Rolf Niedermeier.
Invitation to fixed-parameter algorithms.

10

TREE DECOMPOSITIONS OF GRAPHS

Many hard graph problems become easy when restricted to trees. For instance, VERTEX COVER and DOMINATING SET can be solved in linear time when restricted to trees: start at the leaves and perform a straightforward bottom-up approach. Based on this fact, one would naturally like to find out what makes trees such an algorithmically nice class of graphs—for the moment ignoring the "exceptions" of *NP*-complete problems such as MULTICUT IN TREES (Section 1.3) and MULTICOMMODITY DEMAND FLOW IN TREES (Section 9.4)—and whether and how these properties can be extended to more general classes of graphs. These considerations lead to the following central question.

How "tree-like" is a given graph?

This question is the cradle of the concept of tree decompositions of graphs, introduced by Neil Robertson and Paul D. Seymour about twenty years ago. Tree decompositions nowadays play a central role in algorithmic graph theory. In this chapter, which is far from giving an exhaustive presentation, we will survey several important aspects of tree decompositions of graphs and their algorithmic use with respect to fixed-parameter tractability.

The notion of *tree decomposition* and the related *treewidth* measure—the smaller the treewidth number the more tree-like the graph is—are introduced as a kind of compromise between the generality of graphs and the algorithmic feasibility of trees. In a nutshell, tree decompositions of small width demonstrate algorithmic tractability—in our sense, often fixed-parameter tractability—for many problems on graphs that are "almost" trees. Needless to say, there are several real-world applications of tree decompositions of graphs, ranging from compiler optimization and natural language processing over expert systems and probabilistic inference to telecommunication network design and frequency assignments, to name just a few.

In the following, we begin by introducing basic definitions and facts about tree decompositions. After that, we discuss how to construct a tree decomposition for a given graph, paying special attention to the case of planar graphs. We continue by illustrating the algorithmic use of tree decompositions once found; dynamic programming is again the key technique here and we exhibit in detailed examples concerning VERTEX COVER and DOMINATING SET. In addition, we present monadic second-order logic as a classification tool for deciding on fixed-parameter tractability of graph problems with respect to the parameter treewidth. Finally, we briefly discuss other width metrics for graphs such as pathwidth, local treewidth, and branchwidth, and we conclude by summarizing

the contributions of this chapter. It must be emphasized here, however, that tree decompositions and related concepts constitute a very deep and far-reaching element of algorithmic graph theory that deserves treatment in a book on its own. Thus this chapter only scratches the surface of this important and prospective field.

10.1 Basic definitions and facts

This section is based on the following, at first sight somewhat technical, definition.

Definition 10.1 *Let $G = (V, E)$ be a graph. A* tree decomposition *of G is a pair $\langle \{X_i \mid i \in I\}, T\rangle$ 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 $\{u, v\} \subseteq X_i$; and*
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 *of $\langle \{X_i \mid i \in I\}, T\rangle$ equals $\max\{|X_i| \mid i \in I\} - 1$. The* treewidth *of G is the minimum k such that G has a tree decomposition of width k.*

A clique of n vertices has treewidth $n-1$. The corresponding tree decomposition trivially consists of one bag containing all graph vertices. In fact, no tree decomposition with smaller width is attainable. More generally, it is known that every complete subgraph of a graph G is completely "contained" in a bag of G's tree decomposition. By way of contrast, a tree has treewidth 1 and the bags of the corresponding tree decomposition are simply the two-element vertex sets formed by the edges of the tree. Note that the third, algorithmically most important, requirement in Definition 10.1 is fulfilled in this way. Due to its importance in dynamic programming this third condition is also called *consistency property*. An equivalent formulation of this property is to demand that for every graph vertex v, all bags containing v form a connected subtree. Observe that we have already encountered the consistency property—phrased somewhat differently—in Definition 9.6 (Section 9.5) when introducing tree-like subset collections. The subsets there correspond one-to-one with the bags here. Here and there it is essential for making dynamic programming techniques applicable. Finally, there are several equivalent notions for tree decompositions; amongst others, graphs of treewidth at most k are known as *partial k-trees*. Figure 10.1 shows a graph together with an optimal tree decomposition of width two.

Importantly, an $\ell \times \ell$-grid graph has treewidth ℓ. The upper bound can be easily shown by going—in a row-by-row manner—through the grid, always taking one more vertex from the next row and deleting one from the previous row when constructing the bags of the tree decomposition. Actually, the underlying decomposition tree is only a path, and it can be shown that this tree decomposition is optimal, that is, it yields minimum width. The important consequence of this is, however, that graphs that "contain" large grids have large treewidth.

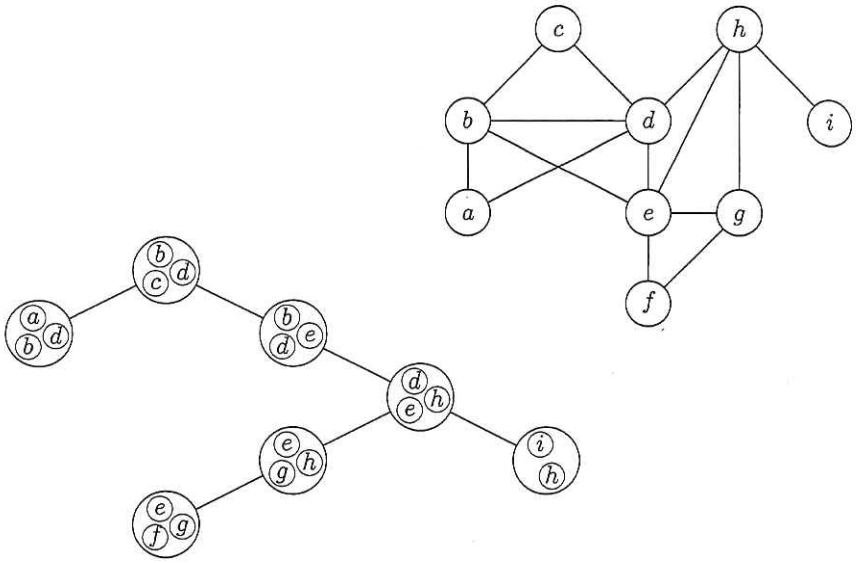

FIG. 10.1. A graph together with a tree decomposition of width 2. Observe that—as demanded by the consistency property—each graph vertex induces a subtree in the decomposition tree.

In particular, because grid graphs are planar it follows that planar graphs in general will not have small treewidth.

There is a very helpful and intuitively appealing characterization of tree decompositions in terms of a game. Consider the following *robber–cop game.* The robber stands on a graph vertex and, at any time, he can run at arbitrary speed to any other vertex of a graph as long as there is a path connecting both. He is not permitted to run through a cop, though. A cop, at any time, either stands at a graph vertex or is in a helicopter (that is, he is above the game board). The goal is to land a helicopter on the vertex occupied by the robber. Note that, due to the use of helicopters, cops are not constrained to move along graph edges. The robber can see a helicopter approaching its landing vertex and he may run to a new vertex before the helicopter actually lands. Thus, for a set of cops the goal is to occupy all vertices adjacent to the robber's vertex and to land one more remaining helicopter on the robber's place. The treewidth of the graph is then simply the minimum number of cops needed to catch a robber minus one.

A tree decomposition with a particularly simple structure is given by the following. Its usefulness will be exhibited when solving problems by dynamic programming on tree decompositions, as shown in Section 10.4.

Definition 10.2 *A tree decomposition* $\langle \{X_i \mid i \in I\}, T \rangle$ *is called a* nice tree decomposition *if the following conditions are satisfied:*

1. *Every node of the tree T has at most two children.*
2. *If a node i has two children j and k, then $X_i = X_j = X_k$; in this case, i is called a* JOIN NODE.
3. *If a node i has one child j, then one of the following situations must hold*
   (a) $|X_i| = |X_j| + 1$ *and* $X_j \subset X_i$; *in this case, i is called an* INTRODUCE NODE, *or*
   (b) $|X_i| = |X_j| - 1$ *and* $X_i \subset X_j$; *in this case, i is called a* FORGET NODE.

It is not hard to transform a given tree decomposition into a nice tree decomposition. More precisely, without stating the proof, the following result holds.

Lemma 10.3 *Given a width-k and n-nodes tree decomposition of a graph G, one can find a width-k and $O(n)$-nodes nice tree decomposition of G in $O(n)$ time.*

Tree decompositions of graphs are connected to another central concept in algorithmic graph theory: *graph separators* are vertex sets whose removal from the graph separates the graph into two or more connected components.

Definition 10.4 *Let $G = (V, E)$ be a connected graph. A subset $S \subseteq V$ is called a* separator *of G if the subgraph $G[V \setminus S]$ is disconnected.*

Actually, each bag of a tree decomposition forms a separator of the corresponding graph. Here, however, we are more interested in the reverse direction, that is, constructing tree decompositions from graph separators. The fundamental idea is to find small separators of the graph and to merge tree decompositions of the resulting subgraphs using the separator sets as "interfaces".

For any given separator splitting a graph into different components, we obtain a simple upper bound for the treewidth of this graph which depends on the size of the separator and the treewidth of the resulting components.

