15

The Open Graph Drawing Framework (OGDF)

Markus Chimani [1]
Friedrich-Schiller-Universitaet Jena

Carsten Gutwenger
TU Dortmund

Michael Jünger
University of Cologne

Gunnar W. Klau
Centrum Wiskunde & Informatica

Karsten Klein
TU Dortmund

Petra Mutzel
TU Dortmund

15.1 Introduction

We present the Open Graph Drawing Framework (OGDF), a C++ library of algorithms and data structures for graph drawing. The ultimate goal of the OGDF is to help bridge the gap between theory and practice in the field of automatic graph drawing. The library offers a wide variety of algorithms and data structures, some of them requiring complex and involved implementations, e.g., algorithms for planarity testing and planarization, or data structures for graph decomposition. A substantial part of these algorithms and data structures are building blocks of graph drawing algorithms, and the OGDF aims at providing such functionality in a reusable form, thus also providing a powerful platform for implementing new algorithms.

The OGDF can be obtained from its website at:

`http://www.ogdf.net`

[1] Markus Chimani was funded via a juniorprofessorship by the Carl-Zeiss-Foundation.

0-8493-8597-0/01/$0.00+$1.50

This website also provides further information like tutorials, examples, contact information, and links to related projects. The source code is available under the GNU General Public License (GPL v2 and v3).

15.1.1 The History of the OGDF

Back in 1996, the development of the AGD library [AGMN97] (Algorithms for Graph Drawing) started at the Max-Planck Institute for Computer Science in Saarbrücken, Germany, originating from the DFG-funded project *Design, Analysis, Implementation, and Evaluation of Graph Drawing Algorithms*. The project later moved to Vienna University of Technology. AGD was designed as a C++ library of algorithms for graph drawing, based on the LEDA library [MN99] of efficient data structures and algorithms.

In 1999, a new branch of the library was developed as an internal project at the oreas GmbH and the research center caesar. The main goal was to have a code basis that could be built independently of any other libraries. This resulted in a new design and complete rewrite by starting from scratch, thereby concentrating on the strengths of the library, i.e., planarity and orthogonal layout, and implementing a wealth of required basic data structures and algorithms.

Later on, this internal project was renamed to OGDF and made open source under the GPL. The OGDF is currently maintained and further developed by researchers at the Universities of Dortmund, Cologne, and Jena.

15.1.2 Outline

After introducing the major design concepts and goals in Section 15.2, we dedicate two sections to the algorithms and data structures contained in the library. Section 15.3 introduces general graph algorithms and related data structures, and Section 15.4 focuses on drawing algorithms and layout styles. Finally, we conclude this chapter with selected success stories in Section 15.5.

15.2 Major Design Concepts

Many sophisticated graph drawing algorithms build upon complex data structures and algorithms, thus making new implementations from scratch cumbersome and time-consuming. Obviously, graph drawing libraries can ease the implementation of new algorithms a lot. E.g., the AGD library was very popular in the past, since it covered a wide range of graph drawing algorithms and—together with the LEDA library—data structures. However, the lack of publicly available source-code restricted the portability and extendability, not to mention the understanding of the particular implementations. Other currently available graph drawing libraries suffer from similar problems, or are even only commercially available or limited to some particular graph layout methods.

Our goals for the OGDF were to transfer essential design concepts of AGD and to overcome AGD's main deficiencies for use in academic research. Our main design concepts and goals are the following:

- Provide a wide range of graph drawing algorithms that allow a user to reuse and replace particular algorithm phases by using a dedicated *module mechanism*.
- Include *sophisticated data structures* that are commonly used in graph drawing, equipped with rich public interfaces.

- A *self-contained* source code that does not require additional libraries (except for some optional LP-/ILP-based algorithms).
- *Portable* C++-code that supports the most important compilers for the major operating systems (Linux, MacOS, and Windows) and that is available under an *open source license* (GPL).

15.2.1 Modularization

In the OGDF, an algorithm (e.g., a graph drawing algorithm or an algorithm that can be used as building block for graph drawing algorithms) is represented as a class derived from a base class defining its interface. Such algorithm classes are also called *modules* and their base classes *module types*. E.g., general graph layout algorithms are derived from the module type `LayoutModule`, which defines as interface a `call` method whose parameters provide all the relevant information for the layout algorithm: the graph structure (`Graph`) and its graphical representation like node sizes and coordinates (`GraphAttributes`)[1]. The algorithm then obtains this information and stores the computed layout in the `GraphAttributes`.

Using common interface classes for algorithms allows us to make algorithms exchangeable. We can write an implementation that utilizes several modules, but each module is used only through the interface defined by its module type. Then, we can exchange a module by a different module implementing the same module type. The OGDF provides a mechanism called module options that even makes it possible to exchange modules at runtime. Suppose an algorithm A defines a module option M of a certain type T representing a particular phase of the algorithm, and adds a set-method for this option. A *module option* is simply a pointer to an instance of type T, which is set to a useful default value in A's constructor and called for executing this particular phase of the algorithm. Using the set-method, this implementation can be changed to *any* implementation implementing the module type T, even new implementations not contained in the OGDF itself.

Module options are the key concept for modularizing *algorithm frameworks*, thus allowing users to experiment with different implementations for particular phases of the algorithm, or to evaluate new implementations for phases without having to implement the whole framework from scratch. Figure 15.1 shows how module options are used in `SugiyamaLayout`. In this case, `SugiyamaLayout` is a framework with three customizable phases (ranking, 2-layer crossing minimization, and layout), and the constructor takes care of setting useful initial implementations for each phase. Using a different implementation of, e.g., the crossing minimization step is simple:

```
SugiyamaLayout sugi;
sugi.setCrossMin(new MedianHeuristic);
```

In Section 15.4, we will illustrate the main drawing frameworks available in the OGDF using class diagrams, thereby showing the interconnections between the various classes in the OGDF.

15.2.2 Self-contained and Portable Source Code

[1] More precisely, the `call` method has only one parameter, the `GraphAttributes`, which allows us to get a reference to the `Graph` itself.

```
class SugiyamaLayout : public LayoutModule
{
protected:
    ModuleOption<RankingModule>         m_ranking;
    ModuleOption<TwoLayerCrossMin>      m_crossMin;
    ModuleOption<HierarchyLayoutModule> m_layout;
    ...
public:
    SugiyamaLayout() {
        m_ranking .set(new LongestPathRanking);
        m_crossMin.set(new BarycenterHeuristic);
        m_layout  .set(new FastHierarchyLayout);
        ...
    }
    void setRanking(RankingModule *pRanking)  { m_ranking.set(pRanking); }
    void setCrossMin(TwoLayerCrossMin *pCrossMin) { m_crossMin.set(pCrossMin); }
    void setLayout(HierarchyLayoutModule *pLayout) { m_layout.set(pLayout); }
    ...
};
```

Figure 15.1 Excerpt from the declaration of `SugiyamaLayout` demonstrating the use of module options.

It was important for us to create a library that runs on all important systems, and whose core part can be built without installing any further libraries. Therefore, all required basic data structures are contained in the library, and only a few modules based on linear programming require additional libraries: COIN-OR [Mar10] as LP-solver and ABACUS [JT00] as branch-and-cut framework.

For reasons of portability and generality, the library provides only the drawing algorithms themselves and not any graphical display elements. Such graphical display would force us to use very system-dependent GUI or drawing frameworks, or to have the whole library based on some cross-platform toolkit. Instead of this, the OGDF simply computes basic layout information like coordinates of nodes or bend points, and an application that uses the OGDF can create the required graphical display by using the GUI framework of its choice.

For creating graphics in common image formats, the OGDF project provides the command line utility `gml2pic`[2]. This utility converts graph layouts stored in GML or OGML file formats into images in PNG, JPEG, TIFF, SVG, EPS, or PDF format. We recommend to use the new OGML (*Open Graph Markup Language*) file format, since it offers a wide range of clearly specified formatting options. Hence, it is easy to save graph layouts in OGML format using the OGDF, and then apply `gml2pic` for creating high-quality graphics. All graph layouts in this chapter have been created with `gml2pic`. Figures 15.9 and 15.11, e.g., demonstrate the automatic creation of Bézier curves from ordinary polylines.

15.3 General Algorithms and Data Structures

[2] available at `http://www.ogdf.net/doku.php/project:gml2pic`

The OGDF contains many basic data structures like arrays, lists, hashing tables, and priority queues, as well as fundamental data structures for the representation of graphs (`Graph`, `ClusterGraph`, and associative arrays for nodes, edges, etc.). Many basic graph algorithms can be found in `basic/simple_graph_alg.h`, e.g., functions dealing with parallel edges, connectivity, biconnectivity, and acyclicity. In this section, we focus on the more sophisticated algorithms and data structures in the OGDF.

15.3.1 Augmentation and Subgraph Algorithms

Augmentation Algorithms. Several augmentation modules are currently available in the library for adding edges to a graph to achieve biconnectivity. This can be done either by disregarding the planarity of the graph or by taking care not to introduce non-planar subgraphs.

Augmenting a planar graph to a planar biconnected graph by adding the minimum number of edges is an NP-hard optimization problem. It has been introduced by Kant and Bodlaender [KB91], who also presented a simple 2-approximation algorithm for the problem. They also claimed to have a 3/2-approximation algorithm, but Fialko and Mutzel [FM98] have shown that this algorithm is erroneous and cannot be corrected. However, their suggested 5/3-approximation algorithm was shown to approximate the optimal solution by a factor of 2, only (see [GMZ09a]). Experiments show that the Fialko-Mutzel algorithm performs very good in practice. The module `PlanarAugmentation` implements the Fialko-Mutzel algorithm, which proceeds roughly as follows. The biconnected components of a graph induce a so-called *block tree* whose nodes are the cut vertices and blocks of the graph. The algorithm first constructs a block tree T from the given graph and then iteratively adds edges between blocks of degree one in T. Experiments on a set of benchmark graphs have shown that in about 96% of all the cases the approximation algorithm finds the optimal solution to the planar augmentation problem [FM98].

In addition, the OGDF contains the module `DfsMakeBiconnected`. The underlying algorithm uses depth-first search and adds a new edge whenever a cut vertex is discovered. If the input graph is planar, the augmented graph also remains planar. However, in general, this approach adds a significantly higher number of edges than the `PlanarAugmentation` module.

