Treewidth: Computational Experiments
Treewidth: Computational Experiments

Arie M.C.A. Koster† Hans L. Bodlaender‡ Stan P.M. van Hoesel§

December 28, 2001

Abstract

Many \( \mathcal{NP} \)-hard graph problems can be solved in polynomial time for graphs with bounded treewidth. Equivalent results are known for pathwidth and branchwidth. In recent years, several studies have shown that this result is not only of theoretical interest but can successfully be applied to find (almost) optimal solutions or lower bounds for many optimization problems.

To apply a tree decomposition approach, the treewidth of the graph has to be determined, independently of the application at hand. Although for fixed \( k \), linear time algorithms exist to solve the decision problem “treewidth \( \leq k \)”, their practical use is very limited. The computational tractability of treewidth has been rarely studied so far. In this paper, we compare four heuristics and two lower bounds for instances from applications such as the frequency assignment problem and the vertex coloring problem.

Three of the heuristics are based on well-known algorithms to recognize triangulated graphs. The fourth heuristic recursively improves a tree decomposition by the computation of minimal separating vertex sets in subgraphs. Lower bounds can be computed from maximal cliques and the minimum degree of induced subgraphs. A computational analysis shows that the treewidth of several graphs can be identified by these methods. For other graphs, however, more sophisticated techniques are necessary.

1 Introduction

In the 1980s and early 1990s, Robertson and Seymour defined the graph parameters pathwidth [28], treewidth [29], and branchwidth [30, 31] as well as the associated graph structures path decomposition, tree decomposition, and branch decomposition. Originally these concepts were introduced in the context of their fundamental research on graph minors. In addition, the
notions have proved to be of importance in computational complexity theory. Many $\mathcal{NP}$-hard graph problems can be solved in polynomial time for graphs with path-, tree-, or branchwidth bounded by a constant. Among these problems are many well-known combinatorial optimization problems such as the maximum independent set problem, the Hamiltonian cycle problem, and the Steiner tree problem. Moreover, several $\mathcal{PSPACE}$-complete problems can be solved in linear time for input restricted to graphs of bounded treewidth (cf. [6]).

All these results have been considered primarily of theoretical interest only. In recent years, a number of computational studies has shown that the results can also be applied successfully. A first attempt to benefit from the polynomial time solvability has been undertaken by Cook and Seymour [10]. They use branch decompositions to obtain close-to-optimal solutions of the traveling salesman problem. Path decompositions are used by Verweij [37] to solve lifting problems of cycle inequalities for the independent set problem. In [22, 23], tree decompositions are used to obtain lower bounds and optimal solutions for a special type of frequency assignment problems. Finally, tree decompositions are used to solve problems in the area of expert systems. The currently most efficient algorithm for the inference calculation in probabilistic (or Bayesian) networks builds upon a tree decomposition of a network's moralized graph [20, 24].

All these studies show that (dynamic programming) algorithms based on a path/tree/branch decomposition of the graph can be an alternative for integer programming techniques to solve hard (combinatorial) optimization problems.

The procedure to solve an optimization problem with for instance bounded treewidth involves two steps: (i) computation of a (good) tree decomposition, and (ii) application of an algorithm that solves instances of bounded treewidth in polynomial time (typically by a dynamic programming algorithm based on the tree decomposition). Whereas the second step is application-dependent, the calculation of a good tree decomposition of a graph can be done independently of the application. The need for a tree decomposition with optimal, or second best, good width becomes clear by the time and space requirements of the algorithm in the second step, which typically are exponentially in the width. A similar procedure is due for pathwidth and branchwidth. Henceforth, we focus on the computation of the treewidth of graphs from a practical point of view. Hicks [19] recently studied the practical computation of branchwidth for (specific classes of) graphs.

The treewidth of a graph is, in general, $\mathcal{NP}$-hard to compute. Much research has been carried out to determine the treewidth (or related notions) for classes of graphs. Moreover, for fixed $k$, linear time algorithms to determine whether a graph has treewidth $k$ have been developed. Their use in practice, however, is very limited since the running time contains a large constant with $k$ in the exponent. For small graphs, Shoikhet and Geiger [33] performed various experiments on randomly generated graphs with an adaptation of an algorithm of Arnborg et al. [3]. Approximation with an $O(\log n)$ approximation ratio can be done in polynomial time (here $n$ denotes the number of vertices in the graph); finding a polynomial time constant approximation ratio algorithm remains open. We refer to [6] for a survey on these and other complexity issues for treewidth.

Heuristics without theoretical quality guarantee and easily computable lower bounds emerge as a practical alternative. Motivated by problems from combinatorial optimization and probabilistic networks, we describe a computational analysis of four heuristics and two lower bounds in this paper. We focus on algorithms that are not exponential in the treewidth (like the algorithm
of Becker and Geiger [5]). The heuristics are described in Section 3, whereas Section 4 is devoted to the lower bounds. The computational analysis of the heuristics and lower bounds on several classes of graphs is the topic of Section 5. The used graphs originate from frequency assignment problems, vertex coloring problems, and probabilistic networks. The paper is closed by concluding remarks and directions for further research.

2 Definitions and Notation

We briefly introduce the most important definitions and notation. Let $G = (V, E)$ be an undirected graph with vertex set $V$ and edge set $E$. Let $n = |V|$ and $m = |E|$ throughout this paper. The set of vertices adjacent to a vertex $v \in V$ is denoted by $N(v) = \{w \in V : vw \in E\}$. Let $\delta(v) := |N(v)|$ be the degree of $v$. A set of vertices $Q \subseteq V$ is called a clique in $G$ if there is an edge between every pair of distinct vertices from $Q$. The cardinality $|Q|$ of $Q$, is the size of the clique. For a set $W$ of vertices, the subgraph induced by $W$ is the graph $G[W] := (W, E[W])$, with $E[W] := (W \times W) \cap E$. A graph $H$ is a minor of a graph $G$ if $H$ can be obtained from $G$ by vertex deletions, edge deletions, and edge contractions (the operation that replaces two adjacent vertices $v$ and $w$ by a single vertex that is connected to all neighbors of $v$ and $w$).

The tree decomposition concept and its measure of value treewidth was introduced by Robertson and Seymour [29] in the context of graph minor research.

**Definition 2.1 (Robertson and Seymour [29])** Let $G = (V, E)$ be a graph. A tree decomposition of $G$ is a pair $(T, X)$, where $T = (I, F)$ is a tree with node set $I$ and edge set $F$, and $X = \{X_i : i \in I\}$ is a family of subsets of $V$, one for each node of $T$, such that

(i). $\bigcup_{i \in I} X_i = V$,

(ii). for every edge $vw \in E$, there is an $i \in I$ with $v \in X_i$ and $w \in X_i$, and

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

The width of a tree decomposition is $\max_{i \in I} |X_i| - 1$. The treewidth of a graph $G$, denoted by $tw(G)$, is the minimum width over all possible tree decompositions of $G$.

The third condition of the tree decomposition is equivalent to the condition that for all $v \in V$, the set of nodes $\{i \in I : v \in X_i\}$ is connected in $T$, i.e., they form a subtree. Figure 1 shows an example of a graph together with an optimal tree decomposition.