Proposition 10.5 *If a connected graph can be decomposed into components of treewidth of at most t by means of a separator of size s, then the whole graph has treewidth of at most $t + s$.*

Proof The separator splits the graph into different components. Suppose that we are given the tree decompositions of these components of width at most t. The goal is to construct a tree decomposition for the original graph. This can be achieved by firstly adding the separator to every bag in each of these given tree decompositions. In a second step, add some arbitrary connections preserving acyclicity between the trees corresponding to the components. It is straightforward to check that this forms a tree decomposition of the whole graph of width at most $t + s$. □

10.2 On the construction of tree decompositions

Constructing a tree decomposition of minimum width for a given graph is a difficult task. In fact, the subproblem to determine, given a graph G and an

integer k, whether the treewidth of G is at most k is NP-complete. For several special graph classes (such as bipartite graphs or graphs of maximum degree nine) the problem remains NP-complete, whereas for others (such as chordal graphs or permutation graphs) it is polynomial-time solvable. As to the question of whether this problem is fixed-parameter tractable with respect to the width parameter k, the answer is positive. Hans L. Bodlaender in 1996 published a "linear-time-FPT" algorithm—it runs in linear time when the parameter k is constant. Unfortunately, the hidden constant factor is huge and seems too large for practical purposes. Thus it is a major open question whether there is a significantly better fixed-parameter algorithm. Another longstanding open problem in this context refers to the class of planar graphs. Whereas we learned that the determination of treewidth is NP-complete for general graphs, it is open whether this also holds for planar graphs or whether it is polynomial-time solvable.

To apply the concept of tree decompositions in practice, as a rule heuristic approaches are employed for their construction. Although these methods do not guarantee optimal tree decompositions, their output is often good enough for the desired application. For instance, there is a relatively simple and efficient ratio-4 approximation algorithm for constructing tree decompositions—the obtained treewidth is at most a factor of 4 from the optimum value. This approximation algorithm is a fixed-parameter algorithm in the sense that its running time is exponential in the treewidth. It is a longstanding open problem whether there exists a polynomial-time approximation algorithm for treewidth with a constant approximation factor. To describe general tree decomposition construction methods in detail is beyond the scope of this book. Roughly speaking, many algorithms and, in particular, heuristics for tree decomposition finding are based upon the same principle:

1. Try to find a linear ordering of the vertices (which allows a one-to-one assignment of numbers to them) such that the higher numbered neighbors of every vertex form a clique—in other words, the graph is chordal. Then an optimal tree decomposition can be found using this so-called *perfect elimination scheme*. In this case, the graph can be interpreted as the "intersection graph" of subtrees of trees, naturally leading to a tree decomposition.

2. In general, the ordering found is not a perfect elimination scheme as described before. Hence one has to run some "fill-in" procedure (making the graph chordal, which means that there is no induced cycle of length at least four without a chord) to "triangulate" the graph by adding edges between non-adjacent higher-numbered neighbors of every vertex. After this task is accomplished, again the triangulation is turned into a tree decomposition as indicated in the first step, also giving a tree decomposition for the original graph.

It is important to note, however, that although one usually obtains non-optimal tree decompositions in this way, using these tree decompositions in a

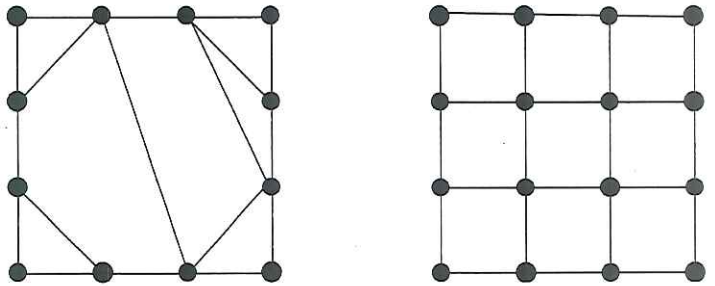

FIG. 10.2. An outerplanar graph (left) and a 2-outerplanar graph (right).

subsequent dynamic programming phase one can nevertheless obtain optimal solutions for the underlying graph problem, such as VERTEX COVER or DOMINATING SET, to be solved. The "only" price one has to pay for non-optimal tree decompositions is that the dynamic programming will consume more time and memory space because both resource requirements usually grow exponentially with respect to the width of the given tree decomposition.

In the context of fixed-parameter tractability with respect to solution size of the underlying graph problem, most algorithms based on tree decompositions deal with planar graphs or slight generalizations thereof. In these cases, efficient construction algorithms are known. That is why we devote the next section solely to planar graphs and how to find "problem-specific tree decompositions" for them. In a few words, by problem-specific we mean that if we know that a planar graph has a vertex cover or dominating set of size k, then we can make use of that to efficiently find a tree decomposition of width only depending on k.

10.3 Planar graphs

It is known that every n-vertex planar graph has treewidth $O(\sqrt{n})$ which, in a certain sense, can also be interpreted as the famous Planar Separator Theorem in disguise. Graph separators play an important role in the construction of tree decompositions. Moreover, in the case of planar graphs, there is a constructive way towards small separators. This is partially based on the "layer view" of planar graphs, expressed by the notion of r-outerplanarity.

Definition 10.6 *A plane graph G is called* outerplanar *if each vertex lies on the boundary of the outer face. A graph G is called* outerplanar *if it admits an outerplanar embedding in the plane.*

See the left-hand side of Figure 10.2 for an example outerplanar graph. The following generalization of the notion of outerplanarity is very helpful in our context.

Definition 10.7 1. *A plane embedding of a graph G is called* r-outerplanar *if, for $r = 1$, the embedding is outerplanar, and, for $r > 1$, inductively,*

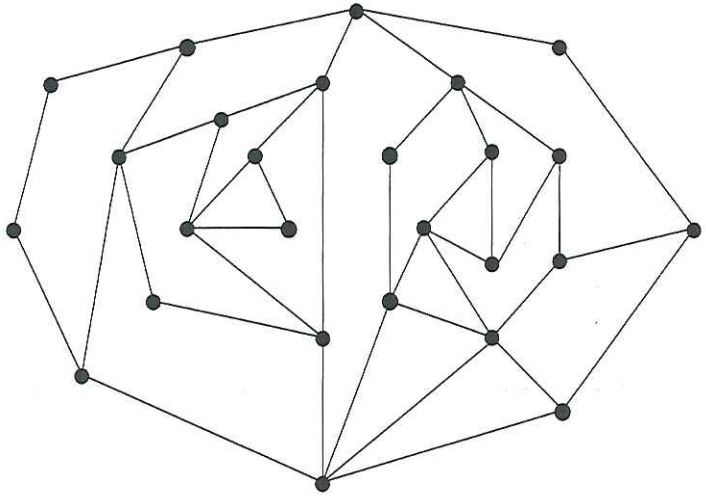

FIG. 10.3. A 3-outerplanar embedding of a graph.

when removing all vertices on the boundary of the outer face and their incident edges the embedding of the remaining subgraph is $(r-1)$*-outerplanar.*

2. *A graph* G *is called* r-outerplanar *if it admits an* r*-outerplanar embedding.*
3. *The smallest number* r *such that* G *is* r*-outerplanar is called the* outerplanarity number.

See the right-hand side of Figure 10.2 for an example 2-outerplanar graph, namely a 4×4-grid graph.

In this way, we may speak of the layers $L_1, \ldots, L_r$ of an embedding of an r-outerplanar graph.

Definition 10.8 *For a given* r*-outerplanar embedding of a graph* $G = (V, E)$*, we define the* ith layer L_i *inductively as follows. Layer* L_1 *consists of the vertices on the boundary of the outer face, and, for* $i > 1$*, the layer* L_i *is the set of vertices that lie on the boundary of the outer face in the embedding of the subgraph* $G[V \setminus (L_1 \cup \ldots \cup L_{i-1})]$.

Figure 10.3 shows a plane graph with three layers. Notably, by peeling off the outer layer one would obtain *two* connected components, both being 2-outerplanar.

Now, using the layer decomposition of plane graphs, there is an iterated version of Proposition 10.5.

Proposition 10.9 *Let* G *be a plane graph with layers* L_i*,* $i = 1, \ldots, r$*. For* $i = 1, \ldots, \ell$*, let* $\mathcal{L}_i$ *be a set of consecutive layers, that is,*

$$\mathcal{L}_i = \{L_{j_i}, L_{j_i+1}, \ldots, L_{j_i+n_i}\},$$

such that $\mathcal{L}_i \cap \mathcal{L}_{i'} = \emptyset$ *for all* $i \neq i'$*. Moreover, suppose that* G *can be decomposed into components, each of treewidth of at most* t*, by means of separators* $S_1, \ldots, S_\ell$*, where* $S_i \subseteq \bigcup_{L \in \mathcal{L}_i} L$ *for all* $i = 1, \ldots, \ell$*.*

Then G *has treewidth of at most* $t + 2s$*, where* $s = \max_{i=1,\ldots,\ell}\{|S_i|\}$.

Proof The proof uses the merging technique illustrated in Proposition 10.5: suppose that, without loss of generality, the sets $\mathcal{L}_i$ appear in successive order, that is, $j_i < j_{i+1}$. For each $i = 0, \ldots, \ell$, consider the component G_i of treewidth at most t which is cut out by the separators S_i and S_{i+1}—by default, we set $S_0 = S_{\ell+1} = \emptyset$. We add S_i and S_{i+1} to every bag in a given tree decomposition of G_i. In order to obtain a tree decomposition of G, we successively add an arbitrary connection between the trees T_i and T_{i+1} of the so-modified tree decompositions that correspond to the subgraphs G_i and G_{i+1}. □

Thus, for plane graphs the goal now can be set as follows: decompose the given graph into various "small" components by using "small" separators, construct a tree decomposition for each component, and get an overall tree decomposition by applying Proposition 10.9. It remains to show how to find these small separators and how to get the tree decompositions for the graph components. This is explained next.