A special variant of the planar augmentation problem is solved by the `PlanarAugmentationFix` module. Here, a planar graph with a fixed planar embedding is given, and this embedding shall be extended such that the graph becomes biconnected. `PlanarAugmentationFix` implements the optimal, linear-time algorithm by Gutwenger, Mutzel, and Zey [GMZ09b].

Acyclic Subgraphs. Two modules are available to compute acyclic subgraphs of a digraph $G = (V, A)$. These modules determine a feedback arc set $F \subset A$ of G, i.e., if G contains no self-loops, an acyclic digraph is obtained by reversing all the arcs in F. `DfsAcyclicSubgraph` computes an acyclic subgraph in linear time by removing all back arcs in a depth-first-search tree of G. On the other hand, `GreedyCycleRemoval` implements the linear-time greedy algorithm by Eades and Lin [EL95]. If G is connected and has no two-cycles, the algorithm guarantees that the number of non-feedback arcs is at least $|A|/2 - |V|/6$.

The OGDF provides further modules for the computation of planar subgraphs. These are covered in the context of graph planarization; see Section 15.3.3.

15.3.2 Graph Decomposition

Besides the basic algorithm for computing the biconnected components of a graph [Tar72, HT73b] (function `biconnectedComponents` in `basic/simple_gaph_alg.h`), the OGDF provides further powerful data structures for graph decomposition. `BCTree` represents the decomposition of a graph into its biconnected components as a BC-tree and `StaticSPQRTree` represents the decomposition of a biconnected graph into its triconnected components as an SPQR-tree [DT89, DT96]. An *SPQR-tree* is a tree whose nodes are associated with the triconnected components (called the *skeletons* of the tree nodes) of the graph: *S-nodes* correspond to serial structures, *P-nodes* to parallel structures, and *R-nodes* to simple, triconnected structures; *Q-nodes* simply correspond to the edges in the graph and are hence not required by an implementation. Both data structures can be build in linear time; the latter constructs the SPQR-tree by applying the corrected version [GM01] of Hopcroft and Tarjan's algorithm [HT73a] for decomposing a graph into its triconnected components. The OGDF is one of the few places where one can find a correct implementation of this complex and involved algorithm (to the best of our knowledge, AGD was the first library providing such an implementation).

BC- and SPQR-trees are important data structures for many graph algorithms dealing with planar graphs, since they efficiently encode all planar embeddings of a planar graph. Notice that a planar graph might have exponentially many planar embeddings. The embeddings of the skeleton graphs of an SPQR-tree induce a unique embedding of the original graph. `StaticPlanarSPQRTree` is a specialized version of `StaticSPQRTree` with additional support for planar graphs. It provides basic operations for changing the currently represented embedding of the graph, like flipping the skeleton of an R-node and permuting the order of the edges in the skeleton of a P-node, and a method for embedding the graph according to the embeddings of the skeletons.

In addition to the static versions of BC- and SPQR-trees, the OGDF also contains efficient implementations of dynamic BC- and SPQR-trees. The supported update operations are insertion of nodes and edges. `DynamicBCTree` implements the update operations as described by Westbrook and Tarjan [WT92], and `DynamicSPQRTree` as described by Di Battista and Tamassia [DT96].

15.3.3 Planarity and Planarization

The OGDF provides a unique collection of algorithms for planar graphs, including algorithms for planarity testing and planar embedding, computation of planar subgraphs, and edge reinsertion. These algorithms can be combined using the planarization approach, yielding excellent crossing minimization heuristics. The planarization approach for crossing minimization is realized by the module `SubgraphPlanarizer`, and the two layout algorithms `PlanarizationLayout` and `PlanarizationGridLayout` implement a complete framework for planarization and layout. Figure 15.2 gives an overview of the OGDF's planarization framework for graph layout, illustrating the interconnection between the modules involved; the various implementations for `EmbedderModule` are shown in Figure 15.3. An in-depth description of this framework can be found in [Gut10].

Planarity Testing and Embedding. The OGDF provides two algorithms for planarity testing. `PlanarModule` implements the node-addition algorithm [LEC67, BL76, CNAO85]) based on PQ-trees, and `BoyerMyrvold` implements the edge-addition algorithm by Boyer and Myrvold [BM04] which is based on depth-first search. Both modules can also compute a planar embedding of the graph.

However, for many graphs it is highly beneficial for a graph layout algorithm not to use

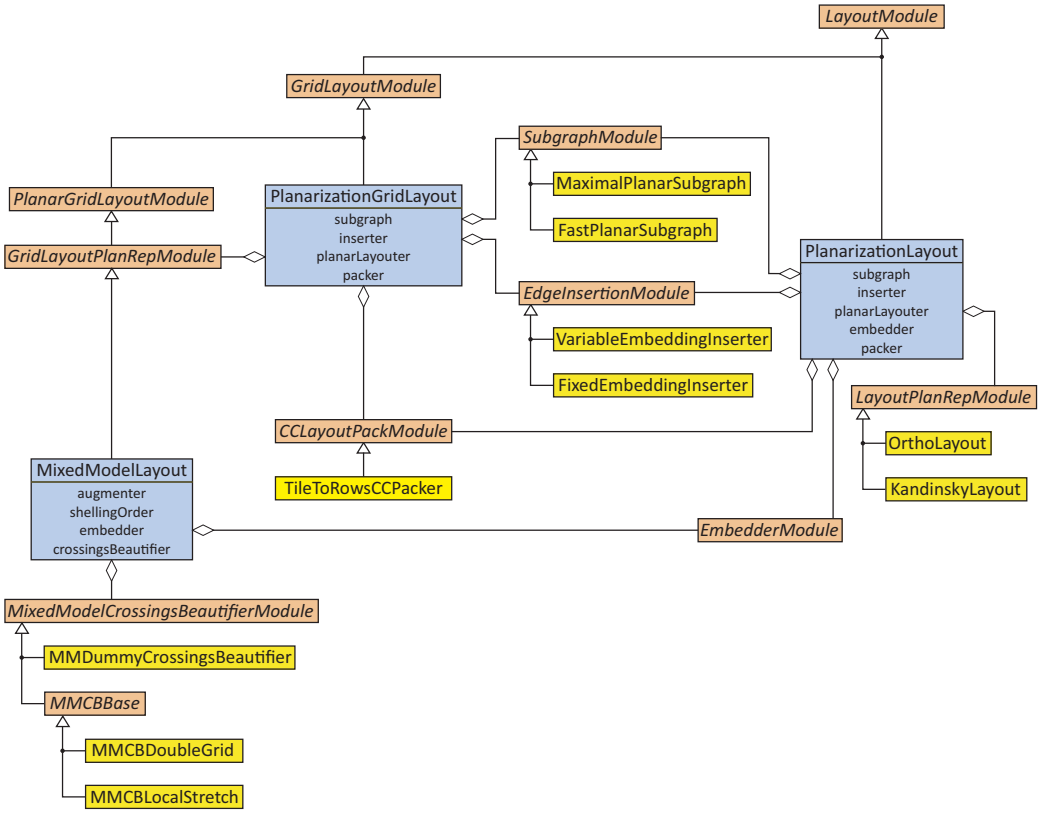

Figure 15.2 The Planarization framework for graph layout in the OGDF library.

just any embedding but an embedding that optimizes certain criteria. The OGDF contains such embedding algorithms which optimize criteria like a large external face or a small *block-nesting depth* (which is a measure for the topological nesting of the biconnected components in the embedding). `EmbedderMinDepthPiTa` implements the algorithm by Pizzonia and Tamassia [PT00], which minimizes the block-nesting depth for fixed embeddings of the blocks. On the other hand, `EmbedderMinDepth` minimizes the block-nesting depth without any restrictions, and `EmbedderMaxFace` maximizes the size of the external face; these two modules implement algorithms presented by Gutwenger and Mutzel [GM03]. Notice that just maximizing the external face still leaves a lot of freedom for embedding inner faces. Therefore, Kerkhof [Ker07] developed an extension of `EmbedderMaxFace`, realized by `EmbedderMaxFaceLayers`, which considers the *layers* of the embedding and the sizes of their *boundaries*. Here, layer i is formed by the faces with distance i to the external face in the dual graph, and the boundary B_i of layer i is roughly given by the edges shared by layers i and $i+1$. Then, the algorithm computes an embedding such that $|B_0|, |B_1|, \ldots$ is lexicographically maximal. There are also combinations of these algorithms, realized by `EmbedderMinDepthMaxFace` and `EmbedderMinDepthMaxFaceLayers`.

Upward Planarity Testing. Although the general upward planarity testing problem is NP-complete [GT01], the problem can be solved efficiently for digraphs with only a single source, also called sT-digraphs. The OGDF provides a linear-time implementation of the sophisticated algorithm by Bertolazzi et al. [BDMT98], which is based on decomposing the underlying undirected graph using SPQR-trees.

Planar Subgraphs. The module `FastPlanarSubgraph` computes a planar subgraph of an input graph G by deleting a set of edges using the PQ-tree data structure [JLM98]. The algorithm is similar to the one by Jayakumar et al. [JTS89] and is one of the best heuristics for the NP-hard *maximum planar subgraph problem* that asks for the smallest set of edges whose removal leads to a planar graph. The heuristic proceeds in a similar manner as PQ-tree-based planarity testing: First, it constructs an s-t-numbering and then adds nodes subsequently to the empty graph in this order while maintaining the PQ-tree data structure. The implementation also provides an option `runs` which runs the algorithm multiple times with randomly chosen (s,t)-edges and then takes the best result. The module `MaximumPlanarSubgraph` solves the maximum planar subgraph problem exactly by applying a branch-and-cut approach.

Edge Reinsertion. Reinserting edges into a planar auxiliary graph such as to introduce as few crossings as possible is an important phase within the planarization approach. The OGDF offers two modules for this task: `FixedEmbeddingInserter` is the standard approach, which reinserts edges iteratively by looking for shortest paths in the dual graph; on the other hand, `VariableEmbeddingInserter` proceeds similarly but solves the subproblem of inserting one edge with the minimum number of crossings under a variable embedding setting to optimality [GMW05]. To achieve this, it applies BC- and SPQR-trees which can encode all planar embeddings of a graph.

