The AGD–Library
User Manual

Version 0.8.3 beta

contact address:

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

participating groups:

Prof. Dr. Stefan Näher
Martin-Luther Universität Halle-Wittenberg
Fachbereich Mathematik und Informatik
Kurt-Mothes-Str. 1
D-06099 Halle (Saale), Germany

Prof. Dr. Michael Jünger
Universität zu Köln
Institut für Informatik
Pohligstraße 1
D-50969 Köln, Germany

Dr. Petra Mutzel
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken, Germany

This work was partially supported by DFG-Grants Ju204/7-3, Mu1129/3-1, Na303/1-3, Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen"

Contents

1 Introduction **1**

1.1 Overview . . . 2

1.1.1 The Structure of the AGD–Library . . . 2

1.1.2 Layout Algorithms in the AGD–Library . . . 3

1.1.3 Standard Interfaces . . . 3

1.2 Implementation of the AGD–Library . . . 5

1.2.1 The Generic Graphics Interface . . . 5

1.2.2 Handling Preconditions and Postconditions of Modules . . . 5

1.3 Using the AGD–Library . . . 6

1.4 Contributing People . . . 7

2 Technical Information **11**

2.1 System Requirements . . . 11

2.2 Installation . . . 11

2.3 Compiling Programs . . . 13

2.4 Documentation (Unix platforms only) . . . 14

3 Base Classes **15**

3.1 AGD Base (AGD) . . . 15

3.2 AGD Modules (AGDModule) . . . 19

3.3 Module Options (AGDModuleOption) . . . 24

3.4 Grid Coordinate Mapper (GridCoordinateMapper) . . . 26

3.5 Layout Interfaces (LayoutInterface) . . . 29

4 Module Types **33**

4.1 Layout Algorithms (LayoutModule) . . . 33

4.2 Grid Layout Algorithms (GridLayoutModule) . . . 37

4.3 Augmentation Modules (AugmentModule) . . . 41

4.4 Planarization Modules (PlanarizerModule) . . . 43

4.5 Subgraph Modules (SubgraphModule) . . . 45

4.6 Edge Insert Modules (EdgeInsertModule) . . . 47

4.7 RankAssignment Modules (RankAssignment) . . . 49

4.8 Two Layer Crossing Minimization (TwoLayerCrossMin) . . . 50

4.9 Hierarchy Layout Modules (HierarchyLayoutModule) . . . 52

5 Layout Algorithms 53

5.1 The de Fraysseix, Pach, Pollack Algorithm (FPPLayout) . . . 53
5.2 The Schnyder Algorithm (SchnyderLayout) . . . 55
5.3 The Convex Algorithm (ConvexLayout) . . . 57
5.4 The ConvexDraw Algorithm (ConvexDrawLayout) . . . 59
5.5 The PlanarStraight Algorithm (PlanarStraightLayout) . . . 61
5.6 The PlanarDraw Algorithm (PlanarDrawLayout) . . . 63
5.7 Visibility Representation (VisibilityRepresentation) . . . 66
5.8 The Mixed-Model Algorithm (MixedModelLayout) . . . 68
5.9 The Orthogonal Drawing Algorithm (Ortho) . . . 71
5.10 The Quasi Orthogonal Drawing Algorithm (Quod) . . . 74
5.11 Walker's Algorithm (TreeLayout) . . . 77
5.12 Spring Embedder (SpringLayout) . . . 79
5.13 Tutte's Algorithm (TutteLayout) . . . 81

6 Generic Layout Algorithms 83

6.1 The Planarization Method (PlanarizationLayout) . . . 83
6.2 Planarization on the Grid (PlanarizationGridLayout) . . . 85
6.3 Sugiyama's Algorithm (SugiyamaLayout) . . . 87

7 Augmentation Modules 91

7.1 Planar Biconnected (PlanAug) . . . 91
7.2 Optimal Planar Biconnected (PlanAugOpt) . . . 93
7.3 LEDA's Make Biconnected (LEDAMakeBiconnected) . . . 95

8 Modules for Planarization 97

8.1 Planarization with Planar Subgraph (SubgraphPlanarizer) . . . 97
8.2 Planar Subgraphs (PlanarSubgraph) . . . 99
8.3 Maximum Planar Subgraphs (MaximumPlanarSubgraph) . . . 101
8.4 Shortest Path Edge Inserter (ShortestPathInserter) . . . 103

9 Two Layer Crossing Minimization 105

9.1 Split Heuristic (SplitHeuristic) . . . 105
9.2 Barycenter Heuristic (BarycenterHeuristic) . . . 106
9.3 Median Heuristic (MedianHeuristic) . . . 107
9.4 Greedy Insert (GreedyInsertHeuristic) . . . 108
9.5 Greedy Switch (GreedySwitchHeuristic) . . . 109

10 Other Modules 111

10.1 Hierarchical Ranking (HierarchicalRanking) . . . 111
10.2 Ranking by Topological Sorting (TopSortRanking) . . . 113
10.3 Ranking by Depth First Search (DfsRanking) . . . 114
10.4 Fast Layout of Hierarchies (FastHierarchyLayout) . . . 115
10.5 Topology Modules (TopologyModule) . . . 117

11 Graph Generators 119

12 Grid Coordinate Mapper 121
12.1 The Grid Coordinate Mapper MaxAll (MaxAllMapper) 121
12.2 The Grid Coordinate Mapper Visibility (VisibilityMapper) 123

13 Layout Interfaces 127
13.1 Interface to GraphWin (GraphWinInterface) 127
13.2 Interface to PostScript (PSInterface) 129
13.3 Layout Buffers (LayoutBuffer) 131
13.4 Layout Interface for Graph Copies (GraphCopyLI) 133
13.5 Layouts (AGDLayout) 134

14 Canonical Orderings 135
14.1 Sets of a Canonical Ordering (OrderSet) 135
14.2 The Canonical Ordering of a Graph (CanonicalOrder) 136
14.3 The Leftmost Canonical Order of a Graph (LmcOrder) 138

15 Handling Pre- and Postconditions 141
15.1 Key Structs (AGD_Key) 141
15.2 Key Mapper (Keymapper) 142
15.3 Key Sets (KeySet) 143
15.4 Tagged Rule Literals (tr_literal) 146
15.5 Tagged Rules (TaggedRule) 147

16 Hierarchies 149
16.1 Layers of Hierarchies (Layer) 149
16.2 Node Hierarchies (Hierarchy) 151

17 Graphs 153
17.1 Copy of a Graph (GraphCopy) 153
17.2 Plane Graph Copies (PlaneGraphCopy) 155
17.3 Leveled Graphs (LeveledGraph) 157

18 Auxiliary Classes 159
18.1 Grid Layouts (GridLayout) 159
18.2 Integer Points (IPoint) 161
18.3 Integer Line Segments (ISegment) 162
18.4 Integer Polylines (IPolyline) 163
18.5 Integer Rectangles (IRect) 164
18.6 Double Points (DPoint) 165
18.7 Double Line Segments (DSegment) 167
18.8 Double Polylines (DPolyline) 168
18.9 Double Rectangles (DRect) 169

A Class Overview **171**

Bibliography **173**

Chapter 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, 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. 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).

The AGD–Library, a library of **A**lgorithms for **G**raph **D**rawing, offers a broad range of existing algorithms for two-dimensional graph drawing and tools for implementing new algorithms. It 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".

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. Moreover, some modules used by algorithms are exchangeable allowing to replace particular phases of an algorithm by new implementations.

1.1 Overview

1.1.1 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.

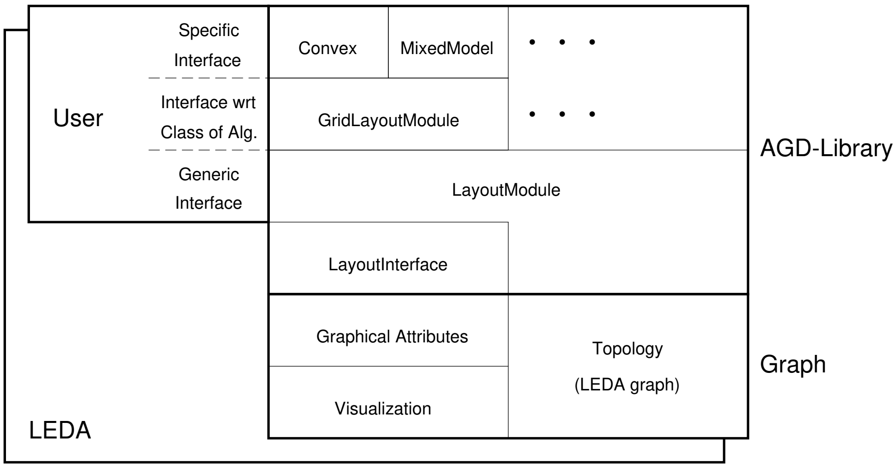


Figure 1.1: A Sketch of the User's View of the AGD–Library

Figure 1.1 shows an overview of the AGD–Library from the user's perspective. The class `GridLayoutModule` is a base class of the specific layout algorithms `Convex` or `MixedModel`. `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 [7] are already available) or by deriving her or his own class from `LayoutInterface`.

1.1.2 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 [2]), `SchnyderLayout` (Schnyder [18]), `Convex` (Kant [11]), `ConvexDraw` (Chrobak and Kant [1]), `PlanarStraight` (Gutwenger and Mutzel [6]), `PlanarDraw` (Gutwenger and Mutzel [6]), `VisibilityRepresentation` (Rosenstiehl and Tarjan [17])

- drawing planar graphs on the grid with edge bends

  `MixedModelLayout` (extension of Kant's algorithm [11, 6]), `Ortho` (Tamassia [20]), `Quod` (extension of Tamassia's algorithm [20, 12])

- drawing special classes of planar graphs

  `TreeLayout` (Reingold and Tilford, Walker [16, 22, 13])

- drawing general graphs

  `PlanarizationLayout`, `PlanarizationGridLayout` (allow to use all planar graph drawing algorithms for drawing general graphs), `SugiyamaLayout` (Sugiyama, Tagawa, and Toda [19]), `SpringLayout` (Fruchterman, Reingold [5]), `TutteLayout` (Tutte [21])

1.1.3 Standard Interfaces

Figure 1.2 gives an overview of the standard interface provided by layout algorithm modules. Note, that methods defined in base classes are inherited by derived classes, e.g., the complete standard interface of `AGDModule` for retrieving common informations about a module is inherited by `LayoutModule`.

