AGD–Library: A Library of Algorithms for Graph Drawing[1]

David Alberts
Martin-Luther-Universität Halle-Wittenberg
D-06099 Halle, Germany
e-mail: alberts@informatik.uni-halle.de

Carsten Gutwenger
Max-Planck-Institut für Informatik, Im Stadtwald
D-66123 Saarbrücken, Germany
e-mail: gutwenge@mpi-sb.mpg.de

Petra Mutzel
Max-Planck-Institut für Informatik, Im Stadtwald
D-66123 Saarbrücken, Germany
e-mail: mutzel@mpi-sb.mpg.de

and

Stefan Näher
Martin-Luther-Universität Halle-Wittenberg
D-06099 Halle, Germany
e-mail: naeher@informatik.uni-halle.de

ABSTRACT

A graph drawing algorithm produces a layout of a graph in two- or three-dimensional space that should be readable and easy to understand. Since the aesthetic criteria differ from one application area to another, it is unlikely that a definition of the "optimal drawing" of a graph in a strict mathematical sense exists. A large number of graph drawing algorithms taking different aesthetic criteria into account have already been proposed. In this paper we describe the design and implementation of the AGD–Library, a library of **A**lgorithms for **G**raph **D**rawing. The library offers a broad range of existing algorithms for two-dimensional graph drawing and tools for implementing new algorithms. The library is written in C++ using the LEDA platform for combinatorial and geometric computing ([16, 17]). The algorithms are implemented independently of the underlying visualization or graphics system by using a generic layout interface. Most graph drawing algorithms place a set of restrictions on the input graphs like planarity or biconnectivity. We provide a mechanism for declaring this precondition for a particular algorithm and checking it for potential input graphs. A drawing model can be characterized by a set of properties of the drawing. We call these properties the postcondition of the algorithm. There is support for maintaining and retrieving the postcondition of an algorithm.

1. Introduction

Visualization of structural information in form of graph layout diagrams is getting increasing attention. Graphs are widely used to model relational structures such as networks, e.g., of a subway,

[1]This work was supported in part by DFG-Grant Na 303/1-2, Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen"

of computers or the internet. Applications arise in economics (project management, work-flow diagrams, entity-relationship diagrams), computer science (compiler and software development tools, data base modelling, algorithm animation), social science (social networks) and natural science (flow-diagrams in Chemistry, visualization of excavations in Archeology).

A graph drawing algorithm takes as input a graph and computes a layout of the graph, i.e., a drawing in two or three-dimensional space by assigning coordinates to the vertices and mapping each edge to a simple curve. In most cases, these curves are straight lines or polygonal chains.

The layouts produced by a graph drawing algorithm should be "aesthetically nice" and "easy-to-understand". Some important criteria for readable diagrams are a small number of edge crossings, evenly distributed vertices and edges, short edges, few edge bends, and a small layout area or volume. There is a wide variety of graph drawing methods that take different aesthetic criteria into account.

Here, we present a library of **A**lgorithms for **G**raph **D**rawing, AGD–Library [1], that offers a broad range of existing algorithms for two-dimensional graph drawing and tools for implementing new algorithms. Section 2 describes the design goals and contains a comparison to related software packages, Section 3 gives an overview of the AGD–Library, and Section 4 deals with the implementation of the library. In particular, the generic graphics interface and the handling of the preconditions and postconditions of the drawing algorithms are described. Section 5 shows how to use the library and gives a complete example of extending it by a new algorithm. Future plans are described in Section 6.

The AGD–Library is a product of a cooperation of groups in Halle, Köln and Saarbrücken supported by the DFG in the program "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen". The AGD–Library is based on LEDA and will be distributed as a LEDA extension package (LEP). LEDA [16, 17] aims at being a comprehensive software platform for combinatorial and geometric computing. It provides a sizable collection of data types and algorithms. This collection includes most of the data types and algorithms described in the text books of the area. In particular, LEDA offers a very powerful and efficient graph data type that allows to implement graph algorithms very close to the typical text book presentation.

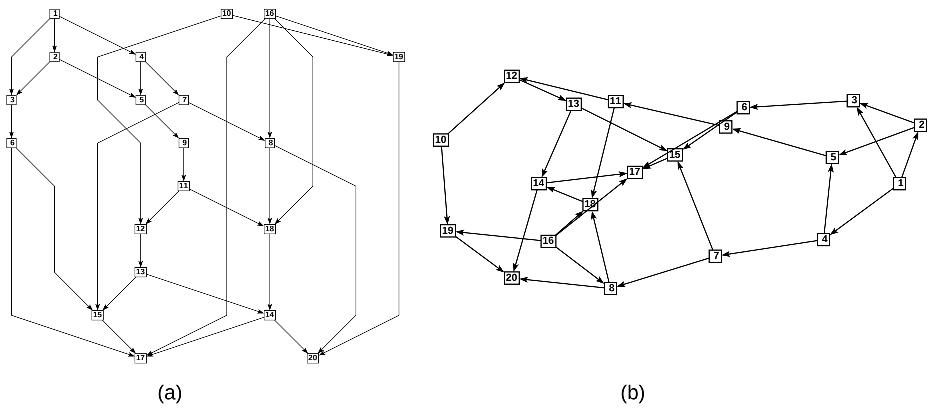

Figure 1: Two drawings of a nonplanar graph using the (a) hierarchical method [21] and (b) the force-directed method [7]

Drawing methods can be classified according to the kind of drawings they produce (e.g., hierarchical drawings, orthogonal drawings, straight-line drawings, circular drawings), the model they use (e.g., force-directed, planarization), and the class of graphs they can be applied to (e.g., planar graphs, directed acyclic graphs, trees).

Most available software packages for graph drawing use the *hierarchical drawing method* suggested

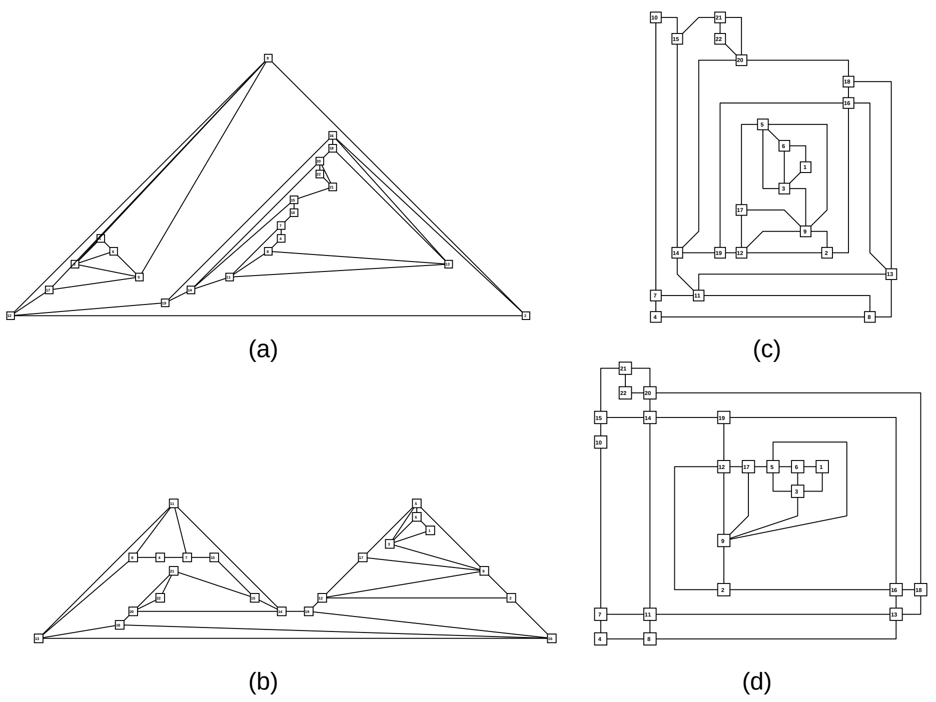

Figure 2: A planar graph drawn with the algorithms (a) in [4] (original) for triangulated graphs (b) in [8] for biconnected graphs (c) in [11, 8] (d) in [22, 12].

by Sugiyama et al. [21], since this method is easy to implement and the produced drawings look pleasant for graphs representing hierarchies (see Figure 1(a)). However, if the given graph does not represent a hierarchy the output can be very poor.

Another popular class of layout algorithms are the *force-directed* algorithms. Most of them are variants of the originally in [5] proposed *spring embedder* algorithm. Here, the energy of a physical system is minimized where the vertices are represented by masses that repel each other and the edges are modelled by springs that attract their endpoints. The idea is that the vertices and edges are equally distributed in the layout-area as soon as the physical system is in equilibrium (see Figure 1(b)). The method produces good results for symmetric graphs. However, since "real world graphs" are, in general, not symmetric, it is not very often used in practical applications.

Circular layout methods are useful for displaying ring and star networks. Here, the vertices are partitioned into groups, and each group of vertices is placed on a circle. The edges are drawn as straight lines.

For restricted classes of graphs various drawing algorithms exist that lead to nice drawings. Let us consider the class of planar graphs, i.e, graphs that can be drawn in the plane without edge crossings. It is well-known that planar graphs can always be drawn with straight edges [6]. Various *planar straight-line algorithms* [4, 11, 3, 8] exist. They proceed in the following way. In a first step, the vertices are sorted according to the requirements of a *canonical ordering*, and in a second step, the vertices are iteratively placed on a grid. The methods differ in the definition of the canonical ordering, that depends on the input graph (biconnected, triconnected or triangulated), and on the layout (convex drawing, layout-area) (see Figure 2(a)-(b)). Some problems of the above methods are that the angles between edges may become too small and that the area of the produced layout is too big. By using a modification called *mixed-model* [11, 8] (see Figure 2(c)) that allows edge bends some of these problems can be avoided.

Orthogonal drawings have no problems with small angles; here, the edges are drawn as chains of vertical and horizontal line segments. Of course, only graphs with maximal degree four can have an orthogonal drawing on a grid. For a fixed combinatorial embedding (plane graph) it is possible to construct a planar orthogonal drawing with the minimal number of bends efficiently [22]. Different approaches have been suggested for using orthogonal drawing algorithms for general planar graphs (with vertex degrees greater than four); the size of the vertices can be increased, the size of the underlying grid can be decreased, or non-orthogonal line segments may be allowed locally. The latter approach is called *quasi-orthogonal drawing* (see Figure 2(d) for an extension [12] of the algorithm in [22]).

By transforming a general graph into a planar graph via introducing additional vertices at edge crossings (planarization), all graph drawing algorithms for planar graphs can be used for drawing general graphs.

Besides the "conventional" graph drawing methods, various other approaches exist, e.g., dynamic graph drawing, proximity drawing, or visibility drawing. In a visibility representation vertices are drawn as horizontal line segments. Every edge is drawn as a vertical line segment connecting the horizontal segments corresponding to its end-vertices at a certain x–coordinate. Apart from these incidences all segments are disjoint.

2. Design Goals and Related Packages

In our opinion the following design goals are the most important in the field of implementing graph drawing algorithms. We try to address them in the AGD–Library as described in the following sections.

Flexibility

Since there are so many different approaches to graph drawing none of which can be ruled out a priori, it is important to provide the possibility of choosing among several different algorithms in order to get good results for a particular graph. Thus, systems offering only a single drawing algorithm are only of limited usefulness.

A related issue is the visualization component. Since graph drawing often appears as a subtask in a complex application, it is necessary to offer the possibility of using an arbitrary visualization component. A system for drawing graphs which is tied to a specific form of visualization significantly reduces the range of possible applications.

Ease of Usage

If a system is hard to use, then only a minority of "expert users" will be able to take advantage of it. Since we want to address a broad audience, we have to provide an easy interface.

Extensibility

Graph drawing is still a very active area of research. We want to help in closing the gap between interesting new ideas and usable software by providing a set of tools which helps in quickly and efficiently implementing new algorithms. In particular the reusage of already written modules appearing in the implementations of several graph drawing algorithms should be possible and easy, and different modules implementing the same functionality should be exchangeable.

At this point we give a brief overview of related libraries of graph drawing algorithms and how we see them in the light of the above formulated design goals.

Graph Layout Toolkit [15] is a commercial system containing four C++ class libraries for integration into application programs. The libraries represent the four layout methods hierarchical, force-directed, orthogonal, and circular drawing. Some of the classes can be reused in user programs. However, integration of new algorithms into the library is not supported.

Graphlet [9] is an object-oriented toolkit for implementing graph editors and graph drawing algorithms. It contains various methods for force-directed layouts, a hierarchical method, a tree layout algorithm, and planar graph drawing methods (partially provided by the AGD–Library). *Graphlet* provides a plugin concept which makes it easy to integrate new algorithms. However, reusability of parts of the algorithms is not intended. Moreover, the integration of graph drawing algorithms into graphical user interface application programs is not supported.

GDToolkit [2] is an experimental library of C++ classes based on LEDA which is still under construction. It will contain orthogonal drawing methods and methods based on planarization. *GDToolkit* is comparable with the AGD–Library in the sense that it allows the combination of various algorithms by the user. In order to deal with modules, in particular, with the precondition and postcondition of algorithms, *GDToolkit* introduces a unique inheritance hierarchy: A (graph drawing) algorithm is seen as a constructor of a class with a higher compatible level generating a new element of a different class with lower level according to the underlying hierarchy (we compare *GDToolkit* with AGD–Library in more detail in Section 4.2). The library can be used together with LEDA and also independently, hence it can be integrated in any user program.

3. Overview of the AGD–Library

3.1. The Structure of the AGD–Library

The two main parts of the AGD–Library are a collection of layout algorithms and a toolbox for extending the library by new implementations.

The design of the AGD–Library is based on the object-oriented features of C++ programming language. In particular, each layout algorithm is implemented as a C++ class. There is a common base class for all layout algorithms called `LayoutModule`. The main advantages of this approach are a generic interface to all layout algorithms, which allows for an easy integration into applications, and a mechanism for handling preconditions and postconditions of algorithms in the base class which is simply inherited by all algorithms. The base class provides two basic methods, `check` for checking the precondition of a particular algorithm given a specific graph, and `call` for calling the particular algorithm. In a specific algorithm, which is a class derived from the base class, the specific precondition is defined and a method for executing the specific algorithm called by `call` is implemented. `call` does not change the input graph, i.e., the input is a `const` parameter.

Figure 3 shows an overview of the AGD–Library from the user's perspective. The specific layout algorithms `Convex` or `MixedModel` are derived from the class `GridLayoutModule`. `GridLayoutModule` provides operations for all layout algorithms which place the vertices on integer coordinates. Moreover, it defines a common user interface for this type of layout algorithms. `GridLayoutModule` inherits from `LayoutModule` which provides common functionality and a basic interface for all layout algorithms.

Depending on the application different types of visualization of the computed layouts may be suitable including no visualization at all (e.g., just using the computed vertex coordinates as input to further computations). Thus it is important that the layout algorithms are not tied to a specific visualization tool. In AGD–Library we achieve this goal by manipulating the graphical attributes of a graph by means of a generic interface which is defined in the virtual class `LayoutInterface`. The user can specify which package to use for representing and displaying graphical attributes of a graph by choosing a predefined specialization of `LayoutInterface` (`GraphWinInterface` for LEDA's graph visualization component *GraphWin* and `GraphletInterface` for the graph editor *Graphlet* are already available) or by deriving her or his own class from `LayoutInterface`.

We are planning to provide a new class `GraphInterface` similar to `LayoutInterface` in order to decouple the layout algorithms from the representation of the topology of the input graph, too.

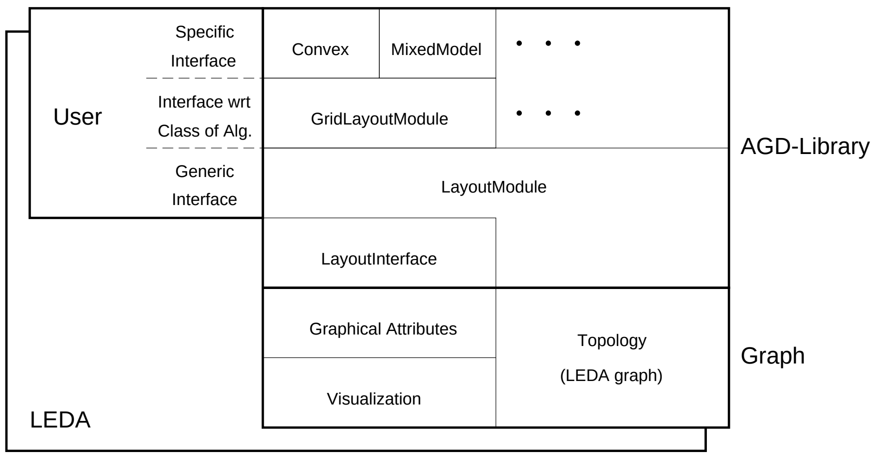

Figure 3: A Sketch of the User's View of the AGD–Library

At the moment a LEDA `graph` has to be used.

3.2. Layout Algorithms in the AGD–Library

The AGD–Library contains classical graph drawing algorithms as well as implementations of new algorithms (in some cases extensions of the former), that have been developed within the DFG project. In particular, AGD–Library is designed to support planar graph drawing algorithms and planarization methods in a flexible way. Moreover, AGD–Library will contain exact algorithms using ABACUS for NP-hard optimization problems occuring in graph drawing, e.g., crossing minimization, maximum planar subgraph. ABACUS is a software system providing a framework for the implementation of branch-and-cut algorithms [10].

Currently, the following graph layout algorithms are available.

- drawing planar graphs on the grid with straight edges

  `FPPLayout` (de Fraysseix, Pach, and Pollack [4]), `SchnyderLayout` (Schnyder [20]), `Convex` (Kant [11]), `ConvexDraw` (Chrobak and Kant [3]), `PlanarDraw` (Gutwenger and Mutzel [8]), `PlanarStraight` (Gutwenger and Mutzel [8]), `VisibilityRepresentation` (Rosenstiehl and Tarjan [19])

- drawing planar graphs on the grid with edge bends

  `MixedModelLayout` (extension of Kant's algorithm [11, 8]), `OrthogonalLayout` (extension of Tamassia's algorithm [22, 12])

- drawing special classes of planar graphs

  `TreeLayout` (Reingold and Tilford, Walker [18, 24, 13])

- drawing general graphs

  `SpringLayout` (Fruchterman, Reingold [7]), `TutteLayout` (Tutte [23]), `SugiyamaLayout` (Sugiyama, Tagawa, and Toda [21])

As soon as the planarization module is integrated, all the planar graph drawing algorithms can be used for drawing general graphs.

4. Implementation of the AGD–Library

4.1. The Generic Graphics Interface

The layout of a graph is expressed by graphical attributes of vertices and edges. Most graph drawing packages define an extended graph data structure, and use this structure with their graph drawing functions.

This approach contains several disadvantages: Many applications that require graph layout algorithms have already defined their own data structures, e.g., the graph editor *Graphlet* uses the class `GT_Graph` to represent a graph together with its graphical attributes. Another disadvantage is that a specific application could need further graphical attributes, such as bitmaps or PostScript graphics, or the provided graph class contains many unnecessary attributes.

The AGD–Library does not define an extended graph data structure, but uses a generic interface class to access graphical attributes in existing data structures. This makes it possible to use any data structure representing a layout by implementing a layout interface for it.

In a drawing of a graph, vertices are represented by graphical objects such as rectangles or circles usually containing a text label, and edges are represented by curves (polylines or splines). A layout algorithm does not need to know the specific details of the graphical representation of a vertex, it is sufficient to know its size and shape. The algorithm computes a position for each vertex, and a list of bends for each edge. Some algorithms, e.g., visibility representation, in addition have to change the size or shape of a vertex and to specify where exactly the edges are connected to the corresponding source and target vertices. These points of connection are often called *edge anchors.*

For the representation of edge anchors we adopted the method used in the *Graphlet* system [9]. Each anchor p is defined by a point with two coordinates describing the relative position of p with respect to the position of the corresponding vertex v. The point $(0, 0)$ corresponds to the center of v. The coordinates in the range of $[-1 \dots 1] \times [-1 \dots 1]$ are linearly mapped to the bounding box of v which is specified by the width and height of v.

The class `LayoutInterface` defines a generic interface to read and manipulate these basic attributes containing the following virtual methods:

double	A.get_width(*node v*)
void	A.set_width(*node v*, *double new_w*)
double	A.get_height(*node v*)
void	A.set_height(*node v*, *double new_h*)
DPoint	A.get_position(*node v*)
void	A.set_position(*node v*, *DPoint pos*)
DPoint	A.get_source_anchor(*edge e*)
void	A.set_source_anchor(*edge e*, *DPoint pos*)
DPoint	A.get_target_anchor(*edge e*)
void	A.set_target_anchor(*edge e*, *DPoint pos*)

The class `LayoutInterface` does not store these attributes, but defines how they are accessed. An implementation of a layout interface is a class derived from `LayoutInterface` that implements the virtual methods for a specific data structure representing a layout.

Furthermore, the class `LayoutInterface` contains three notification methods. The method `init` is called at the start of the computation of a layout, `cleanup` is called, when the layout has been completely computed, and `update` is called, when intermediate changes in the layout shall be made visible (e.g., in animations).

void	A.init(*graph G*)
void	A.cleanup(*graph G*)

void   *A*.update(*graph G*)

The methods `init` and `cleanup` are useful for implementations that buffer some data until the complete layout is computed. The buffer can be initialized in the `init` method, and allocated resources can be freed in the `cleanup` method.

There are already two predefined specializations of `LayoutInterface` available in the AGD–Library: `GraphWinInterface` for LEDA's graph visualization component *GraphWin* [17] and `GraphletInterface` for the graph editor *Graphlet* [9]. For an example of how to use a layout interface with a graph drawing algorithm see Section 5.

4.2. Handling Preconditions and Postconditions of Algorithms

There is no generic algorithm for drawing all types of graphs, but a considerable number of algorithms for different restricted classes of input graphs (e.g., the input graph has to be planar, simple and biconnected). Thus, there is a natural need for handling the precondition on the input graph of a particular algorithm. A simple approach to this task is to describe the precondition in the written documentation of the algorithm, and to simply assume that the algorithm gets only valid input graphs. However, this is quite unsatisfactory.

A related problem lies in coping with the different models of drawing graphs, e.g., edges have bends or not, vertex coordinates are integers or not. Again, a simple solution is to describe the properties of the drawing in the documentation. If in a particular application the user wants to know, which of the available algorithms satisfy a certain property, then there is no elegant way for the application to answer the question.

Our solution to the problems described above is to provide a mechanism for specifying, retrieving and checking the precondition of an algorithm, and a mechanism for specifying and retrieving properties of its output, which we call the postcondition of the algorithm. These mechanisms are implemented in the base classes `AGD` and `AGDModule` by means of hash tables and they are easy to (re–)use.

Let us see an example. A precondition is represented by a set of properties of the input graph, and a postcondition by a set of properties of the layout. They are usually specified in the constructor of a particular algorithm class. The following is the constructor of the class `FPPLayout`, which implements the layout algorithm by de Fraysseix, Pach, and Pollack [4].

```
FPPLayout::FPPLayout () : GridLayoutModule(true) {
    add_precondition (key::planar);
    add_precondition (key::simple);
    add_precondition (key::no_self_loops);

    add_post_rule (key::straight_line);
    add_post_rule (key::no_crossings);
}
```

In this case all input graphs have to be planar and simple (no multi-edges are allowed) and they may not contain selfloops. The drawing that the algorithm produces for a valid input graph is always planar (indicated by `no_crossings`) and the edges are drawn as straight lines. Therefore, the precondition is {planar, simple, no self loops}, and the postcondition is {straight-line, no crossings}.

The constructor of `FPPLayout` passes `true` to the parameter `use_copy` of the constructor of `GridLayoutModule` specifying that the algorithm is working on a copy of the graph rather than the original graph itself. The reason is that the copy can be changed by adding dummy edges temporarily which are ignored in the drawing in order to be able to use a certain algorithm.

The base class `LayoutModule` defines a method `check` which gets a potential input graph and checks whether it satisfies the specified precondition. The default implementation of `check` is able to check all common properties of graphs, like, e.g., planarity, simplicity, biconnectivity. However, there is the possibility to add new properties for particular algorithms, and to extend the `check` method accordingly. The `check` method is not called by default, if an algorithm is executed, because there are possible situations where it does not make sense to do so, e.g., an application can guarantee the validity of the input because there is some preprocessing step which only produces inputs satisfying the precondition.

A different approach to the problem of handling preconditions and postconditions used by the graph drawing library *GDToolkit* [2] works as follows. The input I of an algorithm A belongs to a certain class C_I within the inheritance hierarchy of *GDToolkit*. C_I is the class of all graphs satisfying the precondition of A. I is transformed by A to an output O belonging to a class C_O, which represents the set of all possible outcomes of A and thus the postcondition of A. This way the type system of C++ is used for handling preconditions and postconditions of algorithms.

Advantages of this approach are the lack of code overhead for supporting pre- and postconditions of algorithms and an appealing model of the problem domain in the form of the inheritance hierarchy. The main disadvantage is that the number of classes which is needed to express all possible sets of properties for a graph is exponential in the number of supported independent properties. There has to be a class for every possible tuple in the property space. This leads, e.g., to restricted extensibility. Adding one new property means in the worst case creating a whole bunch of new classes and redesigning the interfaces of many algorithms for which the input or output class was split into two or more classes. Another problem arises for algorithms which do not care about the distinction related to some specific property. They may be able to take input from two or more classes, thus, it is difficult to integrate them into the library without sacrificing the conceptual clarity.

If only a small set of algorithms is used and hence only a small number of properties is needed, the *GDToolkit* approach seems to be suitable, but we think that our approach is better suited for supporting a broad range of different algorithms and easy extensibility.

5. Using the AGD–Library

In the following, we present a complete example of extending the library by a new algorithm (see Figure 4). Moreover, the `main` routine is an easy example of an application. The algorithm added to the library simply assigns random coordinates to the vertices for brevity.

A new class `RandomLayout` is created representing the new algorithm. It is derived from the base class `LayoutModule`. Apart from the constructor, all methods in the class `RandomLayout` are refinements of virtual methods provided in `LayoutModule`.

The constructor declares the precondition (no restriction in this case) and postcondition of the algorithm. The refined public methods provide information about the particular algorithm and implementation. The protected method `call_layout` implements the drawing procedure. It is called by the inherited public method `call` (see `main`). `get_layout_size` tells the application the dimensions of the current drawing. In this simple case, we always use the same dimensions, but usually they depend on a previously calculated drawing.

In `main` we create an instance `GW` of class `GraphWin` as visualization component in line 1. Line 2 displays `GW` on the screen and lets the user edit a graph. We get the graph which has been created in line 3. We create an interface for the generic graph drawing algorithms in the library to `GW` in line 4. We create an instance of the `RandomLayout` algorithm in line 5 and use it on `G` displaying the result again in `GW` in line 6. Finally, we let the user edit the resulting graph in line 7. The graph together with the computed coordinates can be saved to disk, for example.

The `call` method additionally cares about glueing together the drawing algorithm and a specific visualization. For example, `call` initiates the initialization and clean up of the particular instance of class `LayoutInterface` by calling its above described `init` and `cleanup` methods.

Before `call`ing an algorithm it is possible to check whether a certain input graph satisfies the precondition of the algorithm that should be applied using the `check` method. In our example this

```
#include<AGD/LayoutModule.h>
#include<AGD/GraphWinInterface.h>

class RandomLayout : public LayoutModule
{
  public:
    RandomLayout();
    string name() const { return string("Random"); }
    string long_name() const { return string("Random Layout"); }
    string author() const { return string("NA"); }
    string impl_author() const { return string("David Alberts"); }
    string impl_date() const { return string("May 1997"); }
    AGDModule *clone() const { return new RandomLayout; }
  protected:
    bool call_layout(const graph& G, LayoutInterface& a);
    void get_layout_size(DRect& bbox)
      { bbox = DRect(0.0,100.0,0.0,100.0); }
  private:
    random_source rs;
};

RandomLayout::RandomLayout() : LayoutModule()
{
  add_post_rule(key::straight_line);
}

bool RandomLayout::call_layout(const graph& G, LayoutInterface& a)
{
  node v;
  forall_nodes(v,G)
  {
    double x,y;
    rs >> x >> y;   // assigns pseudo-random numbers in the range [0..1]
    x *= 100.0;     // scale to [0..100]
    y *= 100.0;
    a.set_position(v,DPoint(x,y));
  }
  return true;
}

int main()
{
  GraphWin GW;                // create a GraphWin                          (1)
  GW.open();                  // open it, and let user edit a graph         (2)
  graph& G = GW.get_graph();  // get the graph                              (3)
  GraphWinInterface GWI(GW);  // create an interface to GW for AGD alg's    (4)
  RandomLayout RL;            // create an instance RL of RandomLayout      (5)
  RL.call(G,GWI);             // apply RL to G displaying it in GW          (6)
  GW.edit();                  // let the user edit the result               (7)
}
```

Figure 4: A Complete Example

is not necessary, since there are no restrictions on the input.

6. Future Plans

6.1. Modularizing Algorithms

Besides the graph drawing algorithms provided by the AGD–Library themselves, many building blocks of these algorithms can be reused, too. Some examples of such blocks are the following.

- Planarization: Transform a graph G into a graph G' by replacing edge crossings with dummy vertices until G' is planar.
- Augmentation: Transform a graph G into a graph G' by adding edges until G' satisfies a certain condition such as biconnectivity, or triconnectivity.
- Canonical ordering: Compute an ordered partitioning of the nodes of a given graph G that satisfies particular conditions.

Instead of offering an unstructured collection of several classes or functions, we plan to provide enhanced support for reusing those building blocks in the form of independent modules with well-defined interfaces. For example, we can implement a drawing algorithm `PlanarizationLayout`, which is able to draw a general graph by transforming it into a planar graph using a planarization module, and drawing this planar graph using a layout algorithm for planar graphs. Moreover, it should be easy to exchange each module even interactively at run-time.

We can characterize the type of a module by defining the type of its input, the type of its output, and the associated functionality, e.g., augmentation or layout algorithm. A module implementation declares, which precondition the input must satisfy, and which postcondition holds for the output, e.g., a module for planar biconnected augmentation would declare that the input graph must be planar, and that the output graph is planar and biconnected.

If an algorithm allows to exchange a module at run time, it must declare the module type T, the precondition PRE that always holds for the input of the module, and the postcondition $POST$ that the output of the module must satisfy. Then, the module can be exchanged by an implementation M, if M is of type T, $PRE \Rightarrow \mathit{precondition}(M)$, and $\mathit{postcondition}(M) \Rightarrow POST$.

The implementation of the module concept is done in the following way. We express the different types of modules by abstract base classes defining the interface for calling the module. These base classes are derived from `AGDModule`, which provides the management of pre- and postconditions. We also want to provide support for querying if a condition P implies a condition Q by defining dependencies between properties, e.g., "biconnected implies connected", or "tree is equivalent to connected forest".

6.2. Decoupling Algorithms from the Graph Representation

We are planning to insert a further generic interface `GraphInterface` between the layout algorithms and the topology of the graph. Currently, the layout algorithms use the LEDA data type `graph` directly. Such an interface defines a standard set of operations that each graph data structure has to provide. There are specializations by derived classes for specific graph data structures. An important special case is still the LEDA `graph` data type of course, but moreover it becomes possible to have additional problem specific graph data structures, e.g., a data structure which maintains a graph implicitly. Many complex data structures can be viewed as graphs, and their visualization is a useful tool for understanding or debugging them [25, 14]. For all these new graph data structures, one would instantly be able to use all drawing algorithms in the library, once the corresponding interface is written.

References

[1] *The AGD User Manual (Version 0.7)*, 1997. Max-Planck-Institut für Informatik, Im Stadwald, D-66123 Saarbrücken, Germany. See also http://www.mpi-sb.mpg.de/~mutzel/dfgdraw/agdlib.html.

[2] G. Di Battista, W. Didimo, and A. Leonforte. GDToolkit. Work Package 1.2 of the ALCOM-IT ESPRIT LTR Project 20244; contributing sites: Cologne, Paris, Rome and Saarbrücken, http://www.inf.uniroma3.it/people/gdb/wp12, 1997.

[3] M. Chrobak and G. Kant. Convex grid drawings of 3-connected planar graphs. Technical Report RUU-93-45, Dept. of Computer Sci., Utrecht Univ., 1993.

[4] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[5] P. Eades. A heuristic for graph drawing. *Congr. Numer.*, 42:149–160, 1984.

[6] I. Fáry. On straight line representation of planar graphs. *Acta Sci. Math. Szeged*, 11:229–233, 1948.

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

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

[9] M. Himsolt. The Graphlet system. *Proc. Graph Drawing '96, LNCS*, 1190:233–240, 1997.

[10] M. Jünger and S. Thienel. The design of the branch and cut system ABACUS. Technical Report No. 97.260, Institut für Informatik, Universität zu Köln, 1997.

[11] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica, Special Issue on Graph Drawing*, 16(1):4–32, 1996.

[12] G. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Extended Abstract, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1997.

[13] S. Leipert. The tree interface - version 1.0 user manual. Technical Report No. 96.242, Institut für Informatik, Universität zu Köln, 1996.

[14] D. Lütkehaus and A. Zeller. DDD: The data display debugger. Technische Universität Braunschweig, Germany, http://www.cs.tu-bs.de/softech/ddd/, 1997.

[15] B. Madden, P. Madden, S. Powers, and M. Himsolt. Portable graph layout and editing. *Proc. Graph Drawing '95, LNCS*, 1027:385–395, 1996.

[16] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Comm. Assoc. Comput. Mach.*, 38:96–102, 1995.

[17] K. Mehlhorn, S. Näher, and Ch. Uhrig. The LEDA User Manual (Version R 3.5). Max-Planck-Institut für Informatik, Saarbrücken, Germany. See also http://www.mpi-sb.mpg.de/LEDA/leda.html, 1997.

[18] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Trans. Softw. Eng.*, SE-7(2):223–228, 1981.

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

[20] W. Schnyder. Embedding planar graphs on the grid. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 138–148, 1990.

[21] 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.

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

[23] W. T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 13(3):743–768, 1963.

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

[25] A. Zeller and D. Lütkehaus. DDD – a free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1), 1996.