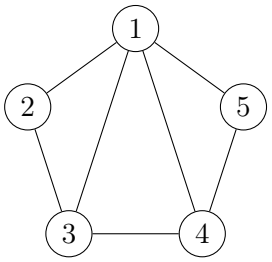


Figure 2.4: Chordal graph G with vertex labels

Before we can establish the connection between chordal graphs and treewidth, we need to look at one last definition. As discussed, one can obtain a chordal graph from an arbitrary graph G by triangulating it, see definition 2.10. We want to apply the same idea to elimination orderings. Since chordal graphs are those graphs for which there exists a perfect elimination ordering, given a graph G and an elimination ordering π of G, we want to add exactly those edges that are missing in order for π to be a perfect elimination ordering of G. The resulting graph will be referred to as G_π^+. We define G_π recursively, where G_i is the graph that results from adding those edges to G_{i-1} which are necessary for the higher numbered neighbors of v_i to be a clique.

Definition 2.19 ([BK10, p. 261])**.** Let π be an elimination ordering of G and v_i be the i-th vertex in π. Define $G_0 := G$ and G_i for $i \in \{1, \dots, n\}$ as

$$G_i := (V(G), E(G_{i-1}) \cup \{\{x, y\} \mid x, y \in N_{G_{i-1}}(v_i) \wedge \pi(x), \pi(y) > i\}).$$

Then we call $G_\pi^+ := G_n$ the filled graph.

The aspects just discussed culminate in the next theorem, which provides an alternative characterization of graphs with bounded treewidth. A formal proof requires other prerequisites which are not as relevant for this work. Therefore, only a sketch of the proof is provided. However, the implication $(iv) \implies (i)$ is proven in lemma 4.1. See [BK10, p. 262] for a complete proof.

Theorem 2.20 ([BK10, p. 262])**.** *Let $k \leq n$ be a non negative integer. The following are equivalent.*

(i) G has treewidth at most k.
(ii) G has a triangulation H such that the maximum size of a clique in H is at most $k + 1$.
(iii) There is an elimination ordering π, such that the maximum size of a clique of G_π^+ is at most $k + 1$.
(iv) There is an elimination ordering π, such that no vertex $v \in V$ has more than k neighbors with a higher number in π in G_π^+.

Proof. $(i) \implies (ii)$: Given a graph G with treewidth at most k, there exists a tree decomposition of G with bag size at most $k + 1$. One can now construct a triangulation by adding edges between vertices that are in the same bag and are not yet adjacent. It can be shown that this yields a chordal graph since cycles in the original graph induce circuits (walks that start and end at the same vertex) in the tree decomposition. From this one can deduce that cycles in the original graph must contain chords.

$(ii) \implies (iii)$: Since H is a triangulation, H is a chordal graph and thus has a perfect elimination ordering π. One can show that $H = G_\pi^+$, which shows the desired.

$(iii) \implies (iv)$: Given the elimination ordering π with the properties from (iii), suppose there exists a vertex $v \in V$ with more than k neighbors with a higher number than v in π in G_π^+. One can show that by construction of G_π^+, v and its neighbors with a higher number in π induce a clique in G_π^+, which would be a clique of size greater than $k + 1$, which is a contradiction.

$(iv) \implies (i)$: Let π and G_π^+ have the properties described in (iv). One can construct a tree decomposition of width k by a method described and proven in the proof of lemma 4.1. □

Since chordal graphs are triangulations of themselves, this implies with corollary 2.7 that, given a chordal graph G, $\mathrm{tw}(G) = \omega(G) - 1$.

The above theorem 2.20 can now be used to develop a class of heuristics that provide upper bounds on the treewidth of a graph by constructing a triangulation.

Given a graph G and an elimination ordering π, we can construct a corresponding triangulation graph $H := G_\pi^+$ such that π is a perfect elimination ordering of H. Then, theorem 2.20 gives us an upper bound on the treewidth of G.

Based on the above description, one can develop an algorithm that constructs an eliminination ordering for a graph G and thus obtain an upper bound on the treewidth. The algorithm GreedyX as proposed in [BK10] does exactly that. It successively builds an elimination ordering of a given graph using different greedy heuristics for choosing the next vertex in the ordering. We will use GreedyX to contextualize the results of the heuristic developed in this thesis later on in the evaluation, see subsection 5.2.4.

3 Clique operator

In the following chapter, we will introduce the main concept underlying the heuristic, the clique (graph) operator. It is defined as an intersection graph and is thus part of intersection graph theory. This branch of graph theory studies graphs that represent the pattern of intersections of a family of sets.

In this area of research, the book *Topics in Intersection Graph Theory* by Terry A. McKee and F. R. McMorris, see [MM99], is a standard reference, which we will use for the basic definitions of this chapter.

We will start by defining what the clique operator is, look at an example and continue with the interaction of the clique operator with some basic binary graph operators. Then we will discuss a well known characterization of clique graphs and go on to look at the application of the clique operator on known classes of graphs such as chordal graphs. At last, we present some sufficient criteria for a graph to have a clique graph of polynomial size.

3.1 Definition

To define the clique graph operator, we must first look at the fundamental concept of intersection graph theory, the intersection graph.

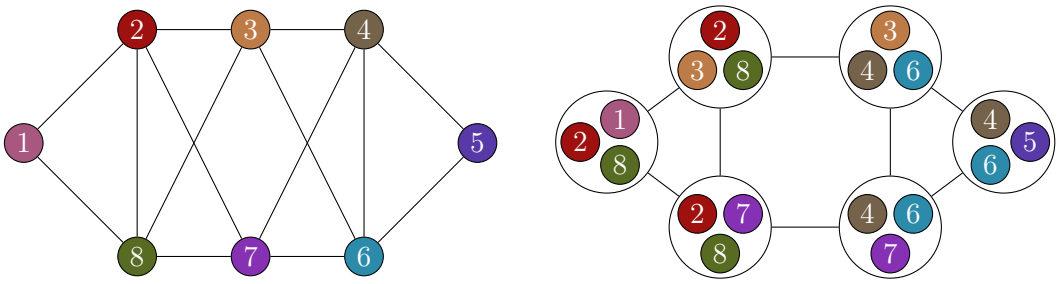
Definition 3.1 (Intersection graph [MM99, p. 1])**.** Let $\mathcal{F} = \{S_1, \ldots, S_n\}$ be a family of sets. The intersection graph of $\mathcal{F}$, denoted by $\Omega(\mathcal{F})$, is the graph with $\mathcal{F}$ as the vertex set with S_i adjacent to S_j if and only if $i \neq j$ and $S_i \cap S_j \neq \emptyset$.

If the sets of $\mathcal{F}$ refer to a given graph G, then the intersection graph of $\mathcal{F}$ can be interpreted as an operator on G. This is what we want to do in the following definition, with the goal of abstracting the structure of a graph using its cliques.

Definition 3.2 (Clique graph operator [MM99, p. 13])**.** The clique graph operator $K(\cdot)$ maps any graph G to the intersection graph of the set of maximal cliques of G. A graph is a clique graph if it is isomorphic to $K(H)$ for some graph H.

Since this definition may seem abstract at first, let us look at the following figure 3.1, which shows a graph G and its clique graph $K(G)$. Note that the colors are just for visibility. As can easily be seen, G consists of 6 maximal cliques, each of size 3. These correspond to the vertices in the clique graph $K(G)$ in subfigure 3.2(b). The clique graph $K(G)$ consists of one vertex for each maximal clique and an edge between two vertices if and only if the corresponding maximal cliques share at least one vertex. Because of this

inherent connection between vertices in the clique graph and maximal cliques, we will refer to vertices in the clique graph as bags and the corresponding maximal clique as the contents of the bag. This also simplifies the distinction between vertices in the original graph G and vertices in the clique graph $K(G)$.



(a) G (b) $K(G)$

Figure 3.1: Graph G and clique graph $K(G)$

The idea of the clique graph operator can be canonically related to the concept of tree decompositions we saw in definiton 2.1. In that case, the maximal cliques, which are the vertices in the resulting clique graph, represent the bags in the tree decomposition. It is clear that the clique graph of a graph satisfies the first and second property of the treewidth definition 2.1. We will elaborate on this idea in the next chapter, where we will explain the heuristic and how it modifies the clique graph to satisfy the other properties of a tree decomposition.

Before doing that, we want to further understand the clique operator by looking at how it interacts with other, more basic operations on graphs.

3.2 Operations on clique graphs

In the following, we want to study the interactions of the clique operator with some basic binary operators. To do so, we must first define such operators. For the following definitions and corollary, let G_1 and G_2 be graphs and $X \times Y$ be the set of sets $\{x, y\}$ with $x \in X$ and $y \in Y$ for sets X, Y. To avoid notational ambiguity, we shall assume without loss of generality that $V(G_1) \cap V(G_2) = \emptyset$.

The first of our binary operators is the union and consists of simply disjointly merging two graphs into one, i.e., two graphs with one component each would be merged into one graph with two components where the components are exactly the two original graphs.

Definition 3.3 (Union graph [Szw03, p. 111])**.** $G_1 \cup G_2$ is the graph such that $V(G_1 \cup G_2) = V(G_1) \cup V(G_2)$ and $E(G_1 \cup G_2) = E(G_1) \cup E(G_2)$.

The following definition is the join of two graphs and builds on the union of two graphs by adding edges between two vertices if they are from different graphs.

Definition 3.4 (Join graph [Szw03, p. 111])**.** $G_1 + G_2$ is the graph where $V(G_1 + G_2) = V(G_1) \cup V(G_2)$ and $E(G_1 + G_2) = E(G_1) \cup E(G_2) \cup [V(G_1) \times V(G_2)]$.

Next, we define the cartesian product of two graphs. The vertex set is the cartesian product of the vertices of the two graphs. Two pairs of vertices are adjacent if the corresponding vertices in the graphs are adjacent.

Definition 3.5 (Cartesian product graph [Szw03, p. 111])**.** $G_1 \times G_2$ is the graph where $V(G_1 \times G_2) = V(G_1) \times V(G_2)$ and for $v_1, w_1 \in V(G_1)$ and $v_2, w_2 \in V(G_2)$, $\{v_1, v_2\}$ and $\{w_1, w_2\}$ are adjacent vertices in $G_1 \times G_2$ if and only if $\{v_1, w_1\} \in E(G_1)$ and $\{v_2, w_2\} \in E(G_2)$.

The last definition is the dot product, which is similar to the cartesian product defined above, except that pairs whose intersection is not empty can also be adjacent.

Definition 3.6 (Dot product graph [Szw03, p. 111])**.** $G_1 \circ G_2$ is the graph where $V(G_1 \circ G_2) = V(G_1) \times V(G_2)$ and edges are defined as follows. For $v_1, w_1 \in V(G_1)$ and $v_2, w_2 \in V(G_2)$, $\{v_1, v_2\}$ and $\{w_1, w_2\}$ are adjacent vertices in $G_1 \circ G_2$ if and only if *(i)* $v_1 = w_1$ and $\{v_2, w_2\} \in E(G_2)$, or *(ii)* $v_2 = w_2$ and $\{v_1, w_1\} \in E(G_1)$, or *(iii)* $\{v_1, w_1\} \in E(G_1)$ and $\{v_2, w_2\} \in E(G_2)$.

We now want to apply the clique operator to the binary graph operations defined above and try to represent the results using known operators. This will be done in the next corollary. The given proof is based on the proof in [Szw03].

Corollary 3.7 ([Szw03, p. 111])**.** *It holds that:*

(i) $K(G_1 \cup G_2) \simeq K(G_1) \cup K(G_2)$.

(ii) $\overline{K(G_1 + G_2)} \simeq \overline{K(G_1)} \times \overline{K(G_2)}$.

(iii) $K(G_1 \circ G_2) \simeq K(G_1) \circ K(G_2)$.

Proof. (i): The isomorphy follows directly from the definition as follows: Cliques in $G_1 \cup G_2$ are either completely contained in G_1 and thus a bag in $K(G_1)$ or completely contained in G_2 and thus a bag in $K(G_2)$. Edges in $K(G_1 \cup G_2)$ exist only between two cliques from the same graph. As a result $K(G_1 \cup G_2)$ is isomorphic to the union of the two clique graphs $K(G_1) \cup K(G_2)$.

(ii): Since two vertices v_1, v_2 with $v_1 \in V(G_1)$ and $v_2 \in V(G_2)$ are by definition adjacent in $G_1 + G_2$, it follows that maximal cliques in $G_1 + G_2$ always consist of a maximal clique C_1 in G_1 and a maximal clique C_2 in G_2. It follows directly that this induces a bijection between $V(K(G_1 + G_2))$ and $V(K(G_1)) \times V(K(G_2)) = V(K(G_1) \times K(G_2))$. In $\overline{K(G_1)}$, two cliques C_1, C_1' are adjacent if and only if $C_1 \cap C_1' = \emptyset$. So by definition two bags $C = (C_1, C_2), C' = (C_1', C_2')$ in $\overline{K(G_1)} \times \overline{K(G_2)}$ are adjacent, if and only if $C_1 \cap C_1' = \emptyset$ (C_1 and C_2' are adjacent in $\overline{K(G_1)}$) and $C_2 \cap C_2' = \emptyset$. Since in $K(G_1 + G_2)$ two cliques C and C' are adjacent if and only if their corresponding pair of cliques C_1, C_2

and C_1', C_2' in G_1 and G_2 satisfy $C_1 \cap C_1 \neq \emptyset$ or $C_2 \cap C_2' \neq \emptyset$. It follows that C and C' are adjacent in $\overline{K(G_1 + G_2)}$ if and only if $C_1 \cap C_1' = \emptyset$ and $C_2 \cap C_2' = \emptyset$, which proves the isomorphy.

(*iii*): Similar to (*ii*), the maximal cliques in $G_1 \circ G_2$ consist of a maximal clique in G_1 and one in G_2 by definition of the graph dot product. Therefore, we obtain a bijection from $V(K(G_1 \circ G_2))$ to $V(K(G_1) \circ K(G_2))$. Furthermore, two cliques C and C' are adjacent in $K(G_1 \circ G_2)$ if and only if they intersect nontrivially, which is the case if and only if at least one of the corresponding clique pairs C_1, C_1' and C_2, C_2' in G_1 and G_2 has a non-empty intersection. Likewise, two bags $C = (C_1, C_2)$ and $C' = (C_1', C_2)$ in $K(G_1) \circ K(G_2)$ are adjacent if and only if $C_1 \cap C_1' \neq \emptyset$ or $C_2 \cap C_2' \neq \emptyset$ by definition of the graph dot product. □

3.3 Characterization of clique graphs

A natural question that might arise when thinking about the clique graph operator is, whether for any graph G, there exists a graph H such that $K(H) = G$. In other words: Is every graph a clique graph?

It turns out that this question was answered quite early in the study of the clique graph operator. In the following, we will look at a characterization of clique graphs based on the Helly property first proven in [RS71]. We will see that not every graph is a clique graph and find a characterization of exactly those graphs that are.

To do so, we first need to define the Helly property for general sets.

Definition 3.8 (Helly property [Szw03, p. 111])**.** Let $\mathcal{F}$ be a family of subsets of some set. $\mathcal{F}$ satisfies the Helly property if for every subfamily $\{S_1, \ldots, S_n\} \subseteq \mathcal{F}$ with $S_i \cap S_j \neq \emptyset$ for all $i, j \in \{1, \ldots, n\}$, the total intersection is non-empty, i.e.

$$\bigcap_{i=1}^{n} S_i \neq \emptyset.$$

Note in the above definition that we only want to consider subfamilies $\{S_1, \ldots, S_n\} \subseteq \mathcal{F}$ such that the pairwise intersection of S_i and S_j is non-empty, as opposed to arbitrary subfamilies. For our purposes, we will choose $\mathcal{F}$ such that it is a family of subsets of $V(G)$. More precisely, $\mathcal{F}$ will consist of (maximal) cliques. This is specified in the next definition.

Definition 3.9 (Clique-Helly graph [Szw03, p. 111])**.** A graph is clique-Helly if the set of all maximal cliques satisfies the Helly property.

The figure 3.2 shows a clique-Helly graph. Note that all maximal cliques have the vertex labeled 2 in common.

From this property we can derive a sufficient condition for clique graphs, as in the following theorem. The proof of the theorem is based on the proof in [Ham68, p. 195].

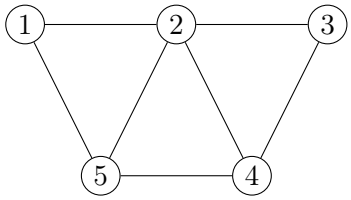


Figure 3.2: Clique-Helly graph

Theorem 3.10 ([Ham68, p. 195])**.** *Clique-Helly graphs are clique graphs.*

Proof. Let G be a clique-Helly graph and $\mathcal{F}$ be the set of all maximal cliques of G. We want to construct a graph H such that $K(H) \simeq G$. For this, define H as follows. $V(H) = V(G) \cup \mathcal{F}$ and for $v, w \in V(G)$ and $S_i, S_j \in \mathcal{F}$:

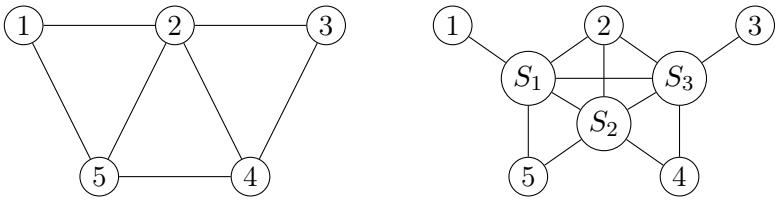
$\{v, S_i\} \in E(H)$ if and only if $v \in S_i$.

$\{S_i, S_j\} \in E(H)$ if and only if $S_i \neq S_j$ and $S_i \cap S_j \neq \emptyset$.

$\{v, w\} \notin E(H)$.

Now consider $C(v) = \{v\} \cup \{S_i \in \mathcal{F} \mid v \in S_i\}$ for $v \in V(G)$. It is easy to see that $C(v)$ is a maximal clique in H, since all $w \in V(G)$ are not adjacent to v, all S_j adjacent to v are already in $C(v)$ and $\{v\} \subseteq S_i \cap S_j \neq \emptyset$ for $S_i, S_j \in C(v)$ by construction. We now claim that there are no other maximal cliques in H. Let $C \subseteq V(H)$ be a clique in H. If $v \in C$ for some $v \in V(G)$ we get that $C \subseteq C(v)$. Otherwise C is a set of pairwise intersecting sets in $\mathcal{F}$ which implies that there is a common vertex $v \in V(G)$ since $\mathcal{F}$ satisfies the Helly property and thus $C \subsetneq C(v)$. This proves that $\varphi : V(G) \to V(K(H)), v \mapsto C(v)$ is a bijection.
Moreover, it can be shown that φ preserves adjacency, since for every edge $xy \in E(G)$ there exists a maximal clique S_{xy} containing xy, which is adjacent to both x and y in H. Consequently, $S_{xy} \in C(x) \cap C(y)$, which implies that $C(x)C(y) = \varphi(x)\varphi(y) \in E(K(H))$ as desired. Conversely, if $\varphi(x)\varphi(y) \in E(K(H))$, we can identify $\varphi(x)$ and $\varphi(y)$ with $C(x)$ and $C(y)$, respectively. Since $C(x)C(y) \in E(K(H))$, it follows by construction that there exists $S_i \in \mathcal{F}$ such that $x, y \in S_i$. This in turn implies that $xy \in E(G)$ as desired. Thus, we have shown that $K(H) \simeq G$. □

The technique of constructing a graph H such that $K(H) \simeq G$ for some given G, as used in the proof above, is commonly used in the context of the clique graph operator. We will reuse this construction for the proof of theorem 3.21. Since we have explicitly constructed a clique graph inverse, we can give an inverse clique graph for the clique-Helly graph in figure 3.2. This can be seen in figure 3.3.

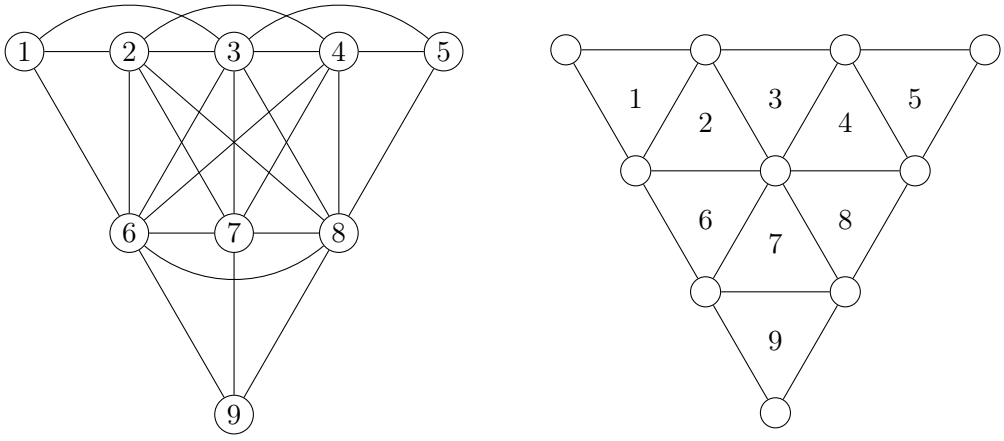


(a) Clique-Helly Graph (b) Clique operator inverse

Figure 3.3: Clique-Helly graph and the clique graph operator inverse according to the construction in the proof of theorem 3.10

However, we will see that the clique-Helly condition in the above theorem is too strict for a characterization of clique graphs. The following example shows a graph that is not a clique-Helly graph, but is still the clique graph of another graph.

