Studienarbeit

Tuning Algorithms for Hard Planar Graph Problems

Frederic Dorn

Betreuer: Dipl.-Math. Jochen Alber
Dr. rer. nat. Rolf Niedermeier

Lehrstuhl Formale Sprachen/Theoretische Informatik
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Contents

1 Introduction

Many problems of high practical significance turn out to be NP-hard, that is, we do not know any efficient algorithms for them. But many well known NP-hard problems can be stated with a parameter k, so that they have polynomial-time algorithms when k is fixed. (For example, given a graph, decide if it has a VERTEX COVER of size at most k.) Downey and Fellows initiated a systematic complexity analysis of such problems [13]. They called those parameterized problems that can be solved by an algorithm with running time $O(f(k)n^{\alpha})$ (where α is a constant and n is the number of graph vertices) *fixed-parameter tractable* (FPT). The problems we study here are hard problems for planar graphs, that is, graphs which can be drawn without edge crossings in the plane. For planar graphs, some problems are in FPT which are not for general graphs (like DOMINATING SET) [13]. The heart of the FPT algorithms presented in this paper is the idea of the *treewidth* of a graph. The treewidth expresses, informally speaking, how "tree-like" a graph is. This paper is focused on the implementation of such an algorithm using treewidth to solve hard planar graph problems. Therefore, the algorithm is split into two proper phases. In the first phase of the algorithm the treewidth is determined. With techniques as decomposing the graph into layers and separating the graph into suitable chunks, an upper bound of $O(\sqrt{k})$ for the treewidth is obtained (where k is the size of the optimal solution for a given problem). The treewidth occurs as a parameter in the running time. In the second phase the optimal solution for VERTEX COVER or DOMINATING SET is determined with the help of dynamic programming, which delivers the running time of the form $O(c^{\sqrt{k}}n^{O(1)})$ [1, 2, 4] (where c is a constant and k the size of the optimal solution).

The underlying techniques are discussed in Section 2, where a pseudocode of the whole algorithm is given. The contribution of this work is to report on the implementation of these algorithms. Our aim is to make the usage of the corresponding software package as convenient as possible. This software package, which we are developing, is written in *C++* and based on LEDA [15]. The usage and design is described in Section 3, where an overview on the functionality of the software package is given. In Section 4, all implemented *C++*-classes and procedures are documented, so this section is a kind of manual on the software package. We also introduce several new algorithmic ideas for the solution of problems not considered in the underlying theoretical papers. In Section 5, first experimental studies on the practical behavior of these "$c^{\sqrt{k}}$-algorithms" are presented. The main message from our studies is that they behave much better than could be expected from the theoretical worst case analysis given in [1, 2, 4]. In Section 6, we discuss some possible improvements for future work, for example with regard to the program code, so that our software package might turn into a useful tool for practical applications.

2 Theory and Algorithms

Tree decompositions. The key notion of this work is that of a tree decomposition of a graph. The basic idea behind is that many graph problems can be solved "easily" on trees. Hence, intuitively speaking, it is appealing to try to detect tree-like structures in

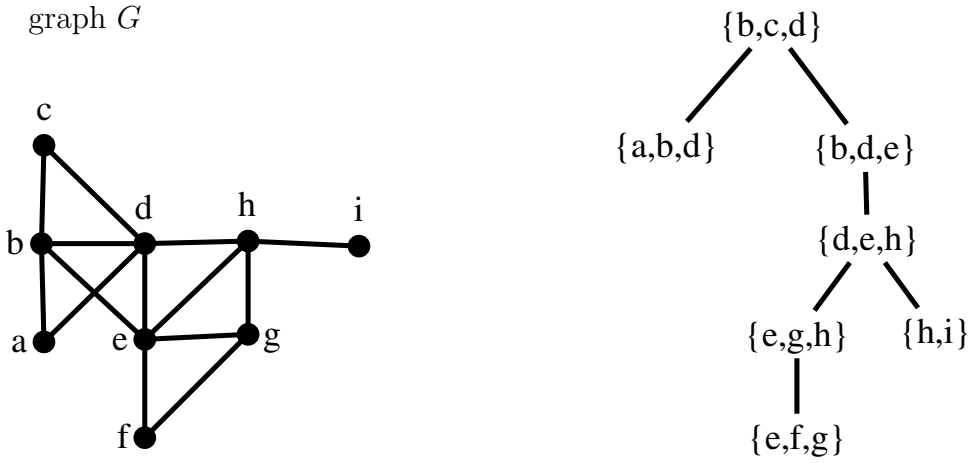


Figure 1: Tree decomposition $\mathfrak{X}$ of a graph G. The biggest bag of $\mathfrak{X}$ has three vertices of G, so the width of $\mathfrak{X}$ is two. Moreover, the treewidth of G is two, since G has several K_3 as subgraphs.

graphs in order to attack hard graph problems. The notions of tree decomposition and treewidth formalize how tree-like a graph is.

Definition 1 *Let $G = (V, E)$ be a graph. A* tree decomposition *of G is a pair $\mathfrak{X} = \langle \{X_i \mid i \in I\}, T \rangle$, where each X_i is a subset of V, called a* bag*, and T is a tree with the elements of I as nodes. The following three properties must hold:*

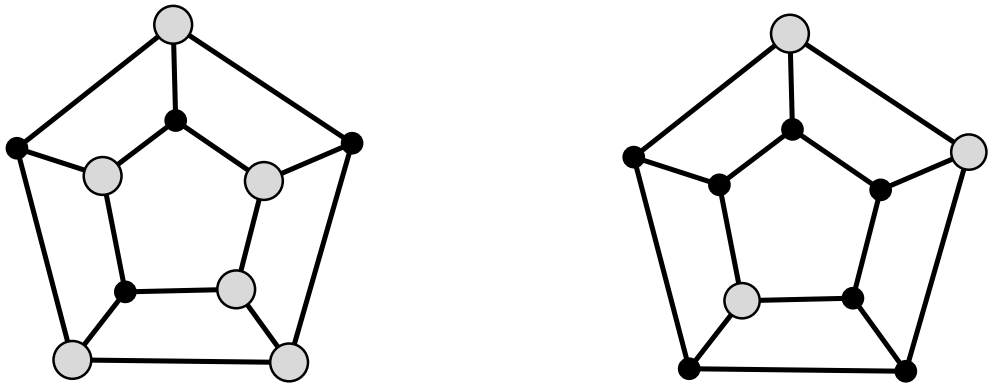
1. $\bigcup_{i \in I} X_i = V$;
2. *for every edge $\{u, v\} \in E$, there is an $i \in I$ such that $\{u, v\} \subseteq X_i$;*
3. *for all $i, j, k \in I$, if j lies on the path between i and k in T, then $X_i \cap X_k \subseteq X_j$.*

The width $\mathrm{tw}(\mathfrak{X})$ *of $\mathfrak{X}$ equals* $\max\{|X_i| \mid i \in I\} - 1$. *The* treewidth $\mathrm{tw}(G)$ *of G is the minimum k such that G has a tree decomposition of width k.*

See Fig. 1 for an example. For example, the treewidth of a tree is one: Just create one bag at any tree vertex and put the tree vertex and its parent vertex inside the corresponding bag. In contrast, the treewidth of a clique K_n of size n is $n - 1$, which can be proven by induction.
The intuition for the algorithm is that hard graph problems like Vertex Cover can be solved in linear running time on trees. Using dynamic programming on the bags, the combinatorial explosion of the running time is reduced to the size of the subgraphs, induced by the bags of the tree decomposition. To solve a graph problem using tree decompositions, one usually proceeds as follows:

phase I: find a tree decomposition of bounded width of the input graph,

phase II: solve the problem using dynamic programming on the tree decomposition (see [9]).



VERTEX COVER of size $k = 6$ DOMINATING SET of size $k = 3$

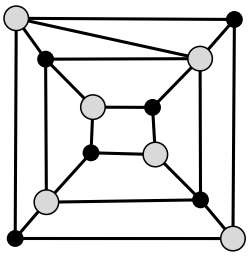
Figure 2: Examples for VERTEX COVER and DOMINATING SET. The vertices of the vertex set and dominating set are highlighted grey in this picture. VERTEX COVER: Any edge has at least one adjacent highlighted vertex. DOMINATING SET: Any vertex has at least one adjacent highlighted vertex or is highlighted itself.

Fixed parameter algorithms on planar graphs. We study planar graphs, i.e., graphs that can be drawn in the plane without edge crossings. Then, (G, ϕ) will denote a plane graph, i.e., a planar graph G together with an embedding ϕ in the plane. In recent work [2, 4], a framework was developed that describes the construction of tree decomposition-based algorithms for a large class of NP-complete problems on planar graphs. For the ease of presentation we focus on the problems VERTEX COVER and DOMINATING SET on planar graphs.

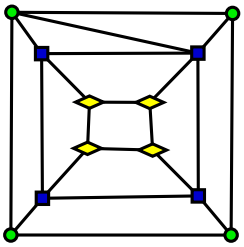
VERTEX COVER. Given an undirected graph $G = (V, E)$ and a positive integer k, find a subset of at most k vertices $V' \subseteq V$ such that each edge in E has at least one of its endpoints in V'.
DOMINATING SET. Given an undirected graph $G = (V, E)$ and a positive integer k, find a subset of at most k vertices $V' \subseteq V$ such that every vertex of G either belongs to V' or has a neighbor in V' (or both) (see Fig. 2 for an example for VERTEX COVER and DOMINATING SET).

The goal was to obtain "efficient" fixed parameter algorithms that solve these problems *optimally.* Obviously, since these problems are NP-complete, we have to accept exponential running times. It could be shown [2, 4], however, that for parameter k being (an upper bound on) the size of the vertex cover or dominating set we search for, and n being the number of graph vertices, the running time achievable by an algorithm that executes the two phases mentioned above has a sublinear exponent in k.

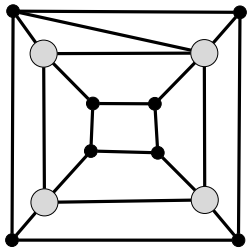
Theorem 2 VERTEX COVER *on planar graphs can be solved in time* $O(c_1^{\sqrt{k}}n)$ *and* DOM-



VERTEX COVER $k = 6$



outerplanarity $r = 3$



one *separator* of size 4

Figure 3: An example for the "Layerwise Separation Property": $|separator| = 4 < \sqrt{3k} = \sqrt{18} \approx 4.2$.

INATING SET *on planar graphs can be solved in time* $O(c_2^{\sqrt{k}}n)$*, where* $c_1 = 2^{4\sqrt{3}}$ *and* $c_2 = 2^{12\sqrt{34}}$. □

On the positive side, this means that "one only pays for what one gets," i.e., the smaller the set we are searching for, the faster we can find it. On the negative side, clearly, the given *worst case* constants c_1 and c_2 are far from being practical. One of the main motivations for this work is to get experimental insight into how far the worst case might be from the average case.

In this section, the algorithm yielding Theorem 2 is shortly outlined. At the end of this section, the whole algorithm is given in pseudocode. In the following, the theoretical background is explained to make an understanding of the algorithm easier.

The algorithm (phase I). We give some theoretical background to explain the pseudocode of the algorithm. The pseudocode can be found at the end of this section. In [4], a general methodology was developed how to, given planar graph problems such as VERTEX COVER or DOMINATING SET with parameter k, construct tree decompositions of width $O(\sqrt{k})$. To do this, one has to separate the graph in a particular, "layerwise" way. The key to this is the so-called "Layerwise Separation Property," which holds for many graph problems (see [4] for details). Here, the term "layer" refers to the following graph decomposition:

Definition 3 *Let* $(G = (V, E), \phi)$ *be a plane graph. The* layer decomposition *of* (G, ϕ) *is a disjoint partition of the vertex set* V *into sets* $L_1, \ldots, L_r$ *(called the* layers*), which are recursively defined as follows:*

- L_1 *is the set of vertices on the exterior face of* G*, and*
- L_i *is the set of vertices on the exterior face of* $G[V - \bigcup_{j=1}^{i-1} L_j]$ *for* $i = 2, \ldots r$.

The (uniquely defined) number r *of different layers is called the* outerplanarity *of* (G, ϕ)*, denoted by* $\mathrm{out}(G, \phi) := r$.