If we restrict the tree $T$ to be a path, a path decomposition of $G$ is obtained. The minimum width over all possible path decompositions of $G$ is called the pathwidth of $G$. Many definitions equivalent to treewidth or pathwidth are known, see [7] for an overview. For the heuristics described in the next section, two of these alternative characterizations are exploited.

3 Heuristics

In this section, we present four heuristics for computing the treewidth of a graph. The first three are based on algorithms for determining whether a graph is triangulated. The fourth one
3.1 Graph Triangulation Heuristics

A graph is called **triangulated** if every cycle of length at least four contains a chord, that is, two non-consecutive vertices on the cycle are adjacent. Triangulated graphs are also called **chordal** due to the existence of a chord in every cycle. Triangulated graphs are among the first classes of graphs for which it has been proved that they are perfect.

The reason for introducing triangulated graphs in this paper is the following property of these graphs:

**Theorem 3.1 (Gavril [16])** A graph $G = (V, E)$ is triangulated if and only if $G$ can be constructed as the intersection graph of subtrees of trees, i.e., there exists a tree $T = (I, F)$ such that one can associate a subtree $T_v = (I_v, F_v)$ of $T$ with each vertex $v \in V$, such that $vw \in E$ if and only if $I_v \cap I_w \neq \emptyset$.

Although the concept of tree decomposition was not yet introduced by the time Gavril formulated Theorem 3.1, exactly this concept was used to characterize triangulated graphs. To state the result in a different way, a graph is triangulated if and only if there exists a tree decomposition with the additional property that $vw \in E$ if and only if $I_v \cap I_w \neq \emptyset$. This additional property...
guarantees that the tree decomposition has minimal width. Let $X_i := \{ v \in V : i \in I_v \}$. Non-adjacent vertices are not in a common subset $X_i$. Hence, a maximum cardinality subset $X_i$ contains the vertices of a maximum cardinality clique $Q$ in $G$. Since it also holds that in any tree decomposition $(T, \mathcal{X})$ and every maximum clique $Q$, there exists a node $i \in I$ with $Q \subseteq X_i$ (cf. Section 4.1), the tree decomposition has minimum width. Moreover, it follows that the treewidth of a triangulated graph equals the maximum clique number minus one, $tw(G) = \omega(G) - 1$.

**Lemma 3.2** For every graph $G = (V, E)$, there exists a triangulation of $G$, $\tilde{G} = (V, E \cup E_i)$, with $tw(\tilde{G}) = tw(G)$.

**Proof.** Let $G$ be a general graph and $(T, \mathcal{X})$ a tree decomposition of minimum width. We construct a graph $\tilde{G} = (V, E)$ by the following rule: $vw \in E$ if and only if $vw \in X_i$ for some $i \in I$. From Theorem 3.1 it is clear that $\tilde{G}$ is triangulated. From the second condition of a tree decomposition, the edge set $E$ can be divided into two parts $E$ and $E_i$. So, $G$ is a triangulation of $G$ by the addition of the triangulation edges $E_i$. Moreover, the treewidth of $G$ and $\tilde{G}$ is equal, $tw(G) = tw(\tilde{G})$. \hfill \blacksquare

**Corollary 3.3** Finding the treewidth of a graph $G$ is equivalent to finding a triangulation $\tilde{G}$ of $G$ with minimum clique size.

Since finding the treewidth of a graph is \textit{NP}-hard, also finding a triangulation with minimum clique number is an \textit{NP}-hard problem. However, the clique number (minus one) of any triangulation $\tilde{G}$ of a graph $G$ provides an upper bound on the treewidth. Moreover, $\omega(\tilde{G})$ can be computed in polynomial time. Therefore, we are looking for algorithms that triangulate a graph, and by that provide upper bounds for the treewidth. There exist several algorithms to test whether a graph is triangulated. These recognition algorithms either detect that the graph is triangulated or return a triangulation of the graph. In the subsequent subsections, three of these algorithms are presented, two variants of the lexicographic breadth first search algorithm of Rose et al. [32], and the maximum cardinality search algorithm of Tarjan and Yannakakis [35]. All three algorithms are based on the characterization of triangulated graphs by a perfect elimination scheme.

A vertex $v \in V$ is called simplicial if its neighbors $N(v)$ induce a complete subgraph of $G$. An ordering $\sigma = [v_1, \ldots, v_n]$ of the vertices is called a perfect elimination scheme if $v_i$ is a simplicial vertex of the induced subgraph $G[\{v_1, \ldots, v_n\}]$ for all $i = 1, \ldots, n$. So, $G$ contains a simplicial vertex $v_1$, and every time a simplicial vertex $v_i$ is removed from the graph, a new simplicial vertex $v_{i+1}$ exists. Fulkerson and Gross [15] proved that a graph is triangulated if and only if it has a perfect elimination scheme. Given a perfect elimination scheme, the maximal cliques of $G$ are of the form $\{u\} \cup M(v)$, where $M(v) = \{ w \in N(v) : \sigma^{-1}(v) < \sigma^{-1}(w) \}$ denotes the set of higher ordered neighbors. The value $\max_{v \in V} |M(v)|$ is also called the dimension of $G$. Arnborg [2] proved that the dimension of $G$ equals the treewidth.

For a non-triangulated graph $G$ and an elimination scheme $\sigma$, a triangulation $\tilde{G}$ can be constructed by Algorithm 1. Moreover, the algorithm returns the treewidth of $\tilde{G}$, which is an upper bound for the treewidth of $G$. The subset $E_i := E \setminus E$ is also called the \textit{fill-in} of $G$. 

5
Algorithm 1 Triangulation of a graph, given an elimination scheme

Input: connected graph \( G = (V, E) \), elimination scheme \( \sigma = [v_1, \ldots, v_n] \)
Output: triangulated graph \( \hat{G} = (V, \hat{E}) \) with \( tw(\hat{G}) \geq tw(G) \)

1: for \( v \in V \) do
2: \( M(v) := N(v) \)
3: end for
4: \( E := E, tw := 0 \)
5: for \( i = 1 \) to \( n \) do
6: if \( |M(v_i)| > tw \) then
7: \( tw := |M(v_i)| \) // update treewidth
8: end if
9: for \( u, w \in M(v_i) \) and \( uw \notin \hat{E} \) do
10: \( E := E \cup \{uw\} \) // add edge between higher ordered neighbors
11: \( M(u) := M(u) \cup \{w\}, M(w) := M(w) \cup \{u\} \) // update adjacency lists
12: end for
13: for \( u \in M(v_i) \) do
14: \( M(u) := M(u) \setminus \{v_i\} \) // \( v_i \) is lower ordered than \( u \)
15: end for
16: end for

The easiest way to recognize triangulated graphs is the construction of an elimination scheme \( \sigma \), which is perfect (i.e., no edges are added by Algorithm 1) in case the graph is indeed triangulated. The following observation plays an important role in such a recognition algorithm.

**Proposition 3.4 (Fulkerson and Gross [15])** Let \( G \) be a triangulated graph. Any simplicial vertex can start a perfect elimination scheme for \( G \).