Example 3.11. The graph G in figure 3.4 is not clique-Helly because the cliques $S_1 = \{1, 2, 3, 6\}, S_2 = \{3, 4, 5, 8\}, S_3 = \{6, 7, 8, 9\}$ intersect pairwise but do not share a common vertex.



(a) Non clique-Helly graph $G = K(H)$ (b) H

Figure 3.4: Graph that is a clique graph but not clique-Helly

Therefore, one might try to weaken the clique-Helly graph condition to allow more graphs, which is exactly what was done in [RS71, p. 103]. The following definition for edge covers (by cliques) is the weaker condition for the subsequent characterization.

Definition 3.12 (Edge cover (by cliques) [Szw03, p. 111])**.** An edge cover (by cliques) of a graph G is a set C of cliques of G, such that every edge of G has both its ends in some clique in C.

It is clear that if a graph is a clique-Helly graph, it admits an edge cover (by cliques) satisfying the Helly property. Hence admitting an edge cover by cliques is a weaker condition than the clique-Helly property. The proof of theorem 3.10 can be reused almost exactly for one of the directions of the following theorem which characterizes clique graphs with the helly property. The other direction of the proof is based on the proof presented in [RS71].

Theorem 3.13 ([RS71, p. 103])**.** *A graph is a clique graph if and only if it admits an edge cover (by cliques) which satisfies the Helly property.*

Proof. Let G and H be two graphs such that $K(H) \simeq G$. We want to show that G admits an edge cover by cliques that satisfies the Helly property. For this, let $V(H) =$

$\{h_1, \ldots, h_n\}$, $V(G) = \{g_1, \ldots, g_m\}$ and $C_1, \ldots, C_m$ be the corresponding maximal cliques in H. For $i \in \{1, \ldots, n\}$, define $S_i = \{g_j \in V(G) \mid h_i \in C_j\}$. Each S_i is a clique in G, because if $g_j, g_k \in S_i$, then $h_i \in C_j \cap C_k$, which implies that $g_j g_k \in E(G)$.
We now claim that $\mathcal{F} = \{S_i \mid i \in \{1, \ldots, n\}\}$ is an edge cover (by cliques) of G which satisfies the Helly property. For this, consider $g_j, g_k \in V(G)$ with $g_j g_k \in E(G)$. It holds that $C_j \cap C_k \neq \emptyset$. Thus, there exists an $h_i \in C_j \cap C_k$, such that $S_i \in \mathcal{F}$ and $g_i, g_j \in S_i$ which proves the edge cover property. To prove the Helly property, let $S_{i_1}, \ldots, S_{i_p} \in \mathcal{F}$ intersect pairwise. Then for all j, k there is a $g_{jk} \in S_{i_j} \cap S_{i_k}$. By definition, $h_{i_j}, h_{i_k} \in C_{jk}$, which implies that $h_{i_j} h_{i_k} \in E(H)$. Since h_{i_j} is fixed for any fixed S_{i_j} and arbitrary S_{i_k}, it follows that $\{h_{i_1}, \ldots, h_{i_p}\}$ form a clique in H and can therefore be completed to a maximal clique C_s. Thus $g_s \in \bigcap_{l=1}^{p} S_{i_l} \neq \emptyset$, which proves the desired.
Proving the other implication is analogous to the proof of theorem 3.10 if instead $\mathcal{F}$ is chosen as the edge cover (by cliques) that satisfies the Helly property. □

Now that we have derived a characterization of clique graphs and more specifically a necessary condition for a graph to be a clique graph, we can give an example of a non clique graph.

Example 3.14. With theorem 3.13 it can be shown that the graph in figure 3.5 is not a clique graph. The edges of the outer 3-cliques must be covered by 3-cliques, otherwise three 2-cliques (e.g. $\{1, 2\}, \{2, 4\}, \{1, 4\}$) would intersect pairwise and have no common vertex (the case of the middle 3-clique is analogous). Thus any edge cover (by cliques) would have to contain the cliques $\{1, 2, 4\}, \{2, 3, 5\}, \{4, 5, 6\}$, which do not satisfy the Helly property.

The above theorem 3.13 is not the only characterization of clique graphs that has been found, see for example [AG03]. However, none of the characterizations that have been found are practical enough to have led to a fast recognition algorithm. In fact, it has been shown that clique graph recognition, i.e. the question of whether a given graph is a clique graph, is $\mathcal{NP}$-complete [Alc+06].

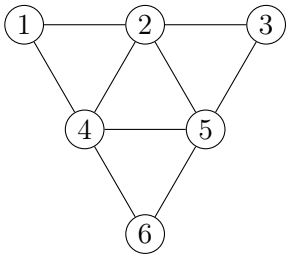


Figure 3.5: Non clique graph

3.4 Clique graphs of classes of graphs

Another question that has been discussed in many papers surrounding the clique operator is the characterization and recognition of $K(\mathcal{A})$ for a given graph class $\mathcal{A}$, see for example [PS99a; Hed84; BP91]. In other words, if the clique graph operator is applied to a given class of graphs, can the resulting class of graphs be characterized? We have already seen that clique-Helly graphs as defined in 3.9 play a special role in the theory of the clique graph operator, since being clique-Helly is a sufficient condition for a graph to be a clique graph, see 3.10. Therefore, we will start by studying the clique graphs of this class of graphs.

It turns out that the class of clique-Helly graphs is fixed under the clique graph operator as described in the following theorem. The provided proof is based on the proof in [Szw03].

Theorem 3.15 ([Szw03, p. 114])**.**

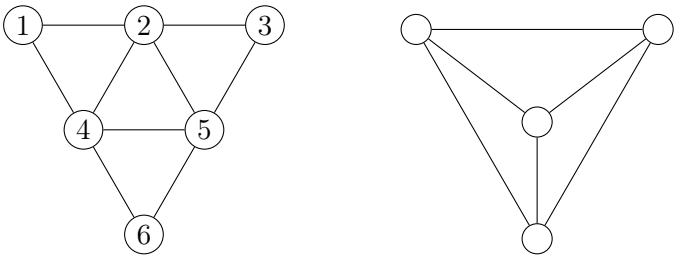
$$K(\text{CLIQUE-HELLY}) = \text{CLIQUE-HELLY}\,.$$

Proof. Let G be a clique-Helly graph and $H = K(G)$. We want to prove that H is clique-Helly, i.e. show that the maximal cliques of H satisfy the Helly property. For this, let $\mathcal{C}$ be a set of pairwise intersecting maximal cliques in H and $C_i \in \mathcal{C}$. Note that the vertices in C_i correspond to pairwise intersecting maximal cliques in G. We will therefore write $v \in C_i$ for some $v \in G$ as shorthand for referring to the corresponding clique in G of $C_i \in V(H)$. Since G is clique-Helly, this implies that there exists a vertex $v_i \in G$ that is common to all vertices in $C_i \subseteq V(H)$. Now, we claim that the set of such vertices v_i, $X = \{v_i \mid C_i \in \mathcal{C}\}$, is a clique in G. For this, let $C_j \in \mathcal{C}$ be another maximal clique in H. By definition of $\mathcal{C}$, C_i and C_j intersect in a vertex $C_{ij} \in H$, which (in G) contains both v_i and v_j. This implies that v_i and v_j are adjacent in G, which shows that X is a clique in G. Now, the completion of X to a maximal clique in G is a vertex in H which is contained in all C_i by construction. Thus, the intersection of the C_i is non-empty and $K(G)$ is clique-Helly as desired.

It remains to show that any clique-Helly graph G is the clique graph of a clique-Helly graph. To do so, consider the construction of the graph H in the proof of theorem 3.10. We want to show that H is a clique-Helly graph which would be sufficient to prove the desired. Therefore, consider a set of pairwise intersecting maximal cliques $\mathcal{C}$ in H. By construction, cliques $C_i, C_j \in \mathcal{C}$ are induced by vertices $v_i, v_j \in G$ and can only intersect in vertices S_{ij} which are themselves cliques in G. Suppose that $S_{ij} \in C_i \cap C_j$, then $v_i, v_j \in S_{ij}$ by construction. This implies that v_i and v_j are adjacent. Thus the v_i which induce the cliques $C_i \in \mathcal{C}$ form a clique C in G. The completion of C to a maximal clique in G is a vertex S in H, which in turn is contained in all C_i. Therefore, the intersection of the C_i is non-empty. This shows that the constructed graph H is a clique-Helly graph. □

Note that the above theorem does not imply that the clique operator inverse of the class of clique-Helly graphs is equal to the class of clique-Helly graphs, i.e. $K^{-1}(\text{CLIQUE-HELLY}) = \text{CLIQUE-HELLY}$. In fact, as seen in figure 3.6, this cannot hold.

Next, we want to return to a class of graphs we have previously considered in the context of treewidth, chordal graphs as seen in definition 2.9. We have already discussed that they can be characterized by the fact that they allow perfect elimination orderings. However, there are several characterizations of chordal graphs, depending on the application. The following is another characterization in the context of intersection graphs, which will help us describe the image of the class of chordal graphs under the clique graph operator.



(a) Non clique-Helly graph G

(b) Clique-Helly graph $K(G) \simeq K_4$

Figure 3.6: Non clique-Helly graph with clique-Helly clique graph

Theorem 3.16 ([Gav74, p. 5])**.** *G is chordal if and only if it is the intersection graph of subtrees of a tree.*

Note that the term intersection graph in the above theorem refers to definition 3.1. We want to prove the above theorem by proving the next theorem 3.18, of which the above is an immediate consequence. For this, however, we need the following lemma about subtrees of trees. It can be interpreted as the Helly property, see definition 3.8, for subtrees of trees. Note that for the proof we need no prerequisites other than trivial tree properties, which is why we were able to use the following lemma in the proof of the previous lemma 2.6.

Lemma 3.17. *Let T be a tree and $\mathcal{F}$ be a family of subtrees of T such that every pair of trees in $\mathcal{F}$ has a non-empty intersection of vertices. Then*

$$\bigcap_{S \in \mathcal{F}} V(S) \neq \emptyset.$$

Proof. Without loss of generality, assume that all subtrees in $\mathcal{F}$ have at least two vertices. Otherwise the claim is trivially true, since all subtrees would contain the unique vertex of the tree with size 1. We want to prove the desired by induction over the number of vertices of T. The case $|V(T)| = 1$ is treated when considering single vertex subtrees. Suppose that the claim holds for all trees with at most $n \geq 1$ vertices and let T be a tree with $n+1$ vertices and $\mathcal{F}$ be a family of subtrees of T with pairwise non-empty intersection of vertex sets. Let v_0 be a vertex of T with degree one (which exists because T is a tree) and v_1 be its unique neighbor. If v_0 is contained in all $S \in \mathcal{F}$, we have nothing to show. Therefore assume that v_0 is not contained in all $S \in \mathcal{F}$. Since the $S \in \mathcal{F}$ pairwise intersect, this implies that there does not exist an $S \in \mathcal{F}$ with $S = \{v_0\}$. Now, let $\mathcal{F}' = \{S - v_0 \mid S \in \mathcal{F}\}$ and $T' = T - v_0$. Note that $\mathcal{F}'$ is a family of non-empty subtrees of T' that have pairwise intersecting vertices, since if $S_0, S_1 \in \mathcal{F}$ intersect at v_0, they both contain v_1. By our induction hypothesis, there exists a common vertex $v \in \bigcap_{S' \in \mathcal{F}'} V(S')$ and by construction $v \in \bigcap_{S \in \mathcal{F}} V(S)$, that is $\bigcap_{S \in \mathcal{F}} V(S) \neq \emptyset$ as desired. □

The above lemma allows us to prove the following theorem about the characterization of chordal graphs as intersection graphs of subtrees.

We will however, only prove the implications $(iii) \implies (ii) \implies (i)$, leaving out the implication $(i) \implies (iii)$, since the latter is an immediate consequence of lemma 4.1 and will be proved later in the context of proving the optimality of our heuristic on chordal graphs. The following proofs are based on the proofs presented in [Gol80, p. 92].

Theorem 3.18 ([Gol80, p. 92])**.** *Let G be a graph. The following statements are equivalent:*

(i) *G is a chordal graph.*
(ii) *G is the intersection graph of a family of subtrees of a tree.*
(iii) *There exists a tree $T = (\mathcal{F}, \mathcal{E})$ whose vertex set $\mathcal{F}$ is the set of maximal cliques of G such that for every $v \in V(G)$, the induced subgraph $T[\{C \in \mathcal{F} \mid v \in C\}]$, consisting of all cliques containing v, is connected.*

Proof. $(iii) \implies (ii)$: Assume that there exists a tree T with the properties in (iii). Let $T_v = T[\{C \in \mathcal{F} \mid v \in C\}]$ for $v \in G$. We now claim that G is (isomorphic to) the intersection graph H of $A = \{T_v \mid v \in V(G)\}$, i.e. $H = \Omega(A)$. For this, let φ be the canonical isomorphism mapping $V(G)$ to A and $v, w \in V(G)$ with $vw \in E(G)$. Then, $v, w \in C$ for some maximal clique $C \in \mathcal{F}$ in G and therefore $C \in V(T_v)$ and $C \in V(T_w)$, i.e. T_v is adjacent to T_w in H. Similarly, if T_v and T_w are adjacent in H, there exists $C \in V(T_v) \cap V(T_w)$ with $v, w \in C$, implying $vw \in E(G)$.

$(ii) \implies (i)$: Suppose G is not chordal, i.e. there exists a chordless cycle $v_0, \dots, v_{k-1}, v_0$ with $k \geq 4$. Let $T_0, \dots, T_{k-1}$ be the associated subtrees of T that exist by our premise (ii). Note that $V(T_i) \cap V(T_j) \neq \emptyset$ if and only if i and j differ by at most 1 modulo k. For the rest of this proof, writing $i-1$ or $i+1$ as indices is shorthand for $i-1 \mod k$ or $i+1 \mod k$, respectively. Furthermore, all of the following paths are meant as paths in T.

For $i \in \{0, \dots, k-1\}$ let a_i be a vertex in $T_i \cap T_{i+1}$. Furthermore, let b_i be the last common vertex on the (unique) paths from a_{i-1} to a_i and a_i to a_{i+1}, i.e., starting at a_i and walking along the path to a_{i+1}, the last vertex that is also in the path from a_{i-1} to a_i. Because $a_{i-1}, a_i \in T_i$ and T_i is connected by construction, the paths from a_{i-1} to a_i and a_i to a_{i+1} lie in T_i and T_{i+1} respectively. Therefore, b_i is in $T_i \cap T_{i+1}$. Now define P_{i+1} as the path connecting b_i and b_{i+1}. Clearly $P_i \subseteq T_i$, since $\{b_{i-1}, b_i\} \subseteq T_i$ and T_i is connected. Therefore, $P_i \cap P_j = \emptyset$ if i and j differ by more than 1 modulo k. Figure 3.7 shows an exemplary visualization of this construction.

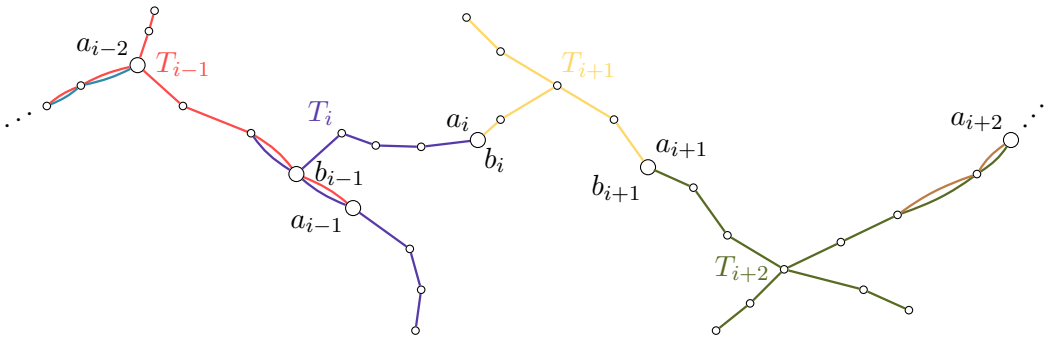


Figure 3.7: Construction used in the proof of theorem 3.18.

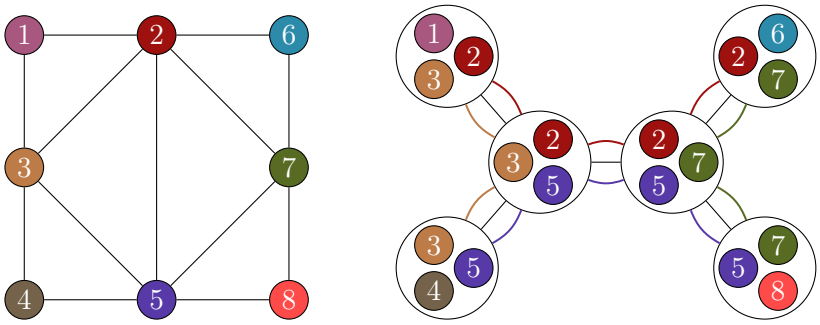
We will now show that $P_i \cap P_{i+1} = \{b_i\}$ for $i \in \{0, \ldots, k-1\}$. Suppose that there exists $x \in P_i \cap P_{i+1}$ with $x \neq b_i$. From $x \in P_i$ it follows that x is on the path from b_{i-1} to b_i. Therefore, by construction of the b_i we can conclude that x is on the path from a_{i-1} to a_i. By analogous arguments, x is on the path from a_i to a_{i+1}. This however means that x is a later common vertex of these two paths than b_i, which is a contradiction. Thus, $P_i \cap P_{i+1} = \{b_i\}$ holds.

In summary, we have shown that for $i \in \{0, \ldots, k-1\}$ it holds that $P_i \cap P_j = \{b_i\}$ if $j = i+1$ and $P_i \cap P_j = \emptyset$ otherwise ($j \notin \{i-1, i\}$). With this and the fact that our supposed chordless cycle is of length ≥ 4, we can conclude that $\bigcup_{i \in \{0,\ldots,k-1\}} P_i$ is a (simple) cycle in T, which is a contradiction to T being a tree. □

We can now formulate the necessary definition for the alternative characterization of chordal graphs, which we will use for the rest of this section.

Definition 3.19 (Chordal graph as the intersection graph of subtrees [SB94, p. 1])**.** Let G be a chordal graph given as the intersection graph of subtrees of a tree T. The subtree of T corresponding to a vertex $v \in V(G)$ is called the representative subtree of v and is denoted by $T(v)$. The tree together with the representative subtrees form a tree representation of G. A minimal representation is a tree representation such that $|V(T)|$ is the smallest possible.

In figure 3.8 a chordal graph is shown with a corresponding representation as an intersection graph. The chordal graph and its representation as an intersection of subtrees is quite similar to the tree decomposition example in figure 2.1. This similarity is not by coincidence and will be revisited when proving the optimality of our heuristic in section 4.5.



(a) Chordal graph G

(b) Representation of G as an intersection graph of subtrees

Figure 3.8: A chordal graph with eight vertices, represented as the intersection graph of eight subtrees of a six node tree. The representative subtree of a vertex v is highlighted with edges in the corresponding color.

It has been shown that a minimal representation is precisely one in which every vertex of T corresponds to a maximal clique of G and for every $v \in V(G)$, the subtree $T(v)$ is formed exactly by the vertices of T corresponding to those maximal cliques of G containing v, similar to 3.18 (*iii*), see [Bun74; Gav74]. We will use this fact in the proof of our characterization of the image of chordal graphs under the clique operator. More precisely, we will characterize the clique graphs of chordal graphs as trees that have been expanded by adding certain edges. This so called class of expanded trees is defined as follows.

Definition 3.20 (Expanded tree [SB94])**.** G is an expanded tree if it admits a spanning tree $T(G)$ such that for every edge $\{v, w\} \in E(G)$, the vertices of the $v-w$ path in T form a clique in G. In this case, we call $T(G)$ a canonical tree of G.

We can now characterize the clique graphs of chordal graphs as the class of expanded trees. The following proof is based on [SB94, p. 2]. In the proof of the implication (*iii*) $\Rightarrow$ (*i*) we will reuse the construction from the proof of theorem 3.10 to construct the clique inverse graph H. Since we cannot rely on the clique-Helly property for this, we will replace its use with lemma 3.17.

Theorem 3.21 ([SB94, p. 2])**.** *The following are equivalent.*

(i) G *is the clique graph of a connected chordal graph* H*.*
(ii) G *admits a spanning tree* T*, such that for every* $X \in V(G), N_G[X] \cap T$ *is a (connected) subtree of* T*.*
(iii) G *is an expanded tree.*

Proof. (*i*) $\Rightarrow$ (*ii*) : Let H be a connected chordal graph, $G = K(H)$ its clique graph and T a minimal representation of H as the intersecion of subtrees as defined in definition 3.19. Now let G' be the graph obtained by adding the edges to T that transform each representative subtree $T(v)$ into a $|T(v)|$-clique. Note that we have a natural bijection from $V(G)$ (the maximal cliques of H) to $V(G')$ (the vertices in T).