See *Algorithm I.1* in the pseudocode at the end of this section for the application of a layer decomposition. See also the middle diagram of Fig. 3 for a graph of outerplanarity 3. The term "separator" refers to the following definition:

Definition 4 *Let $(G = (V, E), \phi)$ be a graph. A subset S of the vertex set V is called a* separator *of G, if the subgraph $G \setminus S$ is disconnected.*

We are searching for special separators. Therefore, the idea of "Layerwise Separation Property" (*LSP*) for graph problems was introduced in [4]. One says, a parameterized problem on planar graphs satisfies the *LSP*, if for each *yes-instance* (G, k) separators of size at most $O(\sqrt{k})$ can be found "layerwisely", so that the remaining residue graphs have outerplanarity of at most $O(\sqrt{k})$. More precisely, for each *LSP*-problem $\mathcal{L}$ a constant s (*size-factor*) and a constant w (*width*) have to be found, so that for $(G, k) \in \mathcal{L}$ a separator of size $\sqrt{\frac{3}{2}s \cdot k}$ occurs in graph G after every $\sqrt{\frac{2}{3}s \cdot k}$ layers. This separator takes at most w layers (see the right diagram of Fig. 3 for an example). *LSP* is used in the algorithm at *Algorithm I.2*.

After having separated the graph G layerwisely by separators $S_1, \ldots, S_\ell$ (each of size bounded by $O(\sqrt{k})$) into its chunks $G_0, \ldots, G_\ell$ (each of outerplanarity $O(\sqrt{k})$) (*Algorithm I.3*), one constructs tree decompositions $\mathfrak{X}_i$ for the graphs G_i, $0 \leq i \leq \ell$ using an algorithm described in [2, 10] (*Algorithm I.3*). Finally, the decompositions $\mathfrak{X}_i$ are "melted" into a (global) tree decomposition for G (*Algorithm I.4*).

The algorithm (phase II). It is common knowledge that, once given a tree decomposition for a graph, many otherwise hard graph problems can be solved easily. More precisely, for treewidth ℓ phase II can be done in time $O(c^\ell N)$, where N is the number of bags of the tree decomposition and c is some constant. In case of Vertex Cover (*Algorithm II.a*), it is not hard to see that $c = 2$. In case of Dominating Set (*Algorithm II.b*), however, this is more complicated. Telle and Proskurowski [17, 18] showed that phase II can be done in time $O(9^\ell N)$, which was recently improved to $O(4^\ell N)$ [2, 6], a significant improvement that some of the experiments in this paper owe their success.

To complete this section, we give the pseudocode of the whole algorithm, solving the graph problems Vertex Cover and Dominating Set, respectively.

The algorithm.

phase I: determine the tree decomposition

input: plane graph $(G = (V, E), \phi)$, parameter k, graph problem $\mathcal{G}$

output: tree decomposition $\mathfrak{X} = \langle \{X_i \mid i \in I\}, T \rangle$ of G

1. determine the layer decomposition $\mathcal{L} = \{L_1, \ldots, L_r\}$ of (G, ϕ) dependent on ϕ
2. determine separators $\mathcal{S} = \{S_1, \ldots, S_\ell\}$ dependent on k and $\mathcal{G}$
3. divide G by $\mathcal{S}$ into the subgraphs $G_0, \ldots, G_\ell$
4. for all subgraphs G_j {
   - determine layers of edges of G_j
   - make a supergraph $\hat{G}_j$ by replacing all vertices v with degree more than three by a path with $(\text{degree}(v) - 2)$ many vertices $\hat{v}_1, \ldots, \hat{v}_f$
   - create a spanning tree ST_j in a layerwise fashion of $\hat{G}_j$
   - make a tree decomposition $\mathfrak{X}_j$ using ST_j
   - replace all "new" supergraph vertices $\hat{v}$ in the bags X_i by the corresponding "old" vertex

   }
5. distribute S_j on $\mathfrak{X}_{j-1}$ and $\mathfrak{X}_j$ (for all $j \in \{1, \ldots, \ell\}$); merge all $\mathfrak{X}_j$ to a global tree decomposition $\mathfrak{X}$

phase II a): solve Vertex Cover using dynamic programming

input: tree decomposition $\mathfrak{X} = \langle \{X_i \mid i \in I\}, T \rangle$ with rooted tree T

output: an optimal Vertex Cover VC

1. for all bags X_i of $\mathfrak{X}$ {
   create a table T_{X_i} with $2^{|X_i|}$ rows and $|X_i| + 1$ columns; each entry, except the entries of the last column, has the value 0 or 1; the entries of the last column consist of the sum of the values of the first $|X_i|$ entries }
2. "bottom up:"
   update $T_{X_{\text{parent}}}$ with $T_{X_{\text{child}}}$ for every child of a node parent in the tree; create pointers $P_{\text{parent}_r -> \text{child}_{r_1,\ldots,r_m}}$: any row r of $T_{X_{\text{parent}}}$ points at the corresponding rows $r_1, \ldots, r_m$ of $T_{X_{\text{child}}}$
3. "top down:"
   determine a row r in $T_{X_{\text{root}}}$ with the minimum value of the last column and follow the pointers $P_{\text{parent}_r -> \text{child}_{r_1,\ldots,r_m}}$ to determine VC beginning with parent = root

phase II b): solve Dominating Set using dynamic programming

input: tree decomposition $\mathfrak{X} = \langle \{X_i \mid i \in I\}, T \rangle$ with rooted tree T

output: an optimal Dominating Set DS

1. determine a "nice" tree decomposition $\mathfrak{X}_N$
2. the remaining algorithm is analogous to phase II a), but with table T_{X_i} with $3^{|X_i|}$ rows, where each entry has the value 0_1, 0_2 or 1

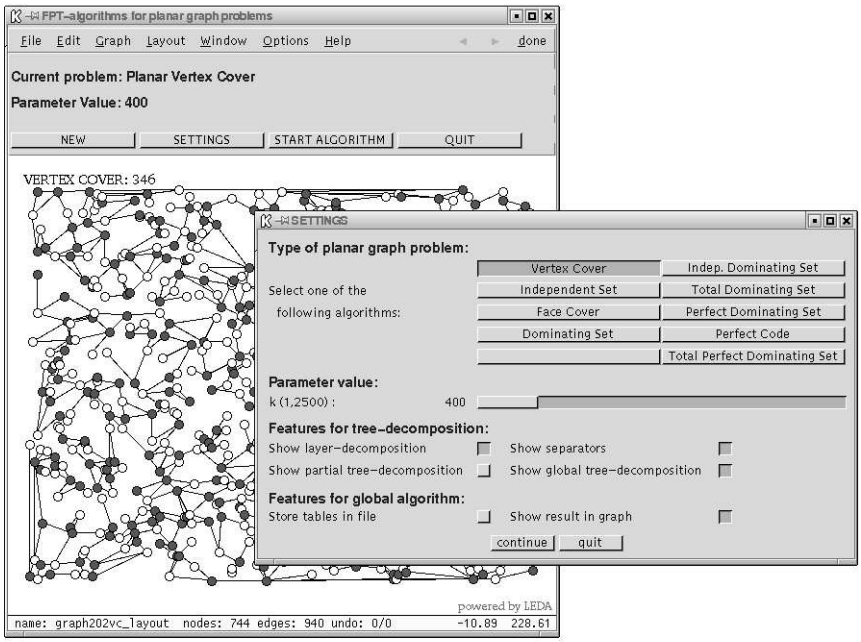


Figure 4: **Software package for planar graph problems.**

3 Design and Use

Using the *C++* library LEDA (Library of Efficient Data structures and Algorithms) [15] we have implemented a software package which is designed to solve some NP-hard problems on planar graphs optimally. More precisely, this package offers algorithms for parameterized graph problems that fit into the framework of [2, 4], i.e., that have the so-called "Layerwise Separation Property." These include the graph problems shown on the panel of the screenshot in Fig. 4. So far, only Vertex Cover, Independent Set, and Dominating Set are implemented. The usage of the package is very easy. We provide an application with buttons for various functions and a window for the graph drawing. The planar input graph can be drawn either directly or can be loaded to the program in `GraphWin gw`-format. The graph should be simple and without selfloops. Since the graphs of the considered graph problems have undirected edges, all edges are treated as if they were undirected (the default representation of an edge in LEDA is directed). The choice of the plane embedding is free, but different embeddings could result in different layer decompositions and, hence, different tree decompositions with different widths (see Fig. 7 in Section 4.3 for an example).

It is also important to choose a crossing-free embedding, because otherwise, the upper bound on treewidth is not guaranteed any more (see Fig. 9 in Section 4.3 for an example). The button SETTINGS in the main window opens a panel for the various settings of the algorithm. The user selects the type of problem that (s)he wants to solve and chooses the parameter value k (i.e., the size of the desired vertex cover, independent set, dominating set, etc. the algorithm is checking for). The current type of problem and the current

value of k are displayed in the main window. Besides, some extra optional features of the output can be adjusted. If *Show layer decomposition* is chosen, during the algorithm, a window will be opened, which shows the layers of the graph. All vertices of the same layer will have the same color. By choosing *Show separators*, a window with the graph will be opened, too. It shows the separator vertices with a red color. The separated subgraphs appear black and white alternatingly. For these subgraphs the algorithm will generate partial tree decompositions. These partial tree decompositions will be put out in various windows when selecting *Show partial tree decompositions. Show global tree decomposition* puts out the global tree decomposition in an extra window. Each window, which contains a tree decomposition, displays the underlying tree with marked vertices. The corresponding bags are listed in a scrollbar in the window. With *Store tables in file* the user can determine a file to store the tables of the dynamic programming in ASCII format. By choosing *Show result in graph*, the vertices, which offer an optimal solution for the chosen type of problem, are highlightened in the graph in the main window. The button START ALGORITHM in the main window runs the algorithm with the chosen settings. The algorithm solves the problem, i.e., it puts out that there is no optimal solution of size at the most k (for minimization problems) or it computes an optimal solution. NEW closes all other windows, so the user can run another algorithm on the graph.

4 Implementation Details

4.1 The LEDA library

LEDA [15] is a library of efficient data types and algorithms and a platform for combinatorial and geometric computing on which application programs can be built. It is written in C++ and (especially) contains many pre-implemented graph algorithms, like Maximum Flow or Spanning Tree, which were used in our software package. LEDA provides also a class `graph`, which owns vertices (there called `node`s) and `edge`s. The class `graph` has a lot of very useful procedures, like `graph` iterators. Furthermore, LEDA contains the class `GraphWin`, which combines the datatypes `graph` and `window`. A `window` is an application, where an object of `graph` can be drawn. An object of type `GraphWin` is a `window`, a `graph`, and a drawing of the `graph`, all at once. In the following documentation of our software package, LEDA datatypes and procedures are written in `bold face`. For further details on the data types like `graph`, `node_map`, `map`, `node_partition`, consult [16].

4.2 Documentation

Here, we give a detailed documentation on the implementation and the use of the datatypes and procedures of our software package. The program is divided into two main parts:

- **phase I**: determine the tree decomposition
- **phase II**: solve a given graph problem $\mathcal{G}$ with a given tree decomposition

With all procedures used for phase I an object TD of the class `tree_decomposition` is generated. This class owns an object T of the class `graph` (which represents a tree) and an object *bag* of the class `node_map<list<node> >` (which represents the bags of TD).

In phase II, the dynamic programming part is executed on TD. We give a documentation for the VERTEX COVER and DOMINATING SET problem.

The two phases are illustrated by process charts, depicted in Fig. 5 and Fig. 6.

4.2.1 phase I

1. `void layer_decomposition(GraphWin&` *gw*`)`

   `DEFINITION:` creates a layer decomposition of the underlying `graph` G of *gw*.
   `INPUT:`

   - `GraphWin` *gw* with a plane `graph` G.

   `OUTPUT:`

   - the global variables `node_map<int>` *level* (saving the layers of the `node`s) and
   - `list<list<node> >` *layer_list*.

   `ALGORITHM:`

   - copy *gw* to `GraphWin` GW;
   - create empty layer list `list<list<node> >` $layer\_list = (L_1, \ldots, L_r)$;
   - run "Space Invaders" algorithm on the underlying `graph` H of GW (starting with $i = 1$):
     while H is not empty {
     - for all `node`s n of H: find `node` n_{min} with minimum x-coordinate min_xcoord;
     - generate "Invader" `node` n_I at $(min\_xcoord - 1, 0)$;
     - generate `edge` $e = (n_I, n_{min})$;
     - sort the adjacency lists L_{Adj_n} of any `node` n in counter clockwise order with `SORT_EDGES(`H`,xcoords of all nodes,ycoords of all nodes)`;
     - "walk" on the outer face using the sorted adjacency lists to determine the first layer:
       start at n_I, go to n_{min}, search in $L_{Adj_{n_{min}}}$ for the next `node` in counter clockwise order;
       go to this `node` and repeat last step until arriving at n_I again;
     - put all visited `node`s into a layer list `<list<node> >` L_i, add L_i to *layer* and delete them from H;
     - $i++$;

     }

2. `void separator(GraphWin&` *gw*`)`

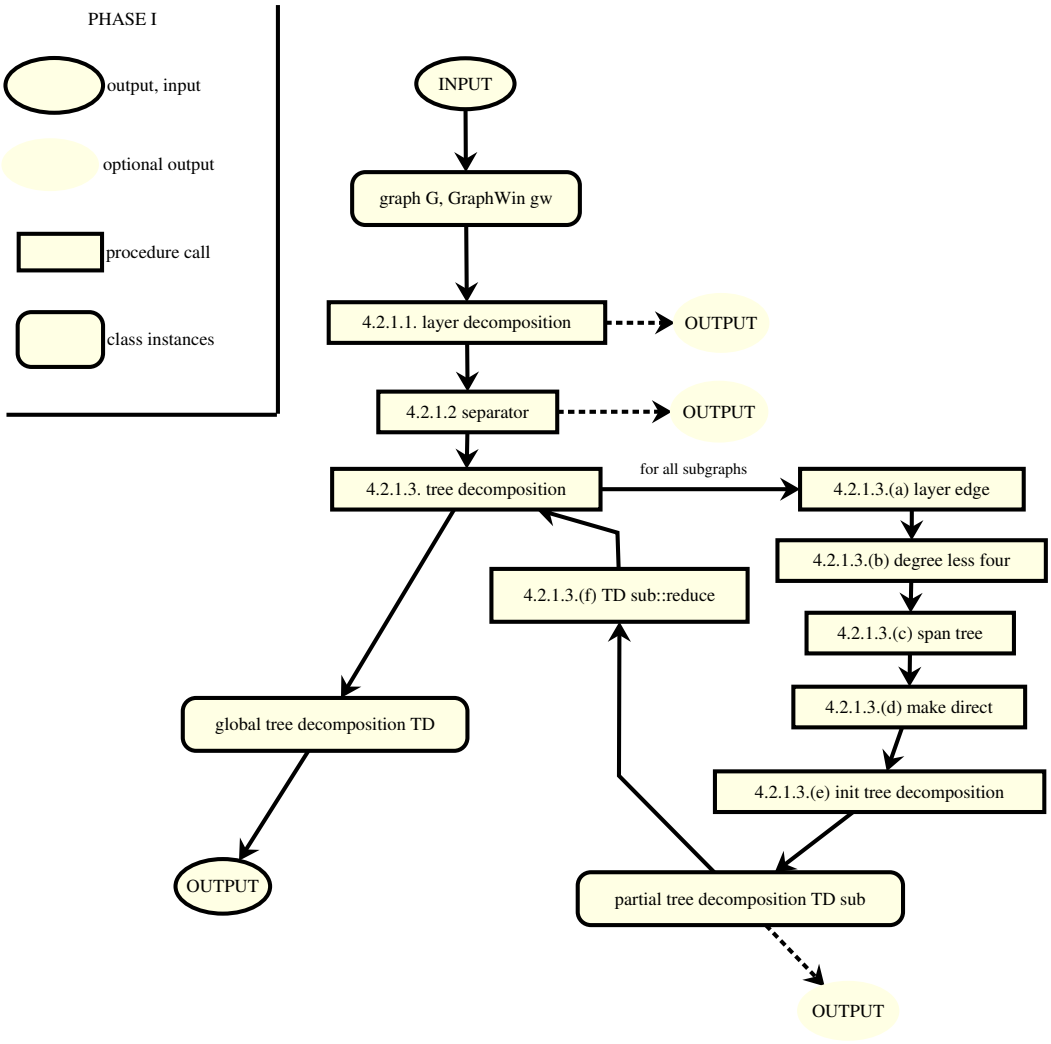


Figure 5: Process chart for phase I: The rectangular boxes correspond to the various procedure calls, numbered after the various paragraphs in this section. The boxes with round corners correspond to the generated class instances. For each subgraph `tree_decomposition` calls a loop of subprocedures.

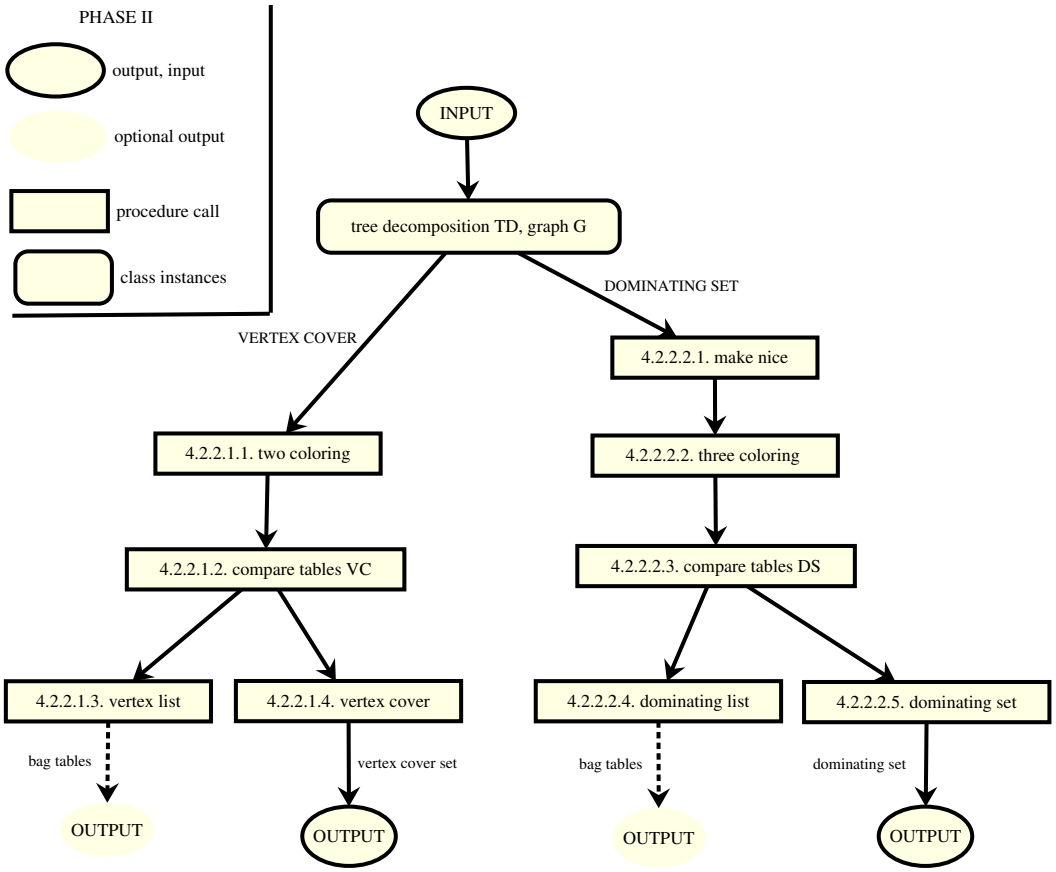


Figure 6: Process chart for phase II: The rectangular boxes correspond to the various procedure calls, numbered after the various paragraphs in this section. The boxes with round corners correspond to the generated class instances.

DEFINITION: determines "layerwise separation" of G, which depends on the parameter value *parameter* and the considered graph problem.
INPUT:

- the global variables `int` *parameter* ,
- `int` *width* ,
- `int` *size_factor* (see [4] for details).

OUTPUT:

- a global variable separator list `list<list<node> >` *separator_list* of size ℓ, which divides G layerwisely into $\ell + 1$ subgraphs;
- the guarantee is that $\ell \in O(\sqrt{parameter})$ and that all subgraphs have outerplanarity $O(\sqrt{parameter})$ (see Section 2 for details on the "Layerwise Separation Property").

ALGORITHM:

- `int` $size = \sqrt{\frac{3}{2}\, size\_factor \cdot parameter}$
  `int` $max\_layer = \sqrt{\frac{2}{3}\, size\_factor \cdot parameter}$;
- create empty separator list `list<list<node> >` $separator\_list = (S_1, \ldots, S_\ell)$;
- for all subgraphs *G_layer* induced by the `node`s of the layers $L_i, \ldots, L_{i+j}$ of *layer_list* with $\mid layer\_list \mid = r$ where $j - i = width$ and $i \in \{1, \ldots, r - j\}$ {
  - if $max\_layer = 0$ print: "No solution for $k = parameter$";
  - generate a source `node` s and a target `node` t;
  - connect s to all `node`s of L_i, which have an `edge` to `node`s of L_{i-1} in G;
  - connect t to all `node`s of L_{i+j}, which have an `edge` to `node`s of L_{i+j+1} in G;
  - use s and t in `MAX_FLOW` to determine separator list `list<node>` *sep*;
  - if $\mid sep \mid < size$
    then add *sep* to *separator_list* and put *max_layer* to its original value
    else *max_layer*– –;

  }

3. `void tree_decomposition(GraphWin&` *gw*`)`

   DEFINITION: creates a tree decomposition for all subgraphs $G_0, \ldots, G_{\ell+1}$ generated by the graph separators and melts them into a global tree decomposition of the original plane graph.
   INPUT:

   - a global variable separator list `list<list<node> >` *separator_list*.

   OUTPUT:

   - a global variable `tree_decomposition` TD.

ALGORITHM:

- for all subgraphs *G_sub* of the original graph that are obtained by a separation according to *separator_list* {

  (a) void layer_edge(GraphWin& *gw_sub*)
  DEFINITION: determines all edge layers of $G_j, (j = 1, \ldots, l)$.
  INPUT:
  - subgraph *G_sub* and
  - its GraphWin *gw_sub*.

  OUTPUT:
  - the global variables edge_map<int> *level_edge* (storing the layer of the edges),
  - the edge layer list list<list<edge> > *layer_edge_list* and
  - the node_map<edge,edge> *in_ccw_out* saving for all nodes two adjacent edges.

  ALGORITHM:
  - copy *gw_sub* to GraphWin *GW_sub*;
  - create empty edge layer list list<list<edge> > *layer_edge_list* $= (L_1, \ldots, L_r)$;
  - create node_map<edge,edge> *in_ccw_out*
  - run "Space Invaders" algorithm on the underlying graph *H_sub* of *GW_sub*:
    while *H_sub* is not empty {
    - for all nodes n of *H_sub*: find node n_{min} with minimum x-coordinate *min_xcoord*;
    - generate "Invader" node n_I at $(min\_xcoord - 1, 0)$;
    - generate edge $e = (n_I, n_{min})$;
    - sort the adjacency lists L_{Adj_n} of any node n in counter clockwise order with SORT_EDGES(*H_sub*,xcoords of all nodes,ycoords of all nodes);
    - "walk" on the outer face using the sorted adjacency lists to determine the first edge layer:
      start at n_I, go to n_{min} search in $L_{Adj_{n_{min}}}$ for the next node in counter clockwise order;
      go to this node and repeat last step until arriving at n_I again; but always add the incoming and the outcoming edges on which this node is visited to this nodes *in_ccw_out*;
    - throw all visited edges into the edge layer list L_i, add L_i to *layer_edge_list* and delete these edges and the isolated nodes from *H_sub*;

    }

  (b) void degree_less_four(GraphWin& *gw_sub*)
  DEFINITION: replaces all nodes with degree more than three by paths of nodes with degree less than four while maintaining the node layer. The

resulting graph is the "supergraph" $\hat{G}_j$.
INPUT:

- subgraph *G_sub*,
- its GraphWin *gw_sub* and
- the global variable node_map<edge,edge> *in_ccw_out*.

OUTPUT:

- the global variable node_map<node> *remember_node* and
- *G_sub*, which is now a supergraph with degree(n)< 4 for all n of *G_sub*.

ALGORITHM:

- sort the adjacency lists L_{Adj_n} of any node n in counter clockwise order with SORT_EDGES(*G_sub*,xcoords of all nodes,ycoords of all nodes);
- create node_map<node> *remember_node* and initialize it for all nodes n of *G_sub* with themselves;
- for all nodes n of *G_sub* {
  - if degree(n) > 3 {
  - generate degree(n)-2 many nodes $n_1, \ldots, n_k$;
  - move the first edge (n, u) of *in_ccw_out* to (n_1, u) and the next edge (n, v) of (n, u) in L_{Adj_n} to (n_1, v);
  - the *remember_node* of n_1 is n;
  - for all nodes n_i, where $i \in \{1, \ldots, k-1\}$ {
    - generate edge (n_i, n_{i+1});
    - move the next edge (n, v_i) in L_{Adj_n} to (n_{i+1}, v_i);
    - the *remember_node* of n_{i+1} is n;

    }
  - move the second edge (n, w) of *in_ccw_out* to (n_k, w);
  - delete n;

    }
  }

(c) void span_tree(GraphWin& *gw_sub*)
DEFINITION: produces a spanning tree ST_j of $\hat{G}_j$, which is generated layerwisely;
INPUT:

- supergraph *G_sub*,
- its GraphWin *gw_sub*,
- the global variables edge_map<int> *level_edge* and
- list<list<edge> > *layer_edge_list*.

OUTPUT:

- a global variable list<edge> ST, which is a spanning tree of *G_sub*.

ALGORITHM:

- create edge_map<edge> *cost* with $cost = \infty$ for all edges e of *G_sub*;
- int $max\_layer = max\{level\_edge(e) \mid e \in E(G\_sub)\}$;

- copy *G_sub* to `graph` H;
- for all `edge`s e of H { if *level_edge*(e) $\neq$ *max_layer* then delete e };
- `list<edge>` ST = `SPANNING_TREE(H)`;
- for all `edge` layers L_i of *layer_edge_list* (from *max_layer* to 0) {
  - copy *G_sub* to `graph` H;
  - for all `edge`s e of H {
    - if $e \in ST$ then *cost*(e) = 0;
    - if *level_edge*(e) < *max_layer* − 1 then delete e from H;
    - *max_layer*– –;
    }
  - ST = `MIN_SPANNING_TREE(`H`,`*cost*`)`;
  }

(d) `void make_direct(GraphWin&` *gw_sub*`)`
`DEFINITION:` determines a root of ST_j and makes ST_j directed from the root to the leaves.
`INPUT:`
- supergraph *G_sub*,
- its `GraphWin` *gw_sub*, and
- its spanning tree `list<edge>` ST.

`OUTPUT:`
- the top-down directed spanning tree `list<edge>` ST of *G_sub*.

`ALGORITHM:`
- determine a root `node` n_{root}, which is in the inner layer of *G_sub*;
- go through ST and flip the direction of the `edge`s from n_{root} towards the leaves;

(e) `void init_tree_decomposition(GraphWin&` *gw_sub*`)`
`DEFINITION:` makes a tree decomposition TD_j out of ST_j. Therefore an object *TD_sub* of the `tree_decomposition` class is generated. This object contains a `graph` *TD_sub*.T and a `node_map<list<node> >` *bag*.
`INPUT:`
- supergraph *G_sub*,
- its `GraphWin` *gw_sub*,
- its top-down directed spanning tree `list<edge>` ST, and
- its `node_map<node>` *remember_node*.

`OUTPUT:`
- the global variables `tree_decomposition` *TD_sub* with
- its `graph` *TD_sub*.T and
- its `node_map<list<node> >` *bag*.

`ALGORITHM:`
- initialize an object `tree_decomposition` *TD_sub* with *TD_sub*.`init(`ST`)` :

- copy ST to the `graph` *TD_sub.T*;
- initialize `node_map<list<node> >` *bag* of any `node` v of *TD_sub.T*;
- for all `node`s n of *G_sub* {
  - add n to the $bag(v_n)$ of *TD_sub.T*;

  }
- for all `edge`s (u, w) of ST {
  - delete (u, w) from *TD_sub.T*;
  - generate path (u, x, w) with a new `node` x in *TD_sub.T*;
  - $bag(x) = bag(u) \cup bag(w)$;

  }
- for all `edge`s (l, m) of $G\_sub \setminus ST$ {
  - add l to any bag on the path between v_l and v_m. This path is determined by Tarjan's least common ancestor algorithm (see ?? for details);

  }
- for all `node`s v of *TD_sub.T* {
  - replace all `node`s n of $bag(v)$ by *remember_node*(n);

  }

(f) *TD_sub*`.reduce()`

`DEFINITION:` is a subroutine of the class `tree_decomposition`. It reduces the size of TD_j by combining two neighboring bags, whenever one *bag* appears to be a subset of the other.

`INPUT:`

- `tree_decomposition` *TD_sub*,
- its `graph` *TD_sub.T*, and
- its `node_map<list<node> >` *bag*.

`OUTPUT:`

- the reduced `tree_decomposition` *TD_sub*.

`ALGORITHM:`

- for all `node`s v of *TD_sub.T* {
  - set `list<node>` $L_{N(v)}$ = adjacent `node`s of v;
  - while $L_{N(v)}$ is not empty {
    - delete first `node` n_i from $L_{N(v)}$;
    - if $bag(n_i) \subseteq bag(v)$ then delete n_i from *TD_sub.T*, add all neighbors of n_i to $L_{N(v)}$ and connect them to v in *TD_sub.T*;

    }

  }

} (End of *for all G_sub*)

- add all S_i from *separator_list* $= (S_1, \ldots, S_\ell)$ to all *bag*s of the tree decompositions of the subgraphs, which S_i had divided;
- connect all *TD_sub*s to one global `tree_decomposition` TD by generating an edge between the roots of two following *TD_sub*s (*TD_sub*s are ordered in the way the *G_sub*s has been separated);

4.2.2 phase II

4.2.2.1 Vertex Cover

For determining an optimal Vertex Cover, the program runs through the following procedures, which are subroutines of the class `tree_decomposition`. We also need the definition of *color*: Each *bag* will have a *bag_table*, where each `node` u of *bag* has a corresponding column. To minimize memory requirement *bag_table* is implemented as a `list<two_tuple<int,int> >`. `Int` *color* is defined by the bit representation of the first entries i of the `two_tuple`s in *bag_table*: if u stands at position j in *bag* then *color*(u) is the bit at position j in i.

1. TD`.two_coloring()`

   `DEFINITION:` creates one table (corresponding to the *bag*s bag_i in TD of size treewidth + 1) with two "colors" for each node n in bag_i of size $2^{|bag_i|}$ for all possible 0,1-permutations of length $|bag_i|$.
   `INPUT:`

   - `tree_decomposition` TD,
   - its `graph` $TD.T$,
   - its `node_map<list<node> >` *bag*, and
   - its treewidth tw of type `int`.

   `OUTPUT:`

   - a global variable `list<two_tuple<int,int> >` *biggest_table*, representing the bag tables of the *bag*s of size $tw + 1$.

   `ALGORITHM:`

   - `int` $i = 0$;
   - create empty `list<two_tuple<int,int> >` *biggest_table*;
   - while $i < 2^{tw+1}${
     - add (i,*min_i*) to *biggest_table*, where *min_i* is the sum of 1 bits in the binary representation of i;
     - $i++$;

     }

2. TD`.compare_tables_VC()`

   `DEFINITION:` the main procedure in this part: it compares the tables in TD in a "bottom up" way.
   `INPUT:`

   - the global variable `list<two_tuple<int,int> >` *biggest_table*,
   - `tree_decomposition` TD.

   `OUTPUT:`

- the global variables `node_map<list<two_tuple<int,int> > >` *bag_table*, representing the bag tables of all *bag*s, and
- `node_map<list<three_tuple<int,node,list<int> > > >` *bag_pointer* of the form {(row r of the corresponding *bag_table*, child `node` n in $TD.T$, rows $r_1, \ldots, r_\ell$ in the *bag_table* of n)}.

`ALGORITHM:`

- initialize *bag_table* and *bag_pointer* on TD;
- for all `node`s v of $TD.T$ from the leaves to the root{
  - if *bag_table*[v] not created{
    - *bag_table*[v] = the first $2^{|bag[v]|}$ rows of *biggest_table*;
    - for all pairs of `node`s $\{u, w\}$ of *bag*[v] {
      - if there is no `edge` between u and w then delete all rows of *bag_table*[v], where the *color* of u and w are 0;

      }

    }
  - if *bag_table*[*parent*(n)] not created then repeat last part for the parent `node` *parent*(n);
  - sort *bag_table*[*parent*(n)] and *bag_table*[n] with `bucket_sort` (see ?? for details);
  - for all rows (`int`) r in *bag_table*[*parent*(n)] {
    - determine the minimum *min* of the *min_i* of the corresponding i rows in *bag_table*[n] (where (i,*min_i*) are the `two_tuple`s of *bag_table*[n]);
    - replace *min_r* by *min_r* + *min* − sum of the *color*s of equal *bag* `node`s;
    - add (r, n,*list of all i with value min* in *bag_table*[n]) to *bag_pointer*[*parent*(n)];

    }

  }

3. TD.`vertex_list(bool` *file_save*`, file` *dir_file*`)`

   `DEFINITION:` saves (if wanted) all tables in a file.
   `INPUT:`

   - the global variables `node_map<list<two_tuple<int,int> > >` *bag_table*,
   - `node_map<list<three_tuple<int,node,list<int> > > >` *bag_pointer*, and
   - the complete path for the file to save.

   `OUTPUT:`

   - ASCII file with all *bag_table*s and *bag_pointer*s.

4. TD.`vertex_cover()`

   `DEFINITION:` determines an optimal Vertex Cover by evaluating the tables "top down" in TD.
   `INPUT:`

- the global variables `node_map<list<two_tuple<int,int> > >` *bag_table* and
- `node_map<list<three_tuple<int,node,list<int> > > >` *bag_pointer*.

`OUTPUT:`

- `int` *vc_min*, which is the size of the optimal VERTEX COVER and
- `list<node>` *vc_list*, the `node`s of an optimal VERTEX COVER.

`ALGORITHM:`

- create an emtpy `list<node>` *vc_list*;
- determine a row r of *bag_table*[*root*] with minimal *min_r*;
- *vc_min* = *min_r*;
- add all `node`s u of *bag*[*root*] with *color*(u) = 1 in r to *vc_list*;
- from n =*root* to the leaves in B(readth) F(irst) S(earch) {
  - for all rows r_b of *bag_pointer*[n], where r_b.*first()*== r {
    - go to child `node` $m = r_b$.*second()*;
    - choose randomly one row r_m of r_b.*third()* and set $r = r_b$;
    - add all `node`s u of *bag*[m] with *color*(u) = 1 in r to *vc_list*;

  }
}

4.2.2.2 DOMINATING SET

For determining an optimal DOMINATING SET, the program runs through the following procedures, which are also subroutines of the class `tree_decomposition`. We also need a redefinition of *color*: Each *bag* will have a *bag_table*, where each `node` u of *bag* has a corresponding column. To minimize memory requirement *bag_table* is implemented as a `list<two_tuple<int,int> >`. `Int` *color* is defined by the bit representation of the first entries i of the `two_tuple`s in *bag_table*:
if u stands at position j in *bag* then :

- *color*(u) = 0 , if the bit at the $j-$th odd position is 0 and the bit at the $j-$th even position is 0;
- *color*(u) = 1 , if the bit at the $j-$th odd position is 0 and the bit at the $j-$th even position is 1;
- *color*(u) = 2 , if the bit at the $j-$th odd position is 1 and the bit at the $j-$th even position is 1.

1. TD`.make_nice()`

   `DEFINITION:` transforms TD into a "nice" tree decomposition TD_N.
   `INPUT:`

   - `tree_decomposition` TD,

- its `graph` $TD.T$, and
- its `node_map<list<node> >` *bag*.

`OUTPUT:`

- a global variable `tree_decomposition` TD, which is now a "nice" tree decomposition.

`ALGORITHM:`

- for all `node`s n of $TD.T$\{
  - if outdegree(n) > 1\{
    - create $2 \cdot |$ children of $n| - 2$ copies $(\hat{n}_1, \ldots, \hat{n}_r)$ of n (and copy the *bag*s also);
    - make a complete binary tree bt out of $\{n, \hat{n}_1, \ldots, \hat{n}_r\}$ (with root n);
    - connect each child of n in $TD.T$ to one leaf of bt;

    \}

  \}
- for all `node`s n of $TD.T$\{
  - if outdegree(n) $== 1$\{
    - while ($|bag[n]| > 0$ and $bag[n] \neq bag[n] \cap bag[\text{child}(n)]$) {
      - `node` u = first node of $bag[n] - (bag[n] \cap bag[\text{child}(n)])$;
      - delete `edge` (n,child(n));
      - create new `node` m with $bag[m] = bag[n] \setminus u$;
      - create new `edge`s (n, u) and (u,child(n));
      - set $n = u$ and child(n) = child(u);

      }
    - while ($|bag[n]| < |bag[\text{child}(n)]|$) {
      - `node` u = first node of $bag[\text{child}(n)] - bag[\text{child}(n)] \cap bag[n]$;
      - delete `edge` (n,child(n));
      - create new `node` m with $bag[m] = bag[n] \cup u$;
      - create new `edge`s (n, u) and (u,child(n));
      - set $n = u$ and child(n) = child(u);

      }

    \}

  \}

2. TD`.three_coloring()`

   `DEFINITION:` creates the tables (corresponding to the *bag*s bag_i in TD) with three "colors" for each node n in bag_i of size $3^{|bag_i|}$ for all possible 0,1,2-permutations of length $|bag_i|$, where these tables differ a little bit in being generated dependent on the kind of nodes (Insert, Forget, Join (see [2] for details)) .
   `INPUT:`

   - "nice" `tree_decomposition` TD,

- its `graph` $TD.T$,
- its `node_map<list<node> >` *bag*, and
- its treewidth `int` tw.

`OUTPUT:`

- a global variable `list<two_tuple<int,int> >` *bag_table*, representing the bag tables of the *bag*s.

`ALGORITHM:`

- initialize *bag_table* on TD;
- for all `node`s n of $TD.T$\{
  - if *bag_table*[n] not created\{
    - `int` $i = 0$;
    - while $i < 4^{|bag[n]|}$\{
      - add (i,*min_i*) to *bag_table*, if in the binary representation of i there is no 1 bit at an odd position followed by a 0 bit.*min_i* is the sum of the 1 bits in the binary representation of i, which are at the odd position;
      - $i++$;

      \}
    - for all `node`s u of *bag*[n] and $u_1, \ldots, u_\ell$ (:= neighbors of $u \cap$ *bag*[n]) \{
      - delete all rows of *bag_table*[n], where the *color* of u is 0 and all *color*s of $u_1, \ldots, u_\ell$ are not 1

      \}
    - if n is a FORGET `node` and u is the "to forget node" in *bag*[n] then delete all rows of *bag_table*[n] where *color*(u) is 2;

  \}
\}
- delete all rows of *bag_table*[root], which contain the color 2;

3. TD`.compare_tables_DS()`

`DEFINITION:` the main procedure in this part: it compares the tables "bottom up" in TD.
`INPUT:`

- the global variable `list<two_tuple<int,int> >` *bag_table*,
- `tree_decomposition` TD.

`OUTPUT:`

- the global variables `node_map<list<two_tuple<int,int> > >` *bag_table*, representing the bag tables of all *bag*s, and

- `node_map<list<three_tuple<int,node,list<int> > > >` *bag_pointer* of the form {(row r of the corresponding *bag_table*, child `node` n in $TD.T$, rows $r_1, \ldots, r_\ell$ in the *bag_table* of n)}.

`ALGORITHM:`

- initialize *bag_pointer* on TD;
- for all `node`s n of $TD.T$ from the leaves to the root{
  - if *parent*(n) is an INSERT `node` or a FORGET `node`{
    - sort *bag_table*[*parent*(n)] and *bag_table*[n] with `bucket_sort` (see ?? for details);
    - for all rows (`int`) r in *bag_table*[*parent*(n)] {
      - determine the minimum *min* of the *min_i* of the corresponding i rows in *bag_table*[n] (a little bit different for INSERT `node`s, see next subsection);
      - replace *min_r* by *min_r* + *min* − sum of the *color*s of equal *bag* `node`s (*color* 2 counts 0);
      - add (r, n,*list of all i with value min*) to *bag_pointer*[*parent*(n)];
      }
    }
  - if *parent*(n) is a JOIN `node`{
    - for all rows (`int`) r in *bag_table*[*parent*(n)] {
      - the minimum *min* is the sum of the *min_i* of the corresponding row i in *bag_table*[n] and the *min_j* of the corresponding row j in *bag_table*[m] (where m is the other child of *parent*(n)). If the `node` u in *bag*[*parent*(n)] has the *color* 0 at position k then *color*(u) has to be 0 and 2 (or 2 and 0) at position k in i and j;
      - replace *min_r* by *min* − sum of the *color*s of equal *bag* `node`s (*color* 2 counts 0);
      - add (r, n,*list of all i with value min*) to *bag_pointer*[*parent*(n)];
      }
    }
  }

4. TD.`dominating_list(bool` *file_save*`, file` *dir_file*`)`

   `DEFINITION:` saves (if wanted) all tables in a file.
   `INPUT:`

   - the global variables `node_map<list<two_tuple<int,int> > >` *bag_table*,
   - `node_map<list<three_tuple<int,node,list<int> > > >` *bag_pointer*, and
   - the complete path for the file to save.

   `OUTPUT:`

   - ASCII file with all *bag_table*s and *bag_pointer*s.

5. TD.`dominating_set()`

   `DEFINITION:` determines an optimal Dominating Set by evaluating the tables "top down" in TD.
   `INPUT:`

   - the global variables `node_map<list<two_tuple<int,int> > >` *bag_table* and
   - `node_map<list<three_tuple<int,node,list<int> > > >` *bag_pointer*.

   `OUTPUT:`

   - `int` *ds_min*, which is the size of the optimal Dominating Set and
   - `list<node>` *ds_list*, the `node`s of an optimal Dominating Set.

   `ALGORITHM:`

   - create an emtpy `list<node>` *ds_list*;
   - determine a row r of *bag_table*[*root*] with minimal *min_r*;
   - *ds_min* = *min_r*;
   - add all `node`s u of *bag*[*root*] with *color*(u) = 1 in r to *ds_list*;
   - from n =*root* to the leaves in BFS {
     - for all rows r_b of *bag_pointer*[n], where r_b.*first()*== r{
       - go to child `node` $m = r_b$.*second()*;
       - choose randomly one row r_m of r_b.*third()* and set $r = r_b$;
       - add all `node`s u of *bag*[m] with *color*(u) = 1 in r to *ds_list*;

       }
   }

4.3 Difficulties, Workarounds, and Tuning

Although the LEDA package has made many things much easier, the implementation work was fairly challenging, needing several new algorithmic ideas for solving problems not considered in the underlying theoretical papers.

Space Invaders. The problem of determining the layers of a given embedding was one of the problems that had to be dealt with: Note that the number of layers depends heavily on the given embedding (see Fig. 7 for an example). An easy routine to determine the layers of a plane graph would make use of the faces of this embedding. Since there is no implemented function in LEDA to determine the faces of a graph corresponding to a given embedding, we had to develop our own solution for determining the layers. The algorithm, which determines the layers of a given embedding is the so-called "Space Invaders" algorithm, used in the procedures `layer_decomposition` and `layer_edge`:

Since this algorithm makes use of the geometric property of the given graph embedding, a cartesian coordinate system is introduced with the graph at the first quadrant. An "Invader" vertex is created at point $(0, 0)$. The "Invader" vertex is connected with the vertex v of the graph with the minimum x-coordinate. Start a walk at v to the next

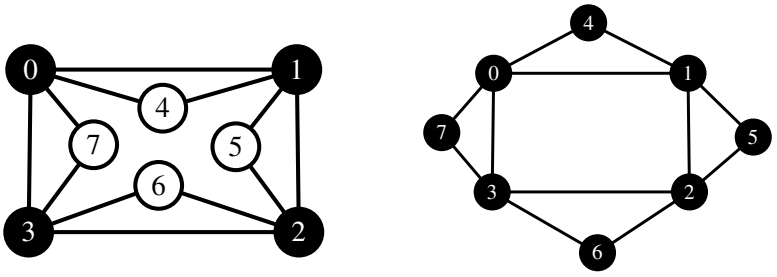


Figure 7: The same graph with two different plane embeddings. The outerplanarity of the left embedding is 2, whereas the right one has outerplanarity 1.

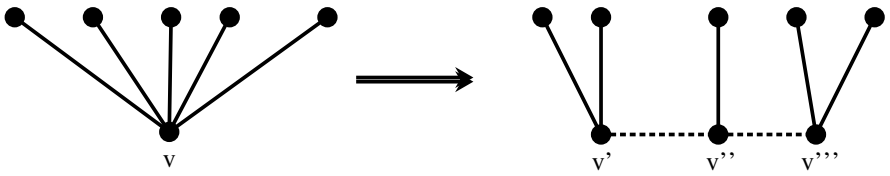


Figure 8: Replacing the vertex v by a path of vertices v', v'', v''' with degree less than four.

adjacent vertex in the adjacency list of v in counter clockwise order. Now the outer face (especially the first layer) is determined by walking on the path (cycle) to the "Invader" vertex. After having removed the first layer from the graph, in this manner, the same procedure is repeated on the remaining graph in order to inductively get all layers.[1]

Replacing high-degree vertices by paths. Since the algorithm guarantees a bounded treewidth only for graphs with vertices of degree less than four, graph vertices of degree more than three have to be replaced by paths of vertices of at most degree three (in the procedure `degree_less_four`, see Fig. 8). This also caused some small difficulties:
First, it is important that a planar graph is given together with a plane embedding. Otherwise, i.e., running the procedure `degree_less_four` on a non-planar embedding, it could happen that the resulting supergraph is not planar anymore (see Fig. 9 for an example). In this case no upper bound for the width of the tree decomposition of the graph can be guaranteed.
A second problem to solve was that the new vertices had to be in the same layer as the replaced vertices. So it is important to which direction the new path is "opened". Therefore it is not sufficient to only know the layers of the adjacent vertices or/and of the adjacent edges (see Fig. 10 and Fig. 11: Two examples, which show the possible consequences of focussing exclusively on the layers of the adjacent vertices or edges).
A way of avoiding this problem is to take two edges adjacent to the vertex in question, which are at the same time adjacent to the face closest to the outer face. These two edges can be determined when running the procedure `layer_edge`. On the walk along the outer face, as described above, *in_ccw_out* always stores the incoming and the outgoing edges of any vertex. If one vertex is turned into a path of vertices each of degree less than four, then the two edges of *in_ccw_out* make up the first and the last edges of this path. This guarantees that the new vertices are in the same layers as the replaced ones (see Fig. 12 and Fig. 13: The vertices in these examples are augmented to paths in a right way).

Minimize memory requirement. In phase II of the algorithm, we worked out a bit-level representation of the vertex colors to avoid a waste of memory. In this bit representation, there are not n table entries per row with values 0 or 1. Instead, these n table entries are treated as if they were the binary representation of one integer. We use only one integer per row of the table. Note that 32 bits are sufficient. This means, that the table could have 2^{32} entries.
This representation fits exactly for Vertex Cover or Independent Set, where only two colors are needed, but for Dominating Set, where we need three colors, a "trinary" representation is needed. So, we masked out two bits for one color, which means that the tables got nearly twice as big. The redundant "fourth" color we used to make the comparisons in `compare_tables_DS()` more time efficient. For example, consider the INSERT NODE comparison (see [2], p.11): The color "0" in a table entry of the parent bag is compared either to the color "0" or "2" in the corresponding table entries of the

[1] The animated version of this algorithm explains the name : the "Invader" vertex moving at one side, "shooting" edges to the graph, which "destroy" vertices, calls to mind the "Space Invaders" arcade game of the 80's.

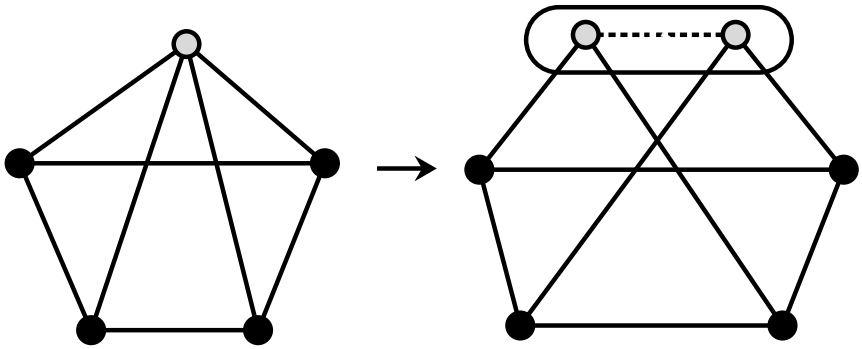


Figure 9: A planar graph turns into a nonplanar graph, after replacing the grey vertex by a path of vertices with degree less than four.

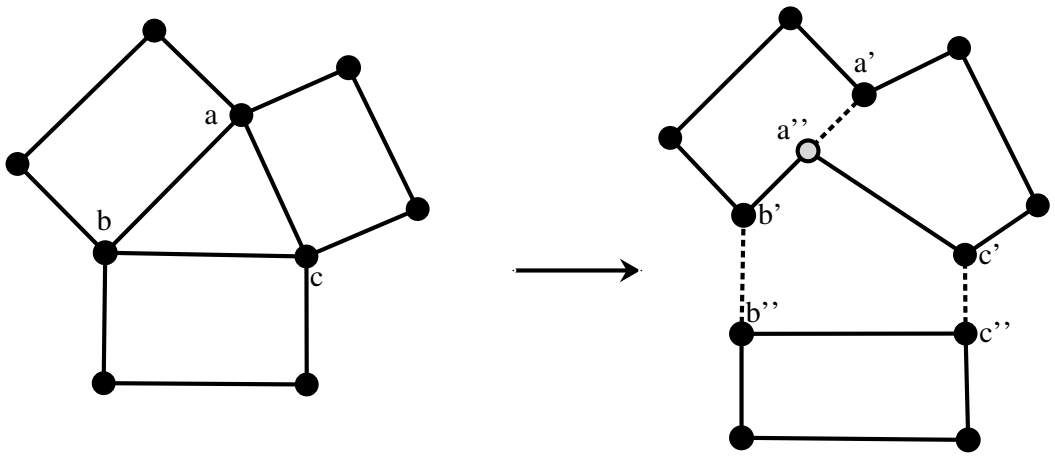


Figure 10: This is an example to illustrate why it is not sufficient to only know the layers of the adjacent vertices. All vertices are in the same layer. But vertex a is replaced by a path of the vertices a' and a'', where a'' is not in the same layer any more.

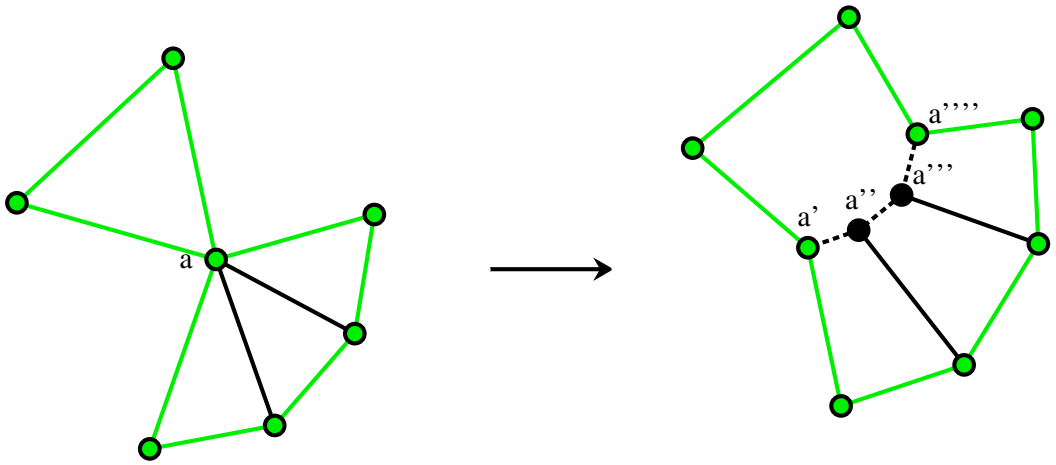


Figure 11: Here is an example for only taking into account the layers of the adjacent edges. The vertices a'' and a''' of the new path are not in the same layer any more.

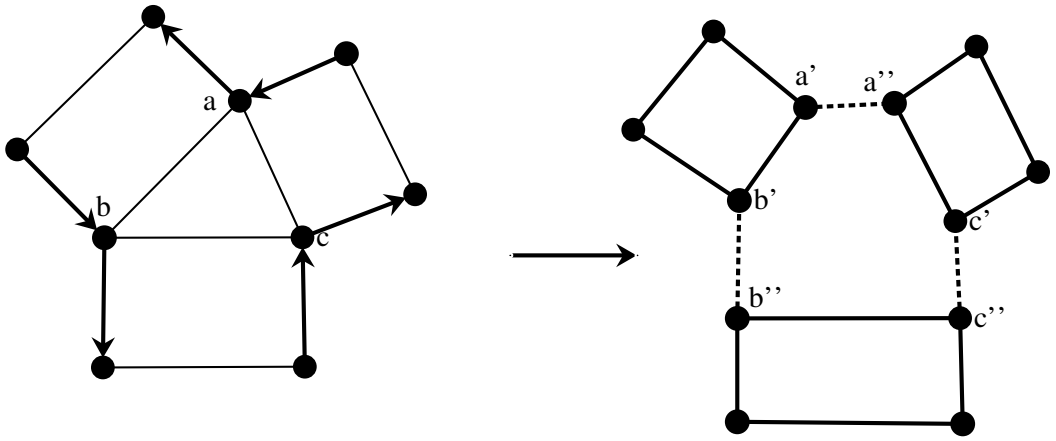


Figure 12: Compare with Fig. 10: The directed edges are the edges stored in *in_ccw_out*. Vertex a is replaced by a path of the vertices a' and a'', which now are in the same layer as a. Note that the paths are generated between the edges of *in_ccw_out*.

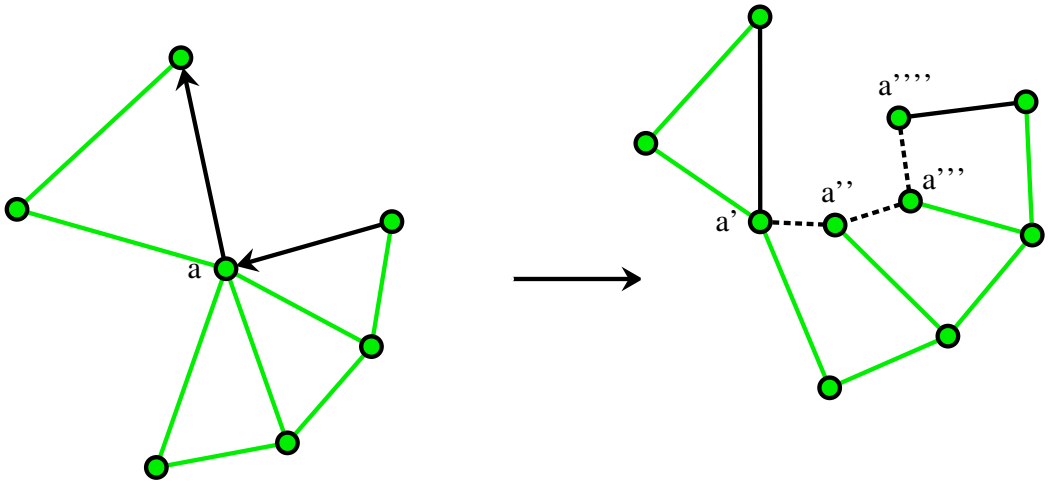


Figure 13: Compare with Fig. 11: The directed edges are the edges stored in *in_ccw_out*. Vertex a is replaced by a path of the vertices a', a'', a''', and a'''', which now are in the same layer as a.

child bag – to which color it is compared depends on the other colors in this table entry of the parent bag. To avoid a loss of time, caused by checking the colors of the same entry, we act as follows: We temporarily exchange these “0”s, which are compared to “2”, by the “fourth” color.

Save running time. In `compare_tables_VC()` (and `compare_tables_DS()`) it is very important to take care of the running time when comparing the tables. Whenever two tables are updated, all rows are allowed to be visited only $O(1)$ often. This means that not every row of the one table can be updated with every row of the other. That is why the tables are sorted. Determine the intersection of the underlying bags. Then the rows are sorted as follows: The bits, that belong to the vertices of the intersection, are masked out. Then the tables are sorted after these bits. Now both tables can be compared by going only once through them.

4.4 Lines of Code, Manyears

So far, the complete software package has been implemented within nine months, of 80h work each month, producing around 5000 lines of C++ code based on the LEDA package [15] ((non-commercial) version 4.2). The underlying machine is a conventional 750 MHz LINUX PC with 720 MB main memory. The implementation still has to be called a prototype—numerous future fine-tuning improvements are foreseeable.

sample set	PG100	PG500	PG750	PG1000	PG1500	PG2000	PG3000	PG4000
size of sample	100	100	100	100	100	50	50	20
# vertices	100	500	750	1000	1500	2000	3000	4000
# edges	201.5	974.6	1483.7	1978.9	2992.0	4016.8	5895.5	8577.6
# layers	3.92	5.12	5.36	5.61	5.84	6.14	6.24	6.89
max. degree	23.2	50.8	61.2	73.3	90.6	103.8	123.1	156.6
avg. degree	4.03	3.90	3.96	3.96	3.99	4.02	3.93	4.29
treewidth (by phase I)	6.99	9.33	10.32	11.11	12.24	13.26	13.67	15.17
avg. bagsize	4.20	4.22	4.33	4.37	4.46	4.67	4.46	4.86
variance of bagsize	2.51	3.59	4.15	4.33	5.02	5.30	5.49	6.41
# bags	75.7	389.9	583.2	779.6	1167.2	1538.8	2346.0	3054.1
depth of tree	19.1	52.8	70.6	82.5	115.2	136.6	173.4	242.1
max. degree in tree	7.0	29.1	37.0	54.3	66.3	79.9	158.8	151.4
time (sec): phase I	0.3	3.0	6.9	12.5	30.2	62.2	146.6	331.7
size of subsample (VC)	100	100	100	99	98	47	44	14
time (sec): phase II (VC)	0.06	5.41	12.35	34.46	116.40	192.17	402.36	1015.91
total time (sec): VC	0.4	8.4	19.3	46.8	145.8	247.4	510.1	1294.0
size of VC	47.2	225.2	342.0	453.2	683.0	924.3	1326.5	1827.8
size of subsample (DS)	92	59	35	29	10	2	2	0
time (sec): phase II (DS)	16.0	208.7	340.5	302.2	936.8	1636.0	1810.5	–
total time (sec): DS	17.1	215.9	365.7	326.1	953.8	1644.6	1837.4	–
size of DS	24.9	155.8	246.8	373.8	532.0	825.5	1182.0	–
sample set	PGD100	PGD500	PGD750	PGD1000	PGD1500	–	–	–
size of sample	100	100	100	50	20	–	–	–
# layers	3.84	6.76	8.41	8.63	10.15	–	–	–
max. degree	7.76	8.85	9.38	9.08	9.35	–	–	–
avg. degree	3.96	4.01	4.24	4.04	4.05	–	–	–
treewidth (by phase I)	9.18	18.45	24.16	24.65	30.30	–	–	–
avg. bagsize	5.62	7.51	8.64	8.34	8.96	–	–	–
variance of bagsize	4.11	15.11	23.34	24.97	32.40	–	–	–

Table 1: Summary of experimental results. The numbers in the various rows are taken as the *average* over graphs in PGn (and PGDn, respectively) of the corresponding column. The abbreviations in the first column are explained in detail in Subsections 5.1, 5.2, 5.3, and 5.4. Each block of rows refers to one Subsection.

5 Experimental Results

In this section, we report on the experimental results obtained by running our software package on various random input samples. This should be considered as a first serious round of tests of our implemented algorithms. Later, we plan to run our tests on a greater variety of not only random input instances and, in particular, we will try to also experiment with planar graphs drawn from practical applications. In the following we partly follow [3].

5.1 Generating Random Graphs

We created a set of sample graphs using the LEDA [15] standard function

```
random_planar_graph (graph& G, int n, int m)
```

for generating (combinatorial) random planar graphs. Here, n and m (with $m \leq 3n-6$) specify the number of vertices and edges of the graph. The function, in a first step, generates a random maximal planar graph in an inductive way: For $n = 3$, as an induction base, a triangle is created. For $n > 3$, a random maximal planar graph of order $n-1$ is generated, an additional vertex v is added to a random face f, and all edges from v to the boundary of f are drawn. In a second step, the function `random_planar_graph` removes all but m edges at random.

Using this function, we created sample sets PGn (PG is short for "planar graphs") of random planar graphs with n vertices (where $n = 100$, 500, 750, 1000, 1500, 2000, 3000, 4000). Here, for each graph in PGn, we chose m as a random number in the interval $[n-1, 3n-6]$. All graphs were given together with a "straight-line embedding" that was computed using the LEDA standard function `STRAIGHT_LINE_EMBEDDING`. Various graph structural data for the sample sets PGn is given in the first block of rows in Table 1.

- *size of sample:* number of random planar graphs in sample PGn;
- *# vertices:* average number of vertices of graphs in PGn;
- *# edges:* average number of edges of graphs in PGn;
- *# layers:* average number of layers of graphs in PGn using the standard straight-line embedding offered by LEDA;
- *max. degree:* average maximum degree of graphs in PGn;
- *avg. degree:* average "average degree" of graphs in PGn;

As outlined in Section 2, the algorithms proceed in two phases. In phase I, a tree decomposition of the given graph is constructed. Phase II then solves the given problem by a dynamic programming approach on the tree decomposition obtained by phase I.

5.2 Phase I: Constructing Tree Decompositions

Phase I of the algorithm, i.e., the construction of a tree decomposition of the given plane graph, is common to all problems offered by our software package.

Evaluation. In this phase, we measured the following figures for a given random input graph:

- *treewidth:* width of tree decomposition obtained;
- *avg. bagsize:* average size of the bags of the tree decomposition;
- *variance of bagsize:* variance of the size of the bags of the tree decomposition;
- *# bags:* number of bags of tree decomposition;
- *depth of tree:* depth of the tree in tree decompositions;
- *max. degree in tree:* maximum degree of the tree in tree decomposition;
- *time phase I:* time needed to perform phase I of the algorithm (in seconds).

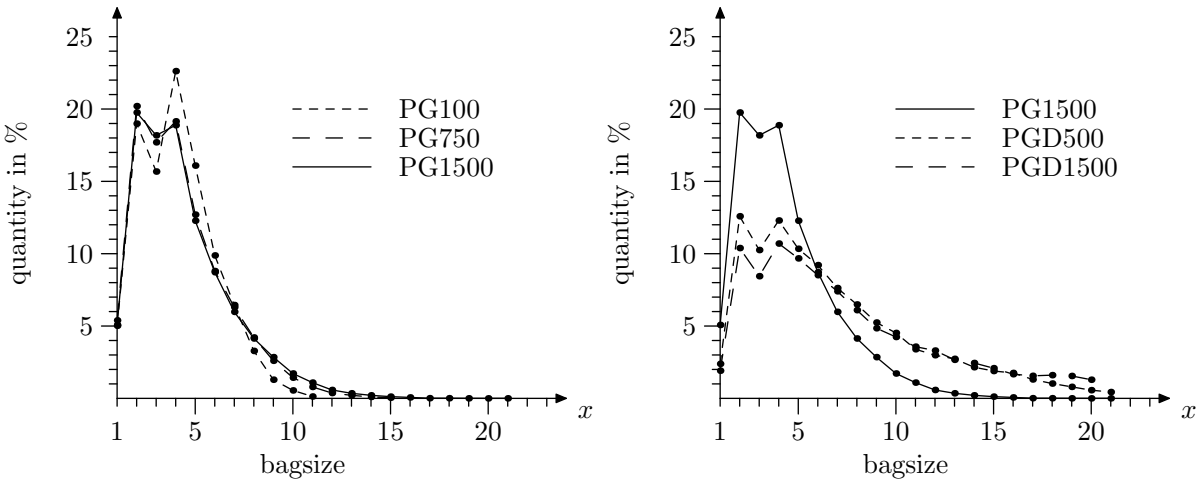


Figure 14: Distribution of sizes of bags in tree decompositions obtained by phase I.

The second block of rows in Table 1 summarizes the corresponding values averaged over each sample set PGn individually. In addition, for each input graph, we investigated the distribution of the various sizes of different bags that appeared in the tree decomposition. More precisely, for each graph G and the tree decomposition $\mathfrak{X}$ obtained, we explored the percentage of bags having size s (where $s \in [1, tw(\mathfrak{X}) + 1]$). This distribution is fundamental for the running time of phase II of the algorithm. The left-hand diagram in Fig. 14 shows this distribution averaged over the graphs of selected sample sets PGn.

Discussion. We comment on several aspects of this part of the algorithm. Note that the treewidth obtained by phase I (see left-hand diagram of Fig. 15) is much below the worst case upper bound derived in [4]. Namely, the upper bound given there was (in terms of the size of a minimum vertex cover)

$$\mathrm{tw}(G) \quad \leq \quad 4\sqrt{3\,\mathrm{vc}(G)},$$

where $\mathrm{vc}(G)$ denotes the size of a minimum vertex cover of the given graph G. As an example, take the sample PG750. Here, the average size of a minimum vertex cover is around 342 (see row "size of VC" in Table 1), the average treewidth obtained by phase I, however, is 10.3 which is by a *factor* 12.4 (sic!) less than the worst case upper bound value $4\sqrt{3 \cdot 342} \approx 128$. This comparison yields an even more drastic gap between worst case upper bound and practice in terms of Dominating Set. Here, the upper bound given in [2, 4] is

$$\mathrm{tw}(G) \quad \leq \quad 6\sqrt{34\,\mathrm{ds}(G)},$$

where $\mathrm{ds}(G)$ denotes the size of a minimum dominating set of the given graph G.

Note that in our setting the random input graphs, in general, turned out to have few layers only. As a consequence, in order to construct a tree decomposition, the algorithm is not forced to firstly execute the step of "layerwisely" separating the graph (see the description of the algorithm in Section 2 and [4] for details). Hence, it directly uses the algorithm for graphs of bounded outerplanarity (see the long version of [2] or [10, Theorem 83]). Still, it is remarkable, that the worst case upper bound $\mathrm{tw}(G) \leq 3\,\mathrm{out}(G, \phi) - 1$,

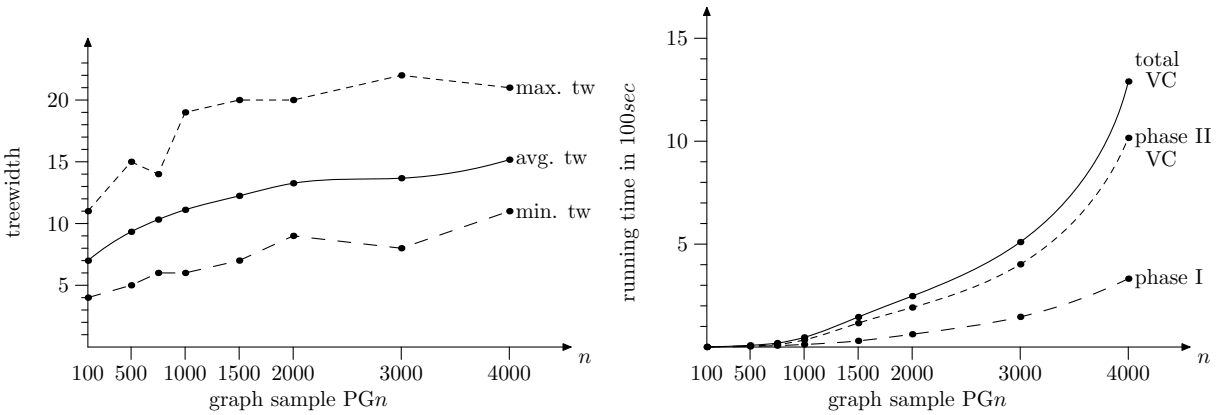


Figure 15: Treewidth and running time. Left diagram illustrates treewidth values obtained by phase I for the graph samples PGn. Right diagram illustrates the total running time of our algorithm for VERTEX COVER split into the two phases of the algorithm.

for a plane graph (G, ϕ) with outerplanarity $\text{out}(G, \phi)$ is too pessimistic. Take again, e.g., the sample PG750 where the average treewidth is 10.3, which is by a factor 1.46 lower than $3 \cdot 5.36 - 1 \approx 15$.

The most remarkable observation is the *structure* of the tree decompositions obtained by this phase of the algorithm. By this we mean, on the one hand, the distribution of the sizes of the bags and, on the other hand, the size of the tree itself. As to the first point, it is interesting that the average size of the bags in the tree decompositions in *all* of the samples PGn is around 4.5 (seemingly independent of the width of the tree decompositions and the size of the input graphs). Also, the variance of the bagsize distribution is very low. To illustrate this, consider Fig. 14, which shows this distribution for various sample sets PGn. Note that the size of a very high percentage of the bags is in the range of 1–6, and only few bags are of big size (determining the width of the tree decomposition).

Also, the number of nodes in the tree decomposition is quite low. Recall that the algorithm (see the long version of [2] or [10, Theorem 83]) for constructing the tree decomposition $\mathfrak{X}$ of a given graph G is based on finding "in a layerwise fashion" a suitable spanning tree T_S for G (more precisely for a supergraph $\hat{G}$ of degree less than four). Then, the tree T of $\mathfrak{X}$ basically is given by a subdivision[2] of T_S. Hence, for an n-vertex graph G we would expect at least $2n - 1$ bags in $\mathfrak{X}$. However, the tree decompositions in average turn out to have only around $0.75n$ many bags. This improvement by a factor of 2.7 over the expected number of bags is due to the `TD.reduce()` function (see Section 4). Both, the distribution of the bagsizes and the number of bags have a direct influence on the running time of phase II of the algorithm.

5.3 Phase II: Dynamic Programming on Tree Decompositions

The second phase of the algorithm is a problem-dependent dynamic programming approach. Since our current software package does not (yet) implement the ideas of [7, 8]

[2]Additional nodes are put on every edge of T_S.

and ourselves to reduce memory requirements,[3] we performed this second phase only on a subset of our original graph samples. In the case of VERTEX COVER (where the space requirement to store all tables is $O(2^{\mathrm{tw}}N)$, if tw is the treewidth of the underlying tree decomposition and N is the number of its bags), we restricted ourselves to those graphs for which phase I yields a tree decomposition of width at most 17; in the case of DOMINATING SET (where the space requirement is $O(3^{\mathrm{tw}}N)$), we only considered graphs with decompositions of width at most 9.

Evaluation. For this part of the algorithm, the following figures were measured:

- *size of subsample* (VC/ DS)*:* number of graphs in PGn that were considered in phase II, i.e., that had treewidth at most 17 in the case of VERTEX COVER (VC) and at most 9 in the case of DOMINATING SET (DS).
- *time phase II* (VC/ DS)*:* time (in seconds) needed to perform phase II of the algorithm in the case of VERTEX COVER/DOMINATING SET on the sample subset;
- *total time* (VC/ DS)*:* time (in seconds) needed to perform phase I and phase II of the algorithm in the case of VERTEX COVER/DOMINATING SET on the sample subset;
- *size of* VC/ DS*:* size of minimum VERTEX COVER/ DOMINATING SET of the sample subset.

Discussion. The time needed to perform the dynamic programming can be determined by considering the distribution of the bagsizes of the tree decomposition obtained by phase I (see Subsection 5.2). More precisely, if, for a given tree decomposition $\mathfrak{X}$ with N nodes, bag(s) (where $s \in [1, tw(\mathfrak{X}) + 1]$) denotes the number of bags having size s, then the running time for phase II basically is $\sum_{s=1}^{tw(\mathfrak{X})+1} \mathrm{bag}(s) \cdot c^s$. Here, c is a constant depending on the problem (for VERTEX COVER, $c = 2$; and for DOMINATING SET, $c = 4$ [6]). Now, clearly, both, the distribution "bag" and the value $N = \sum_{s=1}^{tw(\mathfrak{X})+1} \mathrm{bag}(s)$ do have direct influence on this running time. Together with the observations to be made in Fig. 14, this gives a plausible explanation for the good running times we obtained.

Note that we found vertex covers of average size 1325 in PG3000 in less than 9 minutes average time. Compare this running time with the running time of more than 10^{60} years, which is given by the worst case upper bound from Theorem 2, when plugging in $k = 1325$, $n = 3000$ and when assuming a type of machine we used. To put it the other way round, assuming the same type of machine, the worst case upper bound from Theorem 2 suggests that in 9 minutes we can compute a vertex cover of size 16 only.

5.4 Alternative Random Graphs

Besides the sample sets PGn that were created in a purely combinatorial way using the LEDA function `void random_planar_graph(graph& G,int n,int m)`, we also dealt with a modified random graph generation. The key motivation here was to generate sample sets of graphs with more layers than those in PGn. We achieved this by generating

[3] In our current version, all tables for the dynamic programming are kept in main memory at the same time.

sample sets PGDn (short for "planar graphs Delaunay"). In contrast to PGn, a different, purely geometric procedure was used in order to yield a maximal planar graph of order n. A vertex set of size n was randomly placed in the plane, and then some Delaunay triangulation was computed, from which edges were removed at random. The corresponding embedding (inherited by the Delaunay triangulation) indeed proved to have more layers and, due to the higher "degree of cyclicity" of graphs obtained in such a way, both the absolute treewidth and the average size of the bags of the decomposition obtained by phase I were higher. To give a comparison the last block of rows in Table 1 illustrates the main differences between PGn and PGDn:

- *size of sample:* number of random planar graphs in sample PGDn constructed by the Delaunay triangulation;
- *# layers:* average number of layers of graphs in PGDn using the embedding inherited by the Delaunay triangulation;
- *max. degree:* average maximum degree of graphs in PGDn;
- *avg. degree:* average "average degree" of graphs in PGDn;
- *treewidth:* width of tree decomposition obtained;
- *avg. bagsize:* average size of the bags of the tree decomposition;
- *variance of bagsize:* variance of the size of the bags of the tree decomposition.

Note that the number of layers in PGD1500 is 10.15, which is compared to the number of layers in PG1500 almost twice as high. Hence, the treewidth is almost three times bigger. But we discovered first pieces of circumstantial evidence that the application of the "Layerwise Separation Property" could be improved (see Section 6 for details), so the width of the tree decomposition of graphs with many layers, like in PGDn, might decrease. Also, the distribution of the bags tended to have higher percentage of large bags (see the right-hand diagram in Fig. 14 for a comparison to the sample sets PGn).

Finally, we remark that test sets created by other geometric generating schemes (for example, using maximal planar graphs that were triangulated by a sweeping algorithm instead of a Delaunay triangulation) resulted in similar findings than those of the sample sets PGn.

6 Heuristic Improvements

So far this implementation has been tested for a huge number of graphs. Still, there are quite many improvements foreseeable. In this section, we want to sketch some of them. The improvements intend, on the one hand, to decrease the width of the tree decomposition in phase I, on the other hand to reduce the memory requirement in phase II.

Distribution of separator vertices. The tests in Section 5 were executed with separators of size zero, because otherwise the treewidth would become too big: As implemented in `tree_decomp`, merging two partial tree decompositions, the separator vertices are put in every bag. So far, for the partial tree decompositions $\mathfrak{X}_i$, $i \in \{1, \ldots, n\}$, the treewidth

of the global tree decomposition $\mathfrak{X}$ will be exactly:

$$\max_{i\in\{1,\ldots,n\}}(tw(\mathfrak{X}_i)+ \mid S_{i-1}\cup S_i \mid)$$

(see Section 2 and [4] for details).
As the tests showed, using separators is still inefficient. The resulting treewidth is bigger than the treewidth that was obtained without separating the graph at all and simply applying Bodlaender's algorithm (which produces a tree decomposition of width at most $3r$ for an r-outerplanar graph). One improvement might consist in a more efficient distribution of the separator vertices: Only add a separator vertex n to the bags, which include at least one neighbour of n in the original graph. Then make the tree decomposition consistent, so that the tree decomposition properties hold (see Fig. 16 for an example for this improvement and Fig. 17 for an example for a determined tree decomposition without using this improvement.)
Using the notation above, this might imply the following lower bound on the treewidth of the global tree decomposition:

$$\max_{i\in\{1,\ldots,n\}}(tw(\mathfrak{X}_i)).$$

Note that the separator vertices may not influence the size of the treewidth (see Fig. 16 lower diagrams).

Different embeddings. Another way to decrease the width of the tree decomposion might be to run the program on different plane embeddings in order to reduce the number of layers. However, this seems to be not very efficient: we tested two different embeddings for some graphs. The outerplanarity of these embeddings differed by at most two, in many cases, it was equal.

Decreasing memory requirement. In phase II, the goal should be to cope with bigger treewidths. Hence, we should decrease the memory requirement. As can be seen from the tests, the running time is no obstruction in contrast to space. The current state is that, for each bag, the bag table is generated and then all rows are deleted which lead to invalid solutions. It is important not to create the whole bag tables, but to create bag tables without illegal rows right from the beginning (although the running time might suffer from this).

Memory allocation. All bag tables are created and kept in the memory during the whole algorithm. This is due to the fact that one need to keep in mind all possible solutions for the underlying graph problem until reaching the root vertex of the tree decomposition tree T. To avoid this waste of memory one might only keep the bag tables in the memory that are actually needed. This means: define an ordering (here: postorder) of the vertices of T. Go through T in this order, create bag tables if not yet existing, and delete a bag table of a vertex of T after the comparison with the bag table of the parent vertex has been carried out. In order to minimize the number of "open" bag tables, we

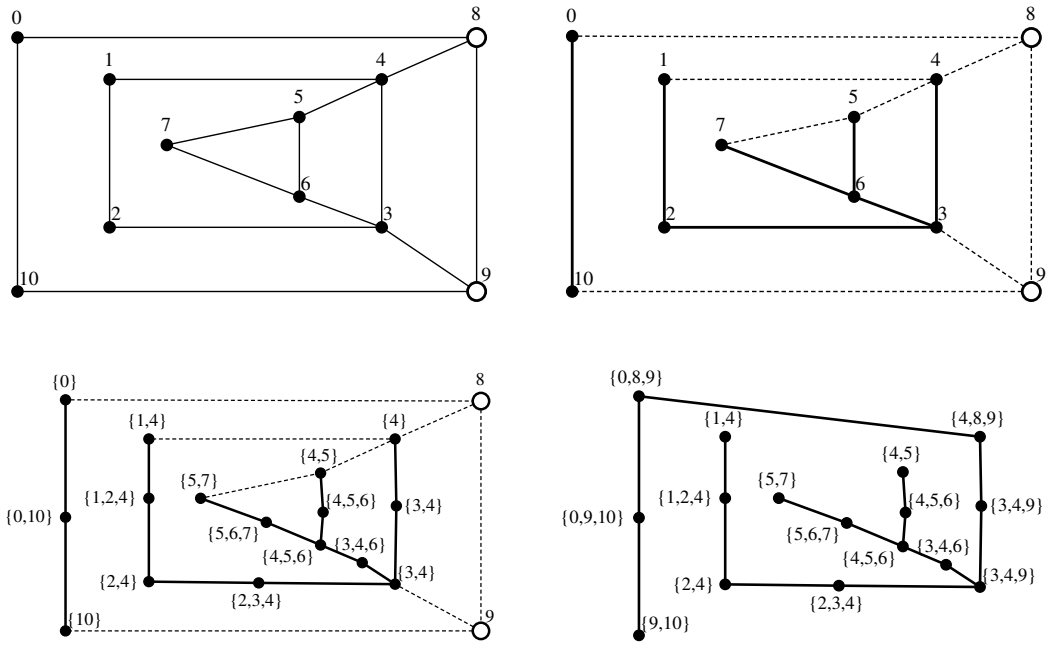


Figure 16: The graph on the upper left side has the separator vertices 8 and 9. The bold edges on the upper right side illustrate a spanning forest after taking out vertices 8 and 9. Lower left: The algorithm creates two partial tree decompositions using the spanning forest of the upper right diagram. Lower right: Using the improvement the global tree decomposition has treewidth two. Otherwise it would have had treewidth four, see Fig. 17.

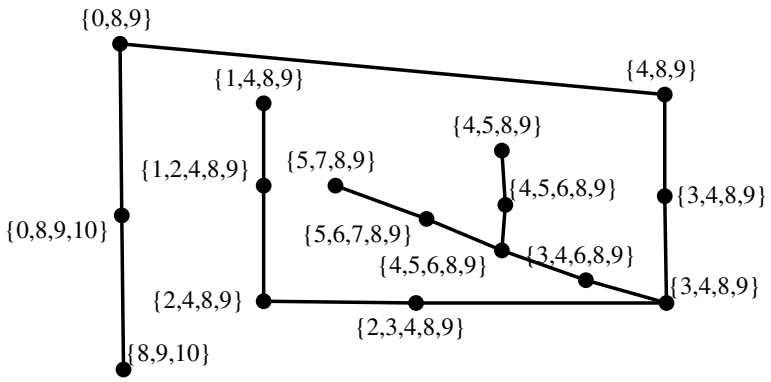


Figure 17: Without the improvement, the separator vertices are put in every bag. This implies the treewidth four.

could in addition use the algorithm suggested in [7]. The important thing will be, to have an efficient data structure for the bag pointers.

Using the decision problem. Another improvement will be to use the decision problem for the different graph problems (i.e., only output is the size of the optimal VERTEX COVER or DOMINATING SET). Hence, one does not have to keep the pointers and the memory requirement will decrease immediately. For example, as can be seen from the tests, the average bagsize of a tree decomposition with 3000 bags is 5. Then each row in the bag table of the root vertex may have the memory requirement of about 2 kilobyte. By way of contrast, the decision problem will only need about 80 bytes per row in any bag table. We may use the decision problem in order to solve the problem constructively with less memory requirement. The idea is to first run the decision problem algorithm, in order to determine the optimal solution size $vc(G)$ or $ds(G)$, and then run the constructive algorithm while deleting all rows in the bag tables, the minimum of which already does exceed the value $vc(G)$ or $ds(G)$.
Another improvement, using the decision problem, would be the following. First, run the decision problem algorithm. Determine a row r_{root} in the bag table of the root vertex, with which the optimal solution size $vc(G)$ or $ds(G)$ is obtained. Then, run the decision problem algorithm again, but only until the child vertices of the root vertex are reached. Then save the rows $r_{t_1}, \ldots, r_{t_i}$ in the bag tables $t_1, \ldots, t_i$ of the child vertices which yield the minimum for r_{root} in the updating process. Repeat this for the vertices one depth below until the leaves are reached. The memory requirement will have the size of the memory required for the decision problem algorithm. Since these steps are repeated at most tree depth times, the worst case running time will only be $O(d \cdot T)$, where d is the depth of the tree decomposition tree and T is the running time for the decision problem algorithm.

7 Conclusion

This work dealt with all phases of an algorithm design for tree decomposition techniques, starting with the theoretical background, over the implementation, the experimental evaluation up to some improvement possibilities. The main part of this work is the documentation. Its aim is to help to get acquainted with the program and to help to find some more improvements. Concerning improvements, one question is, how far can we go? When is the memory requirement definitively too big, when will the running time exceed any reasonable limits?
As one may conclude from Section 5, the program suffers from the memory requirement and not from the running time.
If there were only one bag table of type `list<two_tuple<int, int> >`, where one element of that list requires about 80 bits (including the links of such a list), and if the main memory was about 720 MB, then this bag table could have about 2^{25} elements. This means, that solving the decision problem for VERTEX COVER treewidth 24 would already exceed the memory. However not all elements of such bag tables are needed. Hence, the

bag table turns out to be much smaller. We indicated that solving the decision problem on random graphs for treewidth more than 29 is possible, using the same machine.
In future work we will try to upgrade the code, by, e.g., testing different data types and structures, or exchanging several subprocedures.
Moreover we might use other algorithmic ideas. Using problem kernel reduction (like Nemhauser-Trotter), the program copes now with bigger graphs. Last, but not least we will further ease the use of the software, e.g., by also providing meta-information such as expected remaining running time during the execution.

References

[1] J. Alber. Exact Algorithms for NP-hard Problems on Planar and Related Graphs: Design, Analysis, and Implementation. Dissertation, Universität Tübingen, January 2003.

[2] J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica,* 33(4):461-493, 2002.

[3] J. Alber, F. Dorn, and R. Niedermeier. Empirical Evaluation of a Tree Decomposition Based Algorithm for Vertex Cover on Planar Graphs. To appear in Discrete Applied Mathematics (Elsevier Science).

[4] J. Alber, H. Fernau, and R. Niedermeier. Parameterized complexity: exponential speed-up for planar graph problems. In *Proceedings 28th ICALP*, Springer-Verlag LNCS 2076, pp. 261–272, 2001.
Long version available as Technical Report TR01-023, Electronic Colloquium on Computational Complexity (ECCC), Trier, March 2001.

[5] J. Alber, J. Gramm, and R. Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229: 3–27, 2001.

[6] J. Alber and R. Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *Proceedings 5th LATIN 2002*, Springer-Verlag LNCS 2286, pp. 613-627, 2002.

[7] B. Aspvall, A. Proskurowski, and J. A.Telle. Memory requirements for table computations in partial k-tree algorithms. *Algorithmica*, 27:382–394, 2000.

[8] M. W. Bern, E. L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposbale graphs. *Journal of Algorithms*, 8:216–235, 1987.

[9] H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Proceedings 22nd MFCS'97*, Springer-Verlag LNCS 1295, pp. 19–36, 1997.

[10] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209:1–45, 1998.

[11] V. Bouchitté, D. Kratsch, H. Müller, and I. Todinca. On treewidth approximations. Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW'01).

[12] T. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[13] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer-Verlag, 1999.

[14] M. R. Fellows. Parameterized complexity: the main ideas and some research frontiers. In *Proceedings 12th ISAAC 2001*, Springer-Verlag LNCS 2223, pp.291-307, 2001.

[15] K. Mehlhorn and S. Näher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, 1999.

[16] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. The LEDA User Manual, Version 4.2.

[17] J. A. Telle and A. Proskurowski. Practical algorithms on partial k-trees with an application to domination-like problems. In *Proceedings 3rd WADS*, Springer-Verlag LNCS 709, pp. 610–621, 1993.

[18] J. A. Telle and A. Proskurowski. Algorithms for vertex partitioning problems on partial k-trees. *SIAM Journal on Discrete Mathematics*, 10(4):529–550, 1997.