So an algorithm could be: search for a simplicial vertex in the graph, add it to the elimination scheme and remove it from the graph. If at some point no simplicial vertex exists, the graph is not triangulated, otherwise the graph indeed is. Such an algorithm requires \( n \) searches for a simplicial vertex, which requires an adjacency-test for all pairs of neighbors of a vertex. More efficient algorithms can be constructed by using another observation.

**Proposition 3.5 (Dirac [12])** Every triangulated, non-complete graph has two non-adjacent simplicial vertices.

This observation (combined with Proposition 3.4) implies that at every step of the above described algorithm we have the choice between at least two non-adjacent vertices to place at the next position of the elimination scheme. Hence, without loss of generality, we can select a vertex \( v_n \) for the last position in the elimination scheme. Next, we choose a vertex \( v_{n-1} \) adjacent to \( v_n \) for the \((n-1)\)th position. Further vertices have to be selected in such a way that their already selected neighbors form a clique. In this way, the elimination scheme is constructed backwards. Since, vertices need not be identified as simplicial explicitly, algorithms that construct the elimination scheme are more efficient compared with forward construction algorithms. In the next two subsections, three such backward algorithms are presented.

For non-triangulated graphs these algorithms can be applied to obtain a triangulation, and thus, an upper bound on the treewidth of \( G \). Since in general \( G \) is not triangulated, the choice of the vertex \( v_n \) for the last position of the elimination scheme influences the upper bound returned by the algorithm.
3.1.1 Lexicographic Breadth First Search

In this subsection, we present two variants of what is called a lexicographic breadth first search algorithm. These algorithms as well as the maximum cardinality search algorithm in the next subsection only differ in the way the vertices for positions $n-1, n-2, \ldots, 1$ are determined. All three algorithms guarantee that the elimination scheme is perfect if and only if $G$ is triangulated. Hence, for a triangulated graph $G$, the vertices have to be ordered in such a way that for all $i$, $1 \leq i \leq n$ it should hold that $v_i$ is simplicial in $G[\{v_i, \ldots, v_n\}]$.

The algorithm presented in Rose et al. [32] is known as the lexicographic breadth-first search recognition algorithm (see also [17, Section 4.3]). A first version of this algorithm focuses primarily on the perfectness of the elimination scheme (referred to as LEX$_P$). In the original version, it terminates as soon as it recognizes that no perfect elimination scheme exists, i.e., as soon as it can be used to find a triangulation of $G$. The algorithm labels the vertices with their already ordered neighbors. Each label consists of the lexicographic highest label is selected in every iteration (cf. Algorithm 2). The most efficient implementation of Algorithm 2 requires $O(n + m')$ time and space, where $m' = m + |E_t|$ (see [32] for details). If we start the algorithm $n$ times with every time another vertex $v^*$, the overall running time is $O(n^2 + nm')$.

**Algorithm 2 Lexicographic Breadth First Search, variant Perfect (LEX$_P$)**

**Input:** connected graph $G = (V, E)$, vertex $v^* \in V$

**Output:** triangulated graph $G_T = (V, E \cup E_t)$ with $tw(G_T) \geq tw(G)$; if $G$ is triangulated, then $E_t \equiv \emptyset$

1: $E_t := \emptyset$
2: for $v \in V$ do
3: \hspace{6mm} $label(v) := \emptyset$ \hfill // set of numbers listed in decreasing order
4: end for
5: $S := V$ \hfill // set of unordered vertices
6: for $i = |V|$ to 1 do
7: \hspace{6mm} if $i \equiv |V|$ then
8: \hspace{12mm} $u := v^*$
9: \hspace{6mm} else
10: \hspace{12mm} $u := \arg \max_{v \in S} label(v)$ \hfill // $u$ is a vertex with lexicographic largest label
11: \hspace{12mm} end if
12: \hspace{9mm} $\sigma(i) := u$ \hfill // $u$ to ordering at position $i$
13: \hspace{6mm} for $j, k \in \{i + 1, \ldots, |V|\}, j \neq k$ do
14: \hspace{9mm} if $\{\sigma(i), \sigma(j)\}, \{\sigma(i), \sigma(k)\} \in E \land \{\sigma(j), \sigma(k)\} \notin E \cup E_t$ then
15: \hspace{12mm} $E_t := E_t \cup \{\sigma(j), \sigma(k)\}$ \hfill // add edge to triangulation
16: \hspace{9mm} end if
17: \hspace{6mm} end for
18: $S := S \setminus \{u\}$
19: for $w \in N(u) \cap S$ do
20: \hspace{6mm} $label(w) := label(w) \cup \{i\}$ \hfill // add $i$ to the label of all unordered neighbors
21: end for
22: end for

Compared with the LEX$_P$ algorithm, the second version of the lexicographic breadth first search algorithm has the feature that a minimal triangulation of $G$ is returned (henceforth called LEX$_M$). A triangulation is called minimal if the removal of any edge $e$ of the fill-in $E_t$ results in a non-triangulated graph. To guarantee this property of the triangulation, not only the neighbors of an ordered vertex are labeled, but also all of the vertices that can be reached by a path of unordered
vertices with lower labels (cf. Algorithm 3). Due to this extension of the labeling procedure, the running time of the algorithm increases to \(O(nm')\), whereas the space requirements \(O(n + m')\) are equal to those of LEX_P. Running LEX_M once for all \(v^* \in V\) results in an overall running time of \(O(n^2m')\).

Algorithm 3 Lexicographic Breadth First Search, variant Minimal (LEX_M)

**Input:** connected graph \(G = (V, E)\), vertex \(v^* \in V\)

**Output:** minimal triangulated graph \(G_T = (V, E[t])\) with \(tw(G_T) \geq tw(G)\); if \(G\) is triangulated, then \(E_t = \emptyset\)

1. \(E_t := \emptyset\)
2. for \(v \in V\) do
3. \(\text{label}(v) := \emptyset\) // set of numbers listed in decreasing order
4. end for
5. \(S := V\) // set of unordered vertices
6. for \(i = |V| \) to 1 do
7. if \(i \equiv |V|\) then
8. \(u := v^*\)
9. else
10. \(u := \arg \max_{v \in S} \text{label}(v)\) // \(u\) is a vertex with lexicographic largest label
11. end if
12. \(E_t := E_t \cup \{\text{label}(i)\} \cup \text{label}(k)\) // add edge to triangulation
13. end for
14. \(S := S \setminus \{u\}\)
15. for \(w \in S\) path \(u = v_1, \ldots, v_{k+1} = w\) in \(G\)
16. \(\text{label}(w) := \text{label}(w) \cup \{i\}\) // add \(i\) to the label of all unordered neighbors
17. end for
18. end for

3.1.2 Maximum Cardinality Search

The Maximum Cardinality Search (MCS) recognition algorithm of Tarjan and Yannakakis [35] uses another criterion to select the vertices. Instead of selecting the vertex with the highest lexicographic label, they select the vertex which is adjacent to the highest number of ordered vertices. The algorithm does not guarantee a minimal triangulation and runs in \(O(n + m')\) time.

3.2 Minimum Separating Vertex Set Heuristic

The fourth heuristic described in this paper was developed in the context of solving frequency assignment problems with a tree decomposition approach [21]. It bases on another characterization of tree decompositions. Every tree decomposition can be transformed to a tree decomposition in which the vertex set associated to an internal node separates the graph into at least two components, i.e., the vertices associated with the node form a separating vertex set. The heuristic therefore searches for separating vertex sets. To find a good tree decomposition, we in fact search for minimum separating vertex sets. This explains the name of the heuristic. The choice for minimum separating vertex sets is motivated by the observation of Dirac [12] that a graph \(G\) is triangulated if and only if every minimal separating vertex set induces a complete subgraph of
Algorithm 4 Maximum Cardinality Search (MCS)

Input: connected graph $G = (V, E)$, vertex $v^* \in V$
Output: triangulated graph $G_T = (V, E \cup E_t)$ with $tw(G_T) \geq tw(G)$; if $G$ is triangulated, then $E_t \equiv \emptyset$

1: $E_t := \emptyset$
2: for $v \in V$ do
3: \hspace{1em} $counter(v) := 0$ \hfill // counts the number of ordered neighbors
4: end for
5: $S := V$ \hfill // set of unordered vertices
6: for $i = |V|$ to 1 do
7: \hspace{1em} $u := \arg \max_{v \in S} counter(v)$ \hfill // vertex with largest number of ordered neighbors
8: \hspace{1em} $\sigma(i) := u$ \hfill // add $u$ to ordering at position $i$
9: \hspace{1em} for $j, k \in \{i + 1, \ldots, |V|\}, j \neq k$ do
10: \hspace{2em} if $\{\sigma(i), \sigma(j)\}, \{\sigma(i), \sigma(k)\} \in E \land \{\sigma(j), \sigma(k)\} \notin E \cup E_t$ then
11: \hspace{3em} $E_t := E_t \cup \{\sigma(j), \sigma(k)\}$ \hfill // add edge to triangulation
12: \hspace{2em} end if
13: end for
14: $S := S \setminus \{u\}$
15: for $w \in N(u) \cap S$ do
16: \hspace{1em} $counter(w) := counter(w) + 1$ \hfill // increase counter for all unordered neighbors
17: end for
18: end for

So, for a triangulation $\bar{G}$ of $G$ with $tw(\bar{G}) = tw(G)$, the vertex sets associated with internal nodes of $T$ correspond to minimum vertex separating sets. We are looking for exactly those sets in the original graph $G$. Before describing the minimum separating vertex set ($MSVS$) heuristic, we briefly describe how a separating vertex set of minimum cardinality can be found in a graph.

3.2.1 Minimum Separating Vertex Sets in Graphs

To find a minimum separating vertex set $S$ in a graph $G = (V, E)$, we have to compute the minimum $st$-separating set for all non-adjacent $s, t \in V$.

Definition 3.6 An $st$-separating set of a graph $G = (V, E)$ is a set $S \subseteq V \setminus \{s, t\}$ with the property that any path from $s$ to $t$ passes through a vertex of $S$. The minimum separating vertex set of $G$ is given by the $st$-separating set $S$ with minimum cardinality over all combinations $st \notin E$.

The $st$-separating set with minimum cardinality can be found efficiently using Menger’s theorem (see also Ahuja, Magnanti, and Orlin [1]).

Theorem 3.7 (Menger [26]) Given a graph $G = (V, E)$ and two distinct non-adjacent vertices $s, t \in V$, the minimum number of vertices in an $st$-separating set is equal to the maximum number of vertex-disjoint paths connecting $s$ and $t$.

So, we have to calculate the maximum number of vertex-disjoint paths. This problem is solvable in polynomial time by standard network flow techniques. First, $G$ is transformed into a directed graph $D = (V, A)$, in which each edge $vw$ is replaced by two arcs $(v, w)$ and $(w, v)$. Next, we construct an auxiliary directed graph $D'$, with weights on the arcs, by

- replacing each vertex $v$ by two vertices $v'$ and $v''$,
Then, the minimum number of vertices in an $st$-separating set in $G$ is equal to the minimum weight of an $s'' - t'$ cut in $D'$. So, if we calculate the minimum $s'' - t'$ cut for every non-adjacent pair $s, t \in V$, we obtain the minimum separating vertex set. Note that since the graph $D'$ is a directed graph, we have to solve $O(n^2)$ minimum cut problems. In other words, we cannot use the algorithm of Gomory and Hu [18], which solves the all pairs minimum cut problem for undirected graphs by solving only $O(n)$ minimum cut problems.

3.2.2 MSVS Heuristic

The MSVS heuristic is in fact an improvement heuristic, i.e., it starts with a tree decomposition and tries to improve its width. The heuristic becomes a constructive heuristic if it starts with the trivial tree decomposition $(T, \mathcal{X})$: $|I| = 1$, $F = \emptyset$, and $X_i = V$.

We assume that the tree decomposition $(T, \mathcal{X})$ to start with is minimal in the sense that no $v \in V$ can be removed from a subset $X_i$, $i \in I$, without violating one of the tree decomposition conditions. Suppose there exists an unique node with maximum cardinality associated vertex set. Then, a replacement of this node by a set of nodes, all with smaller associated vertex sets, reduces the width of the tree decomposition. If the node is not unique, then we have to replace all nodes with maximum cardinality vertex set by nodes with smaller associated vertex sets to achieve an improvement in the width.

For each replacement of a node by new nodes, the new nodes have to be connected with the remaining parts of the tree in such a way that again all conditions of a tree decomposition are satisfied. Let $i \in I$ be a node with maximum cardinality vertex set. Suppose we would like to replace $i$ by a new node $i^*$ and $q$ nodes $I^*$ as displayed in Figure 2. So, all new nodes $p \in I^*$ are connected with $i^*$, some of the previous neighbors $j \in N(i)$ are connected with $i^*$, and all other neighbors are connected with $i^*$ via a (unique) node $p \in I^*$. Here, we allow that multiple neighbors $j \in N(i)$ are connected with one and the same new node $p$. But we also allow that not every new node $p \in I^*$ has to be connected with nodes $j \in N(i)$.

We require that the vertex set $X_{i^*}$ of node $i^*$ defines a minimum separating vertex set in a graph $H = (V(H), E(H))$ to be specified in the following. Each of the $q \geq 2$ nodes $p \in I^*$ corresponds with a component of $H[V(H) \setminus X_{i^*}]$. Let $Y_p$ be the vertex set of component $p$. Note that each $v \in X_{i^*}$ is connected with all components, since otherwise $X_{i^*}$ is not a minimum separating vertex set. As a consequence, for every $p \in I^*$, the associated vertex set has to consist of both the vertices in the component and the separating vertex set, $X_p := Y_p \cup X_{i^*}$ (otherwise the condition that for all $vw \in E$ there exist a $i \in I$ with $v, w \in X_i$ cannot be guaranteed afterwards). Now, the graph $H$ can be specified.

**Lemma 3.8** To guarantee that the newly constructed pair $(T, \mathcal{X})$ satisfies all conditions of tree decomposition, the vertex set and edge set of graph $H = (V(H), E(H))$ have to satisfy $X_i \subseteq V(H)$ and $\bigcup_{j \in N(i)} Q(X_i \cap X_j) \cup E_G[X_i] \subseteq E(H)$, respectively. These sets are also sufficient.
Proof. Since we assume that the algorithm starts with a minimal tree decomposition, no vertex $v \in X_i$ can be left out without violating the tree decomposition conditions. As a consequence, every vertex $v \in X_i$ has to be in the vertex set of $H$, otherwise the condition cannot be guaranteed afterwards anymore.