We now claim that this bijection induces an isomorphism between G and G'. For this, let C_1, C_2 be two maximal cliques of H, X_1, X_2 and X_1', X_2' the corresponding vertices in G and G', respectively. Suppose $\{X_1, X_2\} \in E(G)$. Then there exists a $w \in V(H)$ such that $w \in C_1 \cap C_2$. This implies that the subtree $T(w)$ of the minimal representation of H contains $X_1', X_2' \in V(G')$. By construction we have $\{X_1', X_2'\} \in E(G')$. Conversely, if $\{X_1', X_2'\} \in E(G')$, then by construction either $e = \{X_1', X_2'\} \in E(T)$ or $\{X_1', X_2'\} \subseteq V(T(w))$ for some $w \in V(H)$. In the first case $\{w\} \subseteq X_1' \cap X_2' \neq \emptyset$ for some $w \in V(H)$, since H is connected and otherwise $T - e$ would be a (non connected) representation of H as an intersection graph. Therefore, in both cases there exists a representative subtree $T(w)$ containing X_1', X_2'. That is, $w \in C_1 \cap C_2$, which implies that $\{X_1, X_2\} \in E(G)$. So $G \simeq G'$ and since T is a spanning tree of G', it spans G as claimed.

Now notice that for every $X \in V(G)$ and $Y \in N_G[X]$, X and Y correspond to two cliques C_X and C_Y in H. These cliques have a common vertex $w \in H$, since $\{X, Y\} \in E(G)$. This implies that all vertices of the representative subtree $T(w)$ are

in $N_G[X]$, i.e. $V(T(w)) \subseteq N_G[X]$. Since $T(w)$ is a subtree, X and Y are connected in $N_G[X] \cap T$. Since Y was arbitrarily chosen in $N_G[X]$, $N_G[X] \cap T$ is connected.

$(ii) \Rightarrow (iii)$: Let G be a graph admitting a spanning tree T with the properties given in (ii). It suffices to show that T is a canonical tree of G. For this, let $\{v, w\} \in E(G)$ and let $v = v_0, \ldots, v_r = w$ be the $v - w$ path in T. We want to show by induction over r that the vertices $v_0, \ldots, v_r$ form a clique in G. The case $r = 1$ is obvious. Now suppose that $\{v_0, \ldots, v_{r-1}\}$ is a clique in G with $r > 1$. Observe that v and w are adjacent in G, $N_G[v] \cap T$ is connected and similarly $N_G[w] \cap T$ is connected. In particular, there is only one $v - w$ path in T and for $N_G[w] \cap T$ to be connected, if $v \in N_G[w]$, $v_0, \ldots, v_{r-1}$ must be in $N_G[w]$. This implies that $\{v_0, \ldots, v_r\}$ is a clique in G, i.e. T is a canonical tree of G.

$(iii) \Rightarrow (i)$: Consider an expanded tree G. We construct a connected chordal graph H such that $K(H) = G$. For this, let $\mathcal{F}$ be the set of maximal cliques in G and let H be the same graph as constructed in the proof of theorem 3.10 using G and $\mathcal{F}$. Furthermore, let T denote a canonical tree of G. We show that every maximal clique $C \in \mathcal{F}$ of G induces a subtree in T, which is sufficient to show that H is the intersection graph of subtrees. That is, in the construction of H in the proof of theorem 3.10, $V(H)$ consists of cliques $C \in \mathcal{F}$ and vertices $v \in V(G)$. The cliques then correspond to subtrees and the vertices can be seen as single vertex subtrees. Therefore, H as constructed in 3.10 would be the intersection graph of subtrees.

Let v and z be two vertices in $C \in \mathcal{F}$ (G is connected and the case $|V(G)| = 1$ is trivial) and P the $v - z$ path in T. Observe that since T is a tree, for every vertex w of C, the union of the paths $w - v$ and $w - z$ covers P, that is every vertex in P appears in either the $w - v$ or $w - z$ path. Combined with the fact that $\{w, v\}$ and $\{w, z\}$ are edges of the expanded tree G, it follows that w is adjacent to every vertex of P. Overall, this shows that every vertex in C is adjacent to every vertex in P, which implies that the vertices of P belong to C, since C is a maximal clique. Since v and z were chosen arbitrarily in C, $C \cap T$ is connected and C induces a subtree of T. Consequently H is a chordal graph.

It remains to show that $G = K(H)$. Note that, similar to the proof of theorem 3.10, we have an injective function φ mapping vertices $v \in V(G)$ to maximal cliques $C(v)$ in H (v maps to the set containing v and the cliques formed by the maximal cliques of G containing v). We want to show that this function is surjective. For this, let C be a maximal clique in H. If there is a $v \in V(G)$ with $v \in C$, then clearly $C \subseteq C(v)$. Otherwise C consists of maximal cliques $C_0, \ldots, C_k$ of G. As shown above, each clique C_i induces a subtree $C_i \cap T$. These subtrees have pairwise non-empty intersections which implies by lemma 3.17 that there is a common vertex v for which in turn $C \subseteq C(v)$, i.e., the function is surjective. The rest of the proof is the same as in 3.10. □

This concludes the classes of graphs whose image under the clique operator we want to discuss. For an overview of other classes of graphs and their corresponding classes of clique graphs, see table 5.1. in [Szw03, p. 116].

3.5 Clique graphs of bounded size

Especially for computational applications of clique graphs it is interesting to know how big the clique graph of a graph can be. As can be seen directly from definition 3.2 of the clique graph operator, the number of nodes in the clique graph is equal to the number of maximal cliques in a graph. Therefore, we want to use the following chapter to look at different sufficient conditions for a graph to have few maximal cliques and thus a clique graph of bounded size (bounded number of vertices and therefore edges). Note that, as before, we will refer to $|V(G)|$ as n and use $\kappa(G)$ for the number of maximal cliques of G.

Having a (polynomial) bound on the number of maximal cliques in a graph is particularly interesting because, in general, a graph can have up to $3^{\lfloor n/3 \rfloor}$ distinct maximal cliques, i.e. exponentially many in the number of vertices. This result was first published by Moon and Moser in [MM65, p. 23] and first proven in an IBM technical report, see [MM60]. Moreover, one can easily construct a graph that realizes this bound, as shown in figure 3.9 and described in the following example.

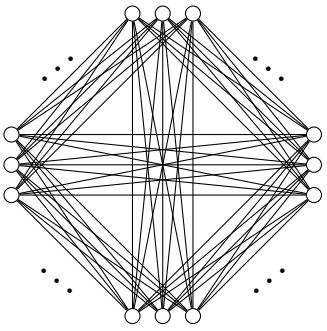


Figure 3.9: Turán graph $T(n, \frac{n}{3})$ with $3 \mid n$

Example 3.22. The figure 3.9 shows the Turán graph $T(n, \frac{n}{3})$ with $n \mid 3$. That is, a complete $\frac{n}{3}$-partite graph. The graph consists of n vertices and $\frac{n}{3}$ equally sized partitions, so each partition contains exactly 3 vertices. Now each maximal clique consists of exactly $\frac{n}{3}$ vertices, each from a different partition. As a result, it is easy to see that the graph contains $3^{n/3}$ maximal cliques, which realizes the upper bound.

We begin by looking at several simple sufficient conditions for a bound on $\kappa(G)$, before looking at two more general results about graphs with few maximal cliques. The latter will allow us to prove for some well-known classes of graphs that their number of maximal cliques is polynomially bounded. The following proofs are based on the proofs given in [PS99b]. Note that with $\mathcal{O}(\cdot)$ we refer to the standard big $\mathcal{O}$ notation.

Lemma 3.23 ([PS99b, p. 4])**.** *If $\omega(K(G))$ is bounded by a constant, then $\kappa(G)$ is in $\mathcal{O}(n)$.*

Proof. Suppose that $\omega(K(G)) \leq k$ for some $k \geq 0$. Note that any vertex v in G cannot belong to more than k maximal cliques, otherwise the cliques containing v would correspond to a clique of size strictly greater than k in $K(G)$. Hence $\kappa(G) \leq nk$, which implies that $\kappa(G) \in \mathcal{O}(n)$. □

The above lemma allows us to easily prove the following sufficient condition for a bounded number of maximal cliques.

Corollary 3.24 ([PS99b, p. 4]). *If $\Delta(K(G))$ is bounded by a constant, then $\kappa(G)$ is in $\mathcal{O}(n)$.*

Proof. Suppose that $\Delta(K(G)) \leq k$, for some positive constant k. Clearly then $\omega(K(G)) \leq \Delta(K(G)) + 1 \leq k + 1$. Therefore, by lemma 3.23, $\kappa(G)$ is in $\mathcal{O}(n)$. □

For the next lemma we want to write P_k to denote a chordless path on k vertices, i.e. a path of k vertices $P_k = v_1 \dots v_k$ such that v_i, v_j are not adjacent for $j \notin \{i-1, i+1\}$. Furthermore, we denote by C_k a chordless cycle on k vertices and by I_k an independent set of k vertices, i.e., k vertices that are all pairwise non-adjacent.

Lemma 3.25 ([PS99b, p. 4]). *Let G be connected. Let H be an induced subgraph of $K(G)$ such that $|V(H)| = k$. If H is isomorphic to either P_k, C_k or I_k, then $k \in \mathcal{O}(n)$.*

Proof. Let $V(H) = \{C_1, \dots, C_k\}$ (the vertices of H correspond to maximal cliques in G). Without loss of generality assume that $k \geq 3$ (the case $k = 1, 2$ is trivial since sufficiently large coefficients can be chosen for n, such that $k \in \mathcal{O}(n)$).

(P_k): Without loss of generality (otherwise rename the C_i), assume that H is the chordless path $C_1C_2 \dots C_k$. In this case, for $1 \leq i < j \leq k$ it holds that, $C_i \cap C_j \neq \emptyset$ for $j = i + 1$ and $C_i \cap C_j = \emptyset$ for $j \neq i + 1$. Let $v_i \in C_i \cap C_{i+1}$ for $1 \leq i \leq k - 1$. The v_i are pairwise distinct, since otherwise two non-consecutive cliques would intersect, which is a contradiction. Therefore, $k - 1 \leq n$, which implies that $k \in \mathcal{O}(n)$.

(C_k): Without loss of generality (otherwise rename the C_i), assume that H is the chordless cycle $C_1C_2 \dots C_{k-1}C_1$. In this case the induced subgraph $K(G)[\{C_1, \dots, C_{k-1}]$ is isomorphic to P_{k-1}. Thus, by the above case, $k - 1 \in \mathcal{O}(n)$, which implies that $k \in \mathcal{O}(n)$.

(I_k): Observe that each vertex $v \in V(G)$ can belong to at most one clique from $C_1, \dots, C_k$, since the C_i are non-adjacent in $K(G)$ and thus have an empty intersection. Furthermore, each C_i must contain at least two vertices, since G is connected and therefore does not contain any maximal cliques of size 1. This implies that $k \leq \frac{n}{2}$ and thus $k \in \mathcal{O}(n)$. □

From this we can easily derive a bound on the maximum number of maximal cliques of graphs whose clique graphs are isomorphic to chordless paths, cycles or independent sets.

Corollary 3.26 ([PS99b, p. 4]). *If $K(G)$ is isomorphic to a chordless path or cycle, then $\kappa(G)$ is in $\mathcal{O}(n)$.*

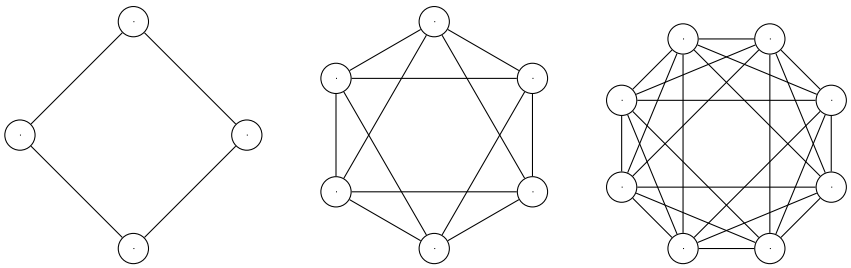
Proof. The claim follows from lemma 3.25 with $H = K(G)$. □

Also, using lemma 3.25, we can give a sufficient condition for a bounded number of maximal cliques using the chromatic number $\chi(K(G))$ of the clique graph.

Corollary 3.27 ([PS99b, p. 5])**.** *If $\chi(K(G))$ is bounded by a constant, then $\kappa(G)$ is in $\mathcal{O}(n)$.*

Proof. Let $\chi(K(G)) = c \leq k$ for some $k \geq 0$. Each color of $K(G)$ is an independent subgraph H_i of $K(G)$. By lemma 3.25 we get that $|V(H_i)| \in \mathcal{O}(n)$ and therefore $\kappa(G) = |V(K(G))| = \sum_{i=1}^{c} |V(H_i)| \in \mathcal{O}(cn) = \mathcal{O}(n)$. □

The above results are useful when information about the clique graph is already given. In most cases, however, only the graph G is known and computing $K(G)$ would involve computing all maximal cliques, which can be very expensive if G has many maximal cliques. The following two theorems solve this problem by allowing to make statements about the number of maximal cliques of graphs or classes of graphs without any information about their respective clique graphs. However, the proofs for both theorems require a lot of preparatory work, which is why we will present both theorems without their respective proofs, which can be found in their respective sources. Both theorems use the $\overline{pK_2}$ graph, which is the complement of the graph on $2p$ vertices, which is the disjoint union of p complete graphs on two vertices. These graphs can be represented as circles of even length, where every vertex is connected to every other vertex except the vertex on the exact opposite side of the circle. Figure 3.10 shows $\overline{pK_2}$ for $p \in \{2, 3, 4\}$.



(a) $\overline{2K_2}$ (b) $\overline{3K_2}$ (c) $\overline{4K_2}$

Figure 3.10: $\overline{pK_2}$ graphs for $p \in \{2, 3, 4\}$

The first theorem gives both a lower bound and an upper bound on the number of cliques of a given graph using the $\overline{pK_2}$ graph.

Theorem 3.28 ([BY89, p. 248])**.** *Let $G = (V, E)$ be a connected graph, let δ be the number of pairs $\{u, v\} \in V \times V$ such that $d(u, v) = 2$, and let p be the largest integer such that G contains an induced subgraph isomorphic to $\overline{pK_2}$. Then*

$$2^p \leq \kappa(G) \leq \delta^p + 1.$$

This theorem now leads to the next theorem, which characterizes classes of graphs with few maximal cliques, i.e. classes with a bound on the number of maximal cliques that is polynomial in the number of vertices. Note that hereditary classes of graphs are classes of graphs closed under induced subgraphs.

Theorem 3.29 ([Pri95, p. 950])**.** *A hereditary class of graphs $\mathcal{C}$ has few maximal cliques if and only if for some constant p it holds that, $\overline{pK_2} \notin \mathcal{C}$.*

We can, of course, use the last theorem 3.28 to easily prove a direction of theorem 3.29 as follows. If for a hereditary class of graphs $\mathcal{C}$ there exists a constant p such that $\overline{pK_2} \notin \mathcal{C}$, then of course $\overline{tK_2} \notin \mathcal{C}$ for all $t \geq p$. Therefore, $k(G) \leq n^p + 1$ for all $G \in \mathcal{C}$ by theorem 3.28. The other direction of the proof is more involved and can be found in [Pri95, p. 950].

We can now use the above two theorems to easily deduce for some well-known graph classes that they have few maximal cliques. Note that if a graph class is not hereditary, theorem 3.29 can simply be applied to its closure under induced subgraphs. A simple class of graphs with few maximal cliques is the class of trees. Although the class is not hereditary, the number of maximal cliques is of course equal to the number of edges $m = n - 1$ and similarly $\overline{2K_2}$ contains a cycle and is therefore neither a tree nor a subgraph of a tree. Similarly, we can show that planar graphs, i.e. graphs that can be embedded in the plane, have few maximal cliques, as done in the following corollary.

Corollary 3.30. *The class of planar graphs has few maximal cliques.*

Proof. Induced subgraphs of planar graphs are planar. Hence the class of planar graphs $\mathcal{C}$ is hereditary. Furthermore, using Kuratowski's theorem, a well-known characterization of planar graphs, see for example [Tho81], we can conclude that $\overline{3K_2}$ is not planar, since it contains the complete bipartite graph with two partitions of size three, i.e. $K_{3,3}$, as a subgraph. Therefore, $\overline{3K_2} \notin \mathcal{C}$, which implies the desired using theorem 3.29. □

Another known class of graphs we discussed already is the class of chordal graphs. Not only does the class have few cliques, but chordal graphs only have linearly many maximal cliques. However, we want to only show the following corollary which is an easy implication of theorem 3.29.

Corollary 3.31. *The class of chordal graphs has few maximal cliques.*

Proof. As already noted below definition 2.9, induced subgraphs of chordal graphs are chordal. Hence, the class of chordal graphs $\mathcal{C}$ is hereditary. The $\overline{2K_2}$ is a cycle of length greater than 3 without a chord and therefore not chordal. Theorem 3.29 now implies the desired. □

4 Clique graph treewidth heuristic

The previous chapters have laid the foundation for the following main chapter, in which we will develop a heuristic for computing an upper bound on the treewidth of a given graph using the clique graph operator.

We will first describe the idea of the heuristic and illustrate it with an example graph in subsection 4.1.2. Subsequently, we will discuss the different variants of the heuristic before evaluating the heuristic with different benchmarks in the next chapter.

Since the treewidth of a graph is the maximum of the treewidths of its components, see 2.5, we will only consider connected graphs. The heuristic can be applied to disconnected graphs by applying it to the components of the graph and taking the maximum over the components.

Note that in the following we will often refer to 'the heuristic', although in reality the heuristic refers to multiple variants of the same heuristic that differ in some parameters. Therefore, when we want to refer to a particular variant of the heuristic, we will specify it. Finally, we will use numbers to refer to the vertices of G, i.e., assume without loss of generality that $V(G) = \{1, \ldots, n\}$.

4.1 Idea

As discussed immediately below the definition of the clique graph operator, see definition 3.2, there is an inherent connection between the clique graph operator and tree decompositions. We have seen that for every tree decomposition of a graph G, for every clique of the graph G there must exist a bag in the tree decomposition containing the clique, see lemma 2.6. This is exactly what is done when constructing the clique graph of a graph G. Each vertex in the clique graph represents a maximal clique in G. Thus, the vertices in the clique graph of a graph can be interpreted as bags containing all the vertices of the corresponding maximal clique, as seen in figure 3.1.

In the following, we will refer to the vertices of the original graph as vertices or more precisely, vertices of the original graph and to the vertices of the clique graph (or vertices of subgraphs of the clique graph) as bags. This may seem ambiguous, since we also use the word bags for bags in tree decompositions, but as we will see, these seemingly different bags will be the same in the following construction. Moreover, when referring to the vertices $i \in I$ of a tree decomposition $(\{X_i \mid i \in I\}, (I, F))$, we will not distinguish between a vertex $i \in I$ and its corresponding bag X_i, e.g. we will refer to edges between bags when we are actually referring to edges between the corresponding vertices $i \in I$ in (I, F).

The above interpretation of clique graph vertices as bags leads us to the core idea of the heuristic. We want to construct the clique graph $K(G)$ of a given graph G and

then modify the clique graph in such a way that it yields a tree decomposition of G, see definition 2.1. In this case, the modified clique graph would be the tree (I, F) of the tree decomposition and the cliques corresponding to the vertices of the clique graph would induce the bags $\{X_i \mid i \in I\}$. In order to achieve this, we must first discuss how the clique graph needs to be adapted to obtain a tree decomposition.

The three criteria of a tree decomposition are listed in definition 2.1. Starting with the first, since every vertex of G forms a 1-clique and every clique is contained in some maximal clique, every vertex belongs to at least one maximal clique and thus to at least one bag of $K(G)$. Thus, the first criterion is already fulfilled. With an analogous argument one can observe that every edge of G is contained in at least one maximal clique in G and thus in a bag of $K(G)$. Therefore the second criterion of the definition is also satisfied.

Before checking the third property, we want to note that for a tree decomposition it is necessary that the graph (I, F) is a tree. In our construction, the graph (I, F) is the clique graph of G. However, in most cases, the clique graph of a graph is not a tree. Instead, we want to satisfy this condition by taking a spanning tree of the clique graph, which we will call the clique graph tree. More specifically, the first version of the heuristic computes a minimum spanning tree. How the weights of the clique graph tree are chosen for this is discussed in section 4.2 and an alternative way to compute a spanning tree of the clique graph is discussed in section 4.3.

Given a clique graph tree, only the third property of the tree decomposition definition remains to be satisfied. We want to consider the alternative formulation of the third property given below definition 2.1. It states that when considering the path between two bags X_i and X_j in the clique graph tree, each bag visited along the path must contain all vertices from the intersection of the bags X_i and X_j. We want to achieve this by 'filling the bags' with the necessary vertices along the paths between all pairs of bags in the clique tree graph.

The following figure 4.1 visualizes the process of filling bags along the path between two vertices. In the actual heuristic, this must be done between all pairs of bags in the clique graph tree, or more precisely, those pairs that have a non-empty intersection. Note that since our clique graph is a tree at this point, for each pair of bags there exists exactly one path connecting the two bags. The red bags in figure 4.1 have a non-empty intersection of $\{4, 8\}$, so every bag X_i along the path between the two red bags is updated to $X_i \cup \{4, 8\}$, i.e. filled.