As already mentioned, we aim at *problem-specific* tree decompositions. The reason for this is that in this way we can relate the solution size of the graph problem we want to solve with the size of the separators we can find in the underlying planar graph. In what follows, we proceed in two steps. To this end, the example graph problems we consider are VERTEX COVER and DOMINATING SET. Analogous considerations hold for many other graph problems.

1. In the first step we demonstrate that the vertex cover or domination number k and the treewidth of a planar graph are linearly related. We show this without explicit construction of graph separators.
2. In the second step, we show that the linear bound $O(k)$ from Step 1 can be improved to $O(\sqrt{k})$. Here, me make explicit use of graph separators.

With regard to the first step, one easily observes the following central relation between the vertex cover number and the domination number on the one side and the outerplanarity number of a planar graph on the other side.

Proposition 10.10 1. *If a planar graph* $G = (V, E)$ *has a size-*k *vertex cover then all plane embeddings of* G *can be at most* k*-outerplanar.*
2. *If a planar graph* $G = (V, E)$ *has a size-*k *dominating set then all plane embeddings of* G *can be at most* $3k$*-outerplanar.*

Proof We show only the result for DOMINATING SET. For a given crossing-free embedding of G in the plane, each vertex in the dominating set can dominate vertices from the previous, the next, or its own layer only. Hence each vertex in the dominating set can contribute to at most three new layers. □

To understand the techniques also used in the second step, it is helpful to consider the concept of a layer decomposition of an r-outerplanar embedding of

graph G. A *layer decomposition* of an r-outerplanar embedding of G is a forest of height $r-1$: the trees of the forest correspond to different connected components of G. The nodes correspond to the various layers.

One more result needed is the following relation between r-outerplanarity and treewidth. We skip the proof.

Theorem 10.11 *An r-outerplanar graph has treewidth of at most $3r-1$.*

The proof of Theorem 10.11 can be made constructive, so a tree decomposition of width at most $3r-1$ can be computed in polynomial time.

Proposition 10.10 and Theorem 10.11 immediately imply the following relationship between the domination number and the treewidth of a planar graph.

Corollary 10.12 *If a planar graph has a k-dominating set then its treewidth is at most $9k-1$.*

With regard to the second step mentioned above, the basic idea of constructing a tree decomposition of small width is the following. If the given graph has few layers, then use Theorem 10.11 directly. If not,

1. find small graph separators that decompose the graph into chunks of small outerplanarity,
2. apply Theorem 10.11 to these graph chunks, and
3. finally combine the tree decompositions of the various chunks into a big one for the overall graph using Proposition 10.9.

Next, we describe how to find these small graph separators.

It is clear that in a layer decomposition $(L_1, \ldots, L_r)$ of a given planar graph of outerplanarity r each layer L_i, $1 \leq i \leq r$, forms a separator. What makes the problem mathematically demanding is that the sizes of the layers L_i might be too large. Thus it remains to be shown that, nevertheless, small separators can be found in a layerwise fashion. To this end, one makes use of the special properties of the underlying parameterized graph problem. We focus on VERTEX COVER and DOMINATING SET to see how this works. Both problems possess problem kernels with a number of vertices linearly depending on the size of the solution set; see Sections 7.4 and 7.6. Hence in both cases we trivially have that $|\bigcup_{i=1}^{r} L_i| = O(k)$.

Theorem 10.13 *If a planar graph has a k-vertex cover or a k-dominating set, then its treewidth is bounded from above by $O(\sqrt{k})$.*

Proof Using the graph layers L_i as separators, go through the sequence of layers $L_1, L_2, L_3, \ldots$ and look for separators of size $s(k) := O(\sqrt{k})$. Due to $|\bigcup_{i=1}^{r} L_i| = O(k)$ such separators of size at most $s(k)$ must appear within each $n(k) := O(\sqrt{k})$ layers in the sequence. In this manner, we obtain a set of disjoint separators of size at most $s(k)$ each, such that any two consecutive separators from this set are at most $O(\sqrt{k})$ layers apart. Clearly, the separators chosen in this way fulfill the requirements in Proposition 10.9.

The components cut out in this way each have $O(\sqrt{k})$ layers, and hence their treewidth is bounded by $O(\sqrt{k})$ due to Theorem 10.11. Using Proposition 10.9, we can upperbound the treewidth of the planar graph by $O(\sqrt{k})$. □

The tree structure of the tree decomposition obtained in the above proof corresponds to the structure of the layer decomposition forest.

Remark 1 *Up to constant factors, the relation exhibited in Theorem 10.13 is optimal. This can be seen, for example, by considering a grid graph G_ℓ of size $\ell \times \ell$, that is, with ℓ^2 vertices and $2(l^2 - l)$ edges: it is known that the treewidth of G_ℓ is exactly ℓ and that a minimum vertex cover as well as a minimum dominating set for G_ℓ both consist of $\Theta(l^2)$ vertices.*

A mathematically more refined analysis than the above one—not making use of linear problem kernels as we did here but employing a more direct way of constructing the separators layerwisely—gives upper bounds $O(\sqrt{k})$ on the treewidths with somewhat smaller constants hidden in the O-notation. Still, however, these worst-case factors are fairly large (in the tens but not astronomical).

Summarizing, we join together the preceding considerations into an algorithm that constructs tree decompositions of width $O(\sqrt{k})$ in the case that we are given a planar graph that possesses a vertex cover or a dominating set of size at most k. It is important to note here that the tree decompositions are only constructed for reduced graphs that are obtained by the reduction to a problem kernel for the underlying parameterized problem (VERTEX COVER or DOMINATING SET in this context). The algorithm proceeds in the following steps.

1. Perform a reduction to a problem kernel that yields a reduced planar graph whose number of vertices is $O(k)$.
2. Embed the reduced planar graph $G = (V, E)$ crossing-free into the plane. Linear-time algorithms are known for that. Determine all layers $L_1, \ldots, L_r$ and the outerplanarity number r of this embedding. By default, we set $L_i = \emptyset$ for all $i \leq 0$ and $i > r$.
3. VERTEX COVER: If $r > k$, then exit (there exists no size-k vertex cover). This is justified by Proposition 10.10.
   DOMINATING SET: If $r > 3k$ then exit (there exists no size-k dominating set). This is justified by Proposition 10.10.
4. Find separators of size $O(\sqrt{k})$ according to the proof of Theorem 10.13.
5. Decompose the graph into subgraphs by removing all the graph separators found in the preceding step. Note that each of these subgraphs has outerplanarity $O(\sqrt{k})$.
6. Construct tree decompositions for all the subgraphs using Theorem 10.11. In this way, all subgraphs obtain tree decompositions of width $O(\sqrt{k})$.
7. Merge the tree decompositions of all subgraphs into a tree decomposition of the overall graph. To do so, use the tree decompositions of the subgraphs and the separators that generated these subgraphs (see fifth step above)

and apply the "separator merging technique" described in the proof of Proposition 10.9.

The above algorithm outline constructively shows how to obtain tree decompositions of width $O(\sqrt{k})$ for problems such as VERTEX COVER or DOMINATING SET in planar graphs parameterized by k. Clearly, to demonstrate the definite usefulness of constructing tree decompositions of graphs it remains to be shown how the underlying graph problem can be efficiently solved using dynamic programming on tree decompositions. This will be the topic of Sections 10.4 and 10.5.

What about the constant factors hidden in the O-notation used throughout the previous considerations? Note that the given presentation traded comprehensibility for exact determination of the considered running time and treewidth values. A thorough mathematical analysis can be found in the literature. As already indicated, the proven upper bounds involve quite high constant factors. For instance, based on the above approach and several more technical details the running times $O(2^{4\sqrt{3k}} \cdot n)$ and $O(2^{12\sqrt{34k}} \cdot n)$ were proven for VERTEX COVER and DOMINATING SET in planar graphs, respectively. The bounds are worst-case, however. Recent work has lowered the exponential bound for DOMINATING SET in planar graphs.

10.4 Dynamic programming for Vertex Cover

To understand the practical usefulness of tree decompositions, one has to learn how algorithmic techniques used for graph problems restricted to trees can be generalized to also work for graphs of bounded treewidth. The standard approach here is dynamic programming. Notably, making this dynamic programming as efficient and practical as possible is not only a question of running time but also of memory consumption. Finally, observe that our dynamic programming approaches provide optimal solutions for the problems considered. This also underpins the relevance of approximative or heuristic algorithms for constructing tree decompositions as long as the delivered treewidths remain small enough.

Typically, tree decomposition based algorithms proceed according to the following scheme in two stages:

1. Find a tree decomposition of bounded width for the input graph; and
2. solve the problem by dynamic programming on the tree decomposition.

So far in this chapter on tree decompositions, we have dealt with the first stage. Now, we describe how the second stage works. In fact, what we show is a fixed-parameter algorithm with respect to the parameter treewidth. Notably, the parameter is not the size of the vertex cover set but it is the width of the tree decomposition. This implies that we are able to solve VERTEX COVER for almost arbitrarily large graphs as long as their treewidth is small enough.

