A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth

Emgad H. Bachoore

Hans L. Bodlaender

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2006-012

www.cs.uu.nl

ISSN: 0924-3275

A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth *

Emgad H. Bachoore and Hans L. Bodlaender

Institute of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract. In this paper, a branch and bound algorithm for computing the treewidth of a graph is presented. The method incorporates extensions of existing results, and uses new pruning and reduction rules, based upon properties of the adopted branching strategy. We discuss how the algorithm can not only be used to obtain exact bounds for the treewidth, but also to obtain upper and/or lower bounds. Computational results of the algorithm are presented.

1 Introduction

The notions treewidth, pathwidth and branchwidth have received a growing interest in recent years not only because of their theoretical significance in (algorithmic) graph theory, but also because many problems that are intractable on graphs, including a large number of well-known NP hard problems, have been shown to be polynomial-time and even linear-time solvable on graphs that are given together with a tree decomposition of width at most some constant k. (See e.g., [2–4, 11, 13].)

Arnborg et al. [5] proved that computing the treewidth of a graph is an NP-hard problem. So, in recent years, many different algorithms have been designed to compute the treewidth exactly, approximately, or do preprocessing. Many of these algorithms have been designed for special classes of graphs, but in this paper, we will focus on algorithms that work on general undirected graphs.

A number of these algorithms were approximation algorithms. These are polynomial time approximation algorithms for treewidth, that have approximation ratio $O(\log n)$ [9] or $O(\log k)$ [1], where k is the actual treewidth of the input graph. Other algorithms [1] have a constant approximation ratio, but their running time is exponential in the treewidth. On the other hand there were several proposals for heuristic algorithms for upper and lower bound on the treewidth. Some of these algorithms are based on the concept of graph triangulation. These are Maximum Cardinality Search, Lexicographic Breadth First search algorithms, Minimum Degree, Minimum Fill-in, MFEO1, MFEO2, RATIO1 and RATIO2. Other heuristics are based on other ideas, e.g., the Minimum Separating Vertex Set algorithm. Some examples of lower bound methods are Maximum Minimum Degree, MMD+, D-LB, and contraction and treewidth lower bounds algorithms. See e.g. [6, 9, 11, 14, 16, 18].

Using the above techniques may help to find a close value for the treewidth of a graph, but in many cases, not the exact one. Therefore, there is a need for algorithms that produce the

* This work has been supported by the Netherlands Organization for Scientific Research NWO (project TACO: 'Treewidth And Combinatorial Optimization').

exact treewidth, at least for some graphs. Bodlaender [10] invented a linear-time algorithm to decide whether the treewidth of a graph is at most a constant k. Unfortunately, this algorithm is exponential in a polynomial in k and hence appears to be impractical even for $k = 4$, see [20].

Well known techniques that we can use to design an algorithm for finding the exact treewidth are for example Branch-and-Bound and Integer Linear Programming. In this paper we introduce an algorithm for finding the treewidth of a graph using a Branch-and-Bound technique. The results of this study may suggest further research into the effects of using branch and bound technique to find the treewidth of graphs.

In 2003, Gogate and Dechter [17] reported on work on a branch and bound algorithm on treewidth. Independently, part of the work reported in the current paper had been done before publication of [17]. There are some significant differences in details and results between [17] and this paper. We report here also on these differences.

A generalization of the algorithm can be used to compute the weighted treewidth of weighted graphs. In this paper, we focus on the unweighted case. Furthermore, more than one variant of the algorithm has been developed and implemented to test the effect of using different pruning rules on the efficiency of the algorithm. Finally, it is interesting to note that, for some large graphs, it is infeasible to find the exact treewidth of the graph in a reasonable time by using branch and bound. Therefore, we developed our algorithm such that it yields better lower and/or upper bounds on the treewidth in such cases.

2 Definitions and preliminary results

Throughout this paper $G = (V, E)$ denotes a finite, simple and undirected graph, where V is the set of vertices of the graph and E the set of edges of the graph. A subgraph of a graph $G(V, E)$, induced by a set of vertices $W \subseteq V$, is denoted by $G[W] = (W, \{\{v, w\} \in E | v, w \in W\})$. A graph H is a ***minor*** of graph G, if H can be obtained from G by zero or more vertex deletions, edge deletions, and edge contractions. ***Edge contraction*** is 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 neighbors of a vertex v is denoted $N(v) = \{w \in V | \{v, w\} \in E\}$. The neighbors of v plus v itself is denoted $N[v] = N(v) \cup \{v\}$. In the same manner we define $N^0[v] = \{v\}$, $N^{i+1}[v] = N[N^i[v]]$, $N^{i+1}(v) = N^{i+1}[v] \setminus N^i[v]$. We can extend the above definition to a set of vertices instead of one vertex. Suppose that S is a set of vertices, then $N^0[S] = S,\ N^{i+1}[S] = N[N^i[S]],\ N^{i+1}(S) = N^{i+1}[S] \setminus N^i[S],\ N[S] = \bigcup_{v \in S} N[v],\ N(S) = N[S] \setminus S,\ i \in \mathcal{N}$. A vertex v in G is called ***simplicial***, if all its neighbors $N(v)$ forms a clique in G. A vertex v in G is called ***almost simplicial***, if its neighbors except one form a clique in G, i.e., if v has a neighbor w such that $N(v) - \{w\}$ is a clique. A graph G is called ***triangulated*** (or: chordal) if every cycle of length four of more possesses a chord. A ***chord*** is an edge between two non consecutive vertices of the cycle. A graph $H = (V, F)$ is a ***triangulation of graph*** $G = (V, E)$, if G is a subgraph of H and H is a triangulated graph.

Definition 1. *Let x be a vertex in a graph $G = (V, E)$. The* fill-in *of x in a graph G, is the number of edges that must be added between the neighbors x, $N(x)$, to make x simplicial, i.e.,*

$$\text{fill-in}(x) = |\{\{v, w\} | v, w \in N(x), \{v, w\} \notin E\}|$$

The fill-in-excluding-one *neighbor of x in a graph G is the minimum number of edges that must be added between vertices in $N(x)$ (minus one vertex), such that x is almost simplicial, i.e.,*

$$\text{fill-in-excl-one}(x) = min_{z \in N(x)} |\{\{v, w\} | v, w \in N(x) - \{z\}, \{v, w\} \notin E\}|$$

Definition 2. *A **tree decomposition** of the graph $G = (V, E)$ is a pair (X, T) in which $T = (I, F)$ a tree, and $X = \{X_i | i \in I\}$ a collection of subsets of V, one for each node of T, such that*

- $\bigcup_{i \in I} X_i = V$.
- *For all $\{u, v\} \in E$, there exists an $i \in I$ with $u, v \in X_i$.*
- *For all $i, j, k \in I$: if j is on path from i to k in T, then $X_i \cap X_k \subseteq X_j$.*

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

3 The Branch and Bound Algorithm for Treewidth BB-tw

The main elements of the branch and bound algorithm for finding the treewidth of a graph, BB-tw, are: The space of all feasible solutions, the upper and lower bounds on the treewidth, and the rules for pruning the feasible solutions that do not contain optimal solution. These are described in Sections 3.1 and 3.2. More details are discussed in Sections 3.3 – 3.6.

3.1 Problem description

Several algorithms for determining or approximating the treewidth of a graph are based on triangulations formed from vertex orderings.

Definition 3. *A **linear ordering** of a graph $G = (V, E)$ is a bijection $f : V \to \{1, 2, \cdots, |V|\}$, also denoted as $[f(1), \cdots, f(|V|)]$. A linear ordering of the vertices of a graph G, $\sigma = [v_1, \cdots, v_n]$ is called a **perfect elimination order (p.e.o.)** of G, if for every $1 \leq i \leq n$, v_i is a simplicial vertex in $G[v_i, \cdots, v_n]$, i.e., the higher numbered neighbors of v_i form a clique.*

Lemma 1. *(See [11].) A graph G is triangulated, if and only if G has a perfect elimination order $(p.e.o)$.*

Definition 4. *Let v be a vertex in a graph G. **Eliminating** a vertex v from a graph G, $eliminate(v_G)$, is the procedure of adding an edge between every pair of non-adjacent neighbors of v in G, and then removing v and its incident edges from G. We call the graph G' obtained from eliminating a vertex v from a graph G, a **temporary graph** of G. I.e., $G'(W, F) = G(V - \{v\}, E - \{\{v, w\} | \{v, w\} \in E, \{v, w\} \notin F\})$.*

Lemma 2. *Let $G' = (V', E')$ be a minor of graph $G = (V, E)$. Then $treewidth(G') \leq treewidth(G)$.*

Let $\sigma = [v_1, \cdots, v_n]$ be a linear ordering of $G = (V, E), n = |V|$. Construct a graph H as follows: Set $H = G$; for $i = 1$ to n, add to H an edge between each pair of non-adjacent higher numbered neighbors (in H) of v_i. H is the graph obtained from G by the fill-in procedure with respect to σ; σ is a perfect elimination scheme for H, hence H is chordal. The following theorem is well known.

Theorem 1. *Let $G = (V, E)$ be a graph, $k = |V|$. The following statements are equivalent.*

- *The treewidth of G is at most k.*
- *G has a triangulation with maximum clique size at most $k + 1$.*
- *There is a linear ordering σ, such that the graph H obtained from G by the* fill-in *procedure w.r.t. σ has maximum clique size at most $k + 1$, or, equivalently, fulfils that each $v \in V$ has at most k higher numbered neighbors (in H, w.r.t. σ).*

Lemma 3. *Let H be a triangulation obtained from applying the* fill-in procedure *on a graph G due to an ordering σ of the vertices of G. Then, the treewidth of G is at least the size of the maximum clique in H minus* 1.

Thus, we take as the space of all feasible solutions for the computation of the treewidth of G the set of all possible linear orderings of the vertices of G. We represent these by a *search tree of all possible solutions* T_r by taking a root r, and having for each node a child for each possible choice of the next vertex in the elimination ordering. Thus, a node in the search tree represents a fixed initial part of the linear ordering. From this point, we look for the best ordering of the vertices not in the initial part. This is equivalent to looking for a linear ordering for the graph, obtained by eliminating all vertices in this initial part. Figure 1 shows an example for the space of all feasible solution for a graph consisting of 4 vertices.

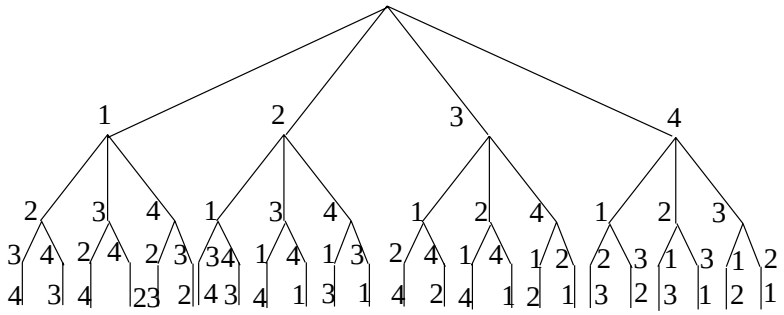

Fig. 1.

Initializing the space of all feasible solutions One of the interesting points in the initialization phase of the algorithm is, how we should initialize or define the search tree of all feasible solutions. Suppose that the given graph is as in Figure 2 and the initial value for the upper bound on the treewidth of this graph equals 9. Furthermore, suppose we use the above method for building

the search tree, and we use only one pruning rule in the branch and bound algorithm, that is, if the degree of the current elimination vertex in the temporary graph is greater than the value of the reported upper bound, then we skip the search operation from the current node in the search tree to its next right sibling. Hence, the first elimination ordering that the algorithm will check is [1, . . . ,27]. This means that we have to visit more nodes in the space of all feasible solutions than that we have to visit if we have build this space in the following manner. Instead of arranging the available nodes in the tree in ascending order, due to their labels, from left to right in each level of each subtree of the search tree, we arrange these nodes in each level due to their sequence in the perfect elimination ordering for finding the best upper bound on the treewidth. Therefore, the initial values of the first elimination ordering for implementing BB-tw algorithm on the graph in Figure 2 will be [10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,2,3,4,5,6,7,8,9,1] when we use for example the elimination ordering obtained from finding the upper bound on the treewidth by using the Minimum Fill-in Heuristic. Thus, the number of visited nodes in the tree becomes smaller by using this method for building the search tree. As a result, the running time of the branch and bound algorithm for finding the treewidth of a graph becomes also smaller.

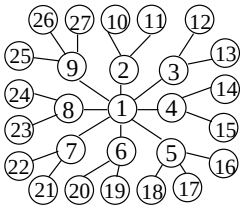

Fig. 2.

3.2 Pruning Rules

In a branch and bound algorithm, we do not search the entire search tree, but speed up the computation by omitting several parts of the search tree, of which we have established that we do not need these parts for finding the optimal solution. We consider a number of pruning rules.

Pruning Rule 1: The upper bound equals to the lower bound Before starting the branch-and-bound search, we compute the upper bounds on the treewidth of the given graph by using the heuristics introduced in [6], and the lower bounds on the treewidth by using two heuristics, namely, the Degeneracy heuristic and Ramachandramurthi γ parameter of the graph [19]. We report the best upper and lower bounds obtained from these heuristics. Then, we check whether the best lower bound equals the best upper bound, and if so, then this value is returned as the treewidth of the given graph; otherwise we start the branch and bound. This comparison is also done when a new (better) upper or lower bound for the treewidth is found during the branch-and-bound search.

Moreover, in nodes in the search tree, we compute the degeneracy lower bound, and prune when this value is not smaller than the best known upper bound.

Pruning Rule 2: Number of vertices in the temporary graph Let $G' = (V', E')$ be a temporary graph of a graph $G = (V, E)$, i.e., the graph obtained by eliminating a set of vertices $X \subseteq V$ from a given graph G. Let max be the maximum degree of all vertices in the initial part σ^0 of the elimination ordering, as represented by the node the search currently is on. Let α be the maximum degree of the vertices in σ^0 at the moment of their elimination. As any linear ordering that starts with σ^0 will yield a treewidth bound that is at most $\beta = max(\alpha, |V'| - 1)$, we prune when $\beta < ub$ with ub the currently best known upper bound, and set ub to β.

Pruning Rule 2 (PR2)
let $r = (G = (V, E), ub, lb)$ be the give instance problem,
$G' = (W, F)$ be the graph obtained from eliminating a set of vertices $X = \{x | x \in V, x \notin W\}$ and their incident edges from the graph G,
max be the maximum degree of all vertices of the set X before they have been exactly eliminated from G, $max_{x \in X}(degree(x))$,
y be the last vertex that has been eliminated from graph G,
ub be the best upper bound reported for the treewidth of the graph G,
if $(|V(G')| \leq max)$
$ub \leftarrow max$;
omit the subtree rooted by the parent of the vertex y from the space of all feasible solutions;

Lemma 4. *Let $G' = (W, F)$ be the graph obtained form eliminating vertex y from graph $G = (V, E)$, $max = degree(y)$ before it has been eliminated from G. If $|W| < max$, then the treewidth of G is at most max.*

Proof. Eliminating the vertices of the graph G' in any order and reporting the maximum of these neighborhood seen during the process will not cause the treewidth of the graph to become greater than max since $max \geq |W|$. □

Pruning Rule 3: The degree of the eliminated vertex If the degree of a vertex v in the temporary graph is larger than or equal to the best upper bound known for the treewidth of the input graph, then we know that eliminating v will give an elimination ordering whose width is at least the degree of v, hence will not yield an improvement to the upper bound. Thus, the branch which selects v as next vertex to be eliminated can be pruned at this point.

Lemma 5. *Let $H = (W, F)$ be a triangulation graph of a graph $G = (V, E)$ and ub be an upper bound on the treewidth of the graph G. If $\exists w \in W,\ degree(w) > ub$, and w is simplicial or w is almost simplicial and $degree(w)$ is at most the treewidth of G, then $tw(G) < tw(H)$.*

Pruning Rule 3 (PR3)
let $x = (G = (V, E), ub, lb)$ be the give instance problem,
v be the current vertex we have to eliminate from a graph G,
ub, lb be the best upper and lower bounds on treewidth of G,
we have reported up to now;
if $(degree(v) \geq ub)$
omit the subtree rooted by v from the search tree;
let the next candidate vertex we have to eliminate be the next sibling
of the vertex v in the search tree.

Pruning Rule 4: Equivalent elimination orderings Gogate and Dechter [17] observed that in some cases, swapping two successive vertices does not affect the width of a linear ordering. We use a simpler but equivalent test. Suppose v and w are successive vertices in a linear ordering σ, and v and w are not adjacent or v and w are adjacent and each has a higher numbered neighbor that is not a neighbor of the other, then the ordering σ', obtained by swapping v and w in σ, has the same width as σ. Thus, we prune the search tree as follows: for such a pair of vertices v, w, when we have looked at a branch representing the elimination orderings starting with $x_1, \ldots, x_i, v, w$, we prune the branch representing the orderings starting with $x_1, \ldots, x_i, w, v$.

Definition 5. *Let S_1 and S_2 be two elimination orderings for the vertices of a graph $G = (V, E)$. If the treewidth, the upper bound on the treewidth or the lower bound on the treewidth we obtain from eliminating the vertices of G due to their orders in S_1 equals what we obtain from eliminating the vertices of the graph due to their orders in S_2, then we call S_1 and S_2* ***equivalent elimination orderings****.*

Theorem 2. *Let S_1 and S_2 be two elimination orderings for the vertices of a graph G. If S_1 differs from S_2 in the positions for one or more pair v and w, such that for each such pair v, w we have that positions of v and w in S_1 are i and $i + 1$ respectively, for some i, whereas the positions of these vertices in S_2 are $i + 1$ and i respectively, and one of the following properties hold:*

- *v and w are not adjacent in G, or*
- *v and w are adjacent in G, and each has at least one neighbor which is not a neighbor of the other.*

Then S_1 and S_2 are equivalent.

Proof. The theorem is trivial in the case of v and w are not adjacent in G. However, if v and w are adjacent, and each has at least one neighbor, which is not neighbor of the other, then let n_{vw} denotes the number of common neighbors between v and w, n_v (n_w) be the number of neighbors of v (w) that are not neighbors of w (v), see Figure 3.
Case1: If we eliminate v before w, namely, if we remove v and turning its neighbors into a clique Q_v, and then remove w and turning its neighbors into a clique Q_w, then we get

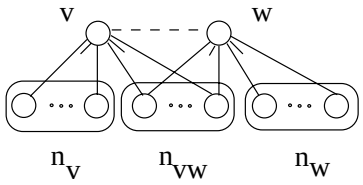

Fig. 3.

$|Q_v| = n_v + n_{vw} + 1$,
$|Q_w| = n_v + n_{vw} + n_w$,
$max_1 = max(|Q_v|, |Q_w|)$, $max_1 = |Q_w|$, if $n_w > 0$, otherwise $max_1 = |Q_v|$.
Case 2: If we eliminate w before v, namely, if we remove w and turning its neighbors into a clique Q_w, and then remove v and turning its neighbors into Q_v, then we get
$|Q_w| = n_w + n_{vw} + 1$,
$|Q_v| = n_w + n_{vw} + n_v$,
$max_2 = max(|Q_w|, |Q_v|)$, $max_2 = |Q_v|$, if $n_v > 0$, otherwise $max_2 = |Q_w|$.

Hence, if n_v and $n_w > 0$, then $max_1 = max_2$, i.e., we obtain the same treewidth or upper bound on treewidth whether we eliminate v before w or w before v. □

Corollary 1. *Consider the same parameters we used in Theorem 2 and let v and w are two adjacent vertices in graph G. Suppose that S_1 and S_2 are two elimination orderings for the vertices of G, such that the order of the vertices in both eliminations are the same except for one pair (v, w) such that the the positions of v and w in S_1 are i and $i + 1$ respectively, and vice versa in S_2. Moreover $n_v = 0$ and $n_w > 0$, then the upper bound on the treewidth we obtain from eliminating the vertices of G due to their orders in S_1 is less than or equal to the upper bound obtained from eliminating the vertices of the graph due to S_2.*

Pruning Rule 5: Simplicial and strongly almost simplicial vertices Given a lower bound β for the treewidth (of the original graph), a vertex is strongly almost simplicial if it is almost simplicial and its degree is at most β. It is known (see [8]) that for each simplicial or strongly almost simplicial vertex v, there is always a linear ordering of minimum width that starts with v. Thus, at each point in the search tree, we check if there is a simplicial or strongly almost simplicial vertex v. If so, we have only one branch, selecting v as the vertex to be eliminated, and we prune all sibling branches selecting a vertex $\neq v$.

Pruning Rule 5 (PR5)
let v be the current vertex we have to eliminate from a graph G;
if v is simplicial or v is strongly almost simplicial
if $degree(v) \geq ub$
omit all subtrees rooted at the parent of v from the search tree,
else omit all subtrees rooted at the sibling of v from the search tree;

Lemma 6. *Let v be a simplicial vertex in a graph $G = (V, E)$. Then the treewidth of G is at least the degree of v, $degree(v)$.*

Lemma 7. *Let v be a strongly almost simplicial vertex in a graph $G = (V, E)$. Then the treewidth of G is at least the degree of v, $degree(v)$.*

3.3 The edge addition rule

Definition 6. *The improved graph $G' = (V, E')$ of a graph $G = (V, E)$ is the graph obtained by adding an edge $\{v, w\}$ to E for all pairs $v, w \in V$ such that, v and w have at least $k+1$ common neighbors of degree at most k in G.*

Gogate and Dechter use another rule, based on the notion of improved graph. If two vertices v and w have at least $ub + 1$ common neighbors, where ub is an upper bound for the treewidth, then adding the edge $\{v, w\}$ does not increase the treewidth (see [12]). Thus, the algorithm of [17] has at nodes a step that looks for such pairs of nonadjacent vertices with many common neighbors, and if found, adds the edge. The hope is that this leads to larger degree vertices that may be pruned by Rule 3.

Theorem 3. *(See Gogate and Dechter [17].)*
Let $G = (V, E)$ be a graph. If ub is an upper bound on the treewidth of G and there exists two vertices v_1 and v_2 in G such that $|N(v1) \cap N(v2)| \geq ub+1$, then there must be an edge between v_1 and v_2 in all possible perfect elimination orderings of G that have treewidth less than or equal to ub.

However, a close analysis of the step shows that it will not help to prune the search tree, and hence only unnecessary spends time: Suppose v and w have at least $ub + 1$ common neighbors, but are not adjacent. As long as no common neighbor of v and w has been eliminated, the degrees of v and w are too large to have these vertices selected; after a common neighbor is eliminated, v and w are anyhow adjacent. Thus, we save time, and do not use this edge addition rule.

3.4 Balancing the use of pruning rules

Using a pruning rule can have a positive or a negative effect on the running time: the time saved by the reduction of the number of considered nodes in the search tree should be less than the time used for testing the validity of the rules. Also, the gain obtained by using some pruning rule may depend on what other rules are also used. Pruning rules 2 and 3 should always be used. For the other rules, we have tested their effect on the running time when used separately and when used in combination with other pruning rules on a large number of graphs.

3.5 The Algorithm

In Figure 4, we give one version of the BB-tw algorithm. In the first step of the algorithm, we check if the best upper and lower bounds for the treewidth of the given graph, obtained form the heuristics, are equal. If so, we return this value as the treewidth of the graph. Otherwise, we

initialize the best upper bound found so far to the treewidth and the perfect elimination ordering for the best upper bound to the elimination ordering for the treewidth. Next, we look for the linear ordering with a better upper bound in the search tree. We prune any ordering from the search tree, which does not yield a better upper bound than that we have reported yet.

We have developed several different versions of the algorithm depending on which pruning rules we incorporate and how we incorporate them in the algorithm. This allows us to see which of the different setups is most effective.

Note: We declare G' and lb in the algorithm as global parameters. The initial value of G' equals the given graph G and the initial value of the lb is lb_h. The values of the parameters of the first call for the algorithm are BB-tw $(peo_{ub_h}, [\,], ub_h, 0)$, where ub_h and lb_h are the best upper and lower bounds obtained from the upper bound and lower bound heuristics for the the treewidth of G, and peo_{ub} is the perfect elimination ordering corresponds to the best upper.

Algorithm BB-tw(S, σ, ub, i)

```
1  if ((n − i) < max) /* PR2 */
2      set ub ← max; max ← max′; σ ← σ′; G′ ← G;
3      if (ub = lb) { return(ub); exit(); } /* PR1 */
4  else
5      foreach v in G′ do
6          if (degree(v) < ub) /* PR3 */
7              if (not((∃ ordering O have been tested before the current
8              elimination ordering σ such that, O[k] = σ[k], k = 1, ··· i − 2&
9              σ[i − 1] = w & O[i − 1] = v & O[i] = w) & (({v, w} ∉ E(G″))
10             or ({v, w} ∈ E(G″) & |N(v)| = 0 or |N(w) = 0)) /* PR4 */
11                 if (degeneracy(G′) < ub)
12                     set G″ ← G′; eliminate v from G′;
13                     set S′ ← S; remove v from S;
14                     set σ′ ← σ; add v to position i in the ordering σ;
15                     if (degree(v) > max)
16                         set max′ ← max; max ← degree(v);
17                     BB-tw (S, σ, ub, i + 1);
18                     G′ ← G″; max ← max′; S ← S′; σ ← σ′;
19                     if(max > ub) return;
20 endif
end BB-tw;
```

Fig. 4. A general scheme for the BB-tw Algorithm.

Theorem 4. *If the BB-tw algorithm terminates normally, then the upper bound obtained from this algorithm equals the exact treewidth of the graph.*

Incorporating the Simplicial and Strongly Almost Simplicial Pruning rule in the BB-tw algorithm

We have tested two methods for incorporating the Simplicial and Strongly Almost Simplicial pruning rule in the branch and bound algorithm. In the first method, we check at each visited node v in the search tree, whether or not v is simplicial or strongly almost simplicial in the temporary graph G'. In the case that v is simplicial or strongly almost simplicial, we test if the degree of v is less than the best upper bound reported up to now, ub. If it satisfies this condition too, then we prune the subtrees rooted at the siblings of this node from the search tree, eliminate the vertex from the graph G' and set the value of the parameter max to the maximum value of its current value and the degree of the eliminated vertex. However, if v is simplicial or strongly almost simplicial and its degree in G' is greater than or equal to the ub, then we prune the subtree rooted at the parent of v from the search tree.

In the second method, at each visited node v in the search tree, we find all simplicial and strongly almost simplicial vertices in G'. Then, if there is no vertex amongst these whose degree is greater than or equal to the reported upper bound, ub, then we eliminate all these vertices from the graph, set the value of max to the maximum value of its current value and the maximum degree of these vertices, and prune all the subtrees rooted at the sibling of each of these vertices, such that if $v_1, \cdots, v_m$ are simplicial vertices in G' and they are siblings of the current visited node v in the search tree, then we eliminate the subtrees rooted at the siblings of v_1, then we eliminate the subtrees rooted at the children of v_1 except the subtree rooted at the node v_2, and so on until the subtree rooted at v_m. However, if there is a vertex amongst these simplicial or strongly almost simplicial vertices its degree is greater than or equal to ub, then we eliminate the subtree rooted at the parent of the node v from the search tree.

3.6 A memory friendly data structure

In order to save memory and time, we use an elegant data structure, which is a variant of the adjacency matrix representation of graphs. Assume the vertices are numbered $1, 2, \ldots, n$. We have an n by n integer matrix A, with for $i < j$, $A_{ij} = 1$ if $\{i, j\} \in E$, and $A_{ij} = 0$ otherwise. The diagonal entries A_{ii} are used to denote if i is already eliminated; $A_{ii} = -1$ if i is not eliminated, and $A_{ii} = k$ if i is the kth eliminated vertex. The lower half of the matrix is used to give fill-in edges: for $j < i$, A_{ij} is initially 0, and becomes k if the edge $\{i, j\}$ is created when eliminating vertex k. We use this matrix as a global variable, and thus have very little parameters to pass on when doing recursive calls in the search, thus giving a considerable gain in speed.

4 Using the Branch and Bound Algorithm for Finding Better Upper and Lower Bounds for the Treewidth

If the branch and bound algorithm does not terminate within reasonable time, then it still often can be used as an upper bound or lower bound heuristic. For this, we initialize the upper bound for the treewidth in the BB-tw algorithm with the best upper bound value obtained from heuristic algorithms. When we terminate the algorithm (e.g., after a pre-determined amount of time), we

report the best upper bound found by the algorithm, with its corresponding linear ordering. Thus, the algorithm either terminates normally, and yields the exact treewidth, or gives an upper bound.

To use the algorithm as a lower bound method, we give the algorithm as upper bound value some integer α. α can be any value, and does not need to be a real upper bound on the treewidth. Now, the algorithm may possibly find a solution of width less than α. If it terminates without having found such a solution, we know that α is a lower bound to the treewidth of G.

We have employed a scheme where we restart the branch and bound algorithm with several different values for the upper bound; each restart is made after the algorithm has run for some fixed amount of time (e.g., 10 minutes or an hour).

Branch and bound flexibility The branch and bound technique has a clear advantage over other treewidth lower and upper bounds techniques: it is often known as the 'any time' property. When given enough time, the algorithm can find the exact bound, but using more time allows to obtain better bounds. Lower or upper bound heuristic give one lower or upper bound value, and when this is not the exact value, giving the heuristic more time does not help to get closer or equal to the exact value.

5 Experimental Results

In this section, we report on computational experiments for the branch and bound algorithm BB-tw. Our experiments are conducted on a large number of graphs and networks. We will show, in this report, some of these results in two tables.

Table 1 shows 15 instances of sizes between 21 and 50 vertices. Table 2 shows 12 instances of sizes between 67 and 450 vertices. The Alarm, Oesoca, Vsd and Wilson are probabilistic networks taken from medical applications; several versions exist of the Myciel networks. The Barley and Mildew networks are used for agricultural purposes, the Water network models a water purification process and Oow-trad, Oow-bas, Oow-solo, and Ship-ship networks are developed for maritime use. The other graphs are obtained from the well-known DIMACS benchmarks for vertex coloring. [1]

The algorithm was implemented using C++ on Windows 2000 PC with Pentium 4, 2.8 GHz processor. The tables shown in this section include besides the basic information for the graph, also columns for the treewidth of the graph (tw), the initial upper bound (ub), the method used for finding the initial upper bound ($ub - heuristic$), and the running time of the algorithm ($time$). We use the character "*" in the column $time$ to indicate that the algorithm did not terminate normally, namely, the algorithm ran out of time. We defined one hour as the maximum limit time for running the algorithm on the input graph, i.e., if the algorithm did not find the exact treewidth within one hour, then it was ended and returned an upper bound value for the treewidth.

All the instances we have chosen to show in Tables 1 and 2 have the following property. The best known upper bound on treewidth of each instance does not equal the best known lower bound on the treewidth of the same instance, obtained from upper and lower bound heuristics. In other words, we have excluded each instance from Tables 1 and 2 whose known upper bound

[1] http://www.cs.uu.nl/people/hansb/treewidthlib, 2004 - 2005.

equals to its known lower bound. The BB-tw algorithm needs less than one second to determine the exact treewidth of any graph when the upper bound on the treewidth equals the lower bound on the treewidth of a graph.

We observe that BB-tw algorithm was able to determine the exact treewidth for all instances of sizes less than 50 vertices except one, namely, Ship-ship, within one hour time. Whereas, the algorithm was able to determine the exact treewidth of only 4 graphs of the large size given in Table 2 within one hour time.

We confirm the observation of Gogate and Dechter [17], that graphs of small treewidth (close to 1) and large (close to n) are easy for BB-tw algorithm also.

Graphname	$\|V\|$	$\|E\|$	lb	ub	ub-heuristic	tw	time
Alarm	37	65	2	4	MFEO1	4	0
Barley	48	126	3	7	MFEO1	7	30.443
Mildew	35	80	2	4	MFEO1	4	0
myciel4	24	71	8	11	MF	10	0.2
myciel5	47	236	14	20	MFEO1	19	178.7
Oesoca+	67	208	9	11	MFEO1	11	0.02
Oow-bas	27	54	2	4	MFEO1	4	0
Oow-solo	40	87	4	6	MFEO1	6	275.608
Oow-trad	33	72	4	6	MFEO1	6	86.772
Queen5_5	25	320	12	18	MFEO1	18	0.62
Queen6_6	36	580	15	26	MF	25	10.62
Ship-ship	50	114	4	8	MFEO1	8	*
VSD	38	62	2	4	MFEO1	4	0
Water	32	123	8	10	MF	9	0.07
Wilson	21	27	2	3	MFEO1	3	0

Table 1.

Graphname	$\|V\|$	$\|E\|$	lb	ub	ub-heuristic	tw	t
anna	138	986	11	12	MFEO1	12	0.911
david	87	812	11	13	MFEO1	13	56.72
dsjc125_1	125	736	15	64	MFEO1	64	*
dsjc125_5	125	3891	55	109	RATIO2	109	*
dsjc250_1	250	3218	43	177	MFEO1	177	*
inithx.i.2	645	13980	31	35	MF	31	0.01
inithx.i.3	621	13969	31	35	MF	31	0.02
games120	120	1276	10	38	RATIO2	38	*
LE450_5A	450	5714	53	304	RATIO2	304	*
myciel6	95	755	12	35	MFEO1	35	*
myciel7	191	2360	31	66	MFEO1	66	*
school1	385	19095	80	209	RATIO2	209	*

Table 2.

6 Conclusions

We can summarize the main results from implementing the BB-tw algorithm as follows:

1. The branch and bound algorithm is efficient for finding the treewidth, namely, it gives the treewidth of a graph in reasonable time, if the given graph has one or more of the following properties:
   - The graph consists of at most 20 vertices.
   - The graph is triangulated or almost triangulated, namely, includes many cliques.
   - The differences between the degrees of the vertices of the graph are relatively large. In other words, the distribution of the degrees of the vertices is irregular.
   - The treewidth of the graph is small (less than 7).
   - The graph has a treewidth which is close to the graph cardinality.
2. The branch and bound algorithm is not efficient for finding the treewidth, if the given graph has one or more of the following specifications:
   - The graph consists of a large number of vertices and a large number of edges.
   - If the distribution of the degrees of the vertices of the graph is regular, namely, the degrees of the vertices of the graph are close to each other.
3. It is difficult to predict the running time of the algorithm when the given graph has a large number of vertices and edges.
4. The ordering of the pruning rules in the BB-tw algorithm has a significant effect on the running time of the algorithm. Two different ordering of the pruning rules in the algorithm may guide, in many instances, to two different running times of the algorithm for the same instance.
5. The efficiency of the algorithm depends critically on the effectiveness of the branching and bounding rules used; bad choices could lead to repeated branching, without any pruning, until the temporary graph becomes very small. In that case, the method would be reduced to an exhaustive enumeration of the domain, which is often impractically large.
6. By using the branch and bound algorithm, we can evaluate the efficiency of any upper or lower bound heuristic, i.e., how much these bounds are close to the exact treewidth of the graph. For example the upper bounds obtained by using MFEO1 [6] heuristic for many tested graphs equal the exact treewidth obtained by using branch and bound algorithm.
7. The branch and bound algorithm BB-tw can be improved in two directions. The first one has to do with the data type we use for representing the graph in the memory. In the current version of the algorithm, we have defined all the values of the adjacency matrix as integers, whereas we believe that the values of the upper part can be defined as bits. The second direction has to do with incorporating more effective pruning and reduction rules in the algorithm. A third approach is to look for a large clique in the input graph, and using the fact that there is an elimination ordering with optimal width that ends with the vertices on the clique.
8. A generalization of the algorithm can be used to compute the weighted treewidth of weighted graphs. This will be reported elsewhere, with other results on weighted treewidth.

Acknowledgments. We would like to thank Gerard Tel for useful comments on earlier versions of this paper, and Arie Koster for joint work on the data structure of Section 3.6.

References

1. E. Amir. Efficient approximation for triangulation of minimum treewidth. *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence, UAI 2001*, pages 7-15, Seattle, Washington, USA, 2001.
2. S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - A survey. *BIT*, 8:277-284, 1987.
3. S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12:308-340, 1991.
4. S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Appl. Math.*, 23:11-24, 1989.
5. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277-284, 1987.
6. E. Bachoore and H. L. Bodlaender. New heuristics for upper bound of treewidth. *Proceedings of the 4th International Workshop, WEA 2005*, pages 216-227, Santorini Island, Greece 2005, Springer Verlag, Lecture Notes in Computer Science, 3503.
7. A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal clique trees. *Artificial Intelligence Journal*, 125:3-17, 2001.
8. H. L. Bodlaender, A. M. C. A. Koster, and F. van den Eijkhof. Pre-processing rules for triangulation of probabilistic networks. *Computational Intelligence*, 21(3):286–305, 2005.
9. H. L. Bodlaender. Discovering treewidth. *Proceedings SOFSEM 2005: Theory and Practice of Computer Science*, pages 1-16, Liptovsky Jan, Slovak Republic, 2005. Springer Verlag, Lecture Notes in Computer Science, vol. 3381.
10. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25:1305–1317, 1996.
11. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2): 1-21,1993.
12. H. L. Bodlaender. Necessary edges in k-chordalizations of graphs. *Journal of Combinatorial Optimization*, 7:283–290, 2003.
13. H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21:358–402, 1996.
14. H.L. Bodlaender, A.M.C.A. Koster and T. Wolle. Contraction and treewidth lower bounds. *Proceedings of 12th Annual European Symposium, Algorithms-ESA*, pages 628-639, Bergen, Norway 2004, Springer Verlag, Lecture Notes in Computer Science, 3221.
15. F. van den Eijkhof, H.L. Bodlaender and A.M.C.A. Koster. Safe reduction rules for weighted treewidth. *Proceedings of the 28th International Workshop in Graph-Theoretic Concepts in Computer Science (WG 2002)*, pages 176-185, Berlin, Germany, 2002. Springer Verlag, Lecture Notes in Computer Science, vol. 2573.
16. F. Clautiaux, A. Moukrim, S. Négre, and J. Carlier. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Operations Research*, 38:13–26, 2004.
17. V. Gogate and R. Dechter. A complete any time algorithm for treewidth. *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, pages 201–208, Banff, Canada, 2004, AUAI Press, Arlington, Virginia.
18. A. M. C. A. Koster, H. L. Bodlaender, and S. van Hoesel. Treewidth: Computational experiments. In *Electronic Notes in Discrete Mathematics*, volume 8. Elsevier Science Publishers, 2001.
19. S. Ramachandramurthi. The structure and number of obstructions to treewidth. *SIAM J. Disc. Math.*, 10:146–157, 1997.
20. H. Röhrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998.