(a) Clique graph tree before filling up bags along the path between the red bags

(b) Clique graph tree after filling up bags along the path between the red bags

Figure 4.1: Part of a clique graph tree to visualize the filling of bags along the path between two bags.

This way, the alternative third criterion for a tree decomposition is satisfied and we obtain a tree decomposition whose width serves as an upper bound on the treewidth of the graph. The width of the decomposition in this case, see definition 2.1, is just the size of the largest bag in the decomposition minus one.

It is easy to see that the method described above is correct in the sense that, given a graph G, it computes a correct upper bound on the treewidth of G.

4.1.1 Pseudo code

Having established the idea of the heuristic, we want to look at possible pseudocode of the heuristic, as seen in listing 4.1.

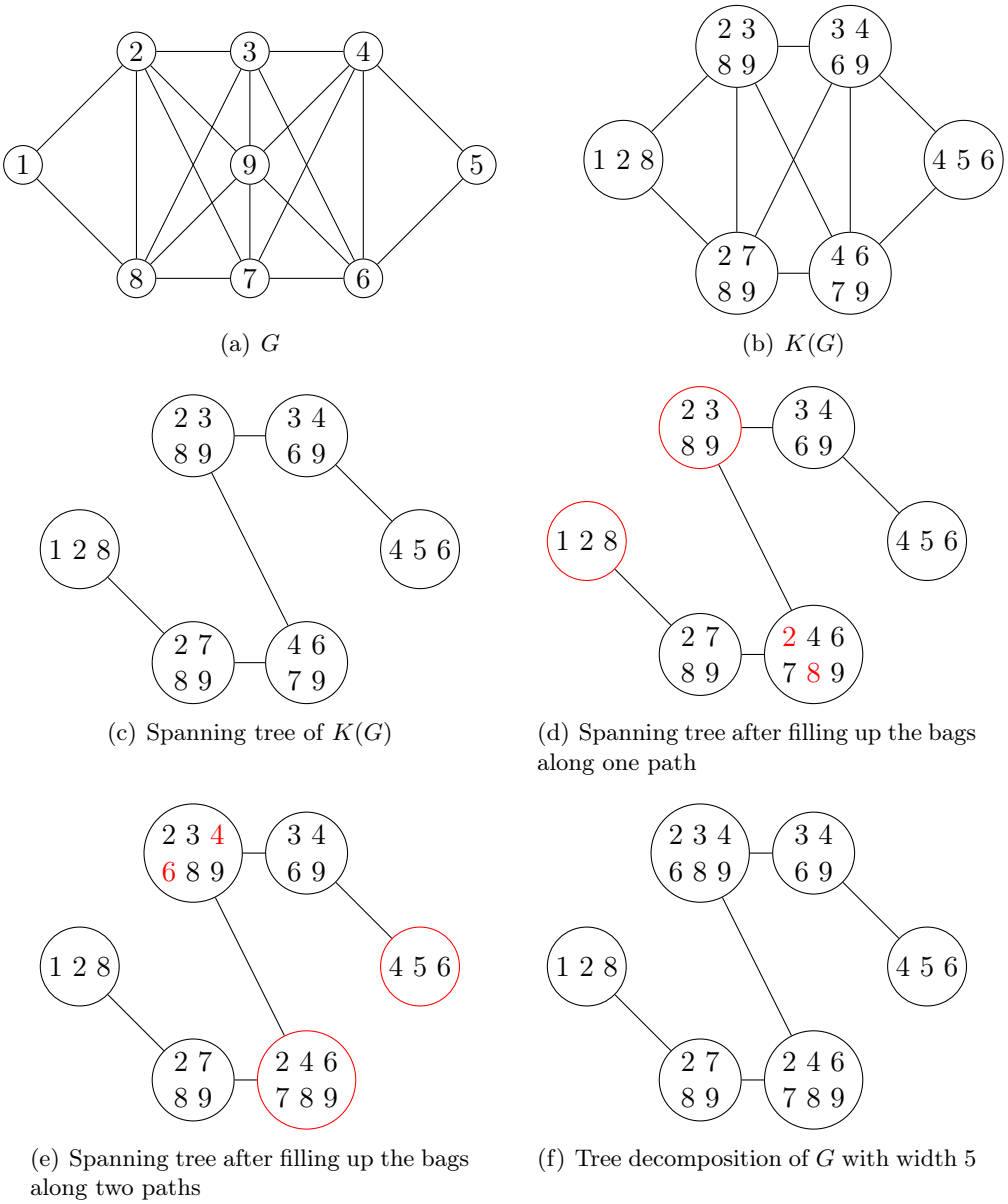
```
def clique_graph_treewidth_heuristic(graph):
    clique_graph = construct_clique_graph(graph)         # step 1
    clique_graph_tree = min_spanning_tree(clique_graph) # step 2

    fill_bags_along_paths(clique_graph_tree)             # step 3

    treewidth = compute_treewidth(clique_graph_tree)     # step 4
    return treewidth
```

Listing 4.1: Pseudocode of the heuristic

The heuristic can be broken down into four steps as follows. The first step of the heuristic is to construct the clique graph of the given graph. Note that the vertices of the clique graph have the corresponding cliques as labels, which we call bags. Then, in a second step, a (minimum) spanning tree is computed. Possible edge weights for the clique graph are discussed in section 4.2. Then, in a third step, to satisfy the third condition of the tree decomposition definition, see definition 2.1, the bags in the clique graph tree are filled. This step was described at the end of section 4.1 and visualized in figure 4.1. As a final fourth step, the treewidth of the constructed tree decomposition is computed, which is done by finding the largest bag in the decomposition, subtracting one from its size and returning that number.

4.1.2 Example

In the following, we want to look at an example graph and all the steps that are made in the heuristic. Looking at figure 4.2, we can see a graph G with 9 vertices in subfigure (a). The subfigure (b) shows the clique graph $K(G)$ of G. As explained at the beginning of this chapter, we want to modify the clique graph in such a way that it yields a tree decomposition. As a first step, the heuristic computes a spanning tree of the clique graph, as shown in subfigure (c). The next two subfigures (d) and (e) depict the necessary filling of the bags along the paths between bags that have a non-empty intersection. Since there are only two such intersecting pairs of bags, we obtain a tree decomposition as shown in subfigure (f).



(a) G

(b) $K(G)$

(c) Spanning tree of $K(G)$

(d) Spanning tree after filling up the bags along one path

(e) Spanning tree after filling up the bags along two paths

(f) Tree decomposition of G with width 5

Figure 4.2: Exemplary application of the heuristic. Changes in each step are highlighted in red

4.2 Edge weights for clique graph

Since a tree decomposition consists of a tree (I, F) and bags $\{X_i \mid i \in I\}$, see definition 2.1, we want to transform the constructed clique graph into a tree. This can be achieved by computing a spanning tree of the clique graph. More specifically, for the naive version of the heurstic, we will compute a minimum spanning tree using known algorithms such

as Prim's or Kruskal's. However, so far we have no weights for the edges of the clique graph. A simple and canonical approach would be to give all edges the same weight. This way, every spanning tree is a minimum spanning tree. In practice, this has led to having to fill many bags in step 3 of the heuristic, see listing 4.1, which in turn leads to very large bags in the resulting tree decomposition and a poor upper bound on the treewidth. Instead, alternative ways of adding weights to the edges of the clique graph can be used that dramatically improve the results, as can be seen in the computational evaluation in chapter 5. The most promising of these are presented below.

4.2.1 Negative intersection

Negative intersection weighting is based on the idea that bags with a large intersection should be close together in the final tree decomposition. This might in turn reduce the length of the paths along which the bags must be filled. Given two bags X_i and X_j in the clique graph, the weight is defined as $-|X_i \cap X_j|$ (the negative cardinality of the intersection). Thus, the edges with low weight connect bags with a large intersection and are more likely to be selected for the minimum spanning tree.

4.2.2 Symmetric difference

Similar to the negative intersection weighting, the symmetric difference weight function is based on the idea of reducing the lengths of the paths along which bags have to be filled in the clique graph tree. However, the approach here is to take the cardinality of the symmetric difference of two bags X_i and X_j in the clique graph, $|X_i \Delta X_j|$, and not invert the sign. This way, the edges with the lowest weights connect those bags that have few vertices that are not in the intersection of the bags.

4.2.3 Other weights

Other edge weights tested include union and disjoint union functions, which compute the cardinality of the union and the sum of the cardinalities of two bags, respectively. We also evaluated intersection and negative symmetric difference weights, which are the sign inverted versions of negative intersection and symmetric difference. Finally, for benchmarking and as a baseline, we also used constant weights to see if choosing an edge weight function other than the canonical one actually improved the heuristic.

4.2.4 Combined weights

While testing the different options of weight functions, it often happened that several edges had the same minimum weight, in which case a random edge was chosen from those with minimum weight to construct the minimum spanning tree. To avoid the randomness in the choice between multiple edges of least weight, one can combine different weighting schemes. Each edge is assigned a tuple of integers as a weight instead of a single integer. The first entry contains the weight according to one edge weight function and the second

entry the weight according to another edge weight function. In this case lexicographic ordering for the tuple edge weights is used.

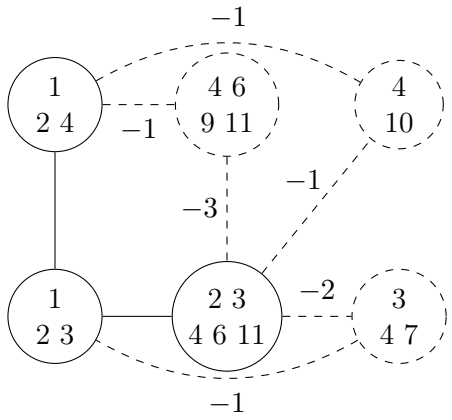
4.3 Alternative spanning tree construction

In the previous section, we discussed different ways of assigning weights to the edges of the clique graph. These weights can be used to compute a minimum spanning tree and then to fill the bags of the resulting tree as shown in figure 4.1.

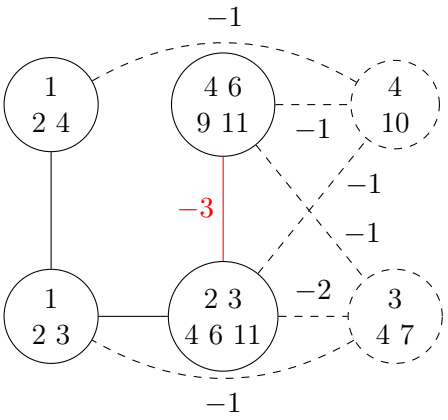
However, the method of simple minimum spanning tree computation leaves out the possibility of adjusting the clique graph tree as it is constructed according to the decisions that have already been made. More precisely, when constructing a spanning tree by successively selecting edges to add, as in Prim's or Kruskal's algorithm, the final spanning tree is being partially determined at each step of the procedure. If at any step two bags are connected in the current subgraph of the final tree, the path between these bags and thus the bags to be filled are already determined. This could be used to fill the bags of the clique graph tree while the spanning tree is still being constructed and subsequently adjust the edge weights according to this change.

There are several ways to incorporate this idea into the heuristic. One option is to use Prim's minimum spanning tree algorithm, which constructs a minimum spanning tree starting at a vertex and selecting edges to add in each step until a spanning tree is obtained. We want to modify this algorithm as follows. After adding a new edge and thus a bag to the current tree, fill all paths from the new bag to all other bags that are already in the current tree. Since this may change some of the bags in the current tree, adjust the edge weights accordingly and continue Prim's algorithm (with the new edge weights). We will refer to this method of constructing a clique graph tree as filling the bags while constructing the spanning tree or 'FillWhile' for short.

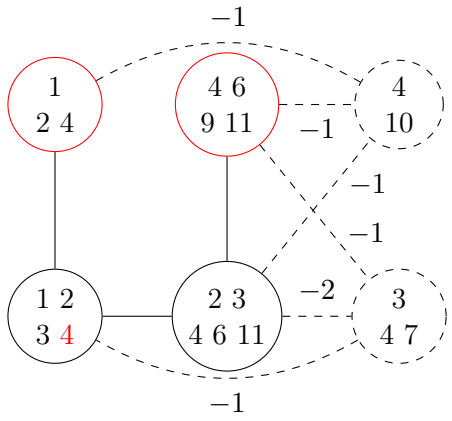
The following figure 4.3 shows a step of the alternative spanning tree construction method FillWhile in combination with negative intersection weights. Subfigure (a) shows a possible current state of the spanning tree construction. The dotted circles and lines are the potential vertices, bags in this case, and edges. Since the edge to the potential bag $\{4, 6, 9, 11\}$ is the cheapest, it is selected as the next edge to be added to the spanning tree according to prim's algorithm. This is depicted in subfigure (b). As a result, the bags that already belong to the spanning tree may need to be updated. The filling of the bags is shown in subfigure (c). The vertex with label 4 from the original graph is added to the bag containing $\{1, 2, 3\}$, since the bag $\{1, 2, 3\}$ is on the path of the bags $\{1, 2, 4\}$ and the newly added bag $\{4, 6, 9, 11\}$, which have a non-empty intersection of $\{4\}$. The last subfigure (d) depicts the change in edge weights resulting from updating the bags in the current tree. In the heuristic these four steps are repeated until a spanning tree is obtained.



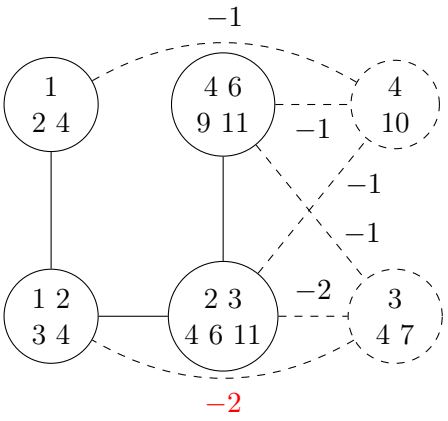
(a) Current state of spanning tree construction. Not yet selected nodes are dashed



(b) Add edge with minimum weight −3 to spanning tree and update other edges accordingly



(c) Fill bags along paths between newly added vertex and other vertices of current subtree



(d) Change edge weights according to changes done while filling bags

Figure 4.3: Filling up the bags while constructing the spanning tree using negative intersection weights. Changes in each step are highlighted in red

We also considered other alternative methods of spanning tree construction. One involves not only updating the edge weights according to the changes made while filling the bags, as shown in figure 4.4(d), but also adding new edges between bags that have a non-empty intersection as a result of the update. In the case of figure 4.3, an edge with weight 1 between the bags $\{1, 2, 3, 4\}$ and $\{4, 10\}$ would be added.

Another approach considered is to minimize the maximum bag size locally for each addition of a new bag to the spanning tree. This means that instead of using an edge

weight function as described in section 4.2, the edge weight would be (the negative of) the maximum bag size of the bags in the current spanning tree if the new bag is added and the bags are updated accordingly. This results in a very large increase in computation time, since all edge weights might need to be recalculated after each addition of a new bag to the current spanning tree.

To distinguish between these different spanning tree construction methods, we will refer to the initial method presented in section 4.1 as 'MinimumSpanningTree', the first method presented in this section as 'FillWhile', the updated variant that adds new edges as 'FillWhileUpdatingEdges' and the variant that minimizes the maximum bag size as 'FillWhileMinimizingBagSize'.

4.4 Time complexity

In the following section, we will discuss the time complexity of the heuristic just developed. We will focus on the first version presented, without the alternative spanning tree construction. As we saw in section 3.5, the number of maximal cliques $\kappa(G)$ of a graph G can be exponential in the number of vertices. Since the heuristic has to construct the clique graph, this leads to a guaranteed exponential running time for such graphs. Therefore, more specifically, we want to focus our time complexity analysis on finding cases where the heuristic could have polynomial running time. In doing so, we will construct an upper bound on the general running time of the algorithm using the number of maximal cliques $\kappa(G)$, $m = |E(G)|$ and $n = |V(G)|$. Note that the upper bound on the running time is not optimal, but rather gives an idea of the approximate running time of the heuristic.

We will use the steps in the pseudocode as shown in listing 4.1 to analyze the running time of the heuristic step by step.

The first step of the heuristic is to construct the clique graph. This is done by computing the maximal cliques of the given graph and constructing the intersection graph of the maximal cliques. In [CN85] an algorithm is presented that returns a list of all maximal cliques in $\mathcal{O}\left(\kappa(G)m^{3/2}\right)$ time. Now the clique graph can be constructed by simply adding all maximal cliques as vertices and then iterating over all $\mathcal{O}\left(|V(K(G))|^2\right) = \mathcal{O}\left(\kappa(G)^2\right)$ pairs of bags in the clique graph and adding an edge if there is an intersection, the latter computation being possible in $\mathcal{O}(n)$ time. Therefore, the first step of the heuristic is in $\mathcal{O}\left(\kappa(G)^2 n + \kappa(G)m^{3/2}\right) \subseteq \mathcal{O}\left(\kappa(G)^2 n + \kappa(G)n^3\right)$. Note that $m \in \mathcal{O}\left(n^2\right)$ since we are only considering simple graphs.

The second step of the heuristic is to compute a spanning tree for the clique graph. With Prim's algorithm, this can be done in $\mathcal{O}\left(\kappa(G)^2\right)$ (even better with binary-/Fibonacci-heaps), i.e. in polynomial time in $\kappa(G)$.

As a third step, the heuristic fills the bags along the paths between pairs of bags in the clique graph. Again, this can be done by iterating over all $\mathcal{O}\left(\kappa(G)^2\right)$ pairs of bags in the graph, where the intersection computation can be done in $O(n)$ time and a path between the pair of bags can be found in $\mathcal{O}\left(|V(K(G))| + |E(K(G))|\right) \subseteq \mathcal{O}\left(\kappa(G)^2\right)$ time using breadth-first search. Now, filling the intersection into all bags along the path takes

$\mathcal{O}(n)$ time per bag, resulting in $\mathcal{O}\left(\kappa(G)^5 n^2\right)$ time (since the path can be up to $\kappa(G)$ long).

This can be improved by using the tree structure in the clique graph tree to find a path between pairs of bags, resulting in $\mathcal{O}(\kappa(G))$ time complexity for finding a path. More specifically, the tree can be implemented as a rooted tree such that each vertex has a unique predecessor (except for the root). Then finding a path between two bags is the same as finding the least common ancestor of the two bags, where the ancestor is a vertex that can be reached by repeatedly looking up the predecessor. Furthermore, one could find the bags between which to fill the paths not by considering all pairs of bags, but by remembering which vertices from the original graph end up in which bags while constructing the clique graph. Then, for each vertex from the original graph, one could use the tree structure to simultaneously find the least common ancestor of all bags containing the current vertex from the original graph and insert the vertex into all bags found along the paths to the least common ancestor.

Seemingly this leads to a runtime of $\mathcal{O}(\kappa(G)n)$ for step 3, since for each vertex v of the original graph, the least common ancestor of the bags containing v has to be found and the number of bags containing v is certainly bounded by $\kappa(G)$. However, finding the least common ancestor of the bags using the canonical approach takes $\mathcal{O}\left(\kappa(G)^2\right)$ time. To see this, let S_v be the set of bags containing v after clique graph construction and let each bag have associated with it not only its predecessor but also its height, i.e., its distance from the root of the tree. One can now find the least common ancestor of the bags in S_v by repeatedly finding the bag with the highest height in S_v, removing that bag from S_v and adding its predecessor to S_v (if it is not already in S_v), stopping when S_v contains only one vertex (the least common ancestor). This takes a maximum of $\kappa(G)$ iterations, since no bag is visited twice. On the other hand, finding the bag with the maximum height in S_v takes $\mathcal{O}(\kappa(G))$. Therefore, this fills the bags in $\mathcal{O}\left(\kappa(G)^2 n\right)$ time (for each vertex v of the n vertices in G, filling up the bags containing v takes $\mathcal{O}\left(\kappa(G)^2\right)$ time). One can further improve this by using a binary max-heap to store the bags in S_v with their heights as keys. This would result in $\mathcal{O}(\log(\kappa(G)))$ time complexity for finding the bag with maximum height in S_v (and removing it from the heap), resulting in a total running time of $\mathcal{O}(\kappa(G)\log(\kappa(G))n)$. For our purposes, it is sufficient to assume that the third step is possible in $\mathcal{O}\left(\kappa(G)^2 n\right)$.

Since the fourth step is just finding the bag with the largest cardinality and subtracting one from it, this can obviously be done in polynomial time in $\kappa(G)$ and n. More precisely, certainly in $\mathcal{O}(\kappa(G)n)$.

In summary, this results in a running time of

$$\mathcal{O}\left(\kappa(G)^2 n + \kappa(G)n^3 + \kappa(G)^2 + \kappa(G)^2 n + \kappa(G)n\right) = \mathcal{O}\left(\kappa(G)^2 n + \kappa(G)n^3\right).$$

Thus, if the number of maximal cliques is polynomially bounded in n, the heuristic runs in polynomial time in n. In section 3.5 we discussed several sufficient criteria for this to be the case.