Theorem 10.14 *For a graph G with given tree decomposition $\langle\{X_i \mid i \in I\}, T\rangle$ an optimal vertex cover can be computed in $O(2^\omega \cdot \omega \cdot |I|)$ time. Here, ω denotes the width of the tree decomposition.*

Proof The basic idea is to check for all of the $|I|$ many bags each time for all of the at most $2^{|X_i|}$ possibilities to obtain a vertex cover for the subgraph $G[X_i]$ of G induced by the vertices from X_i. This information is stored in tables A_i, $i \in I$. Adjacent tables will be updated in a bottom-up process starting at the leaves of the decomposition tree. Each bag of the tree decomposition thus has a table associated with it. During this updating process it is guaranteed that the "local" solutions for each subgraph associated with a bag of the tree decomposition are combined into a "globally optimal" solution for the overall graph G.

The algorithmic details are as follows.

Step 0: For each bag $X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\}$, $|X_i| = n_i$, compute a table

$$A_i = \begin{array}{cccccc|c} x_{i_1} & x_{i_2} & \cdots & x_{i_{n_i-1}} & x_{i_{n_i}} & & m_i() \\ \hline 0 & 0 & \cdots & 0 & 0 & & \\ 0 & 0 & \cdots & 0 & 1 & & \\ 0 & 0 & \cdots & 1 & 0 & & \\ 0 & 0 & \cdots & 1 & 1 & & \\ & & \vdots & & & & \\ 1 & 1 & \ldots & 1 & 0 & & \\ 1 & 1 & \ldots & 1 & 1 & & \end{array} \quad \Big\} 2^{n_i}$$

The table consists of 2^{n_i} rows and $|n_i| + 1$ columns. Each row represents a so-called "coloring" of subgraph $G[X_i]$. By this we mean a 0-1 sequence of length n_i that determines which of the respective bag vertices from X_i should be put into the current vertex cover—this corresponds to value 1—and which should not—this corresponds to value 0. Formally, a coloring is a mapping

$$C_i : X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\} \to \{0, 1\}.$$

For each of the 2^{n_i} different possibilities for a coloring, the table has one further entry. This last column stores, for each specific coloring C_i, the number $m_i(C_i)$ of vertices of a minimal vertex cover that contains those vertices from X_i selected by the coloring C_i. More precisely, this means that it stores the value

$$\begin{aligned} m_i(C_i) := \min\{|V'| : V' \subseteq V \text{ is a vertex cover for } G, \\ \text{such that } v \in V' \text{ for all } v \in (C_i)^{-1}(1) \\ \text{and } v \notin V' \text{ for all } v \in (C_i)^{-1}(0)\}. \end{aligned}$$

Note that $(C_i)^{-1}(1)$ denotes the set of all vertices in X_i that are colored 1, and $(C_i)^{-1}(0)$ denotes all vertices in X_i that are colored 0 under coloring C_i. This value is determined by dynamic programming as described in Step 2 to follow.

Of course, not every possible coloring may lead to a vertex cover. Such a coloring is called *invalid*. To check whether or not a coloring C_i is *valid*,

for each edge $\{u, v\}$ of the subgraph $G[X_i]$ induced by X_i, consider $C_i(u)$ and $C_i(v)$. If there is at least one edge where $C_i(u) = C_i(v) = 0$ then the coloring is invalid; otherwise, it is valid.

Step 1: Table initialization.

For all tables X_i and each coloring $C_i : X_i \to \{0,1\}$ set

$$m_i(C_i) := \begin{cases} \left|(C_i)^{-1}(1)\right|, & \text{if } C_i \text{ is valid;} \\ +\infty, & \text{otherwise.} \end{cases}$$

Clearly, once when value $+\infty$ appears the corresponding "computation path" can be immediately stopped because no vertex cover of the whole graph can be computed in this way.

Step 2: Dynamic programming.

We now go through the decomposition tree T from the leaves to the root and compare the corresponding tables against each other.

Let $i \in I$ be the parent node of $j \in I$. We show how the table for X_i with m_i can be updated by the table for X_j with m_j. To this end, assume that

$$X_i = \{z_1, \ldots, z_s, v_1, \ldots, v_{t_i}\} \text{ and}$$
$$X_j = \{z_1, \ldots, z_s, u_1, \ldots, u_{t_j}\},$$

that is, $X_i \cap X_j = \{z_1, \ldots, z_s\}$.

Subsequently, by an *extension of a coloring* $C : W \to \{0,1\}$ (where $W \subseteq V$) we mean a coloring $\tilde{C} : \tilde{W} \to \{0,1\}$ with $\tilde{W} \supseteq W$ and $\tilde{C}$ restricted to ground set W yields C, in signs, $\tilde{C}|_W = C$. Then, for each possible coloring

$$C : \{z_1, \ldots, z_s\} \to \{0,1\}$$

and each extension $C_i : X_i \to \{0,1\}$ of C we set

$$m_i(C_i) := m_i(C_i) + \min\{\, m_j(C_j) \mid C_j : X_j \to \{0,1\} \text{ is an extension of } C\} - \left|C^{-1}(1)\right|.$$

This is the central instruction of the algorithm. In other words, what happens here is that the value of $m_i(C_i)$ grows by the minimum value for the vertex cover of the subgraph induced by all the vertices contained in the subtree rooted at j. Here, however, one has to take care of double counting with respect to vertices that are also contained in X_i, which is why $|C^{-1}(1)|$ is subtracted. In fact, we record some more (pointer) information in order to be able to construct a solution set efficiently in a subsequent traceback phase.

If a node $i \in V_T$ has several children $j_1, \ldots, j_l \in V_T$ then table A_i is successively updated against all tables $A_{j_1}, \ldots, A_{j_l}$ in the same way. All this is repeated until the root node is finally updated.

Step 3: Construction of a minimum vertex cover V'.

The size of V' is derived from the minimum entry of the last column of the root node table A_r. The coloring of the corresponding row shows which of the vertices of the "root bag" X_r are contained in V'. By recording during Step 2 how the respective minimum of each bag was determined by its "child values", one can easily compute all vertices of an optimal vertex cover by following the pointers mentioned in Step 2.

This concludes the description of the dynamic programming algorithm. It remains to show its correctness and its running time.

1. *Correctness of the algorithm.*

a) The first condition in Definition 10.1, that is, $V = \bigcup_{i \in I} X_i$, makes sure that every graph vertex is taken into account during the computation.

b) The second condition in Definition 10.1, that is, $\forall e \in E\ \exists i_0 \in I : e \in X_{i_0}$, makes sure that after the treatment of invalid colorings right after the initialization in Step 0, during the dynamic programming process only actual vertex covers are dealt with.

c) The third condition in Definition 10.1 guarantees the consistency of the dynamic programming. If a vertex $v \in V$ occurs in two different bags X_{i_1} and X_{i_2} then it is guaranteed that for the computed minimum vertex cover this vertex cannot receive different colors in the two respective rows in the tables A_{i_1} and A_{i_2}. This is handled in the bag of the least common ancestor i_0 of i_1 and i_2 in T.

2. *Running time of the algorithm.*

By keeping the tables sorted adequately, the comparison of a table A_j against a table A_i can be done in time proportional to the maximum table size, that is, $O(2^\omega \cdot \omega)$. For each edge $e \in E_T$ in tree T such a comparison has to be done, that is, the overall running time of the algorithm is $O(2^\omega \cdot \omega \cdot |I|)$. □

Combining Theorem 10.14 with Theorem 10.13 and the corresponding algorithm that constructs a tree decomposition (see Section 10.3) results in a fixed-parameter algorithm for VERTEX COVER IN PLANAR GRAPHS that provides an exponential speedup when compared with the depth-bounded search tree fixed-parameter algorithms for VERTEX COVER on general graphs in Section 8.3. Note, however, that now the constants in the O-term are significantly larger.

Corollary 10.15 VERTEX COVER IN PLANAR GRAPHS *can be solved in* $2^{O(\sqrt{k})} \cdot n$ *time, where* k *denotes the size of the vertex cover and* n *is the number of graph vertices.*

Doing a more refined, deep mathematical analysis, according to the literature the exponential factor in the statement of Corollary 10.15 can be upper-bounded by $2^{4.5\sqrt{k}}$.

10.5 Dynamic programming for Dominating Set

The basic technique for solving DOMINATING SET on a "tree-decomposed" graph is the same as for VERTEX COVER. We have already experienced in earlier parts of this book, however, that from a combinatorial point of view DOMINATING SET is a problem that is more elusive than VERTEX COVER. This also reflects in the larger overhead needed to solve DOMINATING SET efficiently via dynamic programming on tree decompositions. The very first observation is that we need three colors for the dynamic programming tables instead of only two as we had for VERTEX COVER: suppose that $G = (V, E)$ and $V = \{x_1, \ldots, x_n\}$. Assume that the vertices in the bags are given in increasing order when used as indices of the dynamic programming tables, that is, $X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\}$ with $i_1 \leq \ldots \leq i_{n_i}$, $1 \leq i \leq |I|$. We use *three* different "colors" that will be assigned to the vertices in the bag:

- "black": represented by 1, meaning that the vertex belongs to the dominating set;
- "white": represented by 0, meaning that the vertex is already dominated at the current stage of the algorithm; and
- "grey": represented by $\hat{0}$, meaning that, at the current stage of the algorithm, one is still asking for a domination of this vertex.

Again, mapping

$$C_i : \{x_{i_1}, \ldots, x_{i_{n_i}}\} \to \{0, \hat{0}, 1\}$$

will be called a *coloring* for the bag $X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\}$, and the *color* assigned to vertex x_{i_t} by C_i is given by $C_i(x_{i_t})$. Hence, a coloring can be represented as a vector $(C(x_{i_1}), \ldots, C(x_{i_{n_i}}))$.