For the edge set we distinguish between pairs of vertices $v, w \in X_j$ for some $j \in N(i)$ and $v, w \notin X_j$ for all $j \in N(i)$. First, let there be a $j \in N(i)$ with $v, w \in X_j$. Let $p(j)$ be the new node $p \in I^*$ between $j$ and $i^*$ if such a node exists, otherwise let $p(j) = i^*$. By $v \in X_j$ and the definition of the subset $X_{p(j)}$, we have $v \in X_{p(j)}$. The same holds for $w$. Thus $v$ and $w$ have to be in the same component of the graph $H[X_i \setminus X_{i^*}]$, which only can be guaranteed by $vw \in E(H)$. Hence, $Q(X_i \cap X_j) \subset E(H)$ for all $j \in N(i)$.

Now, let $v, w \in X_i$ with $v, w \notin X_j$ for all $j \in N(i)$. So, $i$ is the unique node with $vw \in X_i$. If $vw \in E$, this node is required to guarantee the second condition of a tree decomposition. To guarantee that after the replacement of $i$, there still is a node with both $v$ and $w$ belonging to the associated vertex set, $v$ and $w$ should not end up in two different components. Thus, $vw \in E(H)$. If $vw \notin E$, then separation of $v$ and $w$ does not violate any condition for the resulting tree decomposition. Hence, it is not necessary to include non-adjacent $v$ and $w$ in $E(H)$.

Lemma 3.9 The cardinality of the associated vertex sets decreases, $\max\{|X_i|, \max_{p \in I^*} |X_p|\} < |X_i|$, if and only if $H$ is not a complete graph.

Proof. If $H$ is complete, then no minimum separating vertex set exists, and thus no reduction in the cardinality of the associated vertex sets can be achieved. If $H$ is non-complete, then there exist $v, w \in V(H)$ with $vw \notin E(H)$. The minimum $vw$-separating vertex set $S$ contains at least one, and at most $|V(H)| - 2$ vertices. Every component of $H[V(H) \setminus S]$ contains also at least one vertex. Hence, $\max_{p \in I^*} |X_p| < |X_i|$.

From Lemma 3.9, it directly follows that as long as $H$ is non-complete for all nodes $i \in I$ with
maximum cardinality associated vertex set, improvements on the treewidth are achieved. Note that by construction of $H$ and $Y_p$, the set $X_j$ has a non-empty intersection with at most one set $Y_p, p \in \{1, \ldots, q\}$. Let $v, w \in X_i \cap X_j$, then $vw \in Q(X_i \cap X_j) \subset E(H)$, which implies that $v$ and $w$ cannot be separated by $X_i$. So, either $v, w \in X_i$ or $v, w \in Y_p \cup X_i =: X_p$ for only one $j \in \{1, \ldots, m\}$. Therefore, we connect each neighbor $j \in N(i)$ with the node $p \in \{1, \ldots, q\}$ for which the intersection of $X_j$ and $Y_p$ is non-empty, or in case there is no such $p$ we connect $j$ with $i^*$ (cf. Figure 2). Algorithm 5 shows a pseudo-code implementation of the heuristic.

#### Algorithm 5 Minimum Separating Vertex Sets (MSVS)

**Input:** initial tree decomposition $(\mathcal{T}, \mathcal{X})$

**Output:** modified tree decomposition $(\tilde{T}, \tilde{X})$ with $\text{tw}(\tilde{T}, \tilde{X}) \leq \text{tw}(T, X)$

// if for a maximal subset $X_i$, $H$ is non-complete, the width can be improved
1: \textbf{while} $\exists i \in I : |X_i| = \max_{i \in I} |X_i|$ and $H = (X_i, E(H))$ non-complete
2: \quad $N_{old}(i) := N(i)$ \hfill // store old neighbors of $i$
3: \quad $F := F \setminus \{ij : j \in N_{old}(i)\}$ \hfill // disconnect $i$ from tree
4: \quad let $S \subseteq X_i$ be a minimum separating vertex set in $H$ (cf. Section 3.2.1)
5: \quad let $q$ be the number of components of $H[X_i \setminus S]$
6: \quad $n := |I|$ \hfill // current number of nodes
7: \quad $I := I \cup \{n + 1, \ldots, n + q\}$ \hfill // construct new nodes
8: \quad $F := F \cup \{ij : j = n + 1, \ldots, n + q\}$ \hfill // connect new nodes with $i$
9: \quad \textbf{for} $p = 1$ \textbf{to} $q$
10: \quad \quad \text{let } Y_p \subseteq X_i \text{ be the set of vertices in component } p \text{ of } H[X_i \setminus S]
11: \quad \quad $X_{\text{new}} := Y_p \cup S$ \hfill // define new vertex subsets
12: \quad \textbf{end for}
13: \quad $X_i := S$
14: \quad \textbf{for} $j \in N_{old}(i)$
15: \quad \quad \text{if } \exists p \in \{1, \ldots, q\} \text{ with } X_j \cap Y_p \neq \emptyset$
16: \quad \quad \quad $F := F \cup \{n + p, j\}$ \hfill // reconnect old neighbors
17: \quad \quad \text{else}
18: \quad \quad \quad $F := F \cup \{i, j\}$ \hfill // reconnect old neighbors
19: \quad \textbf{end if}
20: \textbf{end for}
21: \textbf{end while}

The width of the resulting tree decomposition approximates the minimal treewidth. Note that as long as the separating vertex sets $S$ form cliques in the original graph $G$, the algorithm operates in an exact way, since the optimal tree decomposition will contain a node for every clique that separates the graph in multiple components.

## 4 Lower Bounds

By applying the heuristics on a set of graphs, a reciprocative comparison of the algorithms can be carried out. The results, however, do not tell us anything about the quality of the best heuristic. For that purpose, we need to know the treewidth of the graphs, which requires to solve an $\mathcal{NP}$-hard problem. Alternatively, lower bounds on the treewidth can be used to benchmark the heuristics. In this section, two lower bounds that are used in this computational study are presented.
4.1 Maximum Clique Bound

It is well known that the treewidth of a graph does not increase under taking minors. So, the treewidth of a minor $H$ of $G$ is a lower bound for the treewidth of $G$, $tw(H) \leq tw(G)$. The minor of $G$ that provides the best lower bound is of course $G$ itself, or minors $H$ with $tw(H) = tw(G)$. For general minors, computing treewidth is however as difficult as computing the treewidth of $G$.

Since every graph induced by a clique $Q \subseteq V$ in $G$ is a minor of $G$, $tw(G) \geq |Q| - 1$. The best bound derived this way is for the maximum clique. Hence, the maximum clique number $\chi(G)$ of a graph $G$ minus one is a lower bound for the treewidth. Computing the maximum clique number is $\mathcal{NP}$-hard, in general. Nevertheless, several studies have shown that for not too large and not too dense graphs, the maximum clique number can be computed within reasonable time [4, 27].

4.2 Maximum Minimum Degree Bound

A trivial lower bound on the treewidth is given by the minimum degree of the graph: let $(T, X)$ be minimal tree decomposition with optimal width $k$, and let $i \in I$ be a leaf node with predecessor $j$. By minimality of the tree decomposition, there is a vertex $v \in X_i \setminus X_j$. Hence, $\delta(v) \leq k$.

The above implies that the minimum degree of a minor provides a lower bound as well. Since every (induced) subgraph of $G$ is a minor of $G$, the maximum of the minimum degree over all subgraphs, $\text{MMD}(G) = \min_{H \subseteq G} \delta(H)$, bounds the treewidth from below. This so-called maximum minimum degree bound can be computed in polynomial time.

**Lemma 4.1** The maximum minimum degree of $G$ is $\text{MMD}(G) = \max \{\delta(v), \text{MMD}(G[V \setminus \{v\}])\}$ for all $v \in V$ of minimum degree.

**Proof.** Let $v \in V$ be a vertex in $G$ of minimum degree. The minimum degree of all subgraphs $G'$ containing $v$ is at most the minimum degree of $G$. Thus, either $\text{MMD}(G) = \delta(v)$, or the maximum minimum degree bound equals the same bound for the graph without $v$. \[\square\]

This result directly provides an algorithm to compute the maximum minimum degree: remove sequentially minimum degree vertices of the graph and keep track of the maximum of these minimum degrees, see Algorithm 6.

Since the maximum clique of $G$ is a subgraph, $\text{MMD}(G) \geq \omega(G) - 1$. In fact, Szekeres and Wilf [34] proved that $\chi(G) - 1 \leq \text{MMD}(G)$. So, a bound that is at least as good as the maximum clique bound and the minimum coloring bound can be found in polynomial time. Using a Van Emde Boas-tree [36], the algorithm can be implemented in $O(n \log \log n + m)$ time.

Note that the maximum minimum degree $\text{MMD}(G)$ dualizes with another graph theoretic concept called $k$-degeneracy [25]. A graph is called $k$-degenerate if and only if every induced subgraph of $G$ has a vertex of degree at most $k$. The minimum $k$ for which $G$ is $k$-degenerate equals $\text{MMD}(G)$. 

13
Algorithm 6 Maximum Minimum Degree (MMD)

Input: connected graph $G = (V, E)$
Output: $MMD \leq tw(G)$

1: for $v \in V$ do
2: $n(v) := N(v)$
3: end for
4: $S := V$, $MMD := 0$. // $S$ defines induced subgraph
5: while $S \neq \emptyset$ do
6: $u := \arg \min_{v \in S} |n(v)|$ // vertex with minimum degree in subgraph
7: if $|n(u)| > MMD$ then
8: $MMD := |n(u)|$ // update maximum
9: end if
10: for $v \in n(u)$ do
11: $n(v) := n(v) \setminus \{u\}$
12: end for
13: $S := S \setminus \{u\}$ // reduce subset
14: end while

5 Computational Analysis

The algorithms described in the previous sections are tested on graphs obtained from applications like vertex coloring, frequency assignment, and probabilistic networks. All instances are well known in the application area, and are often used for benchmarking newly proposed methodologies. All algorithms have been implemented in C++ on a Linux PC with Pentium III 800 MHz processor. Consecutively, we discuss the results for instances from frequency assignment, probabilistic networks, and vertex coloring.

5.1 Frequency Assignment

The frequency assignment problem deals with the allocation of frequencies to the transmitters of a wireless communication network. Interference between signals transmitted on (almost) the same frequency restricts the assignment of the frequencies. Moreover, certain frequencies can be locally blocked at a transmitter, i.e., it is prohibited to assign the frequency to the transmitter. Depending on the application at hand, we want to minimize the number of used frequencies, the span of the used frequencies, the blocking probability of the network, or the total interference in the network. For more information on frequency assignment, we refer to [13, 21] or to FAP web [14].

In the context of the EUCLID CALMA project [9], instances for a variety of frequency assignment problems have been made available. We restrict ourselves to the instances where the objective is to minimize the overall interference in an assignment. In [21], it was shown that application of a tree decomposition approach yield to good lower bounds and optimal solutions for these instances. In total 11 instances are available with a total of 9 different graphs. Several preprocessing ideas (for frequency assignment) led to an additional set of 7 instances (3 instances were solved by preprocessing, for 1 the graph did not change). We tested the diverse algorithms on these 16 instances.

Table 1 shows statistics for the performance of the algorithms on these instances. The maximum clique size ($MC$) is computed with a standard tree search (branch-and-bound) algorithm. Several remarks can be made. The table shows that the $MMD$ lower bound is indeed better than the clique
| instance   | $|V|$ | $|E|$ | MC | MMD | MCS | LEX_P | LEX_P | MVS |
|------------|-----|-----|----|-----|-----|-------|-------|-----|
|            | min. | avg. | max. | min. | avg. | max. | min. | avg. | max. |
| CELAR06    | 100  | 350  | 10  | 11  | 11.27 | 14 | 11  | 13.69 | 14 | 11  | 13.61 | 14 | 11 |
| CELAR07    | 200  | 817  | 10  | 11  | 22.25 | 26 | 19  | 23.64 | 32 | 17  | 20.64 | 29 | 17 |
| CELAR08    | 458  | 1655 | 10  | 11  | 24.36 | 30 | 21  | 29.16 | 39 | 20  | 27.71 | 38 | 18 |
| CELAR09    | 340  | 1130 | 10  | 11  | 25.09 | 31 | 21  | 29.39 | 39 | 17  | 27.82 | 38 | 18 |
| GRA0005    | 100  | 416  | 8   | 8   | 35.60 | 36 | 29  | 36.60 | 47 | 28  | 35.13 | 43 | 27 |
| GRA0006    | 200  | 843  | 8   | 8   | 61.94 | 74 | 61  | 72.06 | 83 | 59  | 70.93 | 85 | 56 |
| GRA0011    | 340  | 1425 | 7   | 7   | 115.43 | 127 | 106 | 121.88 | 143 | 106 | 120.80 | 139 | 102 |
| GRA0012    | 340  | 1255 | 5   | 5   | 162.31 | 121 | 99  | 114.77 | 128 | 99  | 113.73 | 128 | 93 |
| GRA0013    | 458  | 1877 | 6   | 6   | 148.89 | 162 | 147 | 161.22 | 178 | 147 | 160.55 | 176 | 133 |
| CELAR06PP  | 82   | 327  | 10  | 10  | 11.33 | 14 | 11  | 13.65 | 16 | 11  | 13.52 | 16 | 11 |
| CELAR07PP  | 162  | 764  | 10  | 11  | 21.60 | 26 | 19  | 25.65 | 30 | 17  | 25.62 | 29 | 17 |
| CELAR08PP  | 365  | 1539 | 10  | 11  | 25.04 | 29 | 21  | 32.28 | 38 | 19  | 28.58 | 38 | 18 |
| CELAR09PP  | 67   | 165  | 7   | 7   | 7.00  | 7  | 7   | 7.00  | 7  | 7   | 7.00  | 7  | 7 |
| GRA006PP   | 119  | 348  | 5   | 5   | 23.85 | 30 | 21  | 26.97 | 33 | 20  | 25.83 | 31 | 18 |
| GRA012PP   | 61   | 123  | 4   | 4   | 6.85  | 9  | 5   | 6.48  | 8  | 4   | 5.48  | 7  | 4 |
| GRA013PP   | 456  | 1874 | 6   | 6   | 133  | 163 | 147 | 161.20 | 178 | 147 | 160.52 | 176 | 133 |

Table 1: Statistics for CALMA frequency assignment problems
Table 2: Computation times for CALMA frequency assignment problems

<table>
<thead>
<tr>
<th>instance</th>
<th>CPU time (in seconds)</th>
<th>MC</th>
<th>MCS</th>
<th>LEX_P</th>
<th>LEX_M</th>
<th>MSVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>celar06</td>
<td>0.11</td>
<td>0.07</td>
<td>0.09</td>
<td>0.74</td>
<td>9.30</td>
<td></td>
</tr>
<tr>
<td>celar07</td>
<td>0.44</td>
<td>0.48</td>
<td>0.57</td>
<td>5.90</td>
<td>101.62</td>
<td></td>
</tr>
<tr>
<td>celar08</td>
<td>1.49</td>
<td>5.02</td>
<td>6.08</td>
<td>46.24</td>
<td>603.19</td>
<td></td>
</tr>
<tr>
<td>celar09</td>
<td>0.74</td>
<td>2.55</td>
<td>2.39</td>
<td>28.17</td>
<td>580.30</td>
<td></td>
</tr>
<tr>
<td>Graph05</td>
<td>0.06</td>
<td>0.68</td>
<td>0.84</td>
<td>1.98</td>
<td>34.98</td>
<td></td>
</tr>
<tr>
<td>Graph06</td>
<td>0.20</td>
<td>15.42</td>
<td>20.08</td>
<td>28.88</td>
<td>323.19</td>
<td></td>
</tr>
<tr>
<td>Graph11</td>
<td>0.33</td>
<td>247.30</td>
<td>262.45</td>
<td>292.71</td>
<td>2290.80</td>
<td></td>
</tr>
<tr>
<td>Graph12</td>
<td>0.18</td>
<td>199.57</td>
<td>222.60</td>
<td>246.63</td>
<td>2124.58</td>
<td></td>
</tr>
<tr>
<td>Graph13</td>
<td>0.50</td>
<td>994.37</td>
<td>1118.32</td>
<td>1170.50</td>
<td>7507.91</td>
<td></td>
</tr>
<tr>
<td>celar06pp</td>
<td>0.09</td>
<td>0.04</td>
<td>0.06</td>
<td>0.52</td>
<td>5.67</td>
<td></td>
</tr>
<tr>
<td>celar07pp</td>
<td>0.39</td>
<td>0.36</td>
<td>0.46</td>
<td>4.08</td>
<td>58.65</td>
<td></td>
</tr>
<tr>
<td>celar08pp</td>
<td>1.24</td>
<td>2.51</td>
<td>4.26</td>
<td>28.12</td>
<td>412.07</td>
<td></td>
</tr>
<tr>
<td>celar09pp</td>
<td>0.00</td>
<td>0.03</td>
<td>0.03</td>
<td>0.10</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>Graph06pp</td>
<td>0.01</td>
<td>0.48</td>
<td>0.54</td>
<td>2.26</td>
<td>22.81</td>
<td></td>
</tr>
<tr>
<td>Graph12pp</td>
<td>0.00</td>
<td>0.02</td>
<td>0.01</td>
<td>0.16</td>
<td>2.04</td>
<td></td>
</tr>
<tr>
<td>Graph13pp</td>
<td>0.46</td>
<td>970.08</td>
<td>1092.34</td>
<td>1111.65</td>
<td>7602.44</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Computation times for CALMA frequency assignment problems

bound for some instances. The MSVS heuristic produces the best result for 14 out of 16 instances, MCS 7 times, LEX\_P 3 times, and LEX\_M 7 times. In Table 1, not only the upper bounds computed by the MCS, LEX\_P, and LEX\_M heuristics are reported, but also the average and maximum bound computed in the runs with different vertex $v^*$ (cf. Algorithms 2, 3, and 4). These values show that the bounds computed by LEX\_P and LEX\_M are spread out over a wider range of values than the bounds computed by MCS. For CELAR08, we depict this observation in Figure 3. The range

![Histogram](image1.png)
![Cumulative](image2.png)

Figure 3: Statistics triangulation heuristics for CELAR08

... of the bounds is not only smaller than those for the other heuristics, but also smaller values are more often generated. Next, note that the treewidth for two instances (CELAR09PP and GRAPH12PP) is known since the best lower and upper bound are equal. For CELAR09PP, all algorithms produce the same bound, even for arbitrary starting vertex. On the other hand, the table shows (very) large gaps between lower and upper bounds for the larger GRAPH instances.
This remarkable difference between larger CELAR and GRAPH instances seems to be caused by the origin of the instances: the CELAR instances are taken from real-life whereas the GRAPH instances are computer generated. Finally, we would like to point out that the figures show that the frequency assignment preprocessing is very effective, also for reducing the treewidth of the graphs: the upper bound for the treewidth reduces from 93 to 4 for GRAPH12. Also for other instances, Table 1 shows substantial reductions.

Table 2 shows the computation times of the heuristics as well as the MC bound. The MMD bound is computed in a fraction of a second for all instances. The computation times of the heuristics MCS, LEX_P, and LEX_M do not differ that much. The MSVS heuristic consumes much more time than the other ones.

5.2 Probabilistic networks

We continue our computational study with 8 real-life probabilistic networks from the field of medicine. Table 3 shows the origin of the instances and the size of the network. The most efficient way to compute the inference in a probabilistic network is by the use of the junction-tree propagation algorithm of Lauritzen and Spiegelhalter [24]. This algorithm uses a tree decomposition of the network’s moralized graph. The moralization of a network (or directed graph) $D = (V, A)$ is the undirected graph $G = (V, E)$ obtained from $D$ by adding edges between every pair of non-adjacent vertices that have a common successor, and then dropping arc directions. The size of the edge set $E$ is also reported in Table 3. After the application of pre-processing techniques for computing the treewidth [8], an additional four instances to conduct our heuristics on are available. The size of these four instances is reported in Table 3 as well. After [8], henceforth we refer to these instances as INSTANCENAME_{3,4}. Table 4 shows the results of the heuristics and lower bounds for these instances. The computation times for both lower bounds are negligible. For three instances lower and upper bounds are equal and so the treewidth is reported. For the other instances the gap ranges from 1 to 5. Further the same observations can be made as for the frequency assignment instances. The computation time of MSVS is substantially larger than for the other heuristics, but at the same time the best result for all instances is obtained with this heuristic. Especially, for ICEA(_3) and MUNIN, MSVS outperforms the other heuristics.