On the other hand, if one of the alternative spanning tree construction methods from section 4.3 is used, the running time increases drastically. This is due to the fact that

the bags are filled in each step of the spanning tree construction and as a result all edge weights may have to be recalculated. In particular, we cannot fill the bags along the paths between all bags containing a fixed vertex v from the original graph (the bags in S_v) by finding their least common ancestor as described above. This effect increases even further with the alternative spanning tree construction method FillWhileMinimizingBagSize. We measure the respective increases in running time in the computational evaluation, see 5.2.2.

4.5 Optimality for chordal graphs

As discussed in section 2.3, chordal graphs play an important role in the study of treewidth, have been used to develop several heuristics for upper bounds on treewidth and can be found in various applications. Thus, in the following, we want to prove that the heuristic developed in this thesis provides an optimal upper bound for chordal graphs, i.e., it returns the exact treewidth when given a chordal graph. Note that when we refer to 'the heuristic' in the following, we actually mean the heuristic variant with negative intersection weights and MinimumSpanningTree as spanning tree construction method. Furthermore, we assume that the minimum spanning tree in the heuristic is computed using Prim's algorithm.

We will prove the optimality using the characterization of chordal graphs as graphs with a perfect elimination ordering, see 2.17. More precisely, we will use a specific elimination ordering obtained by an algorithm called Maximum Cardinality Search (MCS). It was first proposed in [TY84] and provides both a test for the chordality of a graph and yields a perfect elimination ordering of the given graph if the test is passed.

The perfect elimination ordering obtained by the algorithm is constructed from right to left with the following procedure. An arbitrary vertex is chosen as v_n (the last vertex in the elimination ordering). Then, at each step, a vertex with the highest number of neighbors among the already chosen vertices is chosen. Usually, in case of a tie, a random vertex is chosen. In our version, we want to break the tie by choosing a vertex that is adjacent to the vertex w chosen in the previous step and that is adjacent to all neighbors of w that have already been chosen. If there is no such vertex, choose one at random (or if there are multiple such vertices, choose one at random among them). Of course, this does not affect the correctness of the algorithm. The selection is repeated until all vertices have been selected. For a full description, pseudocode, and proof of the correctness, see [TY84, p. 569].

To prove the optimality of the heuristic with negative intersection weights on chordal graphs, we first prove two lemmas. As sketched in the proof of theorem 2.20, we will show that given a perfect elimination ordering of a chordal graph, a tree decomposition can be constructed. Then we will see that the obtained decomposition is optimal in the sense that it realizes the treewidth of the chordal graph. Finally, we will show that a tree decomposition of the same form can be obtained using the heuristic with negative intersection weights, which proves optimality.

We begin by constructing a tree decomposition given a perfect elimination π of G. For the rest of this section we denote by v_i the i-th vertex in the ordering π and define $G_i := G[\{v_i, \ldots, v_n\}]$ for $i \in \{1, \ldots, n\}$. In the following lemma we claim not only that a tree composition can be constructed, but also that it realizes the treewidth of G. The proof can therefore be divided into two parts. First, the correctness and well-definedness of the construction of the tree decomposition and second, the proof that the constructed tree decomposition realizes the treewidth of the given chordal graph.

It should be noted that the following proof does not rely on any prerequisites other than corollary 2.7 and the correctness of the MCS algorithm. It was therefore valid to use the following lemma as a proof of the implication $(i) \implies (iii)$ in theorem 3.18. Of course, said implication follows directly from the fact that for every chordal graph there exists a perfect elimination ordering and thus one can construct a tree (decomposition) that satisfies the desired properties using the following lemma. With theorem 3.18 in mind, the following construction can therefore be interpreted as, given a chordal graph and thus a perfect elimination ordering, constructing a minimal representation as the intersection graph of subtrees of a tree, see definition 3.19.

Lemma 4.1. *Given a perfect elimination ordering π of a graph G, one can construct a tree decomposition such that the bags in the tree decomposition are exactly the maximal cliques of G and its width is optimal in the sense that it realizes the treewidth of G.*

Proof. The procedure for constructing a tree decomposition of G can be described as follows. Start with the tree decomposition of G_n, which consists of one bag containing only the vertex v_n. Then iterate over the rest of the elements in the elimination ordering from right to left (starting at v_{n-1}, ending at v_1). For v_i, find a bag X_j containing all neighbors of v_i that appear after v_i in π, i.e. $N_{G_i}(v_i) \subseteq X_j$. If such a bag exists that contains no other vertices, add v_i to that bag. Otherwise, create a new bag containing $\{v_i\} \cup N_{G_i}(v_i)$ and attach it to X_j.

(i) As described, we will first proof the correctness of the above procedure. In order to show the correctness of the described procedure, we must first prove that it is well-defined. To prove this, we need to show that in every step i (except the first one) there exists a bag such that $N_{G_i}(v_i) \subseteq X_j$. We want to do this by induction over the steps of the procedure. The first step is clearly well-defined. Suppose that the previous steps were possible and let v_i be the current vertex from the elimination ordering. Let $v_j \in N_G(v_i)$ be the vertex such that $j > i$ and j is minimal (the vertex among the neighbors of v_i with a higher number in π with the lowest number). By induction, v_j was added to a bag X_k containing all its neighbors. Since π is a perfect elimination ordering, the neighbors of v_i that appear after v_i in π form a clique in G and are therefore all contained in X_k.

Finally, we have to consider the case where v_i has no neighbors with a higher number in π. It is easy to see that this cannot happen (except for $v_i = v_n$), since G is connected by requirement and therefore when constructing the perfect elimination ordering using MCS, a vertex will always be found that has at least one neighbor among the vertices already chosen. Thus, a vertex with no neighbors among the already chosen vertices would

not be selected as the next vertex by MCS. In conclusion, the procedure is well-defined.

We now claim that the procedure yields a tree decomposition. Again, we will prove this by induction, showing that after the $(n+1-i)$-th step we obtain a tree decomposition of G_i. The case $v_i = v_n$ is obvious. Therefore, let v_i be the current vertex and assume that the previous step yielded a tree decomposition of G_{i+1}. To prove that we obtain a tree decomposition of G_i, we distinguish the two cases of how v_i is added to the decomposition of G_{i+1}:

- If v_i can be added to the bag X_j containing its neighbors, the tree decomposition criteria (i),(ii),(iii), see definition 2.1, are easily checked. No bags have been added to the decomposition, therefore it still is a tree. Furthermore, (i) is still satisfied, since the only new vertex from the original graph is v_i, and v_i has just been added to X_j. All neighbors of v_i in G_i are also contained in X_j, which satisfies (ii) by induction. Moreover, v_i is contained in only one bag, so (iii) is also satisfied by induction.

- If a new bag X_l with v_i and $N_G(v_i) \cap X_j$ is created and attached to X_j, the resulting graph is also obviously a tree by induction. Furthermore, analogous to the first case, (i) and (ii) are satisfied. For (iii) note that X_j contains all vertices in X_l except v_i which is only contained in X_j. Therefore, by induction (iii) is also satisfied.

This concludes the first part of the proof.

(ii) We will now prove that the bags in the constructed tree decomposition are exactly the maximal cliques of G and its width is optimal in the sense that it realizes the treewidth of G. To prove that the bags of the constructed tree decomposition correspond to the maximal cliques in G, let C be a maximal clique in G. We will show that there exists a bag X_k in the decomposition such that $C = X_k$. To do this, let v_i be the vertex in C with the lowest index, i.e. the vertex chosen last in our procedure. Note that since π is a perfect elimination ordering, the vertices with a higher number in π than v_i form a clique in G. Therefore, the only such neighbors must be exactly the other vertices in C. By construction, after the $(n+1-i)$-th step (the step in which v_i is added), the decomposition of G_i will contain a bag $X_k = C$. Since C is a maximal clique, no vertices are added to X_k in the following steps. Thus, X_k is the desired bag in the constructed tree decomposition containing exactly the vertices of C.

On the other hand, let X_i be a bag in the constructed tree decomposition. For our claim, it suffices to show that X_i is a maximal clique. For this, let $(n+1-j)$ be the step in which the bag X_i was created, i.e. $v_j \in X_i$, and denote by X_i' the state of X_i after creation. If X_i' is a maximal clique after creation, no vertices are added to X_i' in the following steps, which proves the desired. Otherwise, there exists a vertex $v_k \notin X_i'$, $k < j$, such that $X_i' \cup \{v_k\}$ is a clique. Note that since v_j was just selected in the MCS, the maximum number of neighbors among the already selected vertices was $|X_i'| - 1$ before v_j was selected. Therefore, an upper bound on the number of neighbors among the already selected vertices for the remaining non-chosen vertices is $|X_i'|$. We know that v_k realizes

this bound and is adjacent to all vertices in X'_i, which would break the tie with other vertices in our version of MCS. For every other vertex v_l that also realizes this bound and is adjacent to all vertices in X'_i, it holds that $X'_i \cup \{v_l\}$ is a clique. Therefore, the next element selected in the MCS must be a vertex v_{j-1} such that $X'_i \cup \{v_{j-1}\}$ is a clique. Thus, the next vertices chosen are exactly those that enlarge the bag X_i until this is no longer possible, in which case X_i is a maximal clique. So X_i must be a maximal clique.

Now, for the exact correspondence between the maximal cliques and bags in the tree decomposition, it is sufficient to observe that by construction of the tree decomposition, no two bags can be the same.

As desired, this proves that the width of our constructed tree decomposition of G is $\omega(G) - 1$. Since we know that $\mathrm{tw}(G) \geq \omega(G) - 1$, see corollary 2.7, this implies that $\mathrm{tw}(G) = \omega(G) - 1$, that is, our constructed decomposition is optimal in the sense that it realizes the treewidth of G. □

Since we have just shown that the bags in the constructed tree decomposition correspond to the maximal cliques in G, we can now interpret the constructed tree decomposition as a spanning tree on the clique graph of G. Furthermore, we can deduce that there always exists a spanning tree on the clique graph of G that induces an optimal tree decomposition.

In the following, we want to show that such a spanning tree is obtained by our heuristic with negative intersection weights.

For this, note that the perfect elimination ordering obtained by our version of MCS is not unique. If a random choice between multiple vertices is made in a step of MCS, the resulting perfect elimination ordering may be different depending on that choice. Therefore, there may exist multiple perfect elimination orderings that can be obtained from our version of MCS. We have just shown that they all induce a tree decomposition and that this tree decomposition consists of a spanning tree on the clique graph and has optimal width. Finally, we want to prove that when Prim's minimum spanning tree algorithm is applied to the clique graph $K(G)$ of G with negative intersection weights, the resulting spanning tree corresponds to one of the perfect elimination orderings of G obtained by our version of MCS.

Lemma 4.2. *Given a chordal graph G, the heuristic with negative intersection weights constructs a tree decomposition for which there exists a perfect elimination ordering π which can be obtained by our version of MCS, such that the tree decomposition constructed from π with lemma 4.1 is the decomposition of the heuristic.*

Proof. We prove by induction over the steps of Prim's algorithm (in each step one edge/vertex is selected) that given the current subtree with vertices $C_1, \ldots, C_k$, we can construct a corresponding perfect elimination ordering of $G[\bigcup_{i=1}^{k} C_i]$ using MCS. For the first vertex, i.e. the 0-th edge, which is usually chosen arbitrarily by Prim's algorithm, let C_1 be the corresponding clique in G. Correspondingly, in our MCS we can choose an arbitrary vertex in $x \in C_1$ as the vertex $v_n := x$ in the first step (since this choice is

not fixed by MCS) and choose the other vertices in C_1 in the next steps (since the other vertices in C_1 are adjacent to all previously chosen vertices).

Now suppose we were able to construct a perfect elimination ordering π_k using our version of MCS for the previous edge choices in Prim's algorithm. Suppose the next edge added by Prim's algorithm is $e = \{C_j, C_{k+1}\}$, where $1 \leq j \leq k$ and $C_1, \ldots, C_k$ are the vertices already added in the previous steps. This means that of the currently possible edges to choose from, e is the cheapest, so $|C_j \cap C_{k+1}|$ is maximal. To continue our MCS accordingly, we want to find a vertex v_i that has not yet been selected in the MCS, i.e., as we will see, $v_i \in C_{k+1} \setminus \bigcup_{i=1}^{k} C_i$. Let us therefore assume that $C_{k+1} \subseteq \bigcup_{i=1}^{k} C_i$, i.e. all vertices in C_{k+1} already occur in π_k. Let $v_t \in C_{k+1}$ be the vertex with minimal number in π_k among the vertices of C_{k+1}. Note that when using π_k to construct a tree decomposition of $G[\bigcup_{i=1}^{k} C_i]$ as described in lemma 4.1, a bag X with $X \subseteq C_{k+1}$ is created in the $(n+1-t)$-th step when v_t is added. Therefore, in the resulting tree decomposition induced by π_k, C_{k+1} is a bag, which contradicts the induction hypothesis that π_k induces a tree decomposition corresponding to the current subtree with vertices $C_1, \ldots, C_k$.

Therefore, a vertex $v \in C_{k+1} \setminus \bigcup_{i=1}^{k} C_i$ must exist. Note that this vertex maximizes the number of neighbors in π_k among the vertices not in π_k. This is due to the fact that $|C_j \cap C_{k+1}|$ is maximal by Prim's algorithm. Furthermore, if there is another vertex w with the same number of already chosen neighbors, it cannot break the tie with v, since otherwise one of the cliques $C_1, \ldots, C_k$ could be expanded, which is a contradiction to the C_i being maximal cliques. Therefore v can be chosen as the next vertex in the MCS. We choose the rest of the not yet chosen vertices in C_{k+1} in the next steps similar to the case for C_1. By construction it is easy to see that the resulting perfect elimination ordering π_{k+1} induces the current subtree with vertices $C_1, \ldots, C_{k+1}$. □

This shows that the minimum spanning tree constructed using Prim's algorithm with negative intersection weights induces a tree decomposition with a width of $\omega(G) - 1$. Therefore, no bags need to be filled in the heuristic and the heuristic provides an optimal upper bound as stated in the following theorem.

Theorem 4.3. *Let G be a chordal graph, then the heuristic with negative intersection weights computes an optimal upper bound for G.*

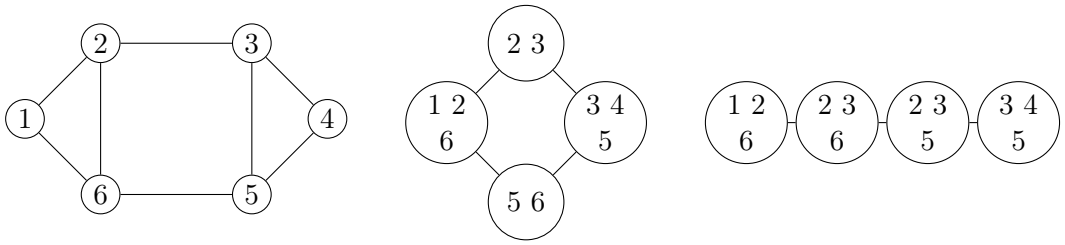
Note that we have just proven optimality for both the MinimumSpanningTree construction method and the alternative FillWhile method. This is due to the fact that no bags need to be filled in the process of spanning tree construction with Prim's algorithm and therefore the FillWhile and MinimumSpanningTree methods behave effectively identically on chordal graphs. Furthermore, the above result implies the optimality of our heuristic on several known subclasses of chordal graphs such as k-trees and thus trees.

Nevertheless, it does not seem reasonable to use our heuristic to recognize chordal graphs or compute their tree decompositions. Maximum cardinality search solves both of these problems in linear time, whereas our heuristic is, at best, polynomial in n, as discussed in section 4.4. The running time for chordal graphs is not exponential in n,

since chordal graphs have only linearly many maximal cliques and are therefore a class with few maximal cliques, see 3.31.

4.6 Non optimality for certain graphs

In the previous section we saw that the heuristic with negative intersection weights computes an optimal upper bound for chordal graphs. Now we want to show that there are graphs for which the heuristic cannot compute an optimal upper bound, no matter the choice of edge weights for the clique graph. That is, consider a graph G for which all spanning trees on the clique graph $K(G)$ result in a non-optimal tree decomposition after filling the bags.



(a) Graph G (b) $K(G)$ (c) Tree decomposition of G

Figure 4.4: Graph G for which the heuristic cannot be optimal

Consider the graphs in figure 4.4. It is easy to see that there are exactly four spanning trees for the graph $K(G)$ shown in figure 4.4(b). Each of which is identified by the edge that is omitted from of $K(G)$. Note that no matter which edge $\{X, Y\} \in E(K(G))$ is omitted, the bags X, Y have a non-empty intersection that must be filled for all other bags. As a result, one of the bags containing three vertices in $K(G)$ is expanded to contain four vertices.

On the other hand, one can easily construct a tree decomposition of G with a width of 2, as shown in figure 4.4(c). Moreover, because G is not a forest, by theorem 2.8 this implies that G has treewidth 2. Therefore, we can conclude that our heuristic cannot compute an optimal upper bound for the graph in figure 4.4(a).

4.7 Bound on the clique size

Before evaluating the heuristic variants developed so far, we want to introduce one additional parameter of the heuristic. To motivate this, let us take a step back and review the idea of the heuristic thus far. We compute the clique graph of a given graph, compute a spanning tree and fill the bags where necessary in order to obtain a tree decomposition. We know by lemma 2.6 that given a tree decomposition, for each clique there must exist a bag that is a superset of the clique. Therefore, it seems like a good idea to start with the maximal cliques and augment them to obtain a tree decomposition. However, we

wondered whether taking the maximal cliques would be too coarse in the sense that it might lose valuable information about the structure of the graph.

With this in mind, we came up with the idea of using cliques instead of maximal cliques. More precisely, we impose a bound k on the size of cliques and divide all cliques larger than k into all subsets of size k. In set notation, we define $\mathcal{F}'$ as follows. Given the set of maximal cliques $\mathcal{F}$ of a graph G and k our bound, we define

$$\mathcal{F}' = \{C \mid (C \in \mathcal{F} \wedge |C| \leq k) \vee \exists C' \in \mathcal{F} : (|C'| > k \wedge C \subseteq C' \wedge |C| = k)\}.$$

Then, in the first step of the heuristic, see listing 4.1, we just use $\mathcal{F}'$ instead of $\mathcal{F}$ to construct the intersection graph $\Omega(\mathcal{F}')$, see definition 3.1, that is, construct and use $\Omega(\mathcal{F}')$ instead of the clique graph for the rest of the heuristic.

Note that this allows us, for example, to use the line graph instead of the clique graph as the starting point of our heuristic by setting $k = 2$.

5 Computational evaluation

In the following chapter, we want to discuss a possible implementation of the heuristic developed in section 4 and computationally evaluate the different variants of the heuristic that have been presented.

5.1 Implementation

All evaluated algorithms were implemented using the Rust programming language [Rust]. Rust was chosen for its high performance, useful compiler messages, multithreading capabilities and existing libraries for modeling graphs. This combination of advantages sets Rust apart from alternatives such as Python and C/C++. The existing Petgraph package, see [Pet], was used to model undirected graphs. It also provided basic graph functionality such as minimum spanning tree and shortest path computation.

Since the heuristic uses many sub-procedures for which an implementation is clear or does not allow much freedom for optimization, we will only discuss certain parts of the implementation. The full implementation can be found on GitHub[1].

5.1.1 Constructing the clique graph

To construct a clique graph according to the clique graph operator 3.2, one must first compute the maximal cliques. Several algorithms have been developed that achieve this, some of which are discussed in [MU04]. It is important to note that, as seen in example 3.22, the number of maximal cliques of a graph can be exponential in the number of vertices n. Therefore, the computation of all maximal cliques is in general not possible in polynomial time in n. Thus, the algorithm used in our implementation, which has a worst-case running time of $O(3^{n/3})$, is optimal in this regard. It was first proposed by [BK73], improved upon by [TTT06] and later discussed by [CK08]. The specific implementation in our case is an adapted version of the algorithms suggested in the citations, in which recursions have been unrolled to yield an iterative algorithm and was implemented for NetworkX, see [HSC08], a popular network algorithm library in Python.

5.1.2 Filling bags using the structure of a tree

In both the alternative version of spanning tree construction, see 4.3, and the MinimumSpanningTree version, it is necessary to fill bags along paths between bags in the clique graph (tree), see 4.1 for the latter version. This can be done naively by iterating over all pairs of bags X_i, X_j in the clique graph tree, computing their unique path in the

[1] https://github.com/RaoulLuque/treewidth-heuristic-using-clique-graphs

tree and filling all bags along the path with the vertices in their intersection $X_i \cap X_j$. Instead, we want to reduce the running time of this procedure in two ways, both of which were already described in section 4.4. First, as we construct the clique graph, for each vertex v from the original graph, we store a list of bags in which v was inserted during clique graph construction (or more generally, intersection graph construction). Having done that, we can iterate over all vertices v of the original graph and know exactly which paths need to be filled with the vertex v. Second, instead of computing the paths between pairs of bags in the clique graph tree, we can use predecessors and levels in the tree. We do this by choosing an arbitrary bag as our root and successively assigning a predecessor to all other bags (starting with the neighbors of the root, continuing with their not yet seen neighbors, and so on). Thus, finding a path between two bags is the same as finding their least common ancestor. This allows us to look at the list of bags in which a vertex has been inserted, find their least common ancestor and fill all of the predecessors' bags along the way, which allows for faster path finding and filling of paths between multiple bags at the same time.