The base classes `AGD` and `AGDModule` implement the handling of preconditions and postconditions. Furthermore, `AGDModule` defines the standard interface to retrieve common informations, e.g, the name or the author of the algorithm.

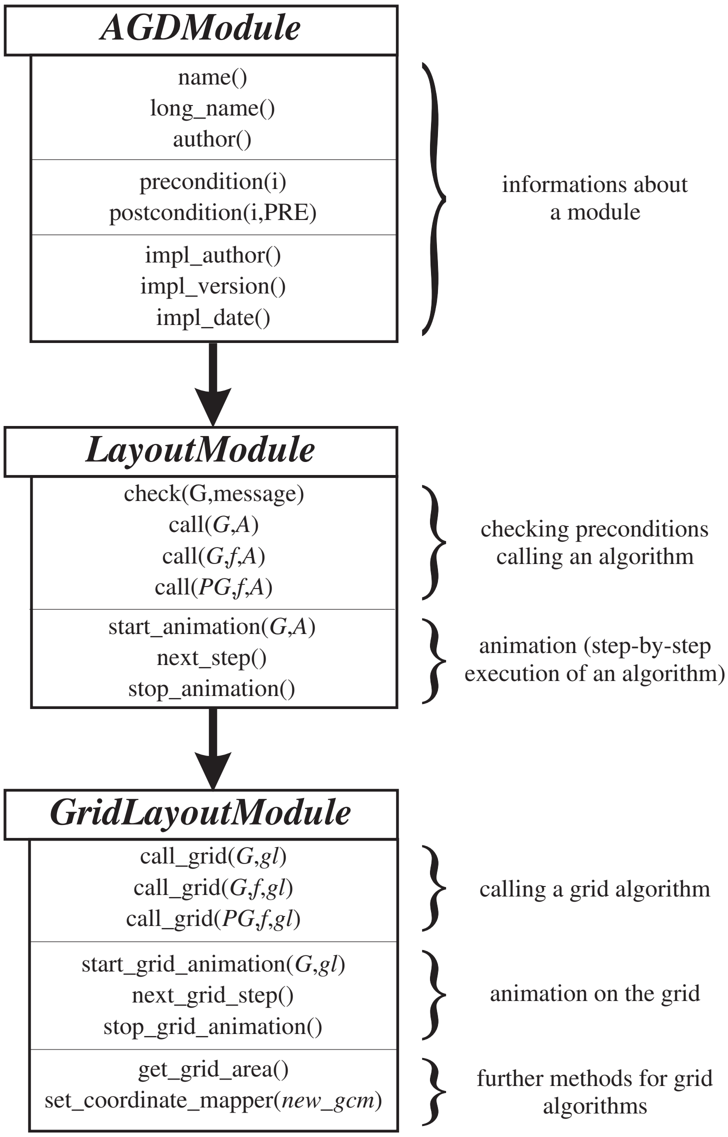


Figure 1.2: Standard Interfaces

`LayoutModule` is the base class for layout algorithms and defines the interface to check the precondition of a layout algorithm given a specific graph, to call the layout algorithm, and to perform an animation sequence (if supported), that is, a step-by-step execution of the algorithm. There is also a special interface for planar layout algorithms, allowing to pass a plane graph (represented by a planar map). Each call function does not change the input graph, i.e., the input is a `const` parameter.

`GridLayoutModule` provides further support for layout algorithms that place the vertices and edge bends on an integer grid, i.e, the method `call_grid` computes the grid layout. There is also an extended interface for planar layout algorithms.

In section 1.3, we give an example of the implementation of a specific layout algorithm and show, how to use it.

1.2 Implementation of the AGD–Library

1.2.1 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 [7]. 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 $[-1\dots1]\times[-1\dots1]$ 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. It does not store them, but defines how these attributes 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 two notification methods. The method `update` is called, when intermediate changes in the layout shall be made visible, and the method `refresh` is called in order to indicate that the topology of the associated graph has changed.

There are already two predefined specializations of `LayoutInterface` available in the AGD–Library: `GraphWinInterface` for LEDA's graph visualization component GRAPHWIN [14] and `GraphletInterface` for the graph editor GRAPHLET [7]. The interface to GRAPHLET is described in detail in the manual "The AGD–Library Graphlet Interface". For an example of how to use a layout interface with a graph drawing algorithm see Section 1.3.

1.2.2 Handling Preconditions and Postconditions of Modules

There is no generic algorithm for drawing all types of graphs, but a wealth 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 and that is it. 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, querying and checking the precondition of an algorithm, and a mechanism for specifying and querying 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 [2].

```
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 parallel edges are allowed) and they may not contain self loops. 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, e.g., by adding dummy edges temporarily which are ignored in the drawing.

The base class `LayoutModule` defines a method `check` which gets a potential input graph and checks whether it satisfies the specified precondition. `check` simply uses the method `check_graph` of its base class `LayoutModule`, which is able to check all common graph properties, like, e.g., planarity, simplicity, biconnectivity. Moreover, there is the possibility to add new properties for particular algorithms, and to extend the `check_graph` 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 all preconditions.

1.3 Using the AGD–Library

In the following, we present a complete example of extending the library by a new algorithm (see Figure 1.4). 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 (none in this case) and postcondition of the algorithm. The refined public methods provide information about the particular algorithm and implementation. The protected method `_call` implements the drawing procedure. It is called by the inherited public method `call` (see `main`).

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 a reference to 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.

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

1.4 Contributing People

This section lists all people that contributed to the AGD–Library.

Universität Halle (Prof. Dr. Näher)

David Alberts	Adaption of Schnyder's layout algorithm. Implementation of the visibility representation (Rosenstiehl, Tarjan), the spring embedder layout algorithm (Fruchtermann, Rheingold), Tutte's layout algorithm and the GraphWin and PostScript^{TM} interface.
Dirk Ambras	Implementation of Schnyder's layout algorithm.
Gunter Koch	Implementation of the AGD-Server.
Stefan Näher	Implementation of the spring embedder layout algorithm (Fruchtermann, Rheingold).

Universität Köln (Prof. Dr. Jünger)

Christoph Buchheim	Implementation of the *fastHierarchyLayout* module (assignment of coordinates in the Sugiyama algorithm).
Sebastian Leipert	Implementation of Walker's layout algorithm, and the computation of a planar subgraph using pq-trees (*PlanarSubgraph* module).

Max-Planck-Institut für Informatik, Saarbrücken (Dr. Mutzel)

Sergey Fialko	Implementation of the planar biconnected augmentation algorithms (modules *PlanAug* and *PlanAugOpt*), and the maximum planar subgraph (*PlanarSubgraphOpt*).
Carsten Gutwenger	Design of the library (base classes, pre- and postcondition handling, auxiliary classes) and design of generic layout algorithms (planarization method and Sugiyama's algorithm). Implementation of the layout algorithms by de Fraysseix, Pach, Pollack, the Convex, ConvexDraw, PlanarStraight, PlanarDraw, and Mixed-Model algorithms, the grid coordinate mapper, the computation of the canonical ordering, and pre- and postprocessing for the *PlanarSubgraph* module. Documentation. Integration of the AGD–Library into GRAPHLET.
Gunnar Klau	Implementation of the orthogonal layout algorithms *Ortho* and *Quod*. Implementation of the *TopologyModule*.
Michael Krüger	Implementation of the heuristics for two layer crossing minimization, the *TopSortRanking* module, and the Sugiyama algorithm (*SugiyamaLayout*).
Thomas Ziegler	Implementation of the edge insert module *ShortestPathInserter*, pre- and postprocessing for the *PlanarSubgraph* module, and heuristics for two layer crossing minimization.

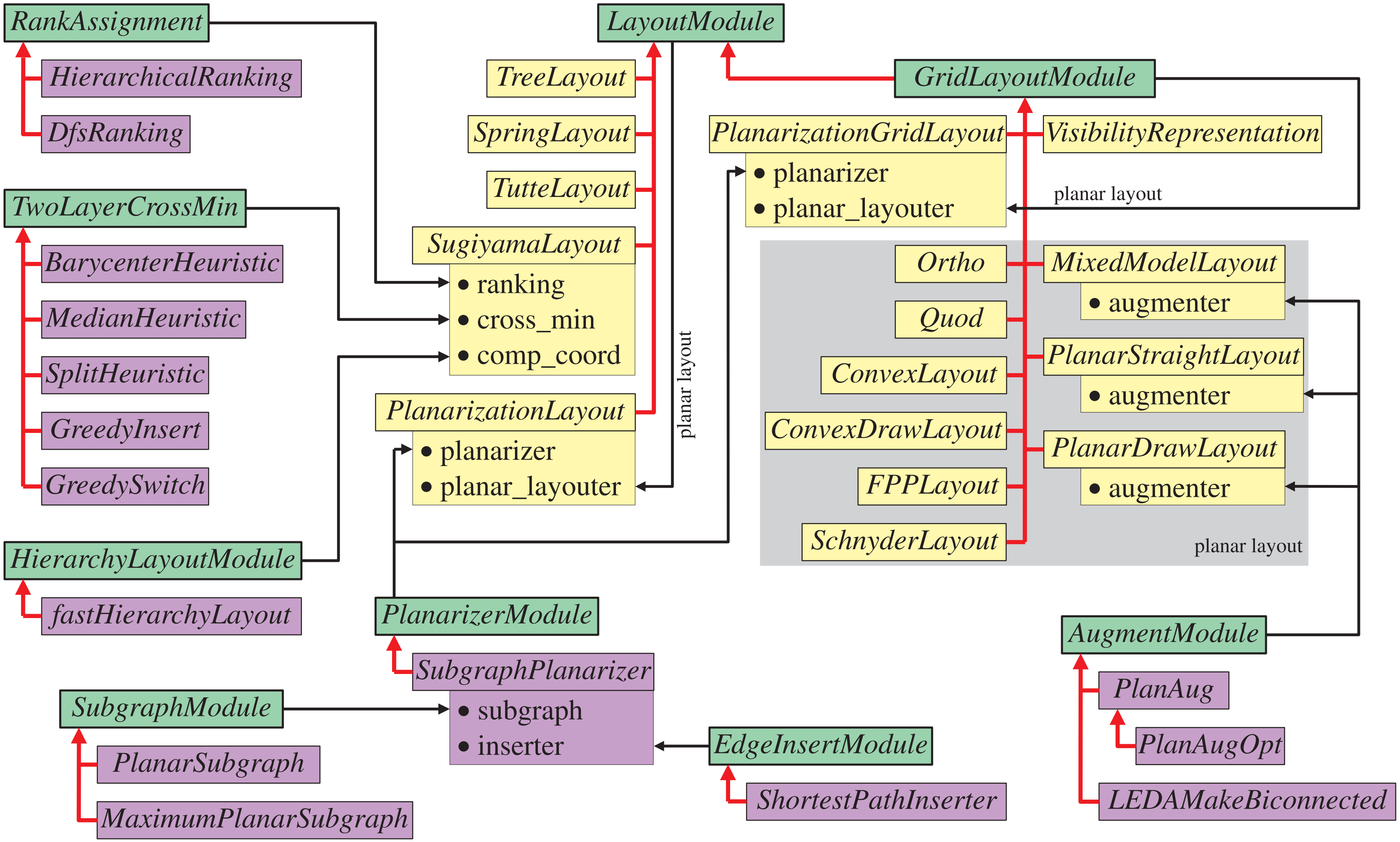


Figure 1.3: Module Overview

```
#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:
    void _call(const graph& G, LayoutInterface& A);
  private:
    random_source rs;
};

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

void RandomLayout::_call(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));
  }
}

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 1.4: A Complete Example

Chapter 2

Technical Information

2.1 System Requirements

The AGD–Library is based on LEDA version 3.7 or higher, so you must have installed LEDA on your system, before you can install the AGD–Library. More information about LEDA are available at the LEDA homepage

http://www.mpi-sb.mpg.de/LEDA/leda.html

The AGD–Library also provides some optimal branch-and-cut algorithms in a separate library `libAGDopt`. Creating and using this library requires that ABACUS [10] (version 2.1) is installed on your system.

Table 2.1 shows, for which platforms the AGD–Library is available.

machine	**operating system**	**compiler**
SUN Sparc	SunOS	SunPRO CC, g++
SUN Sparc	Solaris	SunPRO CC, g++
Intel 80x86	Linux	g++
Intel 80x86	Windows 95, Windows NT	Borland 5.x, Microsoft Visual C++ 4.2, 5.0

Table 2.1: Available platforms

2.2 Installation

The AGD–Library is available at the AGD homepage

http://www.mpi-sb.mpg.de/~mutzel/dfgdraw/agdlib.html

It comes as a source code package in a file `AGD-0.8.3.tar.gz`. Place this file into a directory for which you have write permission and type

```
gunzip AGD-0.8.3.tar.gz
```

and

```
tar xf AGD-0.8.3.tar
```

This will create a directory `AGD-0.8.1`. In the sequel, we will refer with *AGDROOT* to the path name of this directory.

- **Installation on a UNIX platform**

  1. Type: `cd` *AGDROOT*
  2. Type: `lconfig` <*cc*>

     where <*cc*> is the name (or command) of your C++compiler.
     Examples: `lconfig CC`, `lconfig g++`

     `lconfig` without arguments prints a list of known compilers.
  3. If the LEDA header files and libraries are not in the standard path of your compiler (e.g., the include, or lib directory of the compiler), you must tell lconfig the corresponding path names. Therefore, lconfig will ask you at the end of the configuration, if you want to enter the path names of the include and library directory of your LEDA installation. Type `y` and follow the instructions, if these directories are not in the standard path of your compiler, otherwise type `n`.

     If you also want to create the library `libAGDopt`, which contains some optimal branch-and-cut algorithms, you must tell `lconfig` the path name of your ABACUS and CPLEX installations. Then, type `y` and follow the instructions, otherwise type `n`.
  4. Now just type: `make`

     to create the library `libAGD.a` in the directory *AGDROOT* and the demo program `agd_demo` in the directory *AGDROOT*/demo.
     Type: `make opt`

     to create the library `libAGDopt.a` in the directory *AGDROOT* and the demo program `agd_opt_demo` in the directory *AGDROOT*/demo. This requires an ABACUS and CPLEX installation on your system.
  5. Now, you can copy the AGD subdirectory of *AGDROOT*/incl to the standard include directory of your compiler. The library libAGD.a should be copied (or moved) to the standard library directory of your compiler (e.g., usr/lib or usr/local/lib).

  Remarks:

  – You can also use the AGD–Library without copying the include and library files to the default directories by using the `-I` and `-L` option of your compiler. This is described in the following section.

  – The makefile in *AGDROOT* provides the following targets:

target	*description*
`all`	creates `libAGD.a`, `agd_demo` and the test program
`all_g`	as above, but with debugging informations
`opt`	creates `libAGDopt.a` and `agd_opt_demo`
`libAGD`	creates `libAGD.a`
`libAGD_g`	creates `libAGD.a` with debugging informations
`agd_demo`	creates the demo program `agd_demo`
`agd_demo_g`	creates the demo program with debugging informations
`agd_opt_demo`	creates the demo program `agd_opt_demo`
`agd_test`	creates the test program
`agd_test_g`	creates the test program with debugging informations

- **Installation on Windows95 or Windows NT**
  1. Go to *AGDROOT*
  2. Type: `lconfig` <*cc*>

     where <*cc*> is the name (or command) of your C++compiler.

     Examples: `lconfig bcc`, `lconfig msc`

     `lconfig` without arguments prints a list of known compilers.
  3. Now just type:

     ```
     make_lib
     make_demo
     ```

     to make the library `libAGD.lib` in the directory *AGDROOT* and the demo program `agd_demo.exe` in the directory *AGDROOT*/demo.
  4. You should copy the AGD subdirectory of *AGDROOT*/incl to the standard include directory, and the library libAGD.lib to the standard library directory of your compiler.

2.3 Compiling Programs

In the following description, we assume that the library `libAGD.a` and the include directory `AGD` are in the default directories of the compiler.

AGD programs must be linked with the LEDA libraries `libL.a`, `libG.a`, `libP.a` and `libW.a` (if LEDA windows are used).

If the include files and libraries of LEDA are also contained in the default directories of the compiler, you just have to type

```
CC prog.c -lAGD -lG -lL -lm
```

or

```
CC prog.c -lAGD -lW -lP -lG -lL -lX11 -lm
```

if LEDA windows are used.

Note that the libraries must be given in the above order.

Remarks:

- LEDA and the AGD–Library provide a mechanism to check many (not all!) preconditions of operations. This mechanism can be turned off by defining `LEDA_CHECKING_OFF`, e.g.,

  ```
  CC -DLEDA_CHECKING_OFF ...
  ```

  It is a good idea to turn checking off only in the final release version of a program, because the error messages that occur if a check fails are a good hint for errors in the program.

- If using LEDA windows on solaris systems you might have to link with the socket and thread libraries as well:

  `CC` *prog.c* `-lAGD -lW -lP -lG -lL -lX11 -lsocket -lthread -lm`

- If the libraries or include files are not in the default directories of the compiler, you have to use the `-I` and `-L` compiler flags:

  `CC -I`*LEDAROOT*`/incl -I`*AGDROOT*`/incl` *prog.c* `-L`*AGDROOT* `-lAGD ...`

2.4 Documentation (Unix platforms only)

We use slightly modified versions of the LEDA tools for manual production to extract the AGD manual from the AGD header files (see also [14]). The commands for the manual production are contained in the directory *AGDROOT*/Manual/cmd. It is necessary to set the environment variable `AGDROOT` to the directory, where the AGD–Library is installed (e.g., `setenv AGDROOT` *AGDROOT*).

The command `Aman` can be used similar to LEDA's `Lman` to display the manual contained in a specific header file. You should add the directory *AGDROOT*/Manual/cmd to your `PATH` variable. Then, you can type

```
Aman LayoutModule
```

to display the documentation of the class `LayoutModule`.

Chapter 3

Base Classes

3.1 AGD Base (AGD)

1. Definition

The class *AGD* provides a global key mapper, that maps strings to key items of type *agd_key*, and a number of predefined key constants. Furthermore, it defines static methods for testing properties of graphs, and for augmentation (partially wrappers for LEDA functions).

We use key items to represent properties of instances of a particular type, for example a specific instance G of type *graph* could have the property *planar* meaning that G is a planar graph. We represent such properties by arbitrary strings, which are stored in a global hash table by the default keymapper of *AGD*. Strings in this hash table are accessed via key items of type *agd_key*. We use <*s*> to denote a key item for string *s*.

The subclass *key* of *AGD* defines a number of key constants (e.g., *AGD*::*key*::*directed*):

General properties		
animation	"animation"	A layout algorithm supports step by step animation.
grid_animation	"grid animation"	A grid layout algorithm supports step by step animation on the grid.
planar_layout	"planar layout"	A layout algorithm works on a plane graph.

Properties of layouts		
constant	string value	meaning
straight_line	"straight-line"	All edge polylines have no bend points.
orthogonal	"orthogonal"	All edge segments are either horizontal or vertical.
no_crossings	"no crossings"	The edge polylines do not intersect.
upward_planar	"upward planar"	All edges are directed upwards (downwards).
convex	"convex"	All faces are convex.

Properties of graphs		
constant	string value	meaning
directed	"directed"	G is directed.
undirected	"undirected"	G is undirected.
planar	"planar"	G is planar.
simple	"simple"	G has no parallel edges.
no_self_loops	"no self loops"	G has no self loops.
no_reversals	"no reversals"	$(v, w) \in E \Rightarrow (w, v) \notin E$ for all $v, w \in V$
connected	"connected"	G is connected.
biconnected	"biconnected"	G is biconnected.
triconnected	"triconnected"	G is triconnected.
bipartite	"bipartite"	G is bipartite.
four_graph	"four-graph"	$degree(v) \leq 4$ for all $v \in V$
acyclic	"acyclic"	$v_1, \ldots, v_k \in V, k \geq 3, (v_i, v_{i+1}) \in E \Rightarrow v_1 \neq v_k$
forest	"forest"	G represents a forest.
tree	"tree"	G represents a tree.
single_source	"single source"	$\exists! \, v \in G : indeg(v) = 0$
single_sink	"single sink"	$\exists! \, v \in G : outdeg(v) = 0$

where $G = (V, E)$.

2. Operations

a) Testing Properties of Graphs

bool AGD::is_simple(*leda_graph* G)

returns *true* if G is simple, e.g. has no parallel edges, *false* otherwise.

bool AGD::is_connected(*leda_graph* G)

returns *true* if G is connected, *false* otherwise.

bool AGD::is_biconnected(*leda_graph* G)

returns *true* if G is biconnected, *false* otherwise.

bool AGD::is_triconnected(*leda_graph* G)

returns *true* if G is triconnected, *false* otherwise.

bool AGD::is_planar(*leda_graph* G)

returns *true* if G is planar, *false* otherwise.

bool AGD::is_bipartite(*leda_graph* G)

returns *true* if G is bipartite, *false* otherwise.

bool AGD::is_four_graph(*leda_graph* G)

returns *true* if G is a four-graph, e.g. $degree(v) \leq 4$ holds for every node v of G, *false* otherwise.

bool AGD::has_no_self_loops(*leda_graph* G)

returns *true* if G contains no self loops, *false* otherwise.

bool *AGD* :: has_no_reversals(*leda_graph* *G*)

returns *true* if *G* is undirected, or if *G* contains no reversal edges, *false* otherwise.

bool *AGD* :: is_acyclic(*leda_graph* *G*)

returns *true* if *G* is acyclic, *false* otherwise.

bool *AGD* :: is_forest(*leda_graph* *G*)

returns *true* if *G* represents a forest, *false* otherwise.

bool *AGD* :: is_tree(*leda_graph* *G*)

returns *true* if *G* represents a tree, *false* otherwise.

bool *AGD* :: has_single_source(*leda_graph* *G*)

returns *true* if *G* is directed and contains exactly one node with no incoming edges, *false* otherwise.

bool *AGD* :: has_single_sink(*leda_graph* *G*)

returns *true* if *G* is directed and contains exactly one node with no outgoing edges, *false* otherwise.

b) Establishing Properties of Graphs

leda_list<*leda_edge*> *AGD* :: triangulate(*leda_graph*& *G*)

triangulates a directed graph *G* representing a planar map. Returns the list of inserted edges.

leda_list<*leda_edge*> *AGD* :: make_bidirected(*leda_graph*& *G*)

makes *G* bidirected by inserting missing reversal edges. Returns the list of inserted edges.

leda_list<*leda_edge*> *AGD* :: make_biconnected(*leda_graph*& *G*)

makes *G* biconnected. Returns the list of inserted edges.

leda_list<*leda_edge*> *AGD* :: augment_planar_biconnected(*leda_graph*& *G*)

augments a planar graph *G* to a planar and biconnected graph. Returns the list of inserted edges.

leda_list<*leda_edge*> *AGD* :: make_map(*leda_graph*& *G*)

makes *G* bidirected and turns it into a map. Returns the list of inserted edges.

bool *AGD* :: embed_planar_graph(*leda_graph*& *G*)

takes as input a bidirected graph *G*. Returns *true* and turns *G* into a planar map if *G* is planar, *false* otherwise.

c) Access to the Default Keymapper

int AGD :: number_of_keys( )

returns the number of defined keys.

agd_key AGD :: ask_key(*leda_string* s)

returns the key <s> if <s> is already defined, *nil* otherwise.

agd_key AGD :: get_key(*leda_string* s)

returns the key <s>.

KeySet AGD :: get_properties(*leda_graph* G)

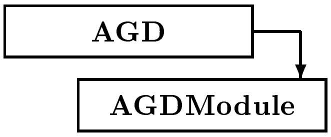
returns the set of all properties that G satisfies.

Iteration

forall_AGD_keys(k)
{ "the keys of AGD's default keymapper are successively assigned to k" }

3.2 AGD Modules (AGDModule)

1. Base Classes



2. Definition

The class *AGDModule* is the base class for modules in the AGD–Library. It provides the handling of pre- and postconditions, and defines a uniform access to common informations about a module.

The Type of a Module

We can characterize the type of a module by defining the type of its input $I_1, \ldots, I_n$, the type of its output $O_1, \ldots, O_m$, and the associated functionality, e.g., augmentation or layout algorithm. We express the different types of modules by abstract base classes derived from *AGDModule*, which define the input and output types, and the functionality.

The input consists of $n \geq 0$ and the output of $m \geq 0$ parameters of an arbitrary abstract (not C++) type. The call method of a particular module uses appropriate classes to represent these types. It is not necessary to represent each abstract type by exactly one class, for example

- the type "grid layout" can be represented by "*node_map*<*int*>& *x*, *node_map*<*int*>& *y*, *edge_map*<*Ipolyline*>& *bends*",
- an input and output parameter "graph" can be represented by a single *graph* reference.

Precondition and Postcondition

AGDModule maintains a set $pre(i)$ of properties for each input parameter I_i, which must be satisfied by I_i if the module is called. Properties are expressed by key items (cf. class 3.1), and sets of properties by the data type *KeySet*. We call $pre(i)$ the precondition of I_i, and $pre(1), \ldots, pre(n)$ the precondition of the module.

The postcondition of the output parameters are sets of properties that the output will always satisfy. We allow that they depend on the properties $P_1, \ldots, P_n$ that the input of the module satisfies. Clearly, these properties must be more general than the required precondition, i.e., $P_i \supseteq pre(i)$ must hold for all $1 \leq i \leq n$. Therefore, we associate a function $post(i) : \mathcal{P}(KEYS)^n \to \mathcal{P}(KEYS)$ with each output parameter O_i, where $KEYS$ is the set of all properties. We also reserve a special postcondition with index 0 for general properties of the module, e.g., we declare that a layout algorithm supports step-by-step animation by adding the key *animation* to this general postcondition.

The function $post(i)$ is defined by a set of rules $post\_rules(i)$, where each rule is of the form

$$(p_1, i_1) \wedge \ldots \wedge (p_k, i_k) \to p \ , \ k \geq 0.$$

AGDModule provides protected methods to insert rules into *post_rules*(i), and to remove rules from *post_rules*(i). Rules in *post_rules*(i) can be accessed via the item type *pr_item*, e.g., to remove a rule. The function defined by *post_rules*(i) is given by

$$\begin{aligned} &post(i)(Q_1, \ldots, Q_n) = \\ &\qquad \{p \mid \exists (p_1, i_1) \wedge \ldots \wedge (p_k, i_k) \rightarrow p \in post\_rules(i) \text{ with } p_j \in Q_{i_j} \forall 1 \leq j \leq k\} \end{aligned}$$

Note, that the so defined functions are monotonous in the following way. If $P_j \subseteq Q_j$ holds for all $1 \leq j \leq n$, then $post(i)(P_1, \ldots, P_n) \subseteq post(i)(Q_1, \ldots, Q_n)$ holds for each $1 \leq i \leq m$.

Common Informations

Every AGD module contains

- a name in a long (*long_name*) and a short (*name*) version.
- an author (*author*).
- the name of the author (*impl_author*), the version (*impl_version*), and the date (*impl_date*) of the implementation.

The default implementations for *name*( ), *author*( ), *impl_author*( ), *impl_version*( ) and *impl_date*( ) return an empty string "" indicating that this information is unknown. *long_name*( ) returns *name*( ) by default.

Exchangeable Modules and Optional Parameters

We recommend to provide a standard interface for exchangeable modules and optional parameters. These recommendation is obeyed by the module implementations in the library.

Exchangeable modules are implemented as module options (see 3.3, page 24) with an additional method set_*name*(*type M*), which sets the module option *name* to *M* if possible.

An optional parameter *name* of type *type* provides two methods

void *name*(*type new_setting*)
type *name*()

for setting and reading the current value of the option. The documentation describes optional parameters in the definition section of its class in the following way

- *type* **name** = *initial_value*
  description

where *initial_value* is the default setting of the option.

3. Initialization

AGDModule $M(int\ n,\ int\ m)$;

initializes a module with n input and m output parameters. Initializes $pre(i)$ and $post(j)$ with the empty set for all $1 \le i \le n$, and $0 \le j \le m$.

4. Operations

a) Queries on Preconditions and Postconditions

int M.num_in_param( )

returns the number of input parameters n.

int M.num_out_param( )

returns the number of output parameters m.

KeySet M.precondition(*int* i)

returns $pre(i)$. *Precondition*: $1 \le i \le n$.

bool M.need_property(*int* i, *leda_string* p)

returns true iff $p \in pre(i)$. *Precondition*: $1 \le i \le n$.

KeySet M.postcondition(*int* i)

returns $post(i)(pre(1), \ldots, pre(n))$. *Precondition*: $0 \le i \le m$.

KeySet M.postcondition(*int* i, *leda_array*<*KeySet*> *PRE*)

returns $post(i)(PRE[1], \ldots, PRE[n])$. *Precondition*: $0 \le i \le m$, PRE.low() = 1 and PRE.high() = n.

bool M.usable(*leda_array*<*KeySet*> *PRE*, *leda_array*<*KeySet*> *POST*)

returns *true* iff M can be used as module for guaranteed preconditions *PRE* and required postconditions *POST*, i.e., $PRE[i] \supseteq pre(i)\ \forall 1 \le i \le n$ and $post(j)(PRE) \subseteq POST[j]\ \forall 0 \le j \le m$.

AGDModule ∗ M.clone( ) returns a pointer to a new instance with the same option settings as M. Must be overridden by module implementations. PURE VIRTUAL

b) Common Informations

leda_string M.name( ) returns the name of the algorithm.

leda_string M.long_name( ) returns the name in an extended version.

leda_string M.author( ) returns the author of the algorithm.

leda_string M.impl_author( ) returns the author of the implementation.

leda_string M.impl_version( ) returns the version of the implementation.

leda_string M.impl_date() returns the date of the last change of the implementation.

c) Protected Methods for Maintaining Preconditions

agd_key M.add_precondition(*leda_string p*, *int i* = 1)

adds the key <p> to $pre(i)$ and returns it. *Precondition*: $1 \leq i \leq n$.

void M.add_precondition(*agd_key k*, *int i* = 1)

adds the key k to $pre(i)$. *Precondition*: $1 \leq i \leq n$.

void M.add_precondition(*agd_key k*, *bool status*, *int i* = 1)

adds the key k to $pre(i)$ if $status = true$, removes it otherwise. *Precondition*: $1 \leq i \leq n$.

void M.add_preconditions(*leda_string ps*, *char delim* = ',', *int i* = 1)

interprets ps as a list of strings seperated by *delim*, and adds the key for each string to $pre(i)$. *Precondition*: $1 \leq i \leq n$.

void M.rem_precondition(*agd_key k*, *int i* = 1)

removes the key k from $pre(i)$. *Precondition*: $1 \leq i \leq n$.

d) Protected Methods for Maintaining Postconditions

pr_item M.add_post_rule(*agd_key k*, *int i* = 1)

adds a new rule item <$\rightarrow k$> to $post\_rules(i)$ and returns it. *Precondition*: $0 \leq i \leq m$.

pr_item M.add_post_rule(*agd_key p1*, *int i1*, *agd_key p*, *int i* = 1)

adds a new rule item <$(p_1, i_1) \rightarrow p$> to $post\_rules(i)$ and returns it. *Precondition*: $0 \leq i \leq m$.

pr_item M.add_post_rule(*TaggedRule r*, *int i* = 1)

adds a new rule item <r> to $post\_rules(i)$ and returns it. *Precondition*: $0 \leq i \leq m$.

void M.rem_post_rule(*pr_item it*, *int i* = 1)

removes the rule item *it* from $post\_rules(i)$. *Precondition*: $0 \leq i \leq m$ and *it* is a rule item in $post\_rules(i)$.

Checking the Precondition of a Module

Particular module types, for example *LayoutModule*, provide a method check that checks if a given input satisfies the precondition of the module. *AGDModule* implements extendable default implementations to check a precondition for specific types, for example graphs. These default implementations can be used by the base class of a particular module type (e.g., *LayoutModule*) to implement its check method.

The method check_graph can test if a given input graph satisfies the following keys: *directed*, *undirected*, *planar*, *simple*, *no_self_loops*, *no_reversals*, *connected*, *biconnected*, *triconnected*, *bipartite*, *four_graph*, *acyclic*, *forest*, *tree*, *single_source*, *single_sink*. If check_graph finds another key it calls check_graph_key.

bool M.check_graph(*int* i, *leda_graph* G, *leda_string*& *message*)

returns *true* if G satisfies the preconditions $pre(i)$. Otherwise, *false* is returned, and *message* contains the error message.

bool M.check_graph_key(*agd_key* p, *leda_graph* G, *leda_string*& *message*)

overridable to extend check_graph. Must return *true* if G satisfies p, otherwise *false* must be returned and *message* must contain the error message.

3.3 Module Options (AGDModuleOption)

1. Definition

The class *AGDModuleOption*<*M*> is the base class for module options ...

2. Creation

AGDModuleOption<*M*> *opt*;

creates an instance *opt* of type *AGDModuleOption*<*M*> ...

3. Operations

void	*opt*.init(*leda_array*<*KeySet*> *PRE*, *leda_array*<*KeySet*> *POST*, *M module*)	initializes *opt* with guaranteed preconditions *PRE*, required postconditions *POST*, and initial module *module*.
bool	*opt*.set(*M module*)	sets the module option to *module* and returns *true* if *module* is usable, returns *false* otherwise.
bool	*opt*.check(*M module*)	returns *true* if *module* is usable, *false* otherwise.
M&	*opt*.get( )	returns a reference to the module contained in *opt*,

Declaring Module Options

The macro **AGD_DECLARE_MODULE**(*type*,*name*) should be used to declare a module option of type *type* with name *name* in a class declaration. This macro expands to

```
protected:
     AGDModuleOption<type> name;
public:
     void set_name (const type &M) {
          name.set(M);
     }
```

4. Example

We create a sample layout algorithm with an optional augmentation module. The augmentation module gets a simple, planar graph and must augment it to a simple, planar, biconnected graph. The call_copy_layout method shows how to use the module, and the main function shows, how to set the module option to another augmenter.

```
class SampleLayout : public LayoutModule
{
  AGD_DECLARE_MODULE(AugmentModule,augmenter);
    ...
```

```
public:
  SampleLayout();
  bool _call(GraphCopy& G, LayoutInterface& A);
};

SampleLayout::SampleLayout() : LayoutModule(true)
{
    ...
  array<KeySet> PRE(1,1), POST(0,1);
  PRE[1]  << key::planar << key::simple;
  POST[1] << key::planar << key::biconnected << key::simple;
  augmenter.init (PRE,POST,PlanAug());
    ...
}

SampleLayout::_call(GraphCopy& G, LayoutInterface& A)
{
    ...
  augmenter.get().call(G,L);  // make G biconnected
    ...
}

int main()
{
  GraphWin GW;
  GW.open();

  GraphWinInterface GWI(GW.get_graph());
  SampleLayout L;

  L.set_augmenter (PlanAugOpt());  // use an optimal augmenter
  L.call(GW,GWI);

    ...
  return 0;
}
```

3.4 Grid Coordinate Mapper (GridCoordinateMapper)

1. Definition

GridCoordinateMapper is the base class for all grid coordinate mappers. Grid coordinate mappers are used to transform a grid layout of a graph into a real layout.

We denote with G_A the graph for which the grid coordinate mapper computes the real layout, and with G_B the graph for which the grid layout is computed. G_B may be equal to G_A, or a copy (*GraphCopy* or *PlanarizedGraph*) of G_A.

Implementation of Grid Coordinate Mappers

We refer to the input grid layout with $column(v)$, $row(v)$ for the position of node v, $ip^e_1, \ldots, ip^e_{k(e)}$ for the polyline of edge e, where $ip^e_i = (col^e_i, row^e_i)$, with $width(v)$, $height(v)$ for the size of the graphics of node v, and with $xmin$, $xmax$, $ymin$, $ymax$ for the bounding box of the grid layout.

The output is given by $x(v)$, $y(v)$ for the position of node v, $line(e)$ for the polyline of edge e, and $width'(v)$, $height'(v)$ for the new width and height of node v.

Basically, a grid coordinate mapper computes a mapping $\varphi : \mathbb{N}^2 \to \mathbb{R}^2$ from the coordinates of a grid coordinate system into the coordinates of the standard coordinate system (see 3.5, The coordinate system):

$$\varphi(column, row) = (\varphi_x(column), \varphi_y(row))$$

That means, the mapping of the x-coordinate is independent of the y-coordinate and vice versa.

Usually, it is sufficient to define φ_x and φ_y by implementing map_x and map_y. The default implementations for the output will then return

$$\begin{aligned} x(v) &= \varphi_x\left(column(v)\right) \\ y(v) &= \varphi_y\left(row(v)\right) \\ line(e) &= \varphi\left(ip^e_1\right), \ldots, \varphi\left(ip^e_{k(e)}\right) \end{aligned} \qquad \begin{aligned} width'(v) &= width(v) \\ height'(v) &= height(v) \end{aligned}$$

The mapping of the coordinates works as shown below. You must override the methods that have no default implementation, and you can change the default implementations if necessary, e.g., if you want to change the size of nodes or to assign special edge anchors.

1. **Initialization**

   The *init* method is called. The default implementation does nothing.

2. **Mapping of node coordinates**

   The *map_nodes* method is called. The default implementation computes the position of the nodes by calling *map_node* for each node in G_A.

   The default implementation of *map_node* calls *map_x* and *map_y* to transform grid coordinates in real coordinates.

3. **Mapping of edge coordinates**

   The *map_edges* method is called. The default implementation computes the position of the edges by calling *map_edge* for each edge in G_A.

   The default implementation of *map_edge* maps each bend point by calling *map_x* and *map_y*, and sets the source and target anchors to $(0, 0)$.

4. **Computing the bounding box of the layout**

   The method *get_layout_size* is called.

5. **Cleaning up**

   Finally, the *cleanup* method is called. The default implementation does nothing.

2. Initialization

GridCoordinateMapper M;

initializes a grid coordinate mapper.

3. Operations

a) Standard Interface

DRect M.call(*leda_graph* G, *GridLayout* *gl*, *IRect* *gbbox*, *LayoutInterface*& A)

calls M for graph G and grid layout *gl*. Assigns the transformed layout to A and returns its bounding box. *Precondition*: *gl* is a grid layout for G, *gbbox* is the bounding box of *gl*, and A is a layout interface for G.

DRect M.call(*GraphCopy* GC, *GridLayout* *gl*, *IRect* *gbbox*, *LayoutInterface*& A)

calls M for graph copy GC and grid layout *gl*. Assigns the transformed layout to A and returns its bounding box. *Precondition*: *gl* is a grid layout for GC, *gbbox* is the bounding box of *gl*, and A is a layout interface for *original*(GC).

b) Access Methods

leda_graph	M.graph_A( )	returns a reference to G_A.
leda_graph	M.graph_B( )	returns a reference to G_B.
leda_node	M.in_B(*leda_node* v)	returns the node in G_B that corresponds to V. *Precondition*: v is a node in G_A.
int	M.x(*leda_node* v)	returns the x-coordinate of v in the grid layout. *Precondition*: $v \in G_A$.
int	M.y(*leda_node* v)	returns the y-coordinate of v in the grid layout. *Precondition*: $v \in G_A$.

IPolyline	M.bends(*leda_edge* e)	returns the list of bend points of e in the grid layout. *Precondition*: e is an edge in G_A.
IRect	M.gbbox( )	returns the bounding box of the grid layout.
double	M.width(*leda_node* v)	returns the width of v. *Precondition*: v is a node in G_A.
void	M.width(*leda_node* v, *double* w)	sets the width of v to w. *Precondition*: v is a node in G_A.
double	M.height(*leda_node* v)	returns the height of v. *Precondition*: v is a node in G_A.
void	M.height(*leda_node* v, *double* h)	sets the height of v to h. *Precondition*: v is a node in G_A.

c) Overridables

	void	M.map_nodes( )	called for assigning node coordinates.
	void	M.map_edges( )	called for assigning edge polylines.
PURE VIRTUAL	*void*	M.get_layout_size(*DRect*& *bbox*)	called to get the bounding box of the layout.
	void	M.cleanup( )	called for cleaning up.
	DPoint	M.map_node(*leda_node* v)	returns the position of v. *Precondition*: v is a node in G_A.
	void	M.map_edge(*leda_edge* e, *DPolyline*& *dpl*, *DPoint*& *s_a*, *DPoint*& *t_a*)	returns the polyline for edge e in *dpl* (bend points), *s_a* (source anchor point) and *t_a* (target anchor point). *Precondition*: e is an edge in G_A.
PURE VIRTUAL	*double*	M.map_x(*int column*)	returns $\varphi_x(column)$.
PURE VIRTUAL	*double*	M.map_y(*int row*)	returns $\varphi_y(row)$.
PURE VIRTUAL	*GridCoordinateMapper* $*$	M.clone( )	returns a pointer to a new *GridCoordinateMapper* instance with the same option settings as M.

3.5 Layout Interfaces (LayoutInterface)

1. Definition

The abstract base class *LayoutInterface* defines the generic interface to access the graphical attributes of a graph. The LEDA *graph* class contains only the topology of a graph, e.g., the nodes and edges. If you want to produce a graphical representation of a graph, you need additional attributes such as the graphics and the position of each node, the line represantation of each edge, and some more.

Our graphical model for graphs contains the following attributes:

- a rectangular bounding box for the graphics of each node.
- a position of type *DPoint* for each node v which is the center of the graphics of v.
- a source and a target anchor point of type *DPoint* for each edge e with coordinates in the range of $[-1 \ldots 1] \times [-1 \ldots 1]$. The anchor points define the start and endpoint of the line representation of e.
- a list of bend points of type *DPolyline* for each edge.

The Coordinate System

We assume a rectilinear coordinate system, where positive x is to the right, positive y is down, and the origin is in the upper left corner. One unit in x- and y-direction should be 1/72 inch. We call this coordinate system the *standard coordinate system*.

2. Initialization

LayoutInterface is an abstract base class, so you cannot create instances of it.

LayoutInterface A(*leda_graph* G);

initializes a layout interface for graph G.

3. Operations

a) Access to Graphical Attributes

The following methods define the generic interface to the graphical attributes of a graph. All of them are pure virtual functions and must be overridden by a specif implementation of *LayoutInterface*, except for *get_position* and *set_position*, which have a default implementation.

double	A.get_x(*leda_node* v)	returns the x-coordinate of the position of node v.	PURE VIRTUAL
void	A.set_x(*leda_node* v, *double* *new_x*)	sets the x-coordinate of the position of node v to *new_x*.	PURE VIRTUAL
double	A.get_y(*leda_node* v)	returns the y-coordinate of the position of node v.	PURE VIRTUAL

PURE VIRTUAL	*void*	A.set_y(*leda_node* v, *double new_y*)	sets the y-coordinate of the position of node v to *new_y*.
	DPoint	A.get_position(*leda_node* v)	returns the position of node v.
	void	A.set_position(*leda_node* v, *DPoint pos*)	sets the position of node v to *pos*.
PURE VIRTUAL	*DPolyline*	A.get_bend_points(*leda_edge* e)	returns the bend points of edge e.
PURE VIRTUAL	*void*	A.set_bend_points(*leda_edge* e, *DPolyline* l)	sets the bend points of edge e to l.
PURE VIRTUAL	*double*	A.get_width(*leda_node* v)	returns the width of the graphics of node v.
PURE VIRTUAL	*void*	A.set_width(*leda_node* v, *double new_w*)	sets the width of the graphics of node v to *new_w*.
PURE VIRTUAL	*double*	A.get_height(*leda_node* v)	returns the height of the graphics of node v.
PURE VIRTUAL	*void*	A.set_height(*leda_node* v, *double new_h*)	sets the height of the graphics of node v to *new_h*.
PURE VIRTUAL	*DPoint*	A.get_source_anchor(*leda_edge* e)	returns the source anchor position of edge e.
PURE VIRTUAL	*void*	A.set_source_anchor(*leda_edge* e, *DPoint pos*)	sets the source anchor position of edge e to *pos*.
PURE VIRTUAL	*DPoint*	A.get_target_anchor(*leda_edge* e)	returns the target anchor position of edge e.
PURE VIRTUAL	*void*	A.set_target_anchor(*leda_edge* e, *DPoint pos*)	sets the target anchor position of edge e to *pos*.

b) Notification Methods

The following methods can be overridden by derived classes if necessary. The default implementations do nothing.

void	A.update()	called to update the layout, e.g., update the window displaying the graph, or set temporarily buffered informations.
void	A.refresh()	called to indicate that the associated graph has been changed, e.g., by adding or deleting nodes or edges.

c) Further Operations

leda_graph	*A*.get_graph( )	returns the associated graph.
DRect	*A*.area( )	computes a tight bounding box.
void	*A*.make_straight_line( )	deletes all edge bends and sets all edge anchors to $(0, 0)$.
void	*A*.set_all_anchors($DPoint\ p\ =\ DPoint(0,\ 0)$)	sets all source and target anchors to p.
void	*A*.set_graph(*leda_graph* G)	protected method for setting the associated graph.

4. Example

The following function translates all nodes displayed in a GraphWin *gw* by a vector p, removes all bends points, and returns the bounding box of the layout.

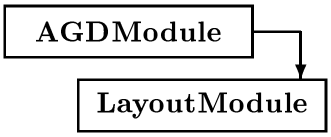
```
DRect translate(GraphWin &gw, DPoint p)
{
  GraphWinInterface gwi(gw);        // create layout interface
  const graph &g = gwi.get_graph();

  gwi.make_straight_line();  // remove alle bends

  // shift all nodes by p
  node v;
  forall_nodes(v,g)
    gwi.set_position(v, p + gwi.get_position());

  gwi.update();  // update the display

  return gwi.area();  // compute bounding box
}
```

Chapter 4

Module Types

4.1 Layout Algorithms (LayoutModule)

1. Base Classes



2. Definition

The class *LayoutModule* is the base class for layout algorithms and defines a generic interface for these algorithms. We define a layout algorithm L as follows:

A layout algorithm L gets as input a graph G and for each node v of G the size of the rectangle surrounding the graphics of v. L computes for each node v a point, which will be the center of the graphics of v, and for each edge a list of bend points with a source and target anchor point. Layout algorithms are also allowed to change the size of the graphics of nodes (this feature is only used by few algorithms).

The graphical attributes of a graph are accessed via a layout interface (see section 3.5).

Input and Output Parameters

input parameter:	graph	$in\_graph = 1$
output parameter:	layout	$out\_layout = 1$

The integer constants *in_graph* and *out_layout* can be used to denote the positions of the input and output parameters.

Implementation of Layout Algorithms

Layout algorithms are not allowed to change the input graph, i.e., the graph is passed as a const reference to the *call* function. Therefore, the AGD provides mechanisms to work on a copy of the input graph. The classes *GraphCopy* and *PlaneGraphCopy* maintain copies of graphs. Implementation of layout algorithms using this mechanism can manipulate the graph copy in certain ways, e.g., inserting dummy edges or splitting edges. Afterwards, the layout of the graph copy is converted automatically in a respective layout for the input graph

The parameter *use_copy* of the constructor of *LayoutModule* decides if such a mechanism is used. Layout algorithms operating on plane graphs have an extended interface in order to pass a plane graph directly. Such an algorithm must set the general postcondition *planar_layout*. An implementation must override the respective _call method depending on the graph type used. This depends on the parameter *use_copy* and the general postcondition *planar_layout*. The following table shows which *_call* method has to be overridden:

use_copy	*planar_layout*	overridable
false	no	*_call(const graph &G, LayoutInterface &A)*
true	no	*_call(GraphCopy &GC, LayoutInterface &A)*
false	yes	*_call(const graph &G, face f, LayoutInterface &A)*
true	yes	*_call(PlaneGraphCopy &GC, face f, LayoutInterface &A)*

3. Initialization

LayoutModule L(*bool use_copy* = *false*);

initializes a layout algorithm. Graph copies are used iff *use_copy* is true.

4. Operations

a) Standard Interface

bool L.check(*leda_graph* G, *leda_string*& *message*)

returns *true* if G satisfies the preconditions of L. Otherwise, *false* is returned, and *message* contains the error message.

void L(*leda_graph* G, *LayoutInterface*& A)

calls the layout algorithm for graph G with layout interface A.

The following methods are only supported by a planar layout module, that is a layout module with general postcondition *planar_layout*, i.e., $planar\_layout \in post(0)(\emptyset)$.

void L.call(*leda_graph* G, *leda_face* f, *LayoutInterface*& A)

calls L for graph G with layout interface A. *Precondition*: G represents a planar map, and f is *nil* or a face in G.

void L.call(*PlaneGraphCopy*& PG, *leda_face* f, *LayoutInterface*& A)

calls L for planarized graph PG with layout buffer A. *Precondition*: f is *nil* or a face in PG.

b) Step by Step Animation

The following methods are only supported by a layout module with general postcondition *animation*, i.e., $animation \in post(0)(\emptyset)$.

bool *L*.start_animation(*leda_graph* G, *LayoutInterface*& A)
initializes step by step animation for graph G with layout interface A. Must be called before $next_step$ is used. Returns *false*, if animation is not possible.

bool *L*.next_step() executes the next step of the animation. Returns *false*, if no more steps are possible, i.e., animation is finished.

void *L*.stop_animation() finish explicitly animation.

c) Implementation of the Module Call

void *L*._call(*leda_graph* G, *LayoutInterface*& A)
implements the computation of the layout for graph G and layout interface A.

void *L*._call(*GraphCopy*& GC, *LayoutInterface*& A)
implements the computation of the layout for graph copy GC and layout interface A.

void *L*._call(*leda_graph* G, *leda_face* f, *LayoutInterface*& A)
implements the computation of the layout for graph G, external face f and layout interface A. f may be *nil* indicating that the algorithm selects the external face.

void *L*._call(*PlaneGraphCopy*& PG, *leda_face* f, *LayoutInterface*& A)
implements the computation of the layout for planarized graph PG, external face f and layout buffer A. f may be *nil* indicating that the algorithm selects the external face.

d) Implementation of Animation

The following methods are called if const graph references are used (*use_copy* is *false*) and L is not a planar layout module:

bool *L*._start_animation(*leda_graph* G, *LayoutInterface*& A)
called to initialize step by step animation for graph G and layout interface A. Returns *true* if animation is possible, *false* otherwise.

bool *L*._next_step(*leda_graph* G, *LayoutInterface*& A)
called to perform the next step of the animation for graph G and layout interface A. Returns *false* if this was the last step, *true* otherwise.

void *L*._stop_animation(*leda_graph* G, *LayoutInterface*& A)
called after the animation has finished (by request of the client or the algorithm).

The following methods are called if graph copies are used (*use_copy* is *true*).

bool L_start_animation(*GraphCopy*& *GC*, *LayoutInterface*& *A*)

called to initialize step by step animation for graph copy *GC* and layout interface *A*. Returns *true* if animation is possible, *false* otherwise.

bool L_next_step(*GraphCopy*& *GC*, *LayoutInterface*& *A*)

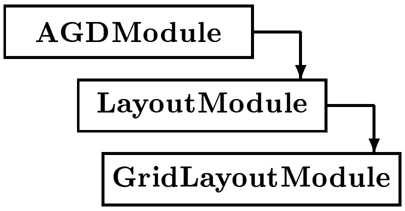
called to perform the next step of the animation for graph copy *GC* and layout interface *A*. Returns *false* if this was the last step, *true* otherwise.

void L_stop_animation(*GraphCopy*& *GC*, *LayoutInterface*& *A*)

called after the animation has finished (by request of the client or the algorithm).

4.2 Grid Layout Algorithms (GridLayoutModule)

1. Base Classes



2. Definition

The class *GridLayoutModule* is the base class for grid layout algorithms. It defines an extended interface for computing a grid layout.

Implementation of Grid Layout Algorithms

The AGD provides mechanisms to work on a copy of the input graph, rather than the input graph itself. An implementation can manipulate the copy in certain ways and compute a grid layout for the resulting graph. Afterwards, this grid layout is transformed automatically into a respective grid layout for the input graph. The type of the graph copy depends on whether the grid layout algorithm is a planar grid layout algorithms (general postcondition *planar_layout* is set, or not. The type is *PlaneGraphCopy* for planar grid layout algorithms, and *GraphCopy* otherwise. The following graph manipulations are possible if *GraphCopy* is used:

- Adding new nodes or new edges. This nodes or edges will not appear in the final grid layout.
- Reversing edges. The respective polylines will be reversed automatically in the final grid layout.
- Splitting edges. The respective polylines will be concatenated in the final grid layout.

If you want to reverse and split an edge, you must first reverse it. Edges resulting from splitting may not be reversed.

An implementation must override the one of the *_call_grid* methods depending on the parameter *use_copy* passed to the constructor of *GridLayoutModule*, and whether the general postcondition *planar_layout* is set or not. The following table shows which *_call_grid* method has to be overridden:

use_copy	*planar_layout*	overridable
false	no	*_call_grid(const graph &G, GridLayout &gl)*
true	no	*_call_grid(GraphCopy &GC, GridLayout &gl)*
false	yes	*_call_grid(const graph &G, face f, GridLayout &gl)*
true	yes	*_call_grid(PlaneGraphCopy &GC, face f, GridLayout &gl)*

Initially, all x- and y-coordinates for nodes are set to 0, and all bend point lists for edges are empty. Finally, all unnecessary bend points are removed automatically. *GridLayoutModule* also maintains the bounding box of the least computed grid layout. If an algorithm computes this bounding box itself, it can set it with *set_grid_bbox*. Otherwise, the bounding box is computed by *GridLayoutModule*.

3. Initialization

GridLayoutModule L(*bool use_copy* = *false*);

initializes a grid layout algorithm.

4. Operations

a) Extended Standard Interface

void *L*.call_grid(*leda_graph* G, *GridLayout*& *gl*)

calls L for graph G. Returns the grid layout in *gl*.

IRect *L*.get_grid_area( ) returns the bounding box of the computed grid layout. *Precondition*: *call_grid* has been called before.

void *L*.set_coordinate_mapper(*GridCoordinateMapper gcm*)

sets the grid coordinate mapper for L to a copy of *gcm*.

GridCoordinateMapper *L*.get_coordinate_mapper( )

returns the currently used grid coordinate mapper.

The following methods are only supported by grid layout modules with general postcondition *planar_layout*, i.e., *planar_layout* $\in$ *post*(0)($\emptyset$).

void *L*.call_grid(*leda_graph* G, *leda_face* f, *GridLayout*& *gl*)

calls L for graph G and external face f. Returns the grid layout in *gl*. *Precondition*: G represents a planar map, and f is *nil* or a face in G.

void *L*.call_grid(*PlaneGraphCopy*& *PG*, *leda_face* f, *GridLayout*& *gl*)

calls L for planarized graph *PG* and external face f. Returns the grid layout *gl*. *Precondition*: f is *nil* or a face in *PG*.

The following methods are only supported by grid layout modules with general postcondition *animation*, i.e., *animation* $\in$ *post*(0)($\emptyset$).

bool *L*.start_grid_animation(*leda_graph* G, *GridLayout*& *gl*)

initializes step by step animation on the grid for graph G; the grid layout is stored in *gl*. Must be called before *next_grid_step* is used. Returns *false*, if animation is not possible.

bool *L*.next_grid_step( ) executes the next step of the grid animation. Returns *false*, if no more steps are possible, i.e., animation is finished.

void *L*.stop_grid_animation( )

finish explicitly grid animation.

c) Implementation of the Module Call

void L_call_grid(*leda_graph G*, *GridLayout& gl*)

implements the computation of the grid layout for graph G. Returns the grid layout in gl.

void L_call_grid(*GraphCopy& GC*, *GridLayout& gl*)

implements the computation of the grid layout for graph copy GC. Returns the grid layout in gl.

void L_call_grid(*leda_graph G*, *leda_face f*, *GridLayout& gl*)

implements the computation of the grid layout for graph G and external face f. G represents a planar map, and f is either a face in G or *nil* indicating that the algorithm selects the external face. Returns the grid layout in gl.

void L_call_grid(*PlaneGraphCopy& PG*, *leda_face f*, *GridLayout& gl*)

implements the computation of the grid layout for planarized graph PG and external face f. f is either a face in PG or *nil* indicating that the algorithm selects the external face. Returns the grid layout in gl.

d) Implementation of Animation

The following methods are called if const graph references are used (*use_copy* is *false*) and L is not a planar layout module:

bool L_start_grid_animation(*leda_graph G*, *GridLayout& gl*)

called to initialize step by step grid animation for graph G and grid layout gl. Returns *true* if animation is possible, *false* otherwise.

bool L_next_grid_step(*leda_graph G*, *GridLayout& gl*)

called to perform the next step of the grid animation for graph G and grid layout gl. Returns *false* if this was the last possible step, *true* otherwise.

void L_stop_grid_animation(*leda_graph G*, *GridLayout& gl*)

called after the grid animation has been finished (by request of the client or the algorithm).

The following methods are called if graph copies are used (*use_copy* is *true*).

bool L_start_grid_animation(*GraphCopy& GC*, *GridLayout& gl*)

called to initialize step by step grid animation for graph copy GC and grid layout gl. Returns *true* if animation is possible, *false* otherwise.

bool L_next_grid_step(*GraphCopy& GC, GridLayout& gl*)

called to perform the next step of the grid animation for graph copy *GC* and grid layout *gl*. Returns *false* if this was the last possible step, *true* otherwise.

void L_stop_grid_animation(*GraphCopy& G, GridLayout& gl*)

called after the grid animation has been finished (by request of the client or the algorithm).

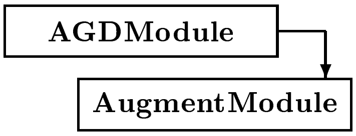
Protected Functions

void L.set_grid_bbox(*IRect bbox*)

sets the computed grid size.

4.3 Augmentation Modules (AugmentModule)

1. Base Classes



2. Definition

The class *AugmentModule* is the base class for augmentation modules. An augmentation module transforms an input graph G into an output graph G' by adding edges, such that G' satisfies a certain condition, e.g., biconnected. This condition is expressed by the postcondition of the module.

Input and Output Parameters

input parameter:	graph	*in_graph* $= 1$
output parameter:	graph	*out_graph* $= 1$

The integer constants *in_graph* and *out_graph* can be used to denote the positions of the input and output parameters.

Implementing Augmentation Algorithms

An implementation of an augmentation module must override the protected method call_aug(G,L), which gets as input a graph reference G. The augmented edges are added to G, and the list of added edges is returned in L.

3. Initialization

AugmentModule A; initializes an augmentation module.

4. Operations

Standard Interface

bool	A.check(*leda_graph* G, *leda_string*& *message*)	returns *true* if G satisfies the precondition of A. Otherwise, *false* is returned, and *message* contains the error message.
void	A.call(*leda_graph*& G)	calls the augmentation module for graph G.
void	A.call(*leda_graph*& G, *leda_list*<*leda_edge*>& L)	calls the augmentation module for graph G. Returns the list of added edges in L.
int	A.num_added_edges( )	returns the number of added edges.

Protected Overridables to Implement Functionality

PURE VIRTUAL

void A_call(*leda_graph*& G, *leda_list*<*leda_edge*>& L)

implements the augmentation algorithm for graph G. Returns the list of added edges in L.

Protected Auxiliary Methods

void *AugmentModule* :: hide_self_loops(*leda_graph*& G, *leda_list*<*leda_edge*>& L)

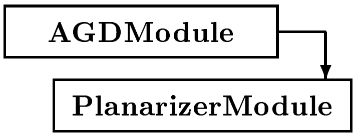
hides all self loops and appends them to L.

void *AugmentModule* :: hide_multiple_edges(*leda_graph*& G, *leda_list*<*leda_edge*>& L)

hides all but one of each set of parallel edges and appends all hided edges to L.

4.4 Planarization Modules (PlanarizerModule)

1. Base Classes



2. Definition

The class *PlanarizerModule* is the base class for planarization modules. A planarization module transforms a graph G into a planar graph PG by replacing edge crossings with dummy nodes, and computes a combinatorial embedding of PG.

Input and Output Parameters

input parameter:	graph	$in\_graph = 1$
output parameter:	graph	$out\_graph = 1$

The integer constants *in_graph* and *out_graph* can be used to denote the positions of the input and output parameters.

Implementing Planarization Algorithms

An implementation of a planarization module must override the protected method call_plan(G,PG), which gets as input a graph $G = (V, E)$ and produces as output a planar graph $PG = (V \cup V_d, E')$. It replaces a set of edges $E_r \subset E$ by chains of edges leading over dummy nodes $\in V_d$. Each dummy node is a node of degree 4, and the output graph PG represents a planar map.

In particular, an edge $e = (v, w) \in E_r$ is replaced with a chain $e_1 = (v_1, v_2), \ldots, e_k = (v_k, v_{k+1})$, where $v_1 = v$, $v_{k+1} = w$, and $v_i \in V_d$ for all $2 \leq i \leq k$. The corresponding chain of edges is stored in PG, i.e., $PG.chain(original(e)) = e_1, \ldots, e_k$. Furthermore, tho following must hold:

- $degree(v) = 4$ for all $v \in V_d$, and $E_r = E \setminus E'$.
- Let $E_d = E' \setminus E$ be the set of inserted new edges.

  Then, there must be a mapping $\varphi : E_d \to E_r$ with $\overset{-1}{\varphi}(e) = chain(e)$ for all $e \in E_r$.

The implementation must also embed the output graph, i.e., the combinatorial embedding must be given by the clockwise ordering of the edges in all adjacency lists.

3. Initialization

PlanarizerModule P; initializes a planarization module.

4. Operations

Standard Interface

bool P.check(*GraphCopy* G, *leda_string*& *message*)

returns *true* if G satisfies the precondition of P. Otherwise, *false* is returned, and *message* contains the error message.

void P.call(*leda_graph* G, *PlaneGraphCopy*& PG)

calls the planarization module for graph PG. Stores the planarization in PG, i.e., for every edge $e \in G$ the ordered list of corresponding edges $chain(e)$ in PG.

Protected Overridables to Implement Functionality

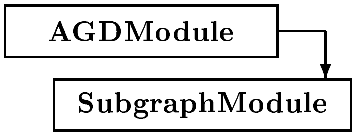
PURE VIRTUAL

void P_call(*leda_graph* G, *PlaneGraphCopy*& PG)

implements the planarization algorithm for graph PG. Stores the planarization in PG, i.e., for every edge $e \in E_r$ the ordered list of corresponding edges $chain(e)$ in PG.

4.5 Subgraph Modules (SubgraphModule)

1. Base Classes



2. Definition

The class *SubgraphModule* is the base class for subgraph modules. A subgraph module computes a subgraph $G' = (V, E')$ of an input graph $G = (V, E)$, where $E' \subseteq E$. Usually, G' satisfies a certain condition, e.g, planar or acyclic. This condition is expressed by the postcondition of the module.

Input and Output Parameters

input parameter:	graph	$in\_graph = 1$
output parameter:	graph	$out\_graph = 1$

The integer constants *in_graph* and *out_graph* can be used to denote the positions of the input and output parameters.

Implementing Subgraph Modules

An implementation of a subgraph module must override the protected method call_subgraph(G,L), which gets as input a reference to a const graph G. It determines the set of edges $\{e_1, \ldots, e_k\} = E \cap E'$, and returns it as a list in L.

3. Initialization

SubgraphModule S; initializes a subgraph module.

4. Operations

Standard Interface

bool S.check(*leda_graph* G, *leda_string*& *message*)

returns *true* if G satisfies the precondition of S. Otherwise, *false* is returned, and *message* contains the error message.

void S.call(*leda_graph* G, *leda_list*<*leda_edge*>& L)

calls the subgraph module for graph G. Returns in L the list of edges $e_1, \ldots, e_k$.

void S.call(*leda_graph*& G)

calls the subgraph module for graph G, and deletes all edges $e_1, \ldots, e_k$ in G.

void S.call(*GraphCopy*& G, *leda_list*<*leda_edge*>& L)

calls the subgraph module for graph copy G. Deletes all edges $e_1, \ldots, e_k$ in G, and returns in L the list of corresponding edges $original(e_1), \ldots, original(e_k)$ in $original(G)$.

int S.num_del_edges( ) returns the number k of edges in $E \cap E'$.

Protected Overridables to Implement Functionality

PURE VIRTUAL

void S._call(*leda_graph* G, *leda_list*<*leda_edge*>& L)

implements the computation of the subgraph for graph copy G. Returns in L the list of edges in $original(G)$ that correspond to deleted edges in G.

4.6 Edge Insert Modules (EdgeInsertModule)

1. Base Classes

2. Definition

The class *EdgeInsertModule* is the base class for edge insert modules. An edge insert module gets as input a planarized graph PG, which is a copy of a graph G, and a list of edges $e_1, \ldots, e_k \in G$. It inserts $e_1, \ldots, e_k$ into PG by replacing edge crossings with dummy nodes.

Edge insert modules are used for computing a planarization with a planar subgraph. The edges that are not in the planar subgraph are reinserted by means of an edge insert module (see 8.1).

Input and Output Parameters

input parameter:	graph	$in\_graph = 1$
	edge list	$in\_list = 2$
output parameter:	graph	$out\_graph = 1$

The integer constants *in_graph*, *in_list* and *out_graph* can be used to denote the positions of the input and output parameters.

Implementing Edge Insert Modules

An implementation of an edge insert module must override the protected method call_insert(PG, *list<edge>& e_insert*), which gets as input a planarized graph PG, and a list of edges $e\_insert = e_1, \ldots, e_k$, which must be inserted.

The planarized graph PG is a copy of G, and $e_1, \ldots, e_k \in G$. For each edge e_i, the implementation inserts a chain of edges starting at $copy(source(e_i))$ and ending at $copy(target(e_i))$ that leads over dummy nodes. The dummy nodes are new nodes resulting from splitting an edge, which will represent edge crossings in a drawing of the graph.

The class *PlaneGraphCopy* stores these chains and provides methods (split, new_edge_chain) for reinserting edges.

3. Initialization

EdgeInsertModule I; initializes an edge insert module.

4. Operations

Standard Interface

bool *I*.check(*PlaneGraphCopy* *G*, *leda_list*<*leda_edge*>& *e_insert*, *leda_string*& *message*)

returns *true* if *G* satisfies the precondition of *I*. Otherwise, *false* is returned, and *message* contains the error message.

void *I*.call(*PlaneGraphCopy*& *PG*, *leda_list*<*leda_edge*>& *e_insert*)

calls the edge insert module for graph *PG*.

Protected Overridables to Implement Functionality

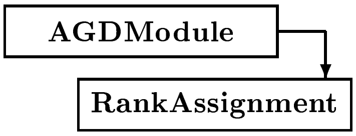
PURE VIRTUAL

void *I*_call(*PlaneGraphCopy*& *G*, *leda_list*<*leda_edge*>& *e_insert*)

implements the edge insert module for the planarized graph *PG*, and the list of edges *e_insert*.

4.7 RankAssignment Modules (RankAssignment)

1. Base Classes



2. Definition

Modules of type *RankAssignment* partition the nodes of a graph G into levels. Each node v is assigned a non-negative rank i, what means that v belongs to the ith level.

Such a ranking can be used to build a node hierarchy of G. The hierarchical drawing method by Sugiyama builds a node hierarchy in a first step and computