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. The source code is available under the GNU General Public License (GPL v2 and v3).

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 [1], OGDF’s predecessor, was very popular in the past, since it covered a wide range of graph drawing algorithms and—together with the LEDA library [23]—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).

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). The algorithm then obtains this information and stores the computed layout in the GraphAttributes.

*Departement of Computer Science, TU Dortmund, Germany, carsten.gutwenger@tu-dortmund.de

This is an abstract related to a tutorial or demo given at GTS 2012. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear in a conference with formal proceedings and/or in a journal.
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. 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.

### 2.2 Self-contained and Portable Source Code

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 [22] as LP-solver and ABACUS [19] 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`.1 This utility converts graph layouts stored in GML or OGML file formats into images (e.g., PNG, JPEG, EPS, PDF, SVG).

### 3 Algorithms and Data Structures

Apart from many basic data structures and algorithms, OGDF contains the following sophisticated algorithms and data structures.

#### 3.1 General Graph Algorithms

**Augmentation and Subgraph 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.

There are two algorithms for augmenting a planar graph to a planar biconnected graph: The module `PlanarAugmentation` implements the Fialko-Mutzel augmentation algorithm [8], which performs very good in practice, and `DfsMakeBiconnected` is a simple, DFS-based algorithm. 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 [15].

Two modules are available to compute acyclic subgraphs of a digraph. `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 [7]. If $G = (V, A)$ is connected and has no two-cycles, the algorithm guarantees that the number of non-feedback arcs is at least $|A|/2 − |V|/6$.

**Graph Decomposition.** Besides the basic algorithms for computing the biconnected components of a graph [25, 18], 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 [5]. Both data structures can be build in linear time; the latter constructs the SPQR-tree by applying the corrected version [14] of Hopcroft

---

1 available at [http://www.ogdf.net/doku.php/project:gml2pic](http://www.ogdf.net/doku.php/project:gml2pic)
and Tarjan’s algorithm [17] 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.

**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.

### 3.2 Graph Drawing Algorithms

Graph drawing algorithms form the heart of the library. The OGDF provides flexible frameworks with interchangeable modules for various drawing paradigms, including the planarization approach 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.

**Planar Drawing Algorithms.** The class PlanarStraightLayout implements planar straight-line drawing algorithms based on a shelling order of the nodes. For a graph with $n$ nodes, the algorithm guarantees to produce a drawing on a \((2n - 4) \times (n - 2)\) grid, with convex faces if the graph is triconnected. An improved version of PlanarStraightLayout is PlanarDrawLayout. It guarantees a smaller grid size of \((n - 2) \times (n - 2)\). Some sample drawings of a triconnected graph are shown in Fig. 1.

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 [13], which is based upon ideas by Kant [21] and also uses a shelling order. 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 $\frac{\pi}{d}$ radians. The grid size is at most \((2n - 6) \times \left(\frac{3}{2}n - \frac{7}{2}\right)\). See Fig. 1 for an example.

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.

**Hierarchical Drawing Algorithms.** The OGDF provides a flexible implementation of Sugiyama’s framework [24] for drawing directed graphs in a hierarchical fashion; see Fig. 2. This framework 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.

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: the upward planarization approach [3] adapts the crossing minimization procedure known from the planarization approach. The OGDF contains a modularized framework for upward planarization as well, however there is currently only a single implementation for each phase.
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.

The OGDF provides implementations for several classical algorithms, such as the force-directed spring embedder algorithm [6], the grid-variant of Fruchterman and Reingold [10], the simulated annealing approach by Davidson and Harel [4], the algorithm by Kamada and Kawai [20], which uses the shortest graph-theoretic distances as ideal pairwise distance values, as well as the GEM algorithm [9] and Tutte’s barycenter method [26].

In addition to these single level algorithms, the OGDF provides a generic framework for the implementation of multilevel algorithms. 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 [27, 11, 16]. 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 [2]. Fig. 3 shows two example drawings of large graphs.

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 by Hachul and Jünger [16], as well as an engineered and optimized version of this algorithm supporting multicore hardware [12].

References

Figure 3: 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).