For each bag X_i with $|X_i| = n_i$, in the same spirit as for VERTEX COVER we use a mapping

$$m_i : \{0, \hat{0}, 1\}^{n_i} \longrightarrow \mathbb{N} \cup \{+\infty\}.$$

For a coloring C_i, the value $m_i(C_i)$ stores how many vertices are needed for a minimum dominating set of the graph visited up to the current stage of the algorithm under the restriction that the color assigned to vertex x_{i_t} is $C_i(x_{i_t})$, $t = 1, \ldots, n_i$. We end up with tables of size 3^{n_i}. Now, by performing a table updating process analogous to the case for VERTEX COVER described in Section 10.4, it is not difficult to finally come up with an algorithm that solves DOMINATING SET on graphs with given tree decomposition of $|I|$ nodes and width ω in $O(9^\omega \cdot \omega \cdot |I|)$ time. The significant increase from base value 2 to 9 is due to the more complicated "dependence structure" in the combinatorics of DOMINATING SET when implemented in a basically straightforward way. Thus, comparing two tables A_i and A_j now takes $O(3^{|X_i|} \cdot 3^{|X_j|} \cdot \max\{|X_i|, |X_j|\}) = O(9^\omega \cdot \omega)$ time.

There is room for improvement, however. This needs some further definitions: to simplify matters, we identify colorings simply by their naturally corresponding vectors $\{0, \hat{0}, 1\}^{n_i}$. On the color set $\{0, \hat{0}, 1\}$, let $\prec$ be the partial ordering given by $\hat{0} \prec 0$ and $d \prec d$ for all $d \in \{0, \hat{0}, 1\}$. This ordering naturally extends to colorings: for $c = (c_1, \ldots, c_m), c' = (c'_1, \ldots, c'_m) \in \{0, \hat{0}, 1\}^m$, let $c \prec c'$ iff $c_t \prec c'_t$ for all $t = 1, \ldots, m$. It is essential for the improved dynamic programming that the mappings m_i are *monotonic* functions from $(\{0, \hat{0}, 1\}, \prec)$ to $(\mathbb{N} \cup \{+\infty\}, \leq)$; that is, for $c, c' \in \{0, \hat{0}, 1\}^{n_i}$, $c \prec c'$ implies that $m_i(c) \leq m_i(c')$. A coloring $c \in \{0, \hat{0}, 1\}^{n_i}$ is *locally invalid* for a bag X_i if

$$\big(\exists s \in \{1, \ldots, n_i\} : c_s = 0\big) \wedge \big(\nexists t \in \{1, \ldots, n_i\} : ((x_{i_t} \in N(x_{i_s})) \wedge (c_t = 1))\big).$$

In other words, a coloring is locally invalid if there is some vertex in the bag that is colored white but this color is not "justified" within the bag, that is, this vertex is not dominated by a vertex colored 1 within the bag. Note that a locally invalid coloring may still be a correct coloring if the white vertex whose color is not justified *within* the bag is dominated by a vertex from bags that have been considered earlier. Also, for a coloring $c = (c_1, \ldots, c_m) \in \{0, \hat{0}, 1\}^m$ and a color $d \in \{0, \hat{0}, 1\}$, we use the notation

$$\#_d(c) := |\{t \in \{1, \ldots, m\} \mid c_t = d\}|.$$

Now we are ready to describe the various steps of the improved dynamic programming. To make things easier, we subsequently assume that we work with a *nice* tree decomposition (see Definition 10.2 and Lemma 10.3 in Section 10.1).

Step 1: Table initialization.

For all tables X_i and each coloring $c \in \{0, \hat{0}, 1\}^{n_i}$ set

$$m_i(c) := \begin{cases} +\infty, & \text{if } c \text{ is locally invalid for } X_i, \\ \#_1(c), & \text{otherwise} \end{cases} \tag{10.1}$$

With this initialization step we make sure that only colorings are taken into consideration where an assignment of color 0 is justified.

Since the check for local invalidity takes $O(n_i)$ time, this step can be carried out in $O(3^{n_i} \cdot n_i)$ time.

Trivially, the mappings m_i of leaf bags are monotonic.

Step 2: Dynamic programming.

After the initialization, we visit the bags of our tree decomposition from the leaves to the root, evaluating the corresponding mappings in each step according to the following rules.

FORGET NODES: Suppose i is a FORGET NODE with child node j and suppose that $X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\}$. Without loss of generality—possibly after rearranging the vertices in j's bag X_j and the entries of m_j accordingly—we may assume that $X_j = \{x_{i_1}, \ldots, x_{i_{n_i}}, x\}$ for some graph vertex x not in X_i. Evaluate the mapping m_i of X_i as follows: For all colorings $c \in \{0, \hat{0}, 1\}^{n_i}$ set

$$m_i(c) := \min_{d \in \{0,1\}} \{m_j(c \times \{d\})\}. \tag{10.2}$$

Note that a coloring $c \times \{\hat{0}\}$ for X_j means that the vertex x is assigned color $\hat{0}$, that is, x is not yet dominated by a graph vertex. Since, by the

consistency property of tree decompositions, the vertex x will never appear in a bag for the rest of the algorithm, a coloring $c \times \{\hat{0}\}$ will not lead to a dominating set because vertex x is not dominated. That is why the minimum in the assignment (10.2) is taken over colors 1 and 0 only.

Clearly, the evaluations can be carried out in $O(3^{n_i} \cdot n_i)$ time. It is trivial to observe that the monotonicity of the mapping m_j implies that m_i also is monotonic.

INSERT NODES: Suppose that i is an INSERT NODE with child node j and suppose that $X_j = \{x_{j_1}, \ldots, x_{j_{n_j}}\}$. Without loss of generality—possibly after rearranging the vertices in X_i and the entries of A_i accordingly—we may assume that $X_i = \{x_{j_1}, \ldots, x_{j_{n_j}}, x\}$. Let

$$N(x) \cap X_j = \{x_{j_{p_1}}, \ldots, x_{j_{p_s}}\}$$

be the neighbors of the "introduced" vertex x which are contained in the bag X_i. We now define a function

$$\phi : \{0, \hat{0}, 1\}^{n_j} \to \{0, \hat{0}, 1\}^{n_j}$$

on the set of colorings of X_j. For $c = (c_1, \ldots, c_{n_j}) \in \{0, \hat{0}, 1\}^{n_j}$, we define $\phi(c) := (c'_1, \ldots, c'_{n_j})$ such that

$$c'_t = \begin{cases} \hat{0}, & \text{if } t \in \{p_1, \ldots, p_s\} \text{ and } c_t = 0, \\ c_t, & \text{otherwise.} \end{cases}$$

Then, evaluate the mapping m_i of X_i as follows: for all colorings $c = (c_1, \ldots, c_{n_j}) \in \{0, \hat{0}, 1\}^{n_j}$ set

$$m_i(c \times \{0\}) := m_j(c) \text{ if } x \text{ has a neighbor } x_{j_q} \in X_i \text{ with } c_q = 1; \quad (10.3)$$

$$m_i(c \times \{1\}) := m_j(\phi(c)) + 1; \quad (10.4)$$

$$m_i(c \times \{\hat{0}\}) := m_j(c). \quad (10.5)$$

Concerning the correctness of the assignments (10.3) and (10.4) we remark the following: it is clear that, if we assign color 0 to vertex x (assignment (10.3)), we again—as already done in the initializing assignment (10.1)—have to check whether this color can be justified at the current stage of the algorithm. Such a justification is given if and only if the coloring under examination already assigns a 1 to some neighbor of x in X_j resp. X_i. This is true, since the consistency property of tree decompositions implies that at the current stage of the dynamic programming process, x can only be dominated by a vertex in X_j (as checked in assignment (10.3)).

If we assign color 1 to vertex x (assignment (10.4)), we thereby dominate all vertices $\{x_{j_{p_1}}, \ldots, x_{j_{p_s}}\}$. Suppose now that we want to evaluate $m_i(c \times \{1\})$ and suppose that some of these vertices are assigned color 0 by c, say $c_{p'_1} = \ldots = c_{p'_q} = 0$, where $\{p'_1, \ldots, p'_q\} \subseteq \{p_1, \ldots, p_s\}$. Since the "1-assignment" of x already justifies the "0-values" of $c_{p'_1}, \ldots, c_{p'_q}$, and since our mapping m_j is monotonic, we obtain $m_i(c \times \{1\})$ by taking entry $m_j(c')$, where $c'_{p'_1} = \ldots = c'_{p'_q} = \hat{0}$, that is, $c' = \phi(c)$.

Since we need time $O(n_i)$ in order to check whether a coloring is locally invalid, the evaluation of m_i can be carried out in $O(3^{n_i} \cdot n_i) \subseteq O(4^\omega)$ time.

Considering assignments (10.4) and (10.5), it is easy to see that m_i is monotonic if m_j is monotonic.

JOIN NODES: Suppose i is a JOIN NODE with children j and k and suppose that $X_i = X_j = X_k = (x_{i_1}, \ldots, x_{i_{n_i}})$.

Let $c = (c_1, \ldots, c_{n_i}) \in \{0, \hat{0}, 1\}^{n_i}$ be a coloring for X_i. We say that $c' = (c'_1, \ldots, c'_{n_i})$, $c'' = (c''_1, \ldots, c''_{n_i}) \in \{0, \hat{0}, 1\}^{n_i}$ *divide* c if

1. $(c_t \in \{\hat{0}, 1\} \Rightarrow c'_t = c''_t = c_t)$, and
2. $\big(c_t = 0 \Rightarrow [(c'_t, c''_t \in \{0, \hat{0}\}) \wedge (c'_t = 0 \vee c''_t = 0)]\big)$.

Then, evaluate the mapping m_i of X_i as follows:

For all colorings $c \in \{0, \hat{0}, 1\}^{n_i}$ set

$$m_i(c) := \min\{m_j(c') + m_k(c'') - \#_1(c) \mid c' \text{ and } c'' \text{ divide } c\}. \quad (10.6)$$

In other words, in order to determine the value $m_i(c)$ we look up the corresponding values for coloring c in m_j (corresponding to the left subtree) and in m_k (corresponding to the right subtree), add the corresponding values, and subtract the number of "1-assignments" in c, since they would be counted twice otherwise.

Clearly, if coloring c of node i assigns the colors 1 or $\hat{0}$ to a vertex x from X_i, we have to make sure that we use colorings c' and c'' of the children j and k which assign the same color to x. However, if c assigns color 0 to x, it is sufficient to justify this color by *only* one of the colorings c' or c''. Observe that, from the monotonicity of m_j and m_k we obtain the same "min" in assignment (10.6) if we replace condition 2 in the definition of "divide" by:

$2'$. $\big(c_t = 0 \Rightarrow [(c'_t, c''_t \in \{0, \hat{0}\}) \wedge (c'_t \neq c''_t)]\big)$.

Note that, for given $c \in \{0, \hat{0}, 1\}^{n_i}$, with $z := \#_0(c)$, we have 2^z many pairs (c', c'') that divide c if we use condition (2') instead of (2). Since there are $2^{n_i - z}\binom{n_i}{z}$ many colorings c with $\#_0(c) = z$, we obtain that

$$\{(c', c'') : c \in \{0, \hat{0}, 1\}^{n_i},\ c' \text{ and } c'' \text{ divide } c\}$$

has size

$$\sum_{z=0}^{n_i} 2^{n_i - z} \binom{n_i}{z} \cdot 2^z = 4^{n_i}.$$

This shows that evaluating m_i can be done in $O(4^{n_i} \cdot n_i)$ time. Again, it is not hard to see that m_i is monotonic if m_j and m_k are monotonic. This basically follows by the definition of "divide".

Step 3: Let r denote the root of T. The domination number is given by

$$\min\{m_r(c) \mid c \in \{0,1\}^{n_r}\}. \tag{10.7}$$

The minimum in (10.7) is taken only over colorings containing only colors 0 and 1 because a color $\hat{0}$ would mean that the corresponding vertex still needs to be dominated.

This concludes the description of the dynamic programming algorithm and leads to the following result.

Theorem 10.16 *For a graph G with given tree decomposition $\langle\{X_i \mid i \in I\}, T\rangle$ an optimal dominating set can be computed in $O(4^\omega \cdot \omega \cdot |I|)$ time. Here, ω denotes the width of the tree decomposition.*

Proof Obviously, the total running time of the algorithm is $O(4^\omega \cdot \omega \cdot |I|)$. For the correctness of the algorithm, we observe the following. In initialization Step 1, as well as in the updating process for INSERT NODES and JOIN NODES of Step 2, we made sure that the assignment of color 0 to a vertex x always guarantees that at the current stage of the algorithm x is already dominated by a vertex from previous bags. Since, by definition of tree decompositions, any pair of neighbors appears in at least one bag, the validity of the colorings was checked for each such pair of neighbors. Finally, the consistency property of tree decompositions together with the comments given in Step 2 of the algorithm imply that the updating of each mapping is done consistently with all mappings that have been visited earlier in the algorithm.

When bookkeeping how the minima in assignments (10.2), (10.6), and (10.7) of Step 2 and Step 3 were obtained, this method also constructs a dominating set D delivering the domination number. □

In conclusion, the most important point in dynamic programming on tree decompositions is the sizes of the tables involved. The table sizes are usually bounded by c^ω, where ω denotes the width of the underlying tree decomposition and c usually depends on the underlying combinatorial problem. Hence two optimization goals are immediate:

1. keep the width of the tree decomposition as small as possible; and
2. closely investigate the combinatorics of the underlying graph problem in order to keep the base c as small as possible.

DOMINATING SET provides a striking example of the second goal, as the constant could be improved from the naturally given 9 to 4. To illustrate the significance of such a result, Table 10.1 compares hypothetical running times of the $O(9^\omega \cdot \omega \cdot |I|)$ algorithm to the $O(4^\omega \cdot \omega \cdot |I|)$ algorithm for some realistic values of ω and $|I| = 1000$. Observe that this gives a realistic comparison of the relative performances since both algorithms incur comparable, small constant factors hidden in the O-notation. The $O(4^\omega \cdot \omega \cdot |I|)$ time algorithm has been successfully implemented. As a non-surprising result, this tells us that improving exponential terms often is a "big issue" for fixed-parameter algorithms.

Runtime	$\omega = 5$	$\omega = 10$	$\omega = 15$	$\omega = 20$
$9^\omega \cdot \omega \cdot \|I\|$	0.25 sec	10 hours	100 years	$8 \cdot 10^6$ years
$4^\omega \cdot \omega \cdot \|I\|$	0.005 sec	10 sec	4.5 hours	260 days

Table 10.1 *Comparing the $O(4^\omega \cdot \omega \cdot |I|)$ algorithm for* DOMINATING SET *with the $O(9^\omega \cdot \omega \cdot |I|)$ algorithm for $|I| = 1000$; we assume a computer executing 10^9 instructions per second and we neglect the constants hidden in the O-terms (which are comparable in both cases).*

Finally, it must be emphasized that besides (exponential) running time (exponential) memory usage is also an important issue in making tree decomposition-based algorithms useful in practice. The exponential table sizes lie at the heart of the exponential running times as well as of the exponential memory usage. In order to avoid the "memory boundedness" of dynamic programming on tree decompositions, all tricks and techniques should be tried—another promising research challenge connected with the development of efficient fixed-parameter algorithms.

In complete analogy to the case of VERTEX COVER, Theorem 10.16 yields the following:

Corollary 10.17 DOMINATING SET IN PLANAR GRAPHS *is solvable in $2^{O(\sqrt{k})} \cdot n$ time, where k denotes the size of the dominating set and n is the number of graph vertices.*

10.6 Monadic second-order logic (MSO)

In the preceding two sections we have explicitly seen two efficient fixed-parameter algorithms with respect to the parameter treewidth to determine optimal vertex covers and dominating sets for graphs given together with their tree decompositions. It might not always be easy to see whether a problem is fixed-parameter tractable with respect to the parameter treewidth in this way. In fact, dynamic programming can become an elusive matter here. There are also results, however, that state that large *classes* of problems can be solved in linear time when a tree decomposition with constant treewidth is known; in other words, these problems are in "linear-time *FPT*". The main work in this direction was done by Bruno Courcelle, providing a powerful classification tool for such problems. It must be emphasized, however, that the now described methodology is of purely theoretical interest because the associated running times suffer from huge constant fac-

tors and combinatorial explosions with respect to the parameter treewidth. Still, it provides an excellent tool for quickly deciding whether a problem is fixed-parameter tractable on graphs parameterized by treewidth. After establishing fixed-parameter tractability in this way, as a second step one should then head for a concrete, problem-specific algorithm with improved efficiency. Because the underlying theory is much beyond the scope of this book, we subsequently focus on describing central concepts and results and how to use them in a profitable way.

Our tool is called *monadic-second order logic* (*MSO*). Roughly speaking, it is an extension of first-order logic that also allows quantification over sets. The point here is that whenever a graph problem is expressible using the formalism of this logic, we can infer that the problem is fixed-parameter tractable with respect to the parameter treewidth. We start by describing the language of MSO.

What do MSO-formulae expressing graph properties look like, that is, what is their syntax? First of all, we have an infinite supply of *"individual" variables*, denoted by small letters x, y, z, etc. Moreover, there is an infinite supply of *set variables*, denoted by capital letters X, Y, Z, etc. To specify graphs and their properties, we use the particular vocabulary $\{E, V, I\}$, where V and E are unary relation symbols interpreted as the vertex and the edge set of a graph, and I is a binary relation symbol interpreted as the incidence relation between vertices and edges. Continuing with MSO syntax for graphs, using mainly postfix notation, we introduce *atomic* MSO-formulae as formulae of the forms $x = y$, Vx, Ex, Ixy, and Xx, where X is a set variable and x, y are individual variables. Based on atomic formulae, more complicated MSO-formulae can be built as follows, thus specifying the whole class of MSO-formulae.

- If ϕ is an MSO-formula, then $\neg\phi$ is one as well.
- If ϕ and ψ are MSO-formulae, then $\phi \wedge \psi$, $\phi \vee \psi$, and $\phi \rightarrow \psi$ are as well.
- If ϕ is an MSO-formula and x is an individual variable and X is a set variable, then $\exists x\phi$, $\forall x\phi$, $\exists X\phi$, and $\forall X\phi$ are as well.

Here, "$\rightarrow$" denotes the Boolean implication. This concludes the description of the syntax. Let us next come to the semantics of MSO-formulae. To simplify notation, for a graph $G = (V, E)$ let $U := V \cup E$. An *assignment* α for an MSO-formula ϕ maps each individual variable of ϕ to an element of U and every set variable to a subset of U. Thus we can inductively define the concept of an assignment α *satisfying* an MSO-formula ϕ, written as $(G, \alpha) \models \phi$ for a given graph G.

- $(G, \alpha) \models x = y$ iff $\alpha(x) = \alpha(y)$;
- $(G, \alpha) \models Vx$ iff $\alpha(x) \in V$;
- $(G, \alpha) \models Ex$ iff $\alpha(x) \in E$;
- $(G, \alpha) \models Ixy$ iff $\alpha(x) \in V$, $\alpha(y) \in E$, and vertex $\alpha(x)$ is endpoint of edge $\alpha(y)$;
- $(G, \alpha) \models Xx$ iff $\alpha(x) \in \alpha(X)$;

- $(G, \alpha) \models \neg\phi$ iff $(G, \alpha) \not\models \phi$;
- $(G, \alpha) \models \phi \wedge \psi$ iff $(G, \alpha) \models \phi$ and $(G, \alpha) \models \psi$, and analogously for the logical or "$\vee$" and the logical implication "$\rightarrow$";
- $(G, \alpha) \models \exists x\phi$ iff there exists an $a \in U$ such that $(G, \alpha\frac{a}{x}) \models \phi$, where $\alpha\frac{a}{x}$ denotes the assignment with $\alpha\frac{a}{x}(x) = a$ and $\alpha\frac{a}{x}(v) = \alpha(v)$ for all $v \neq x$;
- $(G, \alpha) \models \forall x\phi$ iff for all $a \in U$ we have $(G, \alpha\frac{a}{x}) \models \phi$;
- $(G, \alpha) \models \exists X\phi$ iff there exists an $A \subseteq U$ such that $(G, \alpha\frac{A}{X}) \models \phi$, and similarly for $\forall X$ meaning "for all $A \subseteq U$".

The relation $(G, \alpha) \models \phi$ depends only on the values of α at the *free variables* of ϕ, that is, those variables v not occurring in the scope of a quantifier $\exists v$ or $\forall v$, where v may denote an individual as well as a set variable. One writes

$$\phi(x_1, \ldots, x_i, X_1, \ldots, X_j)$$

to indicate that the free individual variables of ϕ are $x_1, \ldots, x_k$ and the free set variables are $X_1, \ldots, X_j$. Then for a graph G and $a_1, \ldots, a_i \in U$, $A_1, \ldots, A_j \subseteq U$ one writes

$$G \models \phi(a_1, \ldots, a_i, A_1, \ldots, A_j)$$

if for every assignment α with

$$\alpha(x_1) = a_1, \ldots, \alpha(x_i) = a_i$$

and

$$\alpha(X_1) = A_1, \ldots, \alpha(X_j) = A_j$$

it holds that $(G, \alpha) \models \phi$. A *sentence* is a formula without free variables.

For example, to express that a graph G is bipartite we write $G \models \phi$ with ϕ being the following formula

$$\exists X \exists Y (\forall x (Vx \rightarrow (Xx \vee Yx)) \wedge$$
$$\forall x \forall y (((x \neq y) \wedge \exists z (Ixz \wedge Iyz)) \rightarrow \neg((Xx \wedge Xy) \vee (Yx \wedge Yy)))).$$

The core result in this field, due to Bruno Courcelle, now reads as follows.

Theorem 10.18 *Let $\omega \geq 1$ and let*

$$\phi(x_1, \ldots, x_i, X_1, \ldots, X_j)$$

be an MSO-formula. Then there is a linear-time algorithm, given a graph G with a tree decomposition of width at most ω and $a_1, \ldots, a_i \in U$, $A_1, \ldots, A_j \subseteq U$, decides whether

$$G \models \phi(a_1, \ldots, a_i, A_1, \ldots, A_j).$$

So far, we have seen how MSO-formulae can be used to express decision problems. There are *extensions of MSO* allowing us to deal with optimization problems without giving up linear-time solvability as stated in Theorem 10.18.

For instance, the following formula expresses the (optimization version of the) VERTEX COVER problem:

$$\min X \; \forall y \exists x (Xx \wedge Ey \wedge Ixy).$$

Similarly, the optimization version of DOMINATING SET can be expressed as follows:

$$\min X \; \forall y \exists x \exists z (Xx \wedge Vy \wedge Ez \wedge Ixz \wedge Iyz).$$

In the purely parameterized (decision) version searching for a vertex cover of size k, we would write

$$\exists x_1 \exists x_2 \ldots \exists x_k \forall y (Vx_1 \wedge \ldots \wedge Vx_k \wedge (Ey \rightarrow (Ix_1y \vee \ldots \vee Ix_ky)))$$

instead. The parameterized version of DOMINATING SET can be dealt with similarly. Observe that, of course, the huge constant factor hidden in the O-notation in Theorem 10.18 depends on the formula size and its complexity.

Summarizing, monadic second-order logic offers a *descriptive* way of the classifying the computational complexity of problems restricted to graphs of bounded treewidth. It is a very elegant and powerful tool for quickly deciding about fixed-parameter tractability, but it is far from any efficient implementations. Algorithmic analysis of the concrete problem at hand always seems beneficial in order to come up with more practical results once we know that fixed-parameter tractability is achievable due to a description in the language of MSO.

10.7 Related graph width parameters

Besides treewidth there are several *graph width* parameters of algorithmic use. We briefly mention three of them here.

Pathwidth. This is a simple special case of treewidth. It is obtained by restriction of the underlying trees in the definition of tree decompositions to paths. Pathwidth closely resembles a form of "narrowness" of graphs. Graphs with small pathwidth typically appear in natural language processing. An important problem in VLSI layout theory, the so-called GATE MATRIX LAYOUT problem, is equivalent to the pathwidth problem. Trivially, the treewidth of a graph always gives a lower bound for the pathwidth of a graph. Moreover, trees may have arbitrarily large pathwidth. As for tree decompositions, there are many different equivalent characterizations for graphs of bounded pathwidth. Recall from Section 10.1 that the optimal tree decomposition of a grid graph is actually a path decomposition. Concerning the determination of optimal path decompositions of graphs, again it is known that the corresponding decision problem is *NP*-complete. As to fixed-parameter tractability with respect to the parameter pathwidth for the problem to construct path decompositions, analogous statements as for the treewidth case apply—the algorithms are far from practical. That is why here also approximation and heuristic algorithms are important.

With regard to the algorithmic use of path decompositions, we note only that it is obvious that dynamic programming is easier for path-like than for

tree-like structures. Actually, in Section 9.5 we experienced with the WEIGHTED SET COVER problem polynomial-time solvability for arbitrary "pathwidth" values, whereas it turned out *NP*-complete for tree-structured cases in general. For the latter case, however, we could show fixed-parameter tractability with respect to the parameter "width of the corresponding tree structure". By way of contrast, path decompositions can lead to larger width values than tree decompositions do—simply consider trees which have treewidth one—and so the use of tree decompositions may become favorable when we encounter a combinatorial explosion with respect to the corresponding width.

Local treewidth. In Section 10.3 we saw that graphs which have a size-k vertex cover or a size-k dominating set possess tree decompositions of width $O(\sqrt{k})$. In Section 10.6 we learned that for graphs of bounded treewidth all graph properties (such as having a size-k vertex cover) that are expressible in monadic second-order logic are decidable in linear time. The notion of local treewidth arose in an attempt to extend and generalize these sorts of results.

To this end, we need to define the concept of *r-neighborhood.* Let $G = (V, E)$ be a graph and $v \in V$ be a vertex in G. Then $N_r[v]$ is the set of all vertices with distance at most r to v, that is, $N_r[v]$ includes v and all vertices that are connected to v via a path of at most r edges in G. Let $tw(G)$ denote the treewidth of graph G.

Definition 10.19 *The* local treewidth *of a graph $G = (V, E)$ is*

$$ltw(G, r) := \max\{tw(G[N_r[v]]) \mid v \in V\}.$$

Then, G has bounded local treewidth *if there is a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ such that $ltw(G, r) \leq f(r)$ for $r \in \mathbb{N}$.*

The definition of bounded local treewidth requires that the local treewidth depends only on r and is independent of the number of graph vertices etc. Clearly, if a graph has treewidth t then its local treewidth is bounded from above by t. However, whereas planar graphs have unbounded treewidth, their local treewidth is bounded, that is, it can be shown that $ltw(G, r) \leq 3r$ for an arbitrary planar graph G and $r \geq 1$. Without going into any details here, we only mention in passing that with the help of the concept of bounded local treewidth numerous fixed-parameter tractability results extending those for planar graphs could be achieved. These results, however, suffer from large hidden constant factors, and so far are of only theoretical interest.

Branchwidth. This is a concept similar to treewidth. In contrast to tree decompositions, however, it is known that optimal branch decompositions of planar graphs can be computed in polynomial time. For general graphs, branchwidth determination again becomes an *NP*-complete problem.

Definition 10.20 *Let $G = (V, E)$ be a graph. A* branch decomposition *of G is a pair $\langle \sigma, T = (I, F) \rangle$ where T is a tree with every node in T of degree one or three, and σ is a one-to-one mapping from E to the set of leaves in T.*

The order of an edge $f \in F$ *is the number of vertices* $v \in V$ *with incident edges* $\{v,u\},\{v,w\} \in E$ *such that the path in* T *from* $\sigma(\{v,u\})$ *to* $\sigma(\{v,w\})$ *uses* f.

The width *of branch decomposition* $\langle \sigma, T = (I,F)\rangle$ *is the maximum order over all edges* $f \in F$. *The* branchwidth *of* G *is the minimum* k *such that* G *has a branch decomposition of width* k.

There is a close relationship between treewidth and branchwidth. Let k be the treewidth of a graph and let k' be its branchwidth. Then it is known that

$$\max\{2,k'\} \leq k+1 \leq \max\{2,\lfloor 3\cdot k'/2\rfloor\}.$$

As with tree decompositions, branch decompositions receive particular attention from an algorithmic point of view because they are amenable to dynamic programming techniques. The basic ideas are analogous to dynamic programming on tree decompositions and we omit the details. Note that the currently best fixed-parameter "$c^{\sqrt{k}}$-algorithm" with constant base c solving DOMINATING SET IN PLANAR GRAPHS is based on branch decompositions. More specifically, it runs in $O(36000^{\sqrt{k}}\cdot k+k^4+n^4)$ time. Observe, however, that this refers to proven worst-case bounds concerning the constant c in the base. For the time being it is not clear what the best algorithms in practice are. Moreover, algorithmically the various tree decomposition and branch decomposition based algorithms are similar—the improvements in the worst-case time bounds are mainly due to refined and improved mathematical analysis.

10.8 Summary and concluding remarks

Treewidth is a sophisticated concept one has to get used to. To this end, its characterization by the robber–cop game (see Section 10.1) is extremely useful. It needs time, however, to become familiar with this methodology—only starting material is presented in this chapter. Already the construction of (semi-)optimal tree decompositions is a thing that needs much more attention than we could pay to it here. For the purpose of studying parameterized problems in planar graphs, however, Section 10.3 presents the fundamental ideas. Note that, so far, most known fixed-parameter complexity results using tree decompositions and related concepts are connected to planar and somewhat more general graphs. The algorithmic usefulness of tree decompositions is tightly connected to dynamic programming—here we presented two concrete examples for VERTEX COVER and DOMINATING SET. In the literature one can find a generalized description of how dynamic programming for graphs of bounded treewidth works. With monadic second-order logic a powerful classification tool is given in order to determine the solvability of graph problems for bounded treewidth in a "descriptive way". Intuitively speaking, in particular for graph problems that carry some "non-local" properties—that is, for instance, the choice of a vertex into a solution set may directly affect other vertices at arbitrary distance from that vertex—fixed-parameter tractability with respect to the parameter treewidth is often far from being clear. Monadic second-order logic may help to decide on that. Finally, it must be noted that the concept of tree decompositions has several relatives—local treewidth and branchwidth being two of them which have already played a role in fixed-parameter complexity studies.

Treewidth and related concepts offer a new view of parameterization—that is, *structural parameterization*. Whereas the most conventional way to choose problem parameters is based on the sizes of the desired solution sets, structural parameters are basically independent of these. Thus, structural parameters such as treewidth might also be considered as ways to define special problem instances—graphs of bounded treewidth also belong into the large field of studying special graph classes and their algorithmic properties. It is plausible to assume that future research will reveal more and more connections and mutual interaction between parameterized computational complexity and the vast field of special graph classes. Another point to note here is that it seems less appropriate to search in the case of structural parameterizations such as by treewidth for problem kernels: clearly, a graph of bounded treewidth k may have size completely independent of k, so there appears to be no point in asking for bounding the graph size by its treewidth as is inherent by the concept of reduction to a problem kernel (see Chapter 7).

Tree decomposition-based algorithms bring forward a so far somewhat neglected point in fixed-parameter algorithmics—efficient *usage of memory*. Combinatorial explosions with respect to memory consumption must be kept as small as possible—a system may run almost forever on hard problems, but it will immediately break down when running out of memory. Keeping this in mind is essential for successful applications of tree decomposition-based and related memory-intensive algorithms.

Tree decompositions of graphs, studied thoroughly, deserve a book on their own. The basic concepts and definitions are technically demanding. Nevertheless, it seems as though anybody doing advanced fixed-parameter algorithmics should gain some familiarity with this mathematically beautiful and algorithmically strong methodology. Discovering and exploiting "width parameters" in hard problems is a core issue that fixed-parameter algorithmics should be occupied with.

10.9 Exercises

1. Describe in detail what the width-ℓ tree decomposition of an $\ell \times \ell$-grid looks like.
2. Construct an optimal tree decomposition for a graph that is simply a cycle.
3. Prove Lemma 10.3.
4. Show that a graph that has a vertex cover of size k has pathwidth at most $k+1$.
5. Show that a complete graph with n vertices has treewidth exactly $n-1$.
6. Show that if a graph G has an induced complete subgraph K then every tree decomposition of G must contain a bag that contains all vertices from K.

7. Prove Proposition 10.10 for the case of VERTEX COVER.
8. The *NP*-complete 3-COLORING problem is to color (if possible) each vertex of a graph with one out of three colors such that no pair of neighboring vertices has the same color.
   Show how 3-COLORING can be solved by dynamic programming on graphs of bounded treewidth (with given tree decomposition).
9. Describe bounded treewidth dynamic programs for the following variations of DOMINATING SET:
   (a) INDEPENDENT DOMINATING SET: Here, the desired dominating set must additionally form an independent set.
   (b) TOTAL DOMINATING SET: Here, each vertex in the dominating set additionally must have a neighbor in the dominating set.
   (c) PERFECT DOMINATING SET: Here, every vertex not in the dominating set must have exactly one neighbor in the dominating set.
10. The *NP*-complete FEEDBACK VERTEX SET problem is defined as follows.
    Input: A graph $G = (V, E)$ and a nonnegative integer k.
    Task: Find a subset of vertices $V' \subseteq V$ with k or fewer vertices such that each cycle in G contains at least one vertex from V'. Thus, removing the vertices in V' from G results in a forest.
    Show how to express FEEDBACK VERTEX SET using monadic second-order logic.

10.10 Bibliographical remarks

Tree decompositions of graphs go back to Robertson and Seymour (1986). Survey papers (Bodlaender, 1993; Bodlaender, 1997; Bodlaender, 1998; Bodlaender, 2005) provide good overviews focussing on different aspects. See also the book Kloks (1994). The linear-time algorithm for bounded treewidth (and the construction of corresponding tree decompositions) is due to Bodlaender (1996). The practical heuristic mentioned for constructing tree decompositions is due to Reed (1993). See Bodlaender (2005) for an up-to-date survey of constructing tree decompositions. The description of planar graphs follows Alber *et al.* (2002). The current best bound for a $c^{\sqrt{k}}$-algorithm for DOMINATING SET IN PLANAR GRAPHS appears in Fomin and Thilikos (2003). See also Demaine *et al.* (2004*a*) and Demaine *et al.* (2004*b*) for several more related issues. The dynamic programming for DOMINATING SET is described in Alber *et al.* (2002) and Alber and Niedermeier (2002).

Related graph width parameters are discussed in Bodlaender (1993) and Bodlaender (1998). The polynomial-time algorithm for computing branchwidth in planar graphs is due to Seymour and Thomas (1994). Local treewidth has been introduced by Eppstein (2000). Important theoretical fixed-parameter tractability results in this direction are Frick and Grohe (2001) and Demaine *et al.* (2004*a*).

11

FURTHER ADVANCED TECHNIQUES

In the preceding chapters we presented in some depth—using several concrete problems and the fixed-parameter algorithms for solving them—the fundamental techniques of reduction to a problem kernel, depth-bounded search trees, dynamic programming, and tree decompositions of graphs. Now we describe a few more techniques which belong in the toolkit of every "parametric algorithm designer".

We begin with color-coding, a very elegant randomized method designed in the context of subgraph isomorphism problems. The method can be derandomized and it seems to apply to a whole list of problems. Our illustrative example is LONGEST PATH, which concerns the parameter path length. Future research must show its breadth and how it performs with respect to practical applications.

Our next technique is integer linear programming. The point here is that due to a ground-breaking result of Hendrik W. Lenstra it is known that bounding the number of variables in an integer linear program by a function depending only on the problem parameter yields fixed-parameter tractability with respect to this parameter. Because of the hidden constants and the large combinatorial explosion involved this method seems more suitable as a classification tool, though. Still, the corresponding integer linear programs can be implemented and handled using standard solvers. Our illustrative example is CLOSEST STRING with respect to the parameter number of input strings—no other way to show fixed-parameter tractability is known here.

We continue with a very new and promising method from the year 2004—iterative compression. The point here is to make use of size-$(k+1)$ solutions in order to find size-k solutions in an iterative manner. This technique is designed for minimization problems. It was used for the breakthrough result to show that GRAPH BIPARTIZATION is fixed-parameter tractable with respect to the number of vertices to be deleted. To illustrate the ideas, we employ the conceptually simpler cases VERTEX COVER and FEEDBACK VERTEX SET. We believe that future research might turn this technique into one that fills a whole chapter.

As with iterative compression, the technique called greedy localization is from 2004. It tries to make use of a greedily found solution in order to find a better or even optimal one. It works for maximization problems where small solution set sizes—which are the parameters—might be less frequent than for minimization problems. Still, the method definitely carries potential for more applications. We illustrate the method using the SET SPLITTING and 3-SET PACKING problems.

We conclude our series of advanced techniques with a brief glimpse at the