Although not reported in Table 4, like for the frequency assignment instances, the range of

| network     | origin                     | \(|V|\) | \(|A|\) | \(|E|\) | \(|V|\) | \(|E|\) |
|-------------|----------------------------|-------|-------|-------|-------|-------|
| WILSON      | Wilson's liver disease     | 21    | 23    | 27    | -     | -     |
| ALARM       | anesthesia monitoring      | 37    | 44    | 62    | -     | -     |
| VSD         | prognosis of ventricular septal defect in infants | 38 | 51 | 61 | - | - |
| OESOPHAGUS  | staging of oesophageal cancer | 42 | 57 | 68 | - | - |
| OESOPHAGUS+ | prediction of response to treatment of oesophageal cancer | 67 | 117 | 194 | 26 | 121 |
| MUNIN       | interpretation of electromyographic findings | 1003 | 1244 | 1662 | 175 | 471 |
| ICEA        | prediction of coverage by antibiotics of pathogens causing pneumonia | 89 | 154 | 215 | 59 | 170 |
| PATHFINDER  | diagnosis of lymphatic disease | 109 | 192 | 211 | 14 | 49 |

Table 3: Probabilistic network characteristics
bounds generated by the triangulation heuristics is smaller for \texttt{MCS} compared with \texttt{LEX.P} and \texttt{LEX.M}. This in particular holds for the \texttt{munin} network. Where the bound ranges from 16 to 56 for \texttt{LEX.P} and \texttt{LEX.M}, \texttt{MCS} reports bounds from 10 to 26 (see [8] for details). Hence, it is important not to restrict the computations to a single run of these heuristics.

5.3 Vertex Coloring

Except for the frequency assignment instances and the probabilistic networks, where tree decomposition is indeed reported in the literature, we decided to test the algorithms also on another class of publicly available graphs. We have chosen for the well-known DIMACS benchmarks for vertex coloring [11]. Advantage of these instances is, that for many of them the chromatic number is known. In Section 4.2, we have seen that this value minus one is a lower bound for treewidth as well, and outperforms the clique bound. Details on the 62 instances and results can be found in Table 5 in Appendix A. In Figure 4 and 5, the results are reported in a different, more descriptive way. Figure 4 deals with the lower bounds and their relation to the upper bounds. Figure 5 depicts the quality of the upper bounds.

In Figure 4(a), we compare the chromatic number (provided the value is known) with the \texttt{MMD} bound. We counted the number of times the values are equal, differ one, differ two, etcetera. This results in the histogram of Figure 4(a). Note that the intervals are not equally sized. The diagram shows that in 33 of the 49 cases, the \texttt{MMD} bound is indeed better. The improvement of the lower bound is incidentally over 30.

In Figure 4(b), we compare the \texttt{MMD} bound with the best upper bound achieved by the various heuristics. Again, we count the number of times the difference between best upper bound and \texttt{MMD} lower bound is zero, one, etcetera. In addition, the cumulative percentage below a difference is displayed. Figure 4(b) shows that for 7 (out of 62) instances the optimal value is achieved, whereas for 25 instances (or 37%) the gap is at most 5. On the other side, the gap is extremely large (> 100) for another 22%, with a maximum gap of 327. For many instances it seems to be difficult to find a good lower bound, or a good upper bound, or both.

\footnote{The \texttt{flat} instances as well as the larger \texttt{dsjc} and \texttt{dsjr} instances are left out, since they have a very high density and thus are very time consuming.}
Figures 5(a)–5(d) show the difference between the best upper bound achieved and the heuristics $MCS$, $LEX_P$, $LEX_M$, and $MSVS$, respectively, in a similar way as the difference between best upper bound and $MMD$ bound in Figure 4(b). To simplify a comparison between the histograms, the same intervals and the same scale are used in all diagrams. The number of times $LEX_M$ and $MSVS$ supply the best upper bound attracts our attention at first sight. Respectively, 36 and 33 times the best bound is reported by these heuristics. An even more impressive result is obtained by joining the results of $LEX_M$ and $MSVS$. Together, they are responsible for 59 (out of 62) best upper bounds. The bounds reported by the other heuristics $MCS$ and $LEX_P$ are more spread out.

The maximum difference between best upper bound and the bound by a particular heuristic is smallest for $MSVS$ with 25, followed by $MCS$ with 47, $LEX_M$ with 193, and $LEX_P$ with 198. The maximum difference between best lower bound and the heuristics is for respectively $MCS$, $LEX_P$, $LEX_M$, and $MSVS$, 344, 330, 330, and 327.

Concerning the computation times, again the $MSVS$ heuristic is outperformed by all other heuristics. On average it took more than 20,000 seconds (> 5.5 hours) to compute the $MSVS$ bound, whereas the $MCS$, $LEX_P$, and $LEX_M$ bound are computed in respectively 1,768, 1,974, and 2,167 seconds (30–36 minutes). The computation of the $MMD$ lower bound took on average a (in comparison with the other times) neglectable 1.17 seconds.

6 Concluding Remarks

In recent years, alternatives for integer programming to solve combinatorial optimization problems attract more and more attention. One of these methods is based on a tree decomposition of the graph associated with the problem. In this approach, the treewidth of a graph has to be computed, independently of the application at hand. Studies to the solvability of treewidth have been rarely carried out. In this paper, a computational study to the quality of heuristics and lower bounds for treewidth has been undertaken. A total of four heuristics and two lower
Figure 5: Statistical information heuristics for DIMACS vertex coloring instances

bounds have been compared on graphs from different applications. Three of the heuristics are based on triangulation recognition algorithms, the other one on minimal separating vertex sets. The lower bounds compared are based on maximum cliques and the maximum minimum degree in a subgraph. For the latter one, it is easy to show that it is never worse than the maximum clique bound.

The bounds are tested on graphs from three different applications; frequency assignment, probabilistic networks, and vertex coloring. The best upper bounds were achieved by the MSVS heuristic. A good alternative to this time consuming bound is the LEX$_M$ bound that performs almost as good as MSVS but has the advantage to be computable in substantially less time. The MMD algorithm provides a reasonable lower bound very fast.

For all applications, the treewidth of some graphs could be identified by combining the best lower and upper bound. On the other hand, for several graphs from frequency assignment and vertex

---

Figure 5: Statistical information heuristics for DIMACS vertex coloring instances

(a) MCS - best upper bound

(b) LEX$_P$ - best upper bound

(c) LEX$_M$ - best upper bound

(d) MSVS - best upper bound
coloring the gap between best lower and upper bound is far too large to draw any conclusion about the treewidth. It is needless to say that for these instances better lower and upper bounds are necessary.

To determine the treewidth for graphs with a relatively small gap between lower and upper bound, also the development and implementation of exact solution procedures can be effective. Ideas in this direction are currently under development. Moreover, for all graphs pre-processing can be a powerful tool to simultaneously reduce the size of the graph and obtain lower bounds. In [8], a first attempt in this direction has been undertaken for the graphs from probabilistic networks (cf. Section 5.2). In a follow-up paper more ideas for graph reduction will be presented. Therefore, this paper should be viewed as one of the first (together with [8]) in a series of the authors and other collaborators to study the practical setting of treewidth and the tractability of the tree decomposition approach for combinatorial optimization problems.

References


Table 5: Computational results for DIMACS vertex coloring instances

<table>
<thead>
<tr>
<th>instance</th>
<th></th>
<th>lower bounds</th>
<th>upper bounds</th>
<th>CPU time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Results for vertex coloring instances</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A Results for vertex coloring instances</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>