5.1.3 Nondeterminism of implementation

Because hash maps are used in several parts of the implementation and because the default hash function in Rust is seeded with random numbers generated by the operating system at runtime, the order of elements in a hash map varies in each run of the algorithm. This in turn results in differences in the computed upper bound.

Upon closer inspection, we noticed that in many graphs and more specifically in their clique graphs, when computing a spanning tree, different edges have the same weight and as a consequence the first of these edges with the same weight is chosen when constructing a spanning tree. Since the order of these edges depends on a hash map and thus on the hash function used, the order varies from one run of the algorithm to another. As a result, multiple runs of the algorithm on the same graph will use different spanning trees, resulting in different upper bounds. This is addressed by running each variant of the algorithm multiple times on the same graph and then taking the lowest computed upper bound as the result. In addition, a custom hash function can be passed to the heuristic, which is then used instead of the default to achieve deterministic results. The variance of the results over multiple runs on the same graphs is included in the evaluation, see subsection 5.2.5.

5.2 Evaluation

We want to use the following section to evaluate the heuristic by discussing how each parameter that makes up the different variants of the heuristic affects the quality of the bound and possibly the running time. These parameters have been discussed in chapter 4 and include the different edge weight functions as seen in section 4.2, the alternative spanning tree construction methods presented in section 4.3 and the optional clique size bound as discussed in section 4.7. This will eventually lead us to the version of the heuristic that performed best in our tests.

We will use the following naming scheme for the different variants of the heuristic:

$$\text{Spanning Tree Construction Method} + \text{Edge Weights} + \text{Optional clique size bound}$$

We abbreviate each part of the complete name, respectively. An example would be the following: 'MSTre+NegIn+BC2'. In this case, MinimumSpanningTree (MSTre) was used as the spanning tree construction method. The edge weights used for the spanning tree are negative intersection edge weights, which is indicated by the second part, 'NegIn'. In the case of combined edge weights, we abbreviate the two edge weights and connect them with a 'T' for 'then', e.g. negative intersection then symmetric difference: 'NiTSd'. The third part is optional, depending on whether there is a bound on the size of the cliques, as defined in section 4.7. If so, 'BCn' indicates that the cliques are bounded up to a size of n. Thus, 'BC2' indicates a clique bound of 2, i.e. the line graph was constructed instead of the clique graph. We will also consider $n = \omega(G) - 1$ and $\omega(G) - 2$, i.e. the clique number minus one and two, which we will denote by BC-1 and BC-2, respectively.

For the sake of clarity, we abbreviate the names of the different edge weight functions and spanning tree construction methods, with the following table showing all such abbreviations used in this chapter.

Edge weight function	Negative intersection	Symmetric difference	Union	Disjoint Union	Constant
Abbreviation	NegIn/Ni	SyDif/Sd	Union	DisjU	Const

Spanning tree construction method	Minimum Spanning Tree	FillWhile	FillWhile Updating Edges	FillWhile Minimizing BagSize
Abbreviation	MSTre	FilWh	FWhUE	FWBag

Table 5.1: Tables showing the abbreviations used for edge weight functions and spanning tree construction methods

To evaluate the different variants of the heuristic, we decided to use the procedure proposed by [SG97] and adapted in [BK10] of using partial k-trees to benchmark the heuristics on. Partial k-trees are k-trees, as defined in definition 2.11, from which a certain number of edges have been removed at random. In our case we will remove 30, 40 or 50 percent of the edges. Since k-trees are chordal graphs and by definition have a clique number of $k + 1$ (if $n \geq k + 1$), we can conclude with theorem 2.20 that k-trees have a treewidth of k. Therefore, the treewidth of partial k-trees is upper bounded by k by lemma 2.3. We now guarantee that our constructed partial k-trees have a treewidth of exactly k by running a heuristic that determines a lower bound on the treewidth. If the heuristic returns k as an upper bound, we know that the treewidth must be exactly k, otherwise we construct a new partial k-tree. However, the latter case, where a lower bound other than k is returned, has never occurred in our tests. Knowing the exact treewidth of the test graphs has the advantage that the upper bounds given by the

heuristics can be compared with the actual treewidth and can thus be contextualized better than when only an upper bound from another heuristic is available.

Partial k-trees are characterized by three parameters, n and k as in the definition of k-trees, see definition 2.11, and p which is the percentage of edges removed from the k-tree to obtain the partial k-tree. In our case we will look at partial k-trees with $n \in \{50, 100, 200, 500\}$, $k \in \{5, 10\}$ and $p \in \{30, 40, 50\}$. We wanted to use higher n and k such as 1000 and 20 (or 15), respectively, as done in [BK10], but in our tests it became clear that some variants of the heuristic would take up to days on a single instance to run on such large graphs, which was neither feasible nor the goal of our computational evaluation.

For the same reason, a sample size of 20 per class of partial k-trees was used as a compromise between high overall running time of the benchmarks and a representative sample size. The choice of a relatively small sample size also came down to the fact that we ran each variant of the heuristic several times on the same graph. This is due to the fact that, as described in subsection 5.1.3, the heuristic is nondeterministic in the sense that multiple runs on the same graph can produce different upper bounds. To reduce the influence of this effect on the results, we ran each variant of the heuristic 5 times on each graph instance and used the best upper bound, i.e. the lowest result, as the final result. This in turn multiplied the number of necessary runs per heuristic variant per graph by 5 to a total of 100 runs.

After having found the best variant of the heuristic according to our tests, we will compare it with the 'GreedyDegree+FillIn' heuristic, a known heuristic for computing an upper bound on the treewidth, presented in [BK10, p. 264]. We have already discussed the idea behind the GreedyDegree+FillIn heuristic at the end of chapter 2. For the comparison we will use partial k-trees as well as Dimacs graph coloring instances from the second Dimacs implementation challenge, see [JT96]. The results of the comparisons with GreedyDegree+FillIn are presented in a plot and a table in subsection 5.2.4.

As will become clear from the results of the benchmarks, even the best variants of the heuristic developed in this thesis cannot compete with other known heuristics for upper bounds on the treewidth. Neither in the computed bound nor in the required computation time. Nevertheless, we decided to first discuss how the different parameters of the heuristic affect its quality. With this knowledge, we will then try to analyze and understand the relatively poor performance of our heuristic compared to other known heuristics.

All of the following benchmarks were run on a system with an Intel Core i7-4770 3.40 GHz processor and 20 GB of RAM installed. Due to the long computation times of our heuristic, we decided to use multithreading to run the non-timed benchmarks. The timed benchmarks were run in single threads to ensure their accuracy.

5.2.1 Comparison of edge weights

The first parameter we discussed was the edge weights chosen for the edges in the clique graph. Therefore, we want to start the evaluation by discussing their effect on the heuristic. Figure 5.1 shows the comparison of the different edge weight functions we presented in subsection 4.2, using 'MSTre' for spanning tree construction. The alternative spanning tree construction methods are evaluated in the next subsection 5.2.2.

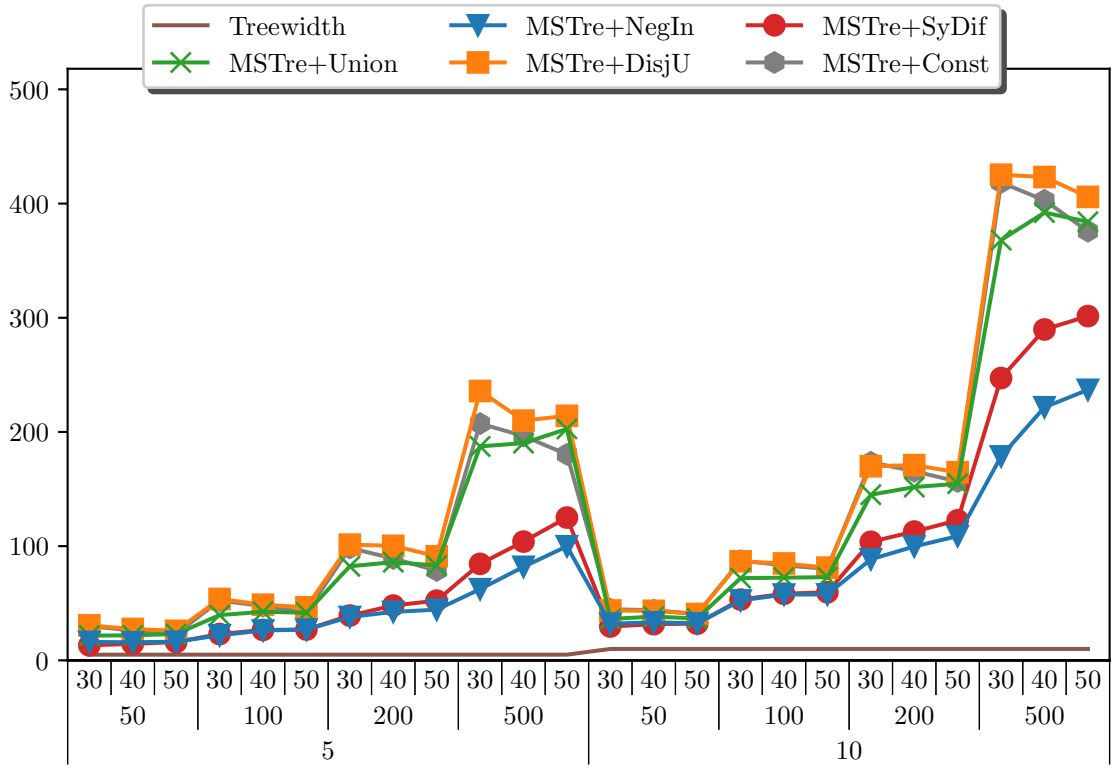


Figure 5.1: Computed upper bound for varying edge weights

Before discussing the results, we first want to explain the structure of the above figure, as it will remain constant for the rest of this evaluation. The x-axis shows the different classes of k-trees used for the benchmark and is divided into three levels. The top row shows the $p \in \{30, 40, 50\}$ of the respective class of k-trees. The second and third rows indicate the n and k, respectively. The y-axis denotes the average upper bound computed over the trees of one class of k-trees, i.e. the average over the 20 trees of a class of k-trees, unless otherwise specified. As discussed above, each heuristic was run 5 times on the same tree and then the lowest bound was used as the result. This result was then used as one of the 20 results for the average. Finally, for heuristic variants that appear in multiple plots, we used consistent colors and tick marks throughout the plots.

The figure 5.1 shows that the negative intersection and symmetric difference weights slightly outperform the other edge weight functions. This result is not surprising, since NegIn and SyDif incentivize the heuristic to minimize the distance between similar bags,

whereas the other edge weight functions use criteria that seem irrelevant to the final tree decomposition. The gap in the computed upper bound seems to increase as the number of vertices in the partial k-trees increases, while the different edge weights perform very similarly for small n. The edge weight functions union and disjoint union seem to perform similarly poorly as choosing no edge weights, i.e. constant edge weights. Looking at fixed n and k, we can see that for the negative intersection and symmetric difference weights, the computed upper bound seems to increase as the percentage of edges removed increases, whereas this effect is inverted for the other edge weights. While negative intersection and symmetric difference weights both outperform the other weights, it can be seen that negative intersection weights are overall the best in this comparison, in particular for larger n.

Next, we want to look at the effect of combined edge weights compared to single edge weights. As a reminder, we came up with the idea of using lexicographically sorted tuples as edge weights by using one edge weight function in the first entry and another in the second entry. This is done to encode more information about the structure of the clique graph in the edge weights, in the hope of obtaining a spanning tree that leads to a lower upper bound.

Since the negative intersection and symmetric difference weights clearly outperformed the other edge weight functions, we will focus on comparing their respective combined edge weights and the negative intersection weight function. Other combinations of edge weights for combined edge weights using functions such as union and disjoint union were also tested, but did not outperform their single edge weight counterparts.

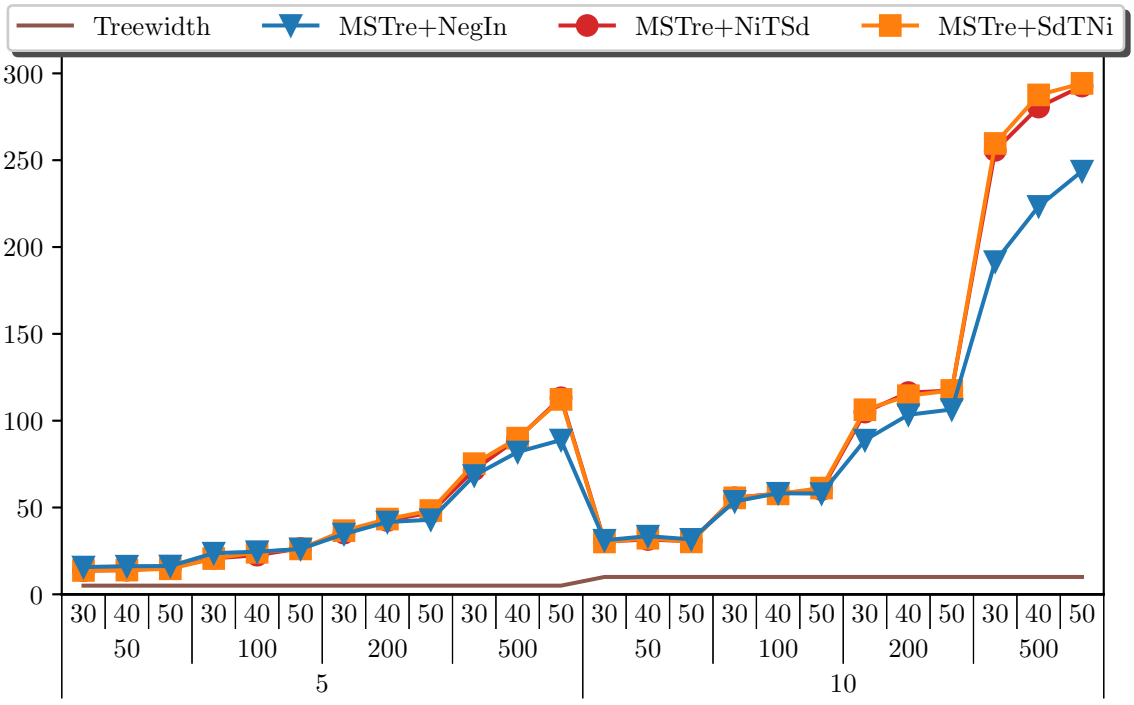


Figure 5.2: Comparison of different combined edge weights in terms of their computed upper bounds

Figure 5.2 shows the comparison of combined and single edge weight functions. Interestingly, it can be seen that both combined edge weights perform slightly worse than just using single negative intersection weights. This result seems to contradict the assumption that combined edge weights outperform their single weight counterparts, given that they provide more information. However, as will be discussed in the next section where we compare spanning tree construction methods, the combined edge weights seem to be more useful when the information is retrieved multiple times, i.e., when edge weights are updated during spanning tree construction, as is done in the alternative spanning tree construction method FillWhile.

5.2.2 Comparison of spanning tree construction methods

So far, we have compared the different edge weights using the default spanning tree construction method, a minimum spanning tree on the clique graph. In the following, we want to compare the different spanning tree construction methods presented in the previous chapters. This will be done both in terms of upper bounds, i.e., quality, and in terms of the running time required by the different spanning tree construction methods.

First, we will compare the default spanning tree construction method and FillWhile. As we will see, the latter turns out to yield the best variant of the heuristic we could find. The comparison is made using both single negative intersection weights and the combined edge weights, NiTSd. The SdTNi variants performed slightly worse than their respective NiTSd counterparts and are therefore omitted for the sake of clarity.

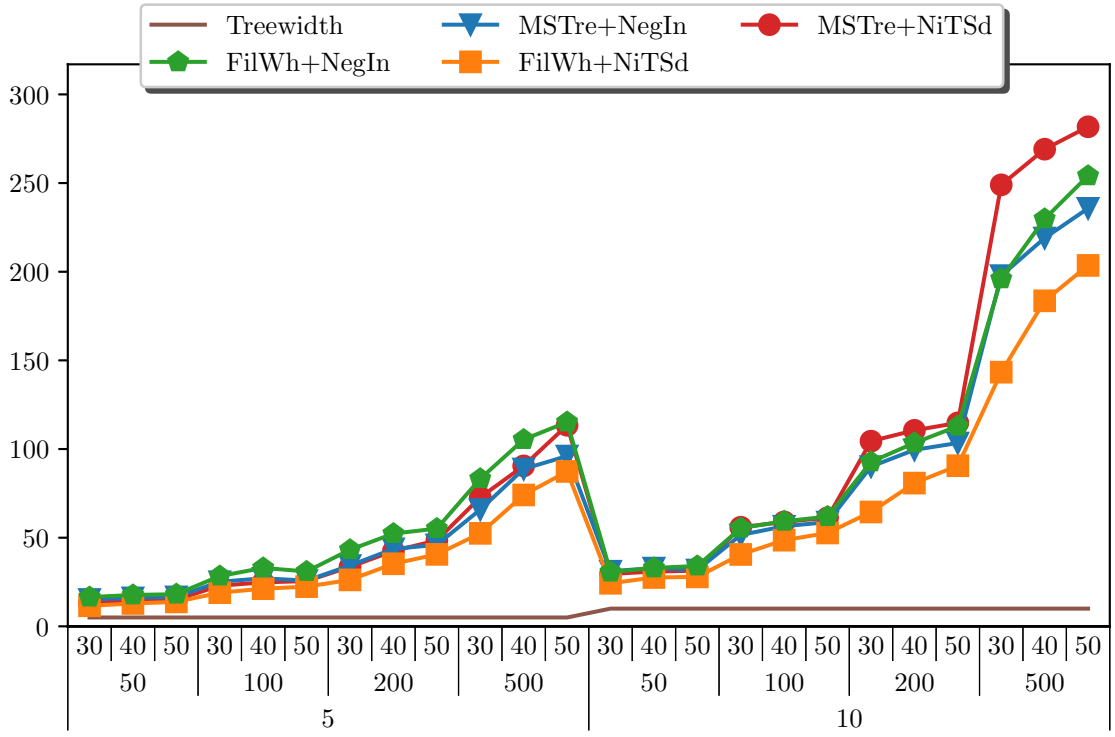


Figure 5.3: Computed upper bounds using MinimumSpanningTree or FillWhile as spanning tree construction methods

The results can be seen in figure 5.3 and show that using the alternative spanning tree construction method leads to better bounds with differences increasing with the number of vertices in the partial k-trees. Furthermore, the combined edge weight NiTSd outperforms the single NegIn weight when FilWh is used as the spanning tree construction method, which is interestingly the opposite of when MSTre is used. In particular, the FilWh single edge weight variant seems to perform worse for $k = 5$ and increases in relative performance with $k = 10$, even slightly outperforming MSTre+NegIn for some partial k-tree classes.

We can conclude that FilWh+NiTSd seems to be the best heuristic variant so far, followed by MSTre+NiTSd and in some cases FilWh+NegIn.

Before comparing the two seemingly best heuristic variants with the other, more elaborate spanning tree construction methods, we want to take a look at the differences in running time when using MSTre or FilWh as spanning tree construction methods. Figure 5.4 shows this comparison of running times including the running time of the GreedyDegree+FillIn heuristic, with which we will compare our heuristic later.

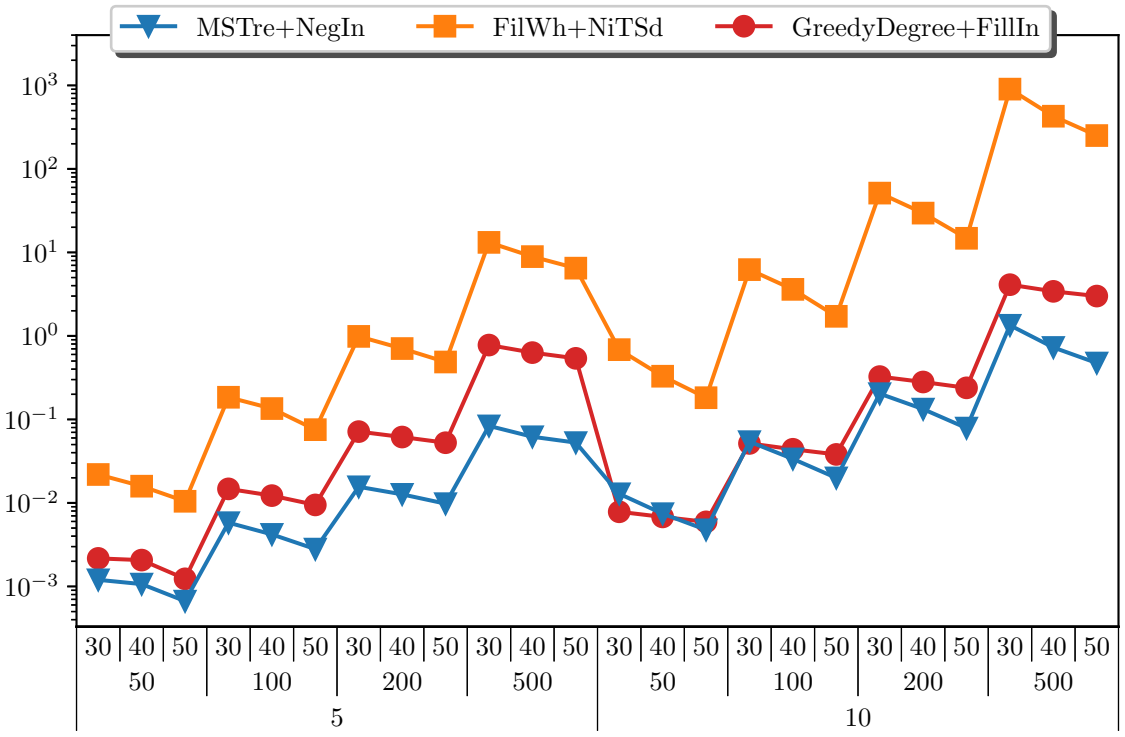


Figure 5.4: Average running time in seconds for different MSTre, FilWh and GreedyDegree+FillIn. The y-axis is scaled logarithmically

Note that the y-axis in the above figure 5.4 is scaled logarithmically. This scaling was chosen due to the drastic increase in the running time for the FillWhile spanning tree construction method for increasing n and k. As can be seen, the running time for one partial k-tree with parameters $(k, n, p) = (10, 500, 30)$ is about 1000 seconds or 16 minutes. Since we had to run this benchmark as a single-threaded benchmark (instead

of multi-threaded), we reduced the number of repetitions per graph to 3 instead of 5 and the number of trees per class of partial k-trees to 10 instead of 20 to keep the total running time of the benchmark in a reasonable time frame.

From the benchmark results shown in figure 5.4, it is not possible to conclude with certainty that the increase in running time from MSTre to FilWh is exponential. However, a drastic increase in the overall running time is clearly visible and the rate at which the running time increases for increasing n seems to be higher for FilWh than for MSTre. Furthermore, a clear pattern in the running time is visible for fixed n and k. The running time is highest for $p = 30$ and lowest for $p = 50$. As discussed in section 4.4, the running time depends mainly on the number of maximal cliques, which would indicate that the number of maximal cliques is highest for $p = 30$ and then decreases for increasing p because possibly too many edges have been removed from the original graph. This observation can also be related to the observation below the comparison of edge weights in figure 5.1, where we saw that for fixed n and k and increasing p, the computed bound increases. Combining these two observations, we can conclude that the increased number of cliques for $p = 30$ compared to $p = 40, 50$ leads to our heuristic computing a better upper bound. This may be because the increased number of maximal cliques allows the heuristic more flexibility in constructing the tree decomposition from the clique graph. On the other hand, the case $p = 30$ may also just be the most similar to $p = 0$, which would be a k-tree for which we have seen that our heuristic computes an optimal upper bound.

After having discussed the running time differences in using FilWh and MSTre as spanning tree construction methods, we want to compare the two seemingly best heuristic variants with the two other, more elaborate, spanning tree construction methods, FillWhileUpdatingEdges (FWhUE) and FillWhileMinimizingBagSize (FWBag), as presented at the end of section 4.3. The comparison is shown in figure 5.5.

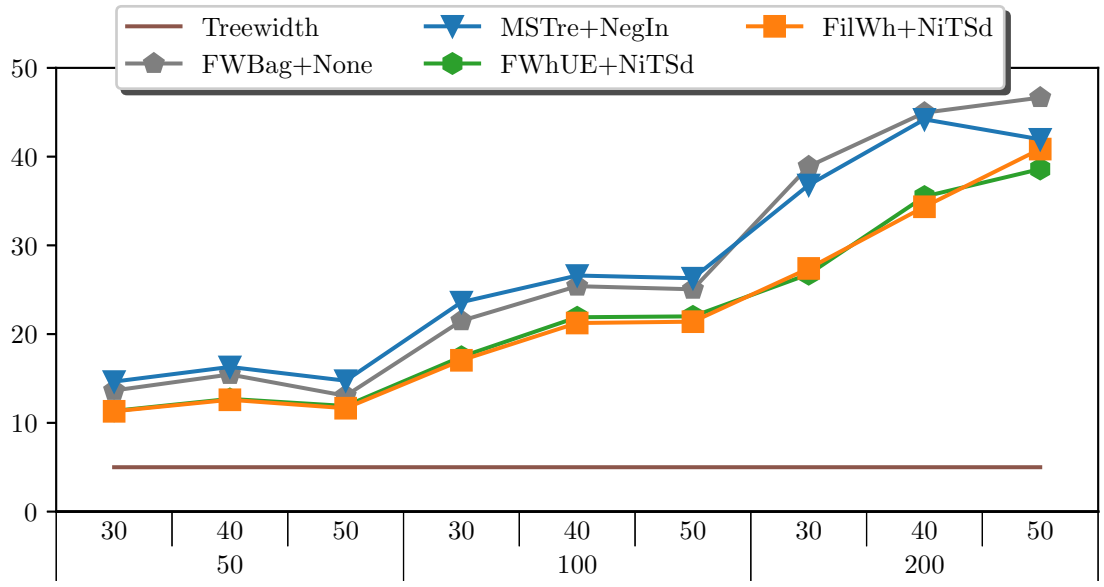


Figure 5.5: Comparison of elaborate spanning tree construction methods by their computed upper bounds for $k = 5$ and $n \leq 200$ only

Note that the comparison of the more elaborate spanning tree construction methods is only for partial k-trees with parameters $n \in \{50, 100, 200\}$, $p \in \{30, 40, 50\}$ and $k = 5$. This is due to the fact that the computation times for the spanning tree construction methods FWhUE and especially FWBag are extremely long, even compared to the already slow FilWh. For instance, one run on a partial k-tree with $n = 500$ takes up to several hours, which forced us to exclude the other classes of k-trees. Nevertheless, we can draw conclusions from the results about the more elaborate spanning tree construction methods. The computationally much more expensive FWBag variant of the heuristic seems to yield worse results than the other two FillWhile variants. This may be because when constructing a tree decomposition, it could be worthwhile to increase the maximum bag size a lot in one step in order to facilitate many of the other bags easily in the next steps. Therefore, trying to greedily minimize the maximum bag size in each step, as FWBag does, might be too conservative in this sense. On the other hand, updating the edges as done in FWhUE seems to have little effect on the computed upper bound. In some cases FWhUE+NiTSd performs a bit better than FilWh+NiTSd, in others it performs a bit worse. However, considering the significant increase in running time, it does not seem worthwhile to update the edges while constructing the spanning tree.

5.2.3 Bounded clique size

As a final parameter, we want to look at the effect of bounding the size of the cliques when constructing the clique graph, as discussed in section 4.7. For the benchmarks, we focused on bounding the clique size either with very small or very large bounds. For example, using 2 as a bound, i.e. constructing the line graph instead of the clique graph. This choice was made mainly for running time reasons. When using 3 instead of 2 (or $\omega(G) - 2$ instead of $\omega(G) - 1$) as clique bound, maximal cliques of size p have to be split into all subcliques of size 3, which are $\binom{p}{3}$ many. For sufficiently large p, this leads to a drastic increase in the number of bags in the computed tree decomposition, which in turn increases the computation time.

The results of the benchmark are shown in figure 5.6. It can be seen that none of the bounded clique variants performed significantly better than their non bounded variants. Moreover, for sufficiently large n, the bounded clique heuristic variants with BC2 produced extremely high upper bounds. On the other hand, the BC-1 variants performed similarly to their non bounded counterparts. This is to be expected, since the $\omega(G) - 1$ bound has only a small effect on the resulting intersection graph used by the heuristic. In turn, by computing the line graph instead of the clique graph, a lot of information about the structure of the original graph is lost. This may be the reason why the variants with BC2 perform so poorly. Furthermore, one can see that using $\omega(G) - 2$ as a bound yields slightly worse results than $\omega(G) - 1$, which would suggest that dividing up the maximal cliques is not promising to begin with. In summary, the results of the bounded clique size benchmarks suggest that using the maximal cliques yields better results than using a bound on the clique size with the heuristic developed in this thesis.

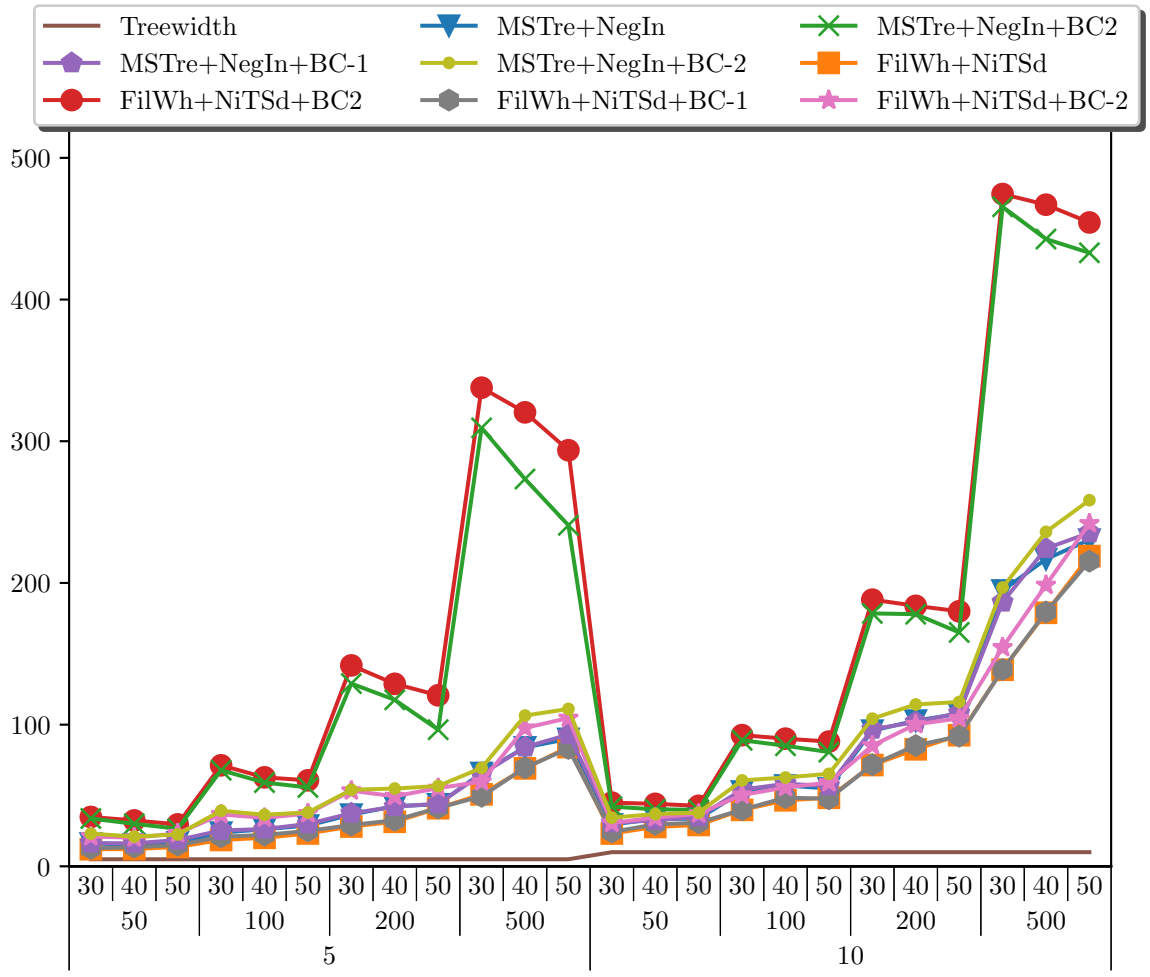


Figure 5.6: Comparison of the upper bounds computed by different clique bounded variants of MSTre+NegIn and FilWh+NiTSd

This completes our comparison of the different variants of our heuristic and we can conclude that FilWh+NiTSd and MSTre+NegIn have been the most performant heuristic variants in our tests, with the former computing a better bound on the treewidth and the latter having a more computation time.

5.2.4 Comparison with known heuristic

In the following, we want to compare the two best variants of our heuristic with the GreedyDegree+FillIn (GD+FI) heuristic as presented in [BK10]. First, we will use the partial k-trees to compare the performance of the heuristics and then look at more practical graph instances from the second Dimacs implementation challenge.

Figure 5.7 shows the results of the comparison on partial k-trees. Note that the plot is normalized so that the actual treewidth is at $y = 1$. From the results, we can clearly see that our heuristic performs much worse than the GD+FI heuristic. The bounds computed by the GD+FI heuristic are very close to 1 times the actual treewidth, while our heuristic computes upper bounds that are up to 15 times the actual treewidth. As seen in the previous results, the computed treewidth increases with the number of vertices in the graph, which may be due to the fact that the number of cliques in the graph increases with the number of vertices.

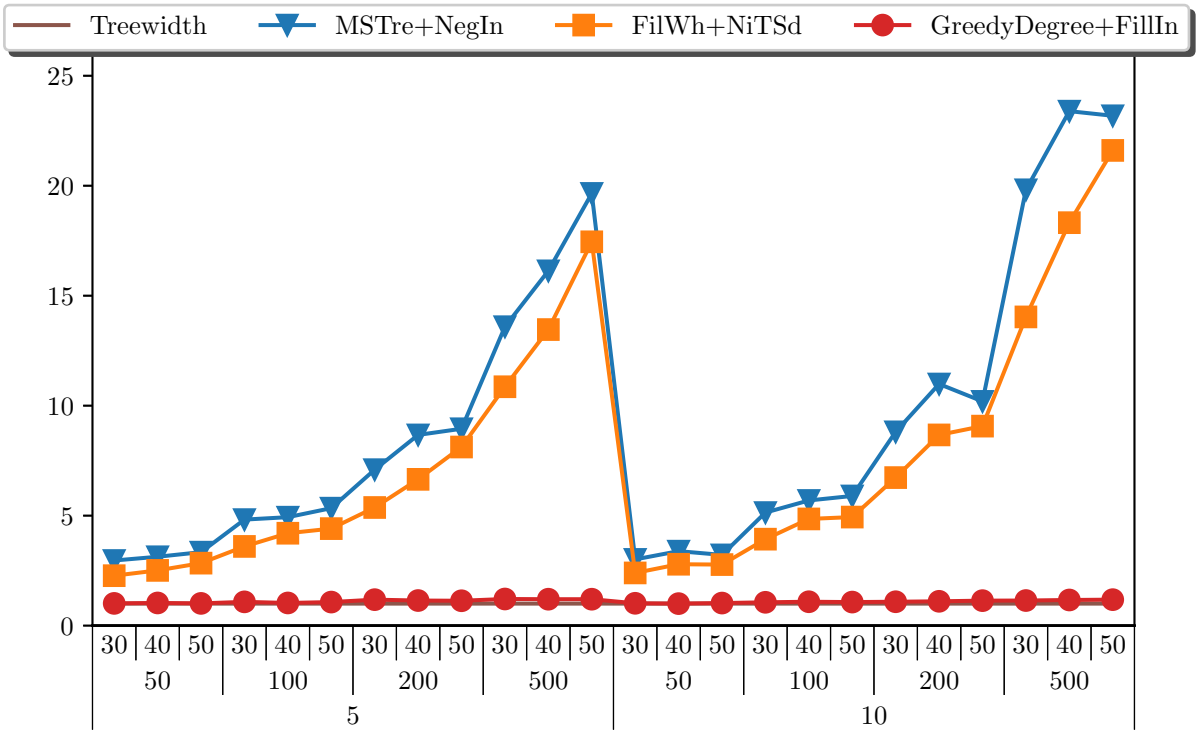


Figure 5.7: Comparison of upper bounds computed by MSTre+NegIn, FilWh+NiTSd and GreedyDegree+FillIn. The y-axis is normalized so that the actual treewidth is at $y = 1$, i.e. $y = 10$ indicates a computed bound of 10 times the actual treewidth.

As a final comparison, we present the computed upper bounds for GD+FI, MSTre+NegIn and FilWh+NiTSd for the coloring graphs of the second Dimacs implementation challenge in table 5.2.

Graph	n	m	LB	UB	GD+FI	MSTre+NegIn	FilWh+NiTSd
fpsol2.i.1	496	11654	66	66	66	101	72
fpsol2.i.2	451	8691	31	31	31	112	43
fpsol2.i.3	425	8688	31	31	31	92	44
inithx.i.1	864	18707	55	56	56	90	65
inithx.i.2	645	13979	31	31	31	101	64
inithx.i.3	621	13969	31	31	31	102	65
le450-5a	450	5714	53	307	315	437	427
le450-5b	450	5734	52	309	318	434	423
le450-5c	450	9803	75	315	315	449	442
le450-5d	450	9757	59	303	303	449	438
le450-15a	450	8168	None	None	290	408	403
le450-15b	450	8169	59	289	291	409	391
le450-25a	450	8260	57	255	255	361	371
le450-25b	450	8263	54	251	265	368	360
mulsol.i.1	197	3925	50	50	50	66	66
mulsol.i.2	188	3885	32	32	32	58	49
mulsol.i.3	184	3916	32	32	32	58	49
mulsol.i.4	185	3946	32	32	32	58	49
mulsol.i.5	186	3973	31	31	32	59	49
zeroin.i.1	211	4100	None	None	50	99	96
zeroin.i.2	211	3541	None	None	33	62	57
zeroin.i.3	206	3540	None	None	33	57	57

Table 5.2: Computed upper bounds of GreedyDegree+FillIn (GD+FI), MSTre+NegIn and FilWh+NiTSd on Dimacs graph coloring instances from the second Dimacs implementation challenge. Known lower and upper bounds for the graphs as presented in [GD12] are given in the fourth and fifth columns, respectively.

It should be noted that the above table 5.2 does not list all the graphs from the second Dimacs implementation challenge. In the case of the graphs not listed, the FilWh+NiTSd heuristic was unable to complete its computations within a reasonable timeframe. Similar to the k-tree benchmarks above, we ran each heuristic 5 times on each graph and used the lowest upper bound as the result.

As evident by the results, our heuristic continues to give relatively poor results in comparison to GD+FI. However, compared to the results on the k-trees in figure 5.7, the bounds computed by our heuristic variants are within 1 to 2 times of the GD+FI heuristic, while this factor was up to 20 for the results on the k-trees. This suggests that our heuristic may perform particularly poorly on k-trees. Furthermore, it is noteworthy that the MSTre+NegIn heuristic produced a lower upper bound on the graph le450-25a, while the FilWh+NiTSd heuristic variant produced significantly better results on almost all other graphs.

We can conclude that the different variants of the heuristic developed in this thesis do not compete with known heuristics for upper bounds on the treewidth. Neither in the upper bound computed nor in the computation time required.

5.2.5 Further analysis of the heuristic

In this final part of the evaluation, our objective is to go more in-depth with our analysis in order to identify some potential underlying aspects that may contribute to the poor performance of the heuristic. To do this, we will focus on FilWh+NiTSd, the most promising variant of the heuristic developed in this thesis. More specifically, we want to understand the process of constructing the spanning tree on the clique graph, i.e. the clique graph tree, since this mainly determines the final tree decomposition. As a first step, we want to discuss figure 5.8, which shows the cardinality of the largest bag for each step of spanning tree construction.

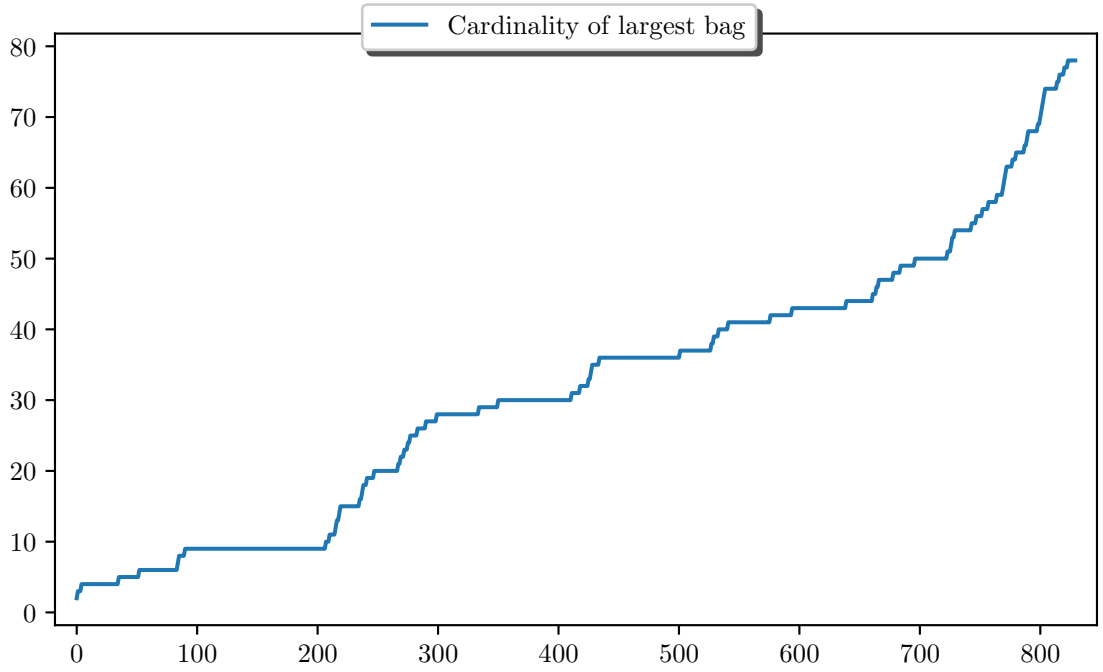


Figure 5.8: Maximum bag size during spanning tree construction using FilWh+NiTSd for one partial k-tree with $(n, k, p) = (500, 5, 40)$. The x-axis denotes the number of bags in the current tree, while the y-axis indicates the cardinality of the largest bag in the current tree.

Note that the number of bags in the current tree exceeds 800, even though the partial k-tree has only 500 vertices. This is due to the fact, that the number of bags in the final clique graph tree is equal to the number of maximal cliques in the partial k-tree, not its number of vertices.

An interesting pattern emerges from the plot in figure 5.8. The plot looks a bit like the steps of a staircase, i.e. the cardinality of the largest bag does not change for up to 100 bags in a row and then increases rapidly, whereupon it stagnates again. Several conclusions can be drawn from this pattern. One is that some, or rather many, of the bags in the final tree decomposition and thus in the clique graph may be redundant. Of course, no two bags in the clique graph are the same, but in the final decomposition there may be bags that are attached to another bag that is a superset of them. This would make such bags redundant. Therefore, one could modify the FilWh method to discard such redundant bags to speed up the spanning tree construction process. Consequently, another conclusion might be that only some of the bags in the clique graph are actually relevant for determining the final decomposition. More precisely, from the figure 5.8 we can deduce that for many steps in spanning tree construction, the choice of which bag to add next to the spanning tree seems easy, indicated by the intervals in which the cardinality of the largest bag remains constant. However, when there are no more bags that can easily be attached to the current spanning tree, one or multiple bags have to be added that drastically increase the cardinality of the largest bag, resulting in the staircase pattern in the plot. We want to refer to such steps in spanning tree construction as 'difficult steps'. One may now conclude that the decision of which bag to add in a difficult step has a major impact on the spanning tree and therefore on the quality of the tree decomposition.

With this in mind, we want to look at the variance in results when using the same heuristic multiple times on the same graph. We have already discussed that the implementation of our heuristic is nondeterministic, in the sense that the results depend on the order of vertices and bags in the graph, which is determined at runtime, see subsection 5.1.3.

Figure 5.9 visualizes the variance of the computed bounds for MSTre+NegIn and FilWh+NiTSd. As can be seen, the results can vary heavily, from up to 120 to less than 60 for the same graph. Using our findings from figure 5.8, which plots the maximum bag size during spanning tree construction, we could attribute this to small changes in the order of bags in the clique graph, which in turn could affect the choice of bags in 'difficult steps', i.e. steps where there is no obvious choice of which bag to attach next to the current spanning tree.

Figure 5.10 shows the same data as 5.9, but visualized as a performance plot. The performance plot again shows that there is a possibility that the computed upper bound is very bad compared to other bounds computed on the same graph using the same heuristic. But we can see that as the bound gets worse, the probability of it being computed decreases very fast.

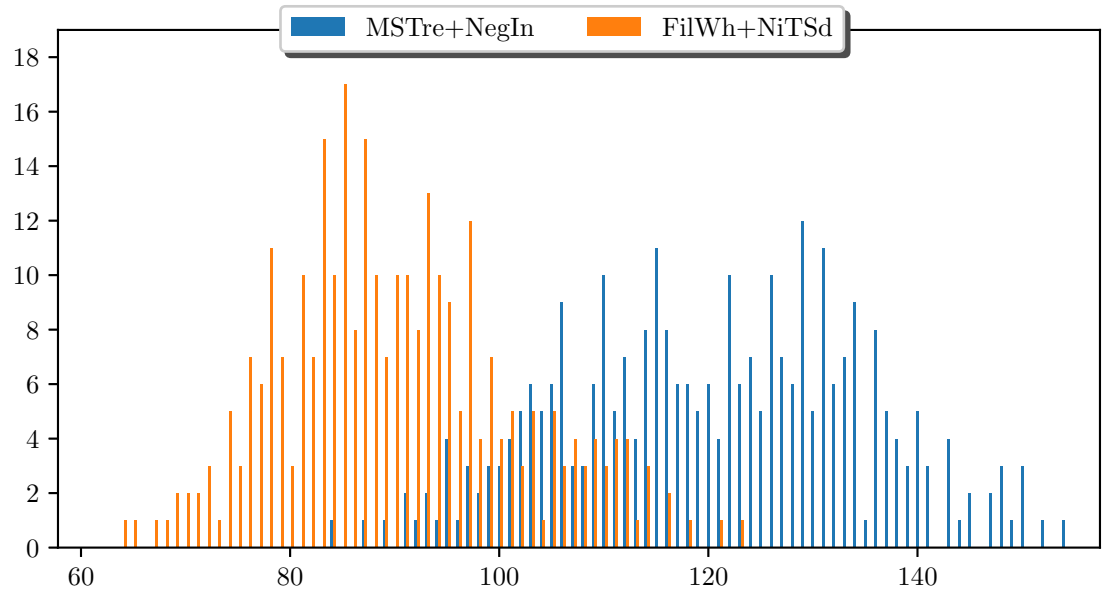


Figure 5.9: Bounds computed by MSTre+NegIn and FilWh+NiTSd over 300 runs on one partial k-tree with $(n, k, p) = (500, 5, 40)$. The x-axis indicates the computed bounds and the y-axis denotes the number of times a bound has been computed.

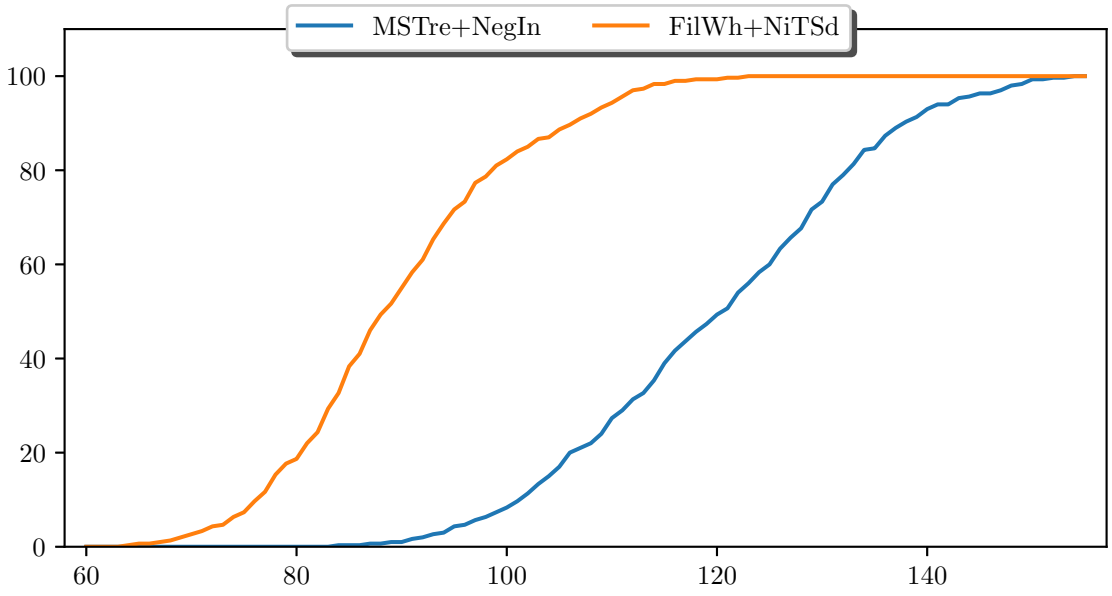


Figure 5.10: Percentage of repetitions that returned an upper bound below a given value comparing MSTre+NegIn and FilWh+NiTSd over 300 runs on one partial k-tree with $(n, k, p) = (500, 5, 40)$. Here, $y(x)$ is the percentage of runs that returned an upper bound less than x, where x denotes a possible upper bound.

6 Conclusion

In this thesis, we presented a novel heuristic for computing an upper bound on the treewidth of undirected simple graphs using the clique operator. We were able to prove its optimality on the class of chordal graphs. However, this turned out to not be of much practical use, since the recognition and construction of a tree decomposition for chordal graphs is possible in linear time using known algorithms such as maximum cardinality search. In contrast, our heuristic has polynomial running time in n on chordal graphs and exponential running time on general graphs. Furthermore, the evaluation has demonstrated that on general graphs, the heuristic developed in this thesis computes extremely poor upper bounds with up to 20 times the actual treewidth in cases where other heuristics compute bounds around 1.1 times the actual treewidth. In the last part of the evaluation, we attempted to identify potential causes of this lack of performance. Although the relative quality of the computed bounds improved significantly on graphs from the second Dimacs implementation challenge, we concluded that our heuristic is not of any direct practical use, since more performant alternatives exist.

Nevertheless, in this thesis we were able to highlight interesting connections between intersection graph theory and the notion of treewidth. One of these is the underlying idea of using the intersection graph of the maximal cliques, that is, the clique graph, as the basis for the heuristic. We generalized this by imposing a bound on the size of the cliques, thereby dividing maximal cliques that exceed a certain threshold into smaller cliques. In this work we only investigated fixed bounds on the clique size. An alternative approach could be to use a flexible bound that splits all cliques into cliques of 0.7 times the size of the original maximal clique. Moreover, to reduce the resulting increase in computation time, one could split maximal cliques only up to a certain size.

Based on this, another interesting aspect could be, whether the approach of using the clique graph as the basis of the heuristic can even lead to good results at all. More precisely, we demonstrated that there exists a graph for which no spanning tree on the clique graph yields an optimal tree decomposition when filling the bags. We tried to generalize this result in order to find a family of graphs for which arbitrarily bad results can be forced, examining families such as grid graphs and Turán graphs, but were unable to reach such a conclusion. Of course, this question can be generalized to whether there exists a spanning tree on any intersection graph such that it yields a good tree decomposition or whether there exists a fixed intersection graph for which there always exists a spanning tree that yields a good tree decomposition.

Bibliography

[AG03] Liliana Alcón and Marisa Gutierrez. "A new characterization of clique graphs". In: *Matemática Contemporânea* 25 (Jan. 2003). DOI: 10.21711/231766362003/rmc251.

[Alc+06] Liliana Alcón et al. "Clique Graph Recognition Is NP-Complete." In: Jan. 2006, pp. 269–277.

[AP89] Stefan Arnborg and Andrzej Proskurowski. "Linear time algorithms for NP-hard problems restricted to partial k-trees". In: *Discrete Applied Mathematics* 23.1 (1989), pp. 11–24. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(89)90031-0. URL: https://www.sciencedirect.com/science/article/pii/0166218X89900310.

[Arn85] Stefan Arnborg. "Efficient algorithms for combinatorial problems on graphs with bounded decomposability — A survey". In: *BIT Numerical Mathematics* 25.1 (Mar. 1985), pp. 1–23. ISSN: 1572-9125. DOI: 10.1007/BF01934985. URL: https://doi.org/10.1007/BF01934985.

[BK07] Hans L. Bodlaender and Arie M. C. A. Koster. "Combinatorial Optimization on Graphs of Bounded Treewidth". In: *The Computer Journal* 51.3 (July 2007), pp. 255–269. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxm037. eprint: https://academic.oup.com/comjnl/article-pdf/51/3/255/1136951/bxm037.pdf. URL: https://doi.org/10.1093/comjnl/bxm037.

[BK10] Hans L. Bodlaender and Arie M.C.A. Koster. "Treewidth computations I. Upper bounds". In: *Information and Computation* 208.3 (2010), pp. 259–275. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2009.03.008. URL: https://www.sciencedirect.com/science/article/pii/S0890540109000947.

[BK73] Coen Bron and Joep Kerbosch. "Algorithm 457: finding all cliques of an undirected graph". In: *Commun. ACM* 16.9 (Sept. 1973), pp. 575–577. ISSN: 0001-0782. DOI: 10.1145/362342.362367. URL: https://doi.org/10.1145/362342.362367.

[BM93] Hans L. Bodlaender and Rolf H. Möhring. "The Pathwidth and Treewidth of Cographs". In: *SIAM Journal on Discrete Mathematics* 6.2 (1993), pp. 181–188. DOI: 10.1137/0406014. eprint: https://doi.org/10.1137/0406014. URL: https://doi.org/10.1137/0406014.

[Bod07] Hans L. Bodlaender. "Treewidth: Structure and Algorithms". In: *Structural Information and Communication Complexity*. Ed. by Giuseppe Prencipe and Shmuel Zaks. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 11–25. ISBN: 978-3-540-72951-8.

[Bod98] Hans L. Bodlaender. "A partial k-arboretum of graphs with bounded treewidth". In: *Theoretical Computer Science* 209.1 (1998), pp. 1–45. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(97)00228-4. URL: https://www.sciencedirect.com/science/article/pii/S0304397597002284.

[BP91] Hans-Jürgen Bandelt and Erich Prisner. "Clique graphs and Helly graphs". In: *Journal of Combinatorial Theory, Series B* 51.1 (1991), pp. 34–45. ISSN: 0095-8956. DOI: https://doi.org/10.1016/0095-8956(91)90004-4. URL: https://www.sciencedirect.com/science/article/pii/0095895691900044.

[Bun74] Peter Buneman. "A characterisation of rigid circuit graphs". In: *Discrete Mathematics* 9.3 (1974), pp. 205–212. ISSN: 0012-365X. DOI: https://doi.org/10.1016/0012-365X(74)90002-8. URL: https://www.sciencedirect.com/science/article/pii/0012365X74900028.

[BY89] Egon Balas and Chang Sung Yu. "On graphs with polynomially solvable maximum-weight clique problem". In: *Networks* 19.2 (1989), pp. 247–253. DOI: https://doi.org/10.1002/net.3230190206. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230190206. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230190206.

[CK08] F. Cazals and C. Karande. "A note on the problem of reporting maximal cliques". In: *Theoretical Computer Science* 407.1 (2008), pp. 564–568. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2008.05.010. URL: https://www.sciencedirect.com/science/article/pii/S0304397508003903.

[CN85] Norishige Chiba and Takao Nishizeki. "Arboricity and Subgraph Listing Algorithms". In: *SIAM Journal on Computing* 14.1 (1985), pp. 210–223. DOI: 10.1137/0214017. eprint: https://doi.org/10.1137/0214017. URL: https://doi.org/10.1137/0214017.

[Die05] R. Diestel. *Graph Theory*. Electronic library of mathematics. Springer, 2005. ISBN: 9783540261834. URL: https://books.google.de/books?id=aR2TMYQr2CMC.

[Dir61] G. A. Dirac. "On rigid circuit graphs". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 25.1 (Apr. 1961), pp. 71–76. ISSN: 1865-8784. DOI: 10.1007/BF02992776. URL: https://doi.org/10.1007/BF02992776.

[DK07] Alex Dow and Richard Korf. "Best-First Search for Treewidth". In: vol. 2. Jan. 2007, pp. 1146–1151.

[Gav74] Fănică Gavril. "The intersection graphs of subtrees in trees are exactly the chordal graphs". In: *Journal of Combinatorial Theory, Series B* 16.1 (1974), pp. 47–56. ISSN: 0095-8956. DOI: https://doi.org/10.1016/0095-8956(74)90094-X. URL: https://www.sciencedirect.com/science/article/pii/009589567490094X.

[GD12] Vibhav Gogate and Rina Dechter. *A Complete Anytime Algorithm for Treewidth*. 2012. arXiv: 1207.4109 [cs.DS]. URL: https://arxiv.org/abs/1207.4109.

[Gol80] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 1980. ISBN: 978-0-12-289260-8. DOI: https://doi.org/10.1016/B978-0-12-289260-8.50008-X. URL: https://www.sciencedirect.com/science/article/pii/B978012289260850008X.

[Ham68] Ronald C. Hamelink. "A partial characterization of clique graphs". In: *Journal of Combinatorial Theory* 5.2 (1968), pp. 192–197. ISSN: 0021-9800. DOI: https://doi.org/10.1016/S0021-9800(68)80055-9. URL: https://www.sciencedirect.com/science/article/pii/S0021980068800559.

[Hed84] Bruce Hedman. "Clique graphs of time graphs". In: *Journal of Combinatorial Theory, Series B* 37.3 (1984), pp. 270–278. ISSN: 0095-8956. DOI: https://doi.org/10.1016/0095-8956(84)90059-5. URL: https://www.sciencedirect.com/science/article/pii/0095895684900595.

[HSC08] Aric Hagberg, Pieter Swart, and Daniel Chult. "Exploring Network Structure, Dynamics, and Function Using NetworkX". In: Jan. 2008.

[JT96] David J. Johnson and Michael A. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. USA: American Mathematical Society, 1996. ISBN: 0821866095.

[Kor22] Tuukka Korhonen. "A Single-Exponential Time 2-Approximation Algorithm for Treewidth". In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. 2022, pp. 184–192. DOI: 10.1109/FOCS52979.2021.00026.

[MM60] R. E. Miller and D. E. Muller. "A problem of maximum consistent subsets". In: *IBM Research Report RC-240* (Mar. 1960).

[MM65] J. W. Moon and L. Moser. "On cliques in graphs". In: *Israel Journal of Mathematics* 3.1 (Mar. 1965), pp. 23–28. ISSN: 1565-8511. DOI: 10.1007/BF02760024. URL: https://doi.org/10.1007/BF02760024.

[MM99] Terry A. McKee and F. R. McMorris. *Topics in Intersection Graph Theory*. Society for Industrial and Applied Mathematics, 1999. DOI: 10.1137/1.9780898719802. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719802. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898719802.

[MU04] Kazuhisa Makino and Takeaki Uno. “New Algorithms for Enumerating All Maximal Cliques”. In: *Algorithm Theory - SWAT 2004*. Ed. by Torben Hagerup and Jyrki Katajainen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 260–272. ISBN: 978-3-540-27810-8.

[Pet] Petgraph maintainer group. *Petgraph*. Version 0.6.4. URL: https://github.com/petgraph/petgraph.

[Pri95] Erich Prisner. “Graphs with few cliques”. In: *Graph theory, combinatorics, and algorithms* 1.2 (1995), pp. 0844–05062.

[PS99a] Erich Prisner and Jayme L. Szwarcfiter. “Recognizing clique graphs of directed and rooted path graphs”. In: *Discrete Applied Mathematics* 94.1 (1999). Proceedings of the Third International Conference on Graphs and Optimization GO-III, pp. 321–328. ISSN: 0166-218X. DOI: https://doi.org/10.1016/S0166-218X(99)00028-1. URL: https://www.sciencedirect.com/science/article/pii/S0166218X99000281.

[PS99b] Fábio Protti and Jayme Luiz Szwarcfiter. “On clique graphs with linear size”. In: *Relatório Técnico NCE* 0799 (1999). URL: http://hdl.handle.net/11422/2650.

[RS71] Fred S. Roberts and Joel H. Spencer. “A characterization of clique graphs”. In: *Journal of Combinatorial Theory, Series B* 10.2 (1971), pp. 102–108. ISSN: 0095-8956. DOI: https://doi.org/10.1016/0095-8956(71)90070-0. URL: https://www.sciencedirect.com/science/article/pii/0095895671900700.

[RS86] Neil Robertson and P.D Seymour. “Graph minors. II. Algorithmic aspects of tree-width”. In: *Journal of Algorithms* 7.3 (1986), pp. 309–322. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(86)90023-4. URL: https://www.sciencedirect.com/science/article/pii/0196677486900234.

[Rust] The Rust Foundation. *Rust*. Version 1.76. URL: https://www.rust-lang.org/.

[SB94] Jayme Szwarcfiter and Claudson Bornstein. “Clique Graphs of Chordal and Path Graphs”. In: *SIAM J. Discrete Math.* 7 (May 1994), pp. 331–336. DOI: 10.1137/S0895480191223191.

[Sch89] Petra Scheffler. “Die Baumweite von Graphen als ein Maß für die Kompliziertheit algorithmischer Probleme”. PhD thesis. Dec. 1989.

[SG97] Kirill Shoikhet and Dan Geiger. “A Practical Algorithm for Finding Optimal Triangulations”. In: *AAAI/IAAI*. 1997. URL: https://api.semanticscholar.org/CorpusID:6072334.

[Szw03] J. L. Szwarcfiter. “A Survey on Clique Graphs”. In: *Recent Advances in Algorithms and Combinatorics*. Ed. by Bruce A. Reed and Cláudia L. Sales. New York, NY: Springer New York, 2003, pp. 109–136. ISBN: 978-0-387-22444-2. DOI: 10.1007/0-387-22444-0_5. URL: https://doi.org/10.1007/0-387-22444-0_5.

[Tho81] Carsten Thomassen. “Kuratowski’s theorem”. In: *Journal of Graph Theory* 5.3 (1981), pp. 225–241. DOI: https://doi.org/10.1002/jgt.3190050304. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jgt.3190050304. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jgt.3190050304.

[TTT06] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. “The worst-case time complexity for generating all maximal cliques and computational experiments”. In: *Theoretical Computer Science* 363.1 (2006). Computing and Combinatorics, pp. 28–42. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2006.06.015. URL: https://www.sciencedirect.com/science/article/pii/S0304397506003586.

[TY84] Robert E. Tarjan and Mihalis Yannakakis. “Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs”. In: *SIAM Journal on Computing* 13.3 (1984), pp. 566–579. DOI: 10.1137/0213035. eprint: https://doi.org/10.1137/0213035. URL: https://doi.org/10.1137/0213035.