15.4 Graph Drawing Algorithms

Graph drawing algorithms form the heart of the library. Traditionally, the focus of the OGDF is on planar drawing algorithms and the planarization approach. However, a large number of drawing algorithms like energy-based layout algorithms or hierarchical drawing methods have been added. Today, the OGDF provides flexible frameworks with interchangeable modules for various drawing paradigms, including the *planarization approach*

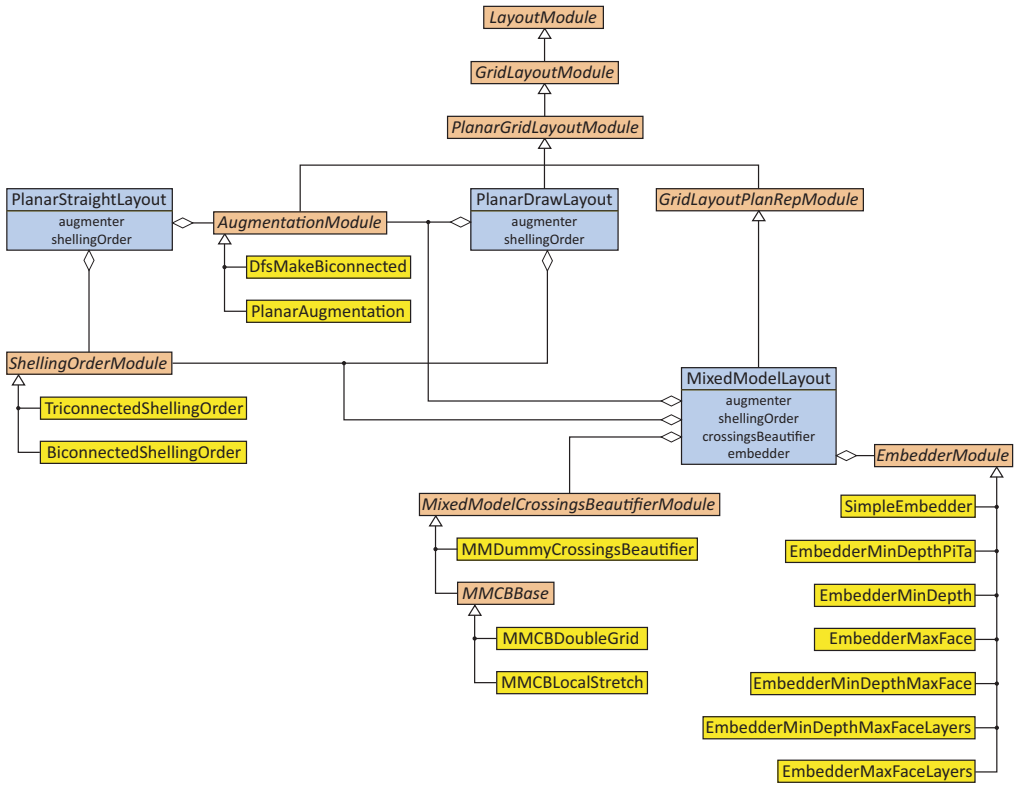

Figure 15.3 Planar graph drawing in the OGDF library.

for drawing general, non-planar graphs, the *Sugiyama framework* for drawing hierarchical graphs, and the *multilevel-mixer*, which is a general framework for multilevel, energy-based graph layout.

15.4.1 Planar Drawing Algorithms

The planar layout algorithms can be divided into those that compute straight-line layouts and those that produce drawings with bends along the edges, in particular orthogonal layouts. Figure 15.3 gives an overview of the available layout algorithms and their module options; the orthogonal layouts (`OrthoLayout` and `KandinskyLayout`) are not covered in this figure, since they are only used within the planarization framework (see Figure 15.2).

Straight-Line Layouts. The class `PlanarStraightLayout` implements planar straight-line drawing algorithms based on a shelling (or canonical) order of the nodes. This order determines the order in which the nodes are placed by the algorithm. `PlanarStraightLayout` provides a module option `shellingOrder` for selecting the shelling order used by the algorithm. There are two implementations in the OGDF: Using the `TriconnectedShellingOrder` realizes the algorithm by Kant [Kan96], which draws triconnected planar graphs such that all internal faces are represented as convex polygons; using the `BiconnectedShellingOrder` realizes the relaxed variant by Gutwenger and Mutzel [GM97] for biconnected graphs. To make the drawing algorithm more generally applicable, it provides the additional module option `augmenter` for setting an augmentation module that is called as a preprocessing step. This augmentation module must ensure that the graph has the required connectivity when computing the shelling order. In all cases, the algorithm

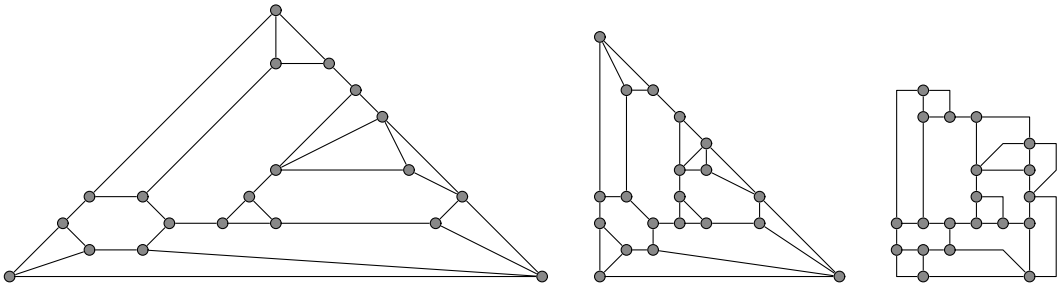

Figure 15.4 A triconnected planar graph drawn with `PlanarStraightLayout`, `PlanarDrawLayout`, and `MixedModelLayout` (from left to right).

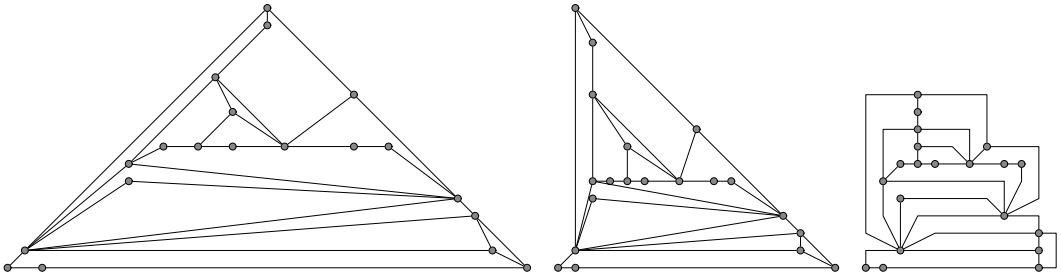

Figure 15.5 A biconnected planar graph drawn with `PlanarStraightLayout`, `PlanarDrawLayout`, and `MixedModelLayout` (from left to right).

guarantees to produce a drawing on a $(2n-4) \times (n-2)$ grid, where $n \geq 3$ is the number of nodes in the graph.

An improved version of `PlanarStraightLayout` is `PlanarDrawLayout`. It provides the same module options but implements a slightly modified drawing algorithm, which guarantees a smaller grid size of $(n-2) \times (n-2)$. Some sample drawings of a tri- and a biconnected graph are shown in Figures 15.4 and 15.5.

Mixed-Model Layouts. In mixed-model layouts, each edge is drawn in an orthogonal fashion, except for a small area around its endpoints. The class `MixedModelLayout` represents the layout algorithm by Gutwenger and Mutzel [GM97], which is based upon ideas by Kant [Kan96]. In particular, Kant's algorithm has been changed concerning the placement phase and the node boxes, which determine the routing of the incident edges around a node. It has also been generalized to work for connected planar graphs.

This algorithm draws a d-planar graph G on a grid such that every edge has at most three bends and the minimum angle between two edges is at least $\frac{2}{d}$ radians. The grid size is at most $(2n-6) \times (\frac{3}{2}n - \frac{7}{2})$, the number of bends is at most $5n-15$, and every edge has length $O(n)$ if G has n nodes.

Similar to the planar straight-line drawing algorithms, `MixedModelLayout` is based on a shelling order (`shellingOrder` module option) and an augmentation module is used to ensure the required connectivity. It also performs an enhancement for the placement of degree-one nodes, which are temporarily removed in a preprocessing step and later considered again when computing the node boxes. A further enhancement improves the drawing of edge crossings when using `MixedModelLayout` within the planarization approach (`PlanarizationGridLayout`). In this case, nodes representing crossings are drawn with four 90° angles, which is not the case for the original version. Figures 15.4 and 15.5 also show the corresponding mixed-model drawings of the graphs drawn with the planar straight-

line methods.

Orthogonal Layouts. Orthogonal drawings represent edges as sequences of horizontal and vertical line segments. Bends occur where these segments change directions. The OGDF provides orthogonal layout algorithms for graphs without degree restrictions; these are embedded in the planarization approach realized by `PlanarizationLayout`. Thereby, the orthogonal layout algorithm receives as input a planarized representation of the possibly non-planar input graph, i.e., a planar graph in which some nodes represent edge crossings.

By default, `PlanarizationLayout` uses `OrthoLayout` as layout algorithm. This is a variation of Tamassia's bend minimizing algorithm [Tam87], generalized to work with graphs of arbitrary node degrees. Tamassia's algorithm requires a planar graph G of maximal node degree four and a planar embedding Γ of G. Notice that pure orthogonal drawings in which the nodes are mapped to points in the grid are only admissible for this class of planar graphs. The computation of the layout follows the so-called *topology-shape-metrics* approach, see, e.g., [DETT99a]. According to the given planar embedding Γ the algorithm constructs a network in which a minimum-cost flow determines a bend-minimal representation of the orthogonal shape of G. In a last phase, a compaction module assigns lengths to this representation and thus fixes the coordinates of the drawing.

The OGDF contains two implementations for orthogonal compaction. `LongestPathCompaction` relies on computing longest paths in the so-called *constraint graphs*, an underlying pair of directed acyclic graphs that code placement relationships. `FlowCompaction` computes a minimum-cost flow in a pair of dual graphs and results in shorter edge lengths. Both algorithms rely on a dissection of the original face structures into rectangular faces. In addition, a branch-and-cut approach that produces provably optimal solutions for the two-dimensional compaction problem [KM99, Kla01] is in preparation.

In order to extend Tamassia's algorithm to graphs of arbitrary node degree, `OrthoLayout` uses ideas from quasi-orthogonal drawings [KM98] and Giotto layout [TDB88], combined with a local orthogonal edge routing algorithm. The common idea is to replace high-degree nodes by artificial faces which will be drawn as larger boxes in an intermediate drawing. The node is then placed within this boxes and its incident edges are routed orthogonally to the corresponding connection points on the surrounding box. An ER-diagram drawn by using `OrthoLayout` with `PlanarizationLayout` is shown in Figure 15.6.

An alternative to `OrthoLayout` is `KandinskyLayout` which extends the basic approach to graphs of arbitrary degree by allowing $0°$ angles between two successive edges adjacent to a node. Nodes in a graph are modeled as square boxes of unified size placed on a coarse grid, whereas edges are routed on a finer grid. Feasibility is achieved by maintaining the so-called *bend-or-end* property: Let e_1 and e_2 be the two edges incident to the same side of a node v in a Kandinsky drawing, e_1 following e_2 in the given embedding, and let f be the face to which e_1 and e_2 are adjacent. Then either e_1 must have a last bend with a $270°$ angle in f or e_2 must have a first bend with $270°$ angle in f. See [FK96] for a detailed description of the Kandinsky drawing model. The `KandinskyLayout` implementation does not use an extension of the original bend-minimization flow network as described in [FK96] to compute a shape for the input graph since this network has a flaw that may lead to suboptimal solutions or not a feasible solution at all [Eig03]. Instead, an ILP formulation is used, and hence `KandinskyLayout` requires COIN-OR.

15.4.2 Hierarchical Drawing Algorithms

Rooted Trees. The `TreeLayout` algorithm draws general trees in linear time. It is based on an efficient implementation [BJL06] of Walker's algorithm [RT81, Wal90] for drawing

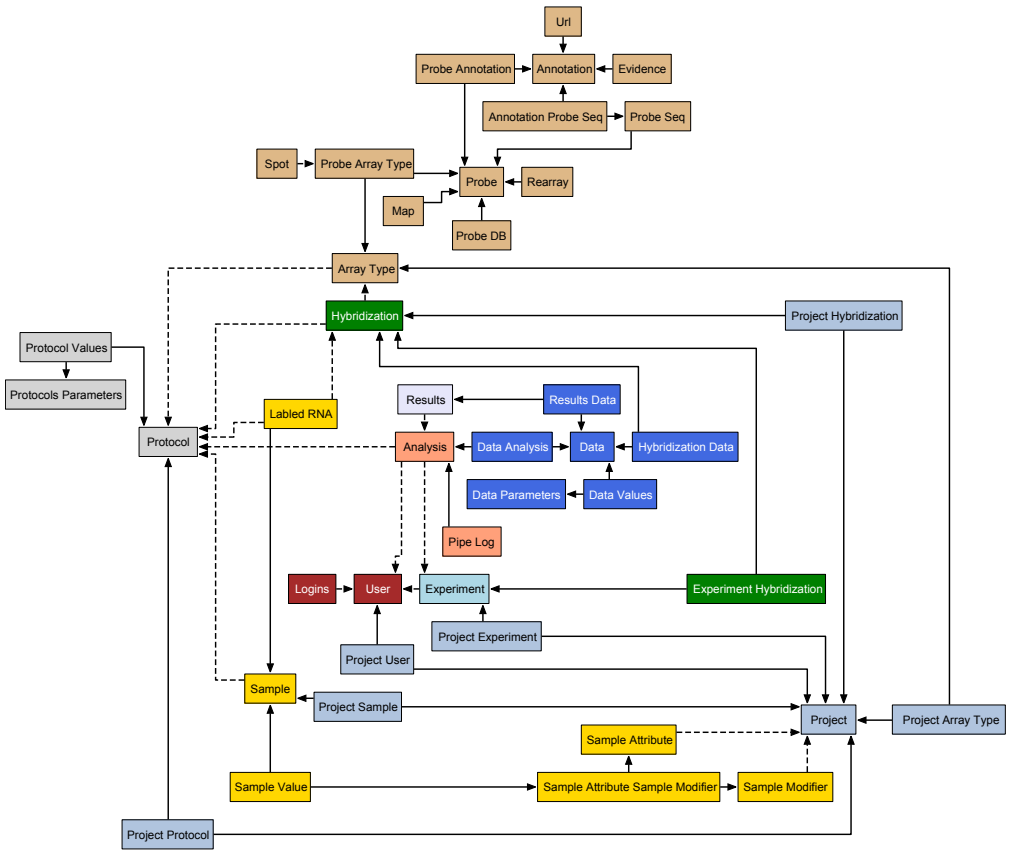

Figure 15.6 An entity-relationship diagram drawn with the planarization approach and `OrthoLayout`.

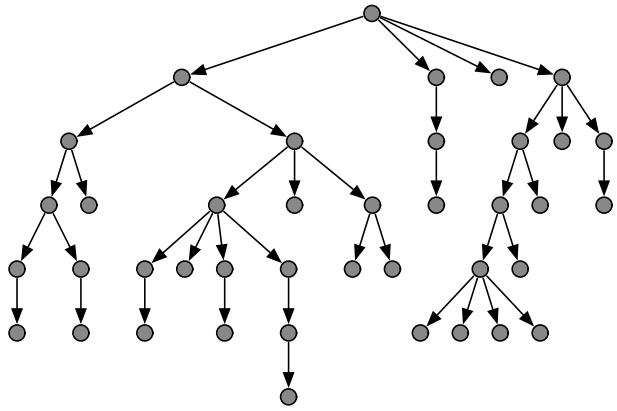

Figure 15.7 A tree drawn with `TreeLayout`.

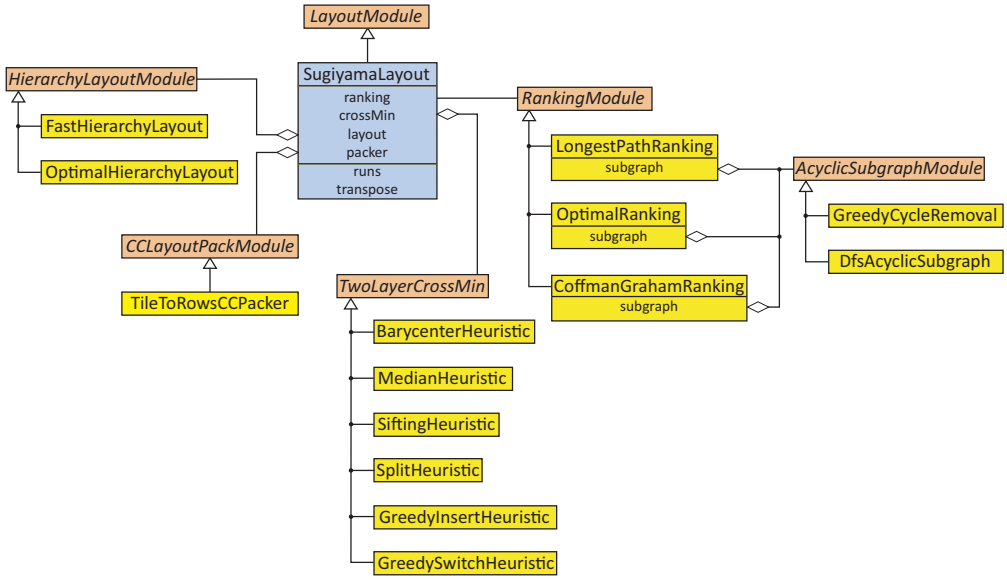

Figure 15.8 Sugiyama's framework for hierarchical graph layout in the OGDF.

trees. In the resulting straight-line drawing nodes on the same level lie on a horizontal line. The algorithm works recursively starting on the lowest level of the tree. In each step, the subtrees of a tree node (that have been laid out already) are placed as closely to each other as possible, resulting in a small size of the layout. `TreeLayout` also provides options for choosing between orthogonal or straight-line edge routing style, for the orientation of the layout (e.g., top to bottom or left to right), and for the selection of the root. Figure 15.7 shows an example drawing.

Sugiyama Framework. The OGDF provides a flexible implementation of Sugiyama's framework [STT81] for drawing directed graphs in a hierarchical fashion. This frameworks basically consists of three phases, and for each phase various methods and variations have been proposed in the literature. The corresponding OGDF implementation `SugiyamaLayout` provides a module option for each of the three phases; optionally, a packing module can be used to pack multiple connected components of the graph. The available OGDF modules and their dependencies are shown in Figure 15.8.

The three phases in Sugiyama's framework and their implementations are:

1. *Rank assignment:* In the first phase (realized by a `RankingModule`), the nodes of the input digraph G are assigned to layers. If G is not acyclic, then we compute a preferably large acyclic subgraph and reverse the edges not contained in the subgraph by one of the modules described in Section 15.3.1. Currently, the OGDF contains three algorithms for computing a layer assignment for an acyclic digraph in which the edges are directed from nodes on a lower level to nodes on a higher level. `LongestPathRanking` is based on the computation of longest paths and minimizes the number of layers (height of the drawing), `OptNodeRanking` minimizes the total edge length [GKNV93] (here the length of an edge is the number of layers it spans), and `CoffmanGrahamRanking` computes a layer assignment with a predefined maximum number of nodes on a layer (width of the drawing) [CG72]. If edges span several layers, they are split by inserting additional artificial nodes such that edges connect only nodes on neighboring layers.
2. *k-layer crossing minimization:* The second phase determines permutations of the nodes on each layer such that the number of edge crossings is small. The corresponding optimization problem is NP-hard. A reasonable method consists of visiting the layers from top to bottom, fixing the order of the nodes on the layer and trying to find a permutation of the nodes on the lower next layer that minimizes the number of crossings between edges connecting the two adjacent layers, also referred to as two-layer crossing minimization (realized by a `TwoLayerCrossMin` module). Then, the algorithm proceeds from bottom to top and so on until the total number of crossings does not decrease anymore. `SugiyamaLayout` contains a sophisticated implementation that uses further improvements like calling the crossing minimization several times (controlled by the parameter `runs`) with different starting permutations, or applying the `transpose` heuristic described in [GKNV93].
   Several heuristics for two-layer crossing minimization have been proposed. The library offers the choice between the barycenter heuristic [STT81], the weighted median heuristic [GKNV93], the sifting heuristic [MSM00], as well as the split, the greedy insert, and the greedy switch heuristics presented in [EK86].
3. *Coordinates assignment:* The third phase (realized by a `HierarchyLayoutModule`) computes the final coordinates of the nodes and bend points of the edges, respecting the layer assignment and ordering of the nodes on each layer. The OGDF contains two implementations for the final coordinate assignment phase. The first, `OptimalHierarchyLayout`, tries to let edges run as vertical as possible by solving a linear program; the second, `FastHierarchyLayout`, proposed by Buchheim, Jünger, and Leipert [BJL00] guarantees at most two bends per edge and draws the whole part between these bends vertically.

Figure 15.9 shows a layered drawing produced by `SugiyamaLayout` using the LP-based coordinate assignment method. The digraph displays the history of the UNIX operating system and the layers correspond to the time line depicted on the right side[3].

Upward Planarization. Though the commonly applied approach for hierarchical graph drawing is based on the Sugiyama framework, there is a much better alternative that produces substantially less edge crossings. This alternative adapts the crossing minimization procedure known from the planarization approach and is thus called *upward planariza-*

[3]Source: `http://en.wikipedia.org/wiki/File:Unix_history-simple.svg`

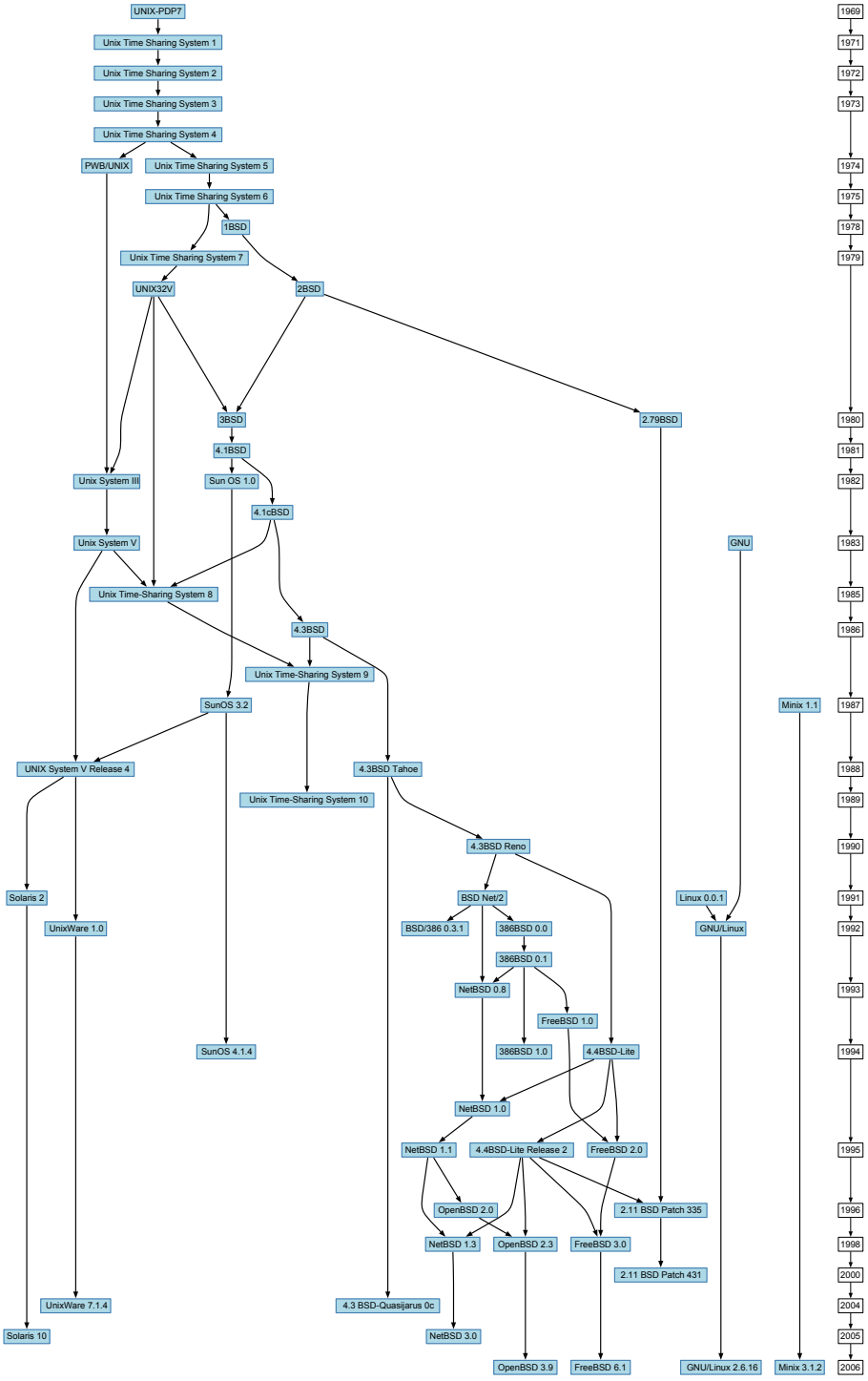

Figure 15.9 A layered digraph illustrating the history of UNIX; each layer represents a point in time. Drawn by applying Sugiyama layout and the LP-based coordinate assignment with angle optimization and special node balancing.

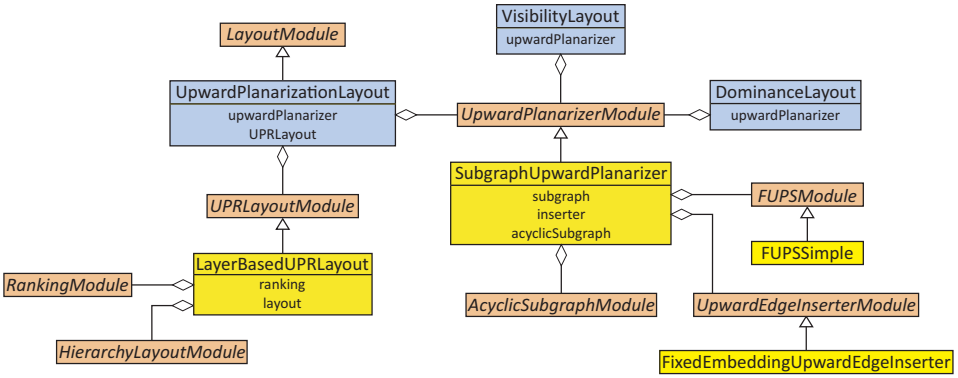

Figure 15.10 The upward planarization framework for hierarchical graph layout in the OGDF; modules for ranking, hierarchy layout, and acyclic subgraphs are omitted and can be found in Figure 15.8.

tion [CGMW10]. Like the traditional planarization approach for undirected graphs, the algorithm consists of two steps: In the first step, a *feasible upward planar subgraph* U is constructed; in the second step, the arcs not yet contained in U are inserted one-by-one so that few crossings arise. These crossings are replaced by dummy nodes so that the digraph in which arcs are inserted can always be considered upward planar. The final outcome of this crossing minimization procedure is an *upward planar representation*; it can be turned into a drawing of the original digraph by replacing the dummy nodes with arc crossings.

The upward planarization framework in the OGDF follows the presentations in [CGMW09] and [CGMW10]; see Figure 15.10. The class `UpwardPlanarizationLayout` represents the layout algorithm, which is implemented in two phases: The first phase realizes the upward crossing minimization procedure and computes an upward planarized representation of the input digraph; the second phase is realized by a `UPRLayoutModule` and computes the final layout. Currently, the layout computation is implemented by reusing modules from Sugiyama's framework, namely the rank assignment and hierarchy layout modules.

The crossing minimization step, realized by an `UpwardPlanarizerModule`, is the heart of the upward planarization. The OGDF modularizes this step similarly as for the planarization approach. First, a feasible upward planar subgraph is computed by a `FUPSModule`, which is implemented by `FUPSSimple`, and then the remaining edges are inserted by an `UpwardEdgeInserterModule`, implemented by applying a fixed embedding approach (`FixedEmbeddingUpwardEdgeInserter`). Figure 15.11 compares two upward drawings of the same digraph, one produced by the Sugiyama approach and the other one by applying upward planarization. Typically, Sugiyama drawings tend to become quite flat, thus enforcing many crossings, whereas the upward planarization approach unfolds the digraph well, thereby saving a lot of crossings and revealing the true structure of the digraph.

The crossing minimization step is also used by two further algorithms: `VisibilityLayout` based on the computation of a visibility representation by Rosenstiehl and Tarjan [RT86] and `DominanceLayout` based on dominance drawings of s-t-planar digraphs. An *s-t-planar digraph* is a directed, acyclic planar graph G with exactly one source s and exactly one sink t. `DominanceLayout` applies the layout algorithm for s-t-planar digraphs by Di Battista, Tamassia, and Tollis [DTT92]. If the input digraph G contains no transitive edges, the algorithm computes a planar dominance grid drawing of G, i.e., a straight-line embedding such that, for any two nodes u and v, there is a directed path from u to v if and only if $x(u) \leq x(v)$ and $y(u) \leq y(v)$. Dominance drawings characterize the transitive closure

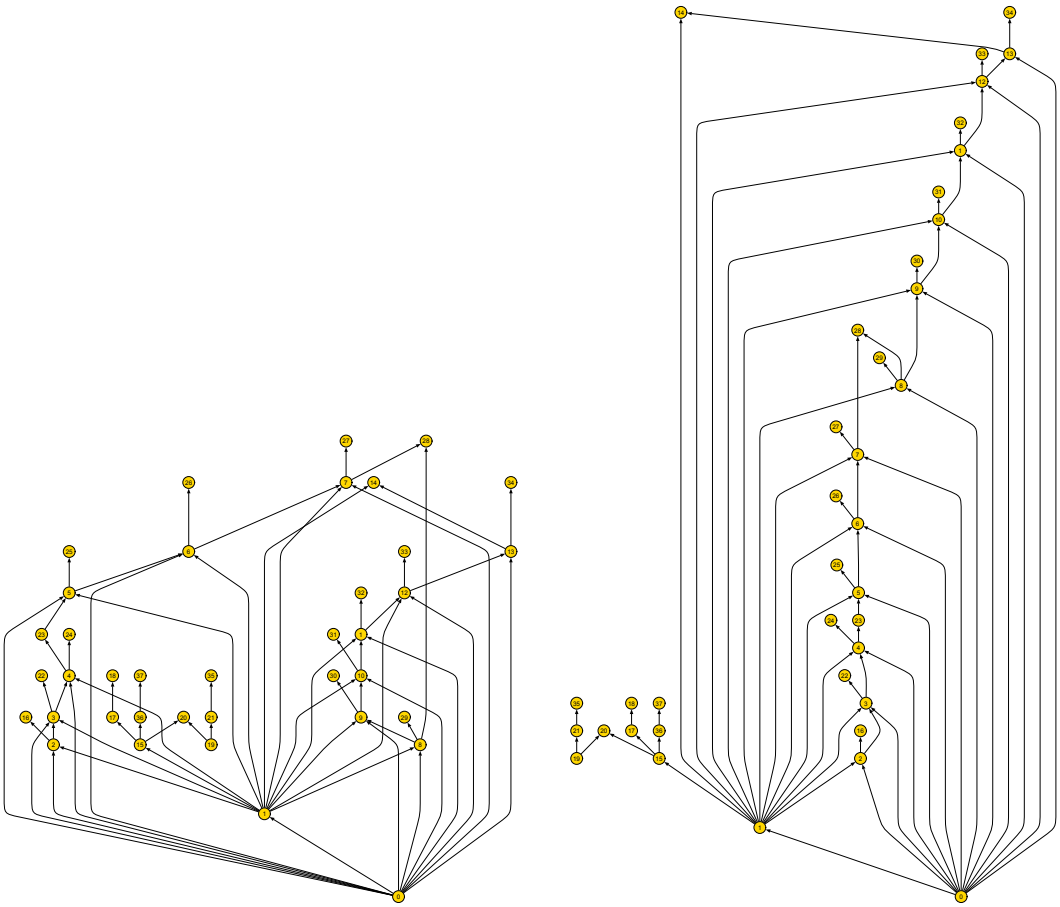

Figure 15.11 Two upward drawings of the same graph, drawn with `SugiyamaLayout` (left, 27 crossings) and `UpwardPlanarizationLayout` (right, 1 crossing).

of a digraph by means of the geometric dominance relation among the nodes [DTT92]. If G does contain transitive edges, the algorithm splits these edges by introducing artificial nodes and computes a dominance drawing for the resulting digraph in which the artificial nodes represent bend points.

15.4.3 Energy-based Drawing Algorithms

Energy-based drawing algorithms constitute the most common drawing approach for undirected graphs. They are reasonably fast for medium sized graphs, intuitive to understand, and easy to implement—at least in their basic versions. The fundamental underlying idea of energy-based methods is to model the graph as a system of interacting objects that contribute to the overall energy of the system, such that an energy-minimized state of the system corresponds to a nice drawing of the graph. In order to achieve such an optimum, an energy or cost function is minimized. There are various models and realizations for this approach, and the flexibility in the definition of both the energy model and the objective function enables a wide range of optimization methods and applications. There is a wealth of publications concerning energy-based layout methods; see [DETT99b, KW01] for an overview and the comprehensive discussion in Chapter **??**.

Single-level Algorithms. The OGDF provides implementations for several classical algorithms, such as the force-directed spring embedder algorithm [Ead84], the grid-variant of Fruchterman and Reingold [FR91] (`SpringEmbedderFR`), and the simulated annealing approach by Davidson and Harel [DH96] (`DavidsonHarelLayout`). We also implemented the energy-based approach by Kamada and Kawai [KK89] (`SpringEmbedderKK`), which uses the shortest graph-theoretic distances as ideal pairwise distance values and subsequently tries to obtain a drawing that minimizes the overall difference between ideal and current distances. Further implementations include the GEM algorithm [FLM95] (`GEMLayout`) and Tutte's barycenter method [Tut63] (`TutteLayout`). All implementations of energy-based drawing algorithms are directly derived from the class `LayoutModule`.

An important advantage of energy-based methods—based on the iterative nature of the numerical methods for computing the layout—is that they provide an animation of the change from a given layout to a new one, thus allowing us to use a given drawing as input. In addition, these algorithms support stopping the computation when either the improvement of successive steps falls under a certain threshold or as soon as a prespecified energy value is reached. Our implementations therefore provide the corresponding interfaces to adjust the respective parameters.

Multi-level Algorithms. In addition to these single level algorithms, the OGDF provides a generic framework for the implementation of multilevel algorithms, realized by the class `ModularMultilevelMixer`. Multilevel approaches can help to overcome local minima and slow convergence problems of single level algorithms. Their result does not depend on the quality of an initial layout, and they are well suited also for large graphs with up to tens or even hundreds of thousands of nodes.

The multilevel framework allows us to obtain results similar to those of many different multilevel layout realizations [Wal03, GK02, HJ04a]. Instead of implementing these versions from scratch, only the main algorithmic phases— coarsening, placement, and single level layout—have to be implemented or reused from existing realizations. The module concept allows us to plug in these implementations into the framework, enabling also a comparison of different combinations as demonstrated in [BGKM10]. Figure 15.12 shows two example drawings of large graphs.

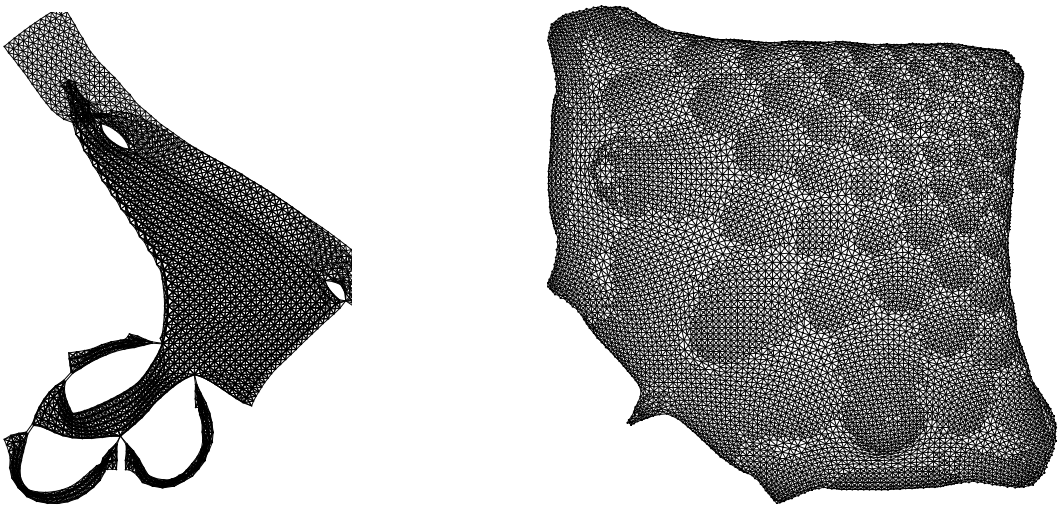

Figure 15.12 Two drawings obtained with OGDF's multilevel framework: graph `data` (left; 2,851 nodes; 15,093 edges) and graph `crack` (right; 10,240 nodes; 30,380 edges).

On the one hand, the multilevel framework provides high flexibility for composing multilevel approaches out of a variety of realizations for the different layout steps. On the other hand, this modularity prohibits fine-tuning of specific combinations by adjusting the different phases to each other. Therefore the OGDF also contains a dedicated implementation of the fast multipole multilevel method (`FMMMLayout`) by Hachul and Jünger [HJ04b], as well as an engineered and optimized version of this algorithm supporting multicore hardware [Gro09] (`FastMultipoleMultilevelEmbedder`). Figure 15.13 shows a drawing of a very large graph with 143,437 nodes, which was obtained—using this engineered version—in just 2.1 seconds on an Intel Xeon E5430 (2.66GHz) quadcore machine.

15.4.4 Drawing Clustered Graphs

A clustered graph $C = (G, T)$ is a tuple consisting of a graph $G = (V, E)$ and a hierarchical structuring T called *cluster tree*. Every node of V is assigned to exactly one inner node of T, which is the cluster to which it belongs.

In the OGDF, a clustered graph C is represented by an instance of class `ClusterGraph`, which stores the necessary information together with a reference to the underlying graph G. The OGDF provides methods to test c-planarity of arbitrary clustered graphs and to draw such graphs in either orthogonal or hierarchical style.

Orthogonal Layout. Similar to the planarization approach for general graphs, we implemented the planarization approach for clustered graphs based on the method by Di Battista et al. [DDM01], which is realized by the class `ClusterPlanarizationLayout`. This method uses the topology-shape-metrics approach and is suitable only for *c-connected* clustered graphs, i.e., clustered graphs with the property that every subgraph induced by the nodes of some cluster c and its subclusters is connected.

The cluster planarization algorithm works as follows: First, we calculate a minimum spanning tree for each cluster, thereby treating its subclusters as simple nodes. Afterwards these trees are joined together. We start to insert the remaining edges one after another if this is possible without introducing any crossings. We call the resulting graph the *maximal planar cluster subgraph* G'. Hence, after this step there generally remains a set of edges

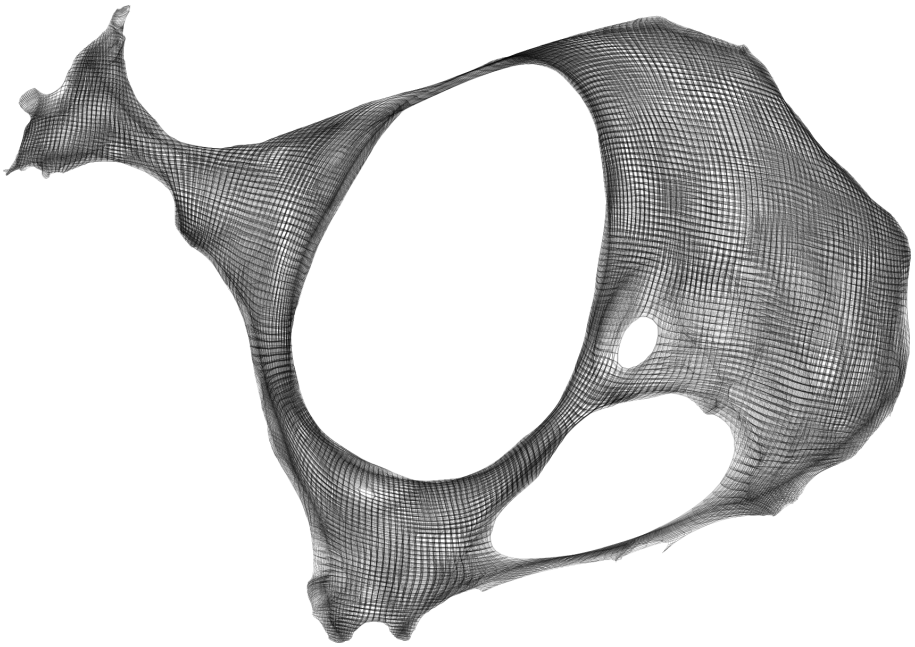

Figure 15.13 The graph `fe_ocean` (143,437 nodes; 409,593 edges) drawn with the `FastMultipoleMultilevelEmbedder` in 2.1 seconds.

which could not yet be inserted.

We apply the following steps for every edge $e = (u, v)$ that still has to be inserted: We generate a dual graph D with the following properties: faces within the same cluster are joined by bidirectional arcs. Arcs between faces of different clusters are only generated if these clusters are on the path between u and v in T. The direction of such arcs is chosen accordingly. Finally we add arcs from u to all of its incident faces, and arcs from all faces incident to v, to v. Then we search for the shortest path between u and v in D and generate edge crossings according to the used arcs.

Thanks to the OGDF's modularity we can easily reuse all available code concerning bend minimization and compaction—originally only intended for planar graphs—without a single change.

In order to also cope with non-c-connected clustered graphs, the OGDF provides two quite different approaches: As a simple and fast heuristic, non-connected clusters are made connected by temporarily adding single dummy edges between the components of the induced subgraphs. A much more sophisticated approach is implemented by the class `MaximumCPlanarSubgraph`: This class applies a branch-and-cut approach that computes the maximum c-planar subgraph of a clustered graph [CGJ+08]. The result can be used for the first step of the cluster planarization approach and at the same time also constitutes the first practical c-planarity testing algorithm for arbitrary (i.e., also non-c-connected) clustered graphs. The branch-and-cut approach is based on the result that every c-planar clustered graph can be augmented to a completely connected clustered graph, i.e., where for each cluster both the cluster and its complement are connected [CW06]. As the call function also returns the set of edges that is eventually added to achieve c-connectivity, these edges can be used in order to make the input graph c-connected, allowing us to apply the planarization and drawing approach without adding unnecessary crossings.

Figure 15.14 shows an example drawing of a c-planar (but not c-connected) clustered

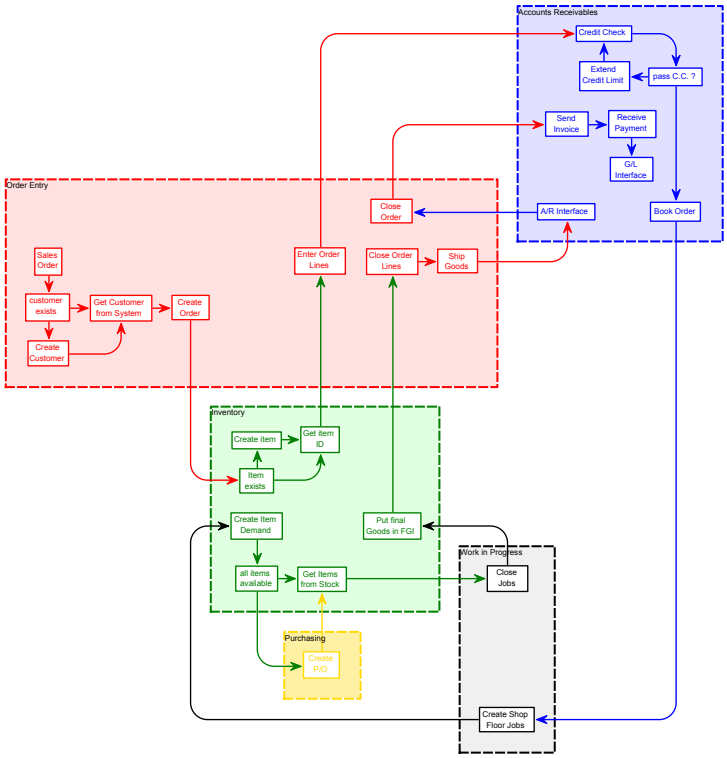

Figure 15.14 A clustered graph drawn in orthogonal style with `ClusterPlanarization-Layout`.

graph, obtained with `ClusterPlanarizationLayout` and the simple heuristic for making the clustered graph c-connected.

Hierarchical Layout. For drawing directed graphs with an additional cluster structure in a hierarchical style, `SugiyamaLayout` provides an additional call method. This methods implements a cluster hierarchical layout algorithm that is based on Sugiyama's framework as described by Sanders [San96a, San96b] and applies improvements for the crossing reduction strategy proposed by Schreiber [Sch01] and Forster [For02].

Though this approach would also allow us to draw *compound graphs*, i.e., a generalization of clustered graphs where edges can also be attached at clusters, the implementation currently supports only clustered graphs. The reason is that there is yet no representation of compound graphs in the OGDF; it will be added in a future release.

15.5 Success Stories

This section showcases some outstanding implementations in the OGDF and the story behind their development. These are also good examples for demonstrating design decisions and engineering aspect.

15.5.1 SPQR-Trees

In the early 1970s, Hopcroft and Tarjan [HT73a] published the first linear-time algorithm for computing the triconnected components of a graph. This decomposition is in particular important in graph drawing—here usually known as the data structure SPQR-tree—, as it allows us to encode all planar embedding of a graph. Hence, this algorithm has been cited over and over again for showing that SPQR-trees can be constructed in linear time. However, for a long time nobody was able to come up with an implementation of this algorithm, since the paper was hard to understand and contained various flaws, thus preventing a straightforward implementation.

This situation is a classical example for showing the need of publicly available reference implementations, revealing all the algorithmic details and simplifying the application of the algorithm. A break-through was achieved in the early 2000s—almost 20 years after the publication of the algorithm—by Gutwenger and Mutzel [GM01]. They described how to fix the flaws in the Hopcroft and Tarjan algorithm, and also provided a stable implementation of SPQR-trees in the AGD library. This implementation is now part of the OGDF and thus open source, allowing everybody to study and understand it. Since this implementation is publicly available, we observed a lot of interest in it, ranging from applications that just apply the data structure to re-implementations, e.g., in other programming languages.

15.5.2 Exact Crossing Minimization

One of the most challenging problems in graph drawing is the *crossing number problem*, i.e.: What is the minimum number of edge crossings required when drawing a given graph? See Chapter **??** for an extensive introduction to this topic. For a long time, no exact algorithms existed that could compute the crossing number for at least some interesting graphs in practice. The classical benchmark instances for evaluating crossing minimization algorithms are the Rome graphs, a benchmark set of quite sparse graphs with up to 100 nodes. The first approach that could compute the exact solutions for a handful of interesting Rome graphs was based on an ILP formulation presented by Buchheim et al. [BEJ$^+$05] and implemented using AGD and CPLEX. In the following years, this approach was revised and reimplemented using the OGDF by Chimani et al. [BCE$^+$08, CGM09], and the resulting implementation was able to solve many more graphs. The key ideas were to use branch-and-cut-and-price (by applying the ABACUS framework) and better primal heuristics (the planarization approach provided by the OGDF). The currently best algorithm for exact crossing minimization [Chi08, CMB08] is also implemented using the OGDF, and now allows us to solve the majority of Rome graphs exactly.

This example demonstrates how the OGDF's modular design supports the development of new algorithms. Here, the planarization approach (see Section 15.3.3) is used as primal heuristic, as well as other important components like testing planarity, extraction of Kuratowski subdivisions [CMS08], and the non-planar core reduction [CG09] as a preprocessing strategy. It also shows that exact algorithms based on ILP formulations require additional frameworks providing an LP-solver and support for the design and implementation of branch-and-cut(-and-price) algorithms. Hence, we decided to (optionally) use the libraries COIN-OR as LP-solver interface (which also allows us to choose CPLEX as LP-solver) and ABACUS in the OGDF.

15.5.3 Upward Graph Drawing

The classical approach for upward drawing of acyclic digraphs is Sugiyama's framework, which was already proposed in the early 1980s. The first step of this framework layers the graph, and the subsequent steps ensure that each node is finally placed on its assigned layer. It is well known that such a fix layer assignment forces unnecessary crossings in the drawing, but the Sugiyama framework is still widely used. A breakthrough in upward crossing minimization was recently achieved by Chimani et al. [CGMW08, CGMW09, CGMW10]. They propose a new method for upward crossing minimization that does not need to layer the graph and report on substantial reduction in crossings (compared to Sugiyama's framework and other approaches) for commonly used benchmark graphs.

They developed this new approach using the OGDF and made use of the OGDF's modular design by reusing some of the modules from Sugiyama's framework (see also Figure 15.10), as well as sophisticated algorithms like upward planarity testing for sT-digraphs. Within their experimental study, they could apply the OGDF's Sugiyama layout algorithm providing state-of-the-art crossing minimization heuristics for the layered approach. The resulting implementation is also modularized and thus allows us to easily replace particular phases of the algorithm with alternative implementations.

This example demonstrates how the OGDF helps in developing alternative approaches, and how new frameworks can be established such that other users can easily experiment with it and modify some of the phases.

Acknowledgement

The OGDF, as it is today, is by far not only the product of the authors of this chapter. It benefits from contributions of many additional supporters, in alphabetical order:[4] *Dino Ahr, Gereon Bartel, Christoph Buchheim, Tobias Dehling, Martin Gronemann, Stefan Hachul, Mathias Jansen, Thorsten Kerkhof, Joachim Kupke, Sebastian Leipert, Daniel Lückerath, Jan Papenfuß, Gerhard Reinelt, Till Schäfer, Jens Schmidt, Michael Schulz, Andrea Wagner, René Weiskircher, Hoi-Ming Wong, Bernd Zey.*

[4] See also `http://www.ogdf.net/doku.php/team:about` for an up-to-date list.

Bibliography

[AGMN97] D. Alberts, C. Gutwenger, P. Mutzel, and S. Näher. AGD-library: A library of algorithms for graph drawing. In *Proc. WAE '97*, pages 112–123, 1997.

[BCE+08] C. Buchheim, M. Chimani, D. Ebner, C. Gutwenger, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. A branch-and-cut approach to the crossing number problem. *Discrete Optimization, Special Issue in Memory of George B. Dantzig*, 5(2):373–388, 2008.

[BDMT98] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM J. Comput.*, 27(1):132–169, 1998.

[BEJ+05] C. Buchheim, D. Ebner, M. Jünger, P. Mutzel, and R. Weiskircher. Exact crossing minimization. In P. Eades and P. Healy, editors, *Graph Drawing (Proc. GD '05)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.

[BGKM10] G. Bartel, C. Gutwenger, K. Klein, and P. Mutzel. An experimental evaluation of multilevel layout methods. In *18th International Symposium on Graph Drawing 2010 (GD10)*, number 6502 in LNCS, pages 80–91. Springer-Verlag, 2010.

[BJL00] C. Buchheim, M. Jünger, and S. Leipert. Fast layout algorithm for k-level graphs. In J. Marks, editor, *Proc. Graph Drawing 2000*, volume 1984 of *LNCS*, pages 229–240. Springer, 2000.

[BJL06] C. Buchheim, M. Jünger, and S. Leipert. Drawing rooted trees in linear time. *Software: Practice and Experience*, 36(6):651–665, 2006.

[BL76] K. Booth and G. Lueker. Testing for the consecutive ones property interval graphs and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13:335–379, 1976.

[BM04] J. M. Boyer and W. Myrvold. On the cutting edge: simplified $o(n)$ planarity by edge addition. *J. Graph Algorithms Appl.*, 8(3):241–273, 2004.

[CG72] E. G. Coffman and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[CG09] M. Chimani and C. Gutwenger. Non-planar core reduction of graphs. *Discrete Mathematics*, 309(7):1838–1855, 2009.

[CGJ+08] M. Chimani, C. Gutwenger, M. Jansen, K. Klein, and P. Mutzel. Computing maximum c-planar subgraphs. In I. G. Tollis and M. Patrignani, editors, *Graph Drawing*, volume 5417 of *Lecture Notes in Computer Science*, pages 114–120. Springer, 2008.

[CGM09] M. Chimani, C. Gutwenger, and P. Mutzel. Experiments on exact crossing minimization using column generation. *ACM Journal of Experimental Algorithmics*, 14(3):4.1–4.18, 2009.

[CGMW08] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Layer-free upward crossing minimization. In C. McGeoch, editor, *Proceedings of the 7th International Symposium on Graph Drawing (WEA 2008)*, volume 5038 of *Lecture Notes in Computer Science*, pages 55–68. Springer-Verlag, 2008.

[CGMW09] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Upward planarization layout. In D. Eppstein and E. Gansner, editors, *Proceedings of the 17th*

0-8493-8597-0/01/$0.00+$1.50

Symposium on Graph Drawing 2009 (GD 2009), volume 5849 of *Lecture Notes in Computer Science*, pages 94–106. Springer-Verlag, 2009.

[CGMW10] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Layer-free upward crossing minimization. *ACM Journal of Experimental Algorithmics*, 15:Article No. 2.2, 2010.

[Chi08] M. Chimani. *Computing crossing numbers*. PhD thesis, TU Dortmund, 2008. `http://hdl.handle.net/2003/25955`.

[CMB08] M. Chimani, P. Mutzel, and I. Bomze. A new approach to exact crossing minimization. In *Proc. ESA '08*, volume 5193 of *LNCS*, pages 284–296. Springer, 2008.

[CMS08] M. Chimani, P. Mutzel, and J. M. Schmidt. Efficient extraction of multiple Kuratowski subdivisions. In *Proc. GD '07*, volume 4875 of *LNCS*, pages 159–170. Springer, 2008.

[CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *J. Comput. Syst. Sci.*, 30(1):54–76, 1985.

[CW06] S. Cornelsen and D. Wagner. Completely connected clustered graphs. *J. Discrete Algorithms*, 4(2):313–323, 2006.

[DDM01] G. Di Battista, W. Didimo, and A. Marcandalli. Planarization of clustered graphs. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 60–74. Springer, 2001.

[DETT99a] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.

[DETT99b] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, 1999.

[DH96] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996.

[DT89] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 436–441, 1989.

[DT96] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.

[DTT92] G. Di Battista, R. Tamassia, and I. G. Tollis. Constrained visibility representations of graphs. *Inform. Process. Lett.*, 41:1–7, 1992.

[Ead84] P. A. Eades. A heuristic for graph drawing. In *Congressus Numerantium*, volume 42, pages 149–160, 1984.

[Eig03] M. Eiglsperger. *Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach*. Phd-thesis, Eberhardt-Karl-Universität (Tübingen), 2003.

[EK86] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria*, 21(A):89–98, 1986.

[EL95] P. Eades and X. Lin. A new heuristic for the feedback arc set problem. *Australian Journal of Combinatorics*, 12:15–26, 1995.

[FK96] U. Fössmeier and M. Kaufmann. Drawing high degree graphs with low bend number. In *Graph Drawing (Proc. GD '95)*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[FLM95] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 388–403, London, UK, 1995. Springer-Verlag.

[FM98] S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*, pages 260–269, San Francisco, California, 1998. ACM Press.

[For02] M. Forster. Applying crossing reduction strategies to layered compound graphs. In S. G. Kobourov and M. T. Goodrich, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 276–284. Springer, 2002.

[FR91] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.

[GK02] P. Gajer and S. G. Kobourov. GRIP: Graph drawing with intelligent placement. *J. Graph Algorithms Appl.*, 6(3):203–224, 2002.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, 1993.

[GM97] C. Gutwenger and P. Mutzel. Grid embedding of biconnected planar graphs. Extended Abstract, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1997.

[GM01] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR trees. In J. Marks, editor, *Proceedings of the 8th International Symposium on Graph Drawing (GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer-Verlag, 2001.

[GM03] C. Gutwenger and P. Mutzel. Graph embedding with minimum depth and maximum external face. In G. Liotta, editor, *11th Symposium on Graph Drawing 2003, Perugia*, volume 2912 of *Lecture Notes in Computer Science*, pages 259–272. Springer-Verlag, 2003.

[GMW05] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005.

[GMZ09a] C. Gutwenger, P. Mutzel, and B. Zey. On the hardness and approximability of planar biconnectivity augmentation. In H. Q. Ngo, editor, *Proceedings of the 15th Annual International Computing and Combinatorics Conference 2009*, volume 5609 of *Lecture Notes in Computer Science*, pages 249–257. Springer-Verlag, 2009.

[GMZ09b] C. Gutwenger, P. Mutzel, and B. Zey. Planar biconnectivity augmentation with fixed embedding. In J. Fiala, J. Kratochvl, and M. Miller, editors, *Proceedings of the 20th International Workshop on Combinatorial Algorithms 2009*, volume 5874 of *Lecture Notes in Computer Science*, pages 289–300. Springer-Verlag, 2009.

[Gro09] M. Gronemann. Engineering the fast-multipole-multilevel method for multicore and SIMD architectures. Master's thesis, Technische Universität Dortmund, 2009.

[GT01] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, 2001.

[Gut10] C. Gutwenger. *Application of SPQR-Trees in the Planarization Approach for Drawing Graphs*. PhD thesis, Technische Universität Dortmund, Germany, Fakultät für Informatik, 2010.

[HJ04a] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In Janos Pach, editor, *Proc. Graph Drawing 2004*, volume 3383 of *LNCS*, pages 285–295. Springer-Verlag, 2004.

[HJ04b] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In Janos Pach, editor, *Proc. Graph Drawing 2004*,

volume 3383 of *LNCS*, pages 285–295. Springer-Verlag, 2004.

[HT73a] J. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[HT73b] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[JLM98] M. Jünger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design*, 17(7):609–612, 1998.

[JT00] M. Jünger and S. Thienel. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Software: Practice and Experience*, 30:1325–1352, 2000. See also http://www.informatik.uni-koeln.de/abacus/.

[JTS89] R. Jayakumar, K. Thulasiraman, and M. N. S. Swamy. $O(n^2)$ algorithms for graph planarization. *IEEE Trans. Comp.-Aided Design*, 8:257–267, 1989.

[Kan96] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16:4–32, 1996. (special issue on Graph Drawing, edited by G. Di Battista and R. Tamassia).

[KB91] G. Kant and H. L. Bodlaender. Planar graph augmentation problems. In *Proc. WADS '91*, volume 519 of *LNCS*, pages 286–298. Springer, 1991.

[Ker07] T. Kerkhof. Algorithmen zur Bestimmung von guten Graph-Einbettungen für orthogonale Zeichnungen. Master's thesis, University of Dortmund, 2007.

[KK89] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[Kla01] G. W. Klau. *A Combinatorial Approach to Orthogonal Placement Problems*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, Fachbereich Informatik, Technische Fakultät I, 2001.

[KM98] G. W. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max Planck Institut für Informatik, Saarbrücken, Germany, 1998.

[KM99] G. W. Klau and P. Mutzel. Optimal compaction of orthogonal grid drawings. In G. Cornuejols, R. E. Burkard, and G. J. Woeginger, editors, *Integer Programming and Combinatorial Optimization*, volume 1610 of *Lecture Notes Comput. Sci.*, pages 304–319. Springer-Verlag, 1999.

[KW01] M. Kaufmann and D. Wagner, editors. *Drawing Graphs*, volume 2025 of *LNCS*. Springer-Verlag, 2001.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs: Internat. Symposium (Rome 1966)*, pages 215–232, New York, 1967. Gordon and Breach.

[Mar10] K. Martin. Tutorial: COIN-OR: Software for the OR community. *Interfaces*, 40(6):465–476, 2010. See also http://www.coin-or.org.

[MN99] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MSM00] C. Matuszewski, R. Schönfeld, and P. Molitor. Using sifting for k-layer crossing minimization. In *Graph Drawing (Proc. GD '99)*, Lecture Notes in Computer Science. Springer-Verlag, 2000. to appear.

[PT00] M. Pizzonia and R. Tamassia. Minimum depth graph embedding. In M. Paterson, editor, *Proceedings of the 8th Annual European Symposium on Algorithms (ESA 2000)*, volume 1879 of *Lecture Notes in Computer Science*, pages 356–367. Springer-Verlag, 2000.

[RT81] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Trans. Softw. Eng.*,

SE-7(2):223–228, 1981.

[RT86] P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete Comput. Geom.*, 1(4):343–353, 1986.

[San96a] G. Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität Saarbrücken, 1996.

[San96b] G. Sander. *Visualisierungstechniken für den Compilerbau*. PhD thesis, Universität Saarbrücken, Germany, 1996.

[Sch01] F. Schreiber. *Visualisierung biochemischer Reaktionsnetze*. PhD thesis, Universität Passau, Germany, 2001.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.

[Tam87] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[TDB88] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.

[Tut63] W. T. Tutte. How to draw a graph. *Proc Lond Math Soc*, 13:743–767, 1963.

[Wal90] J. Q. Walker II. A node-positioning algorithm for general trees. *Softw. – Pract. Exp.*, 20(7):685–705, 1990.

[Wal03] C. Walshaw. A multilevel algorithm for force-directed graph-drawing. *J. Graph Algorithms Appl.*, 7(3):253–285, 2003.

[WT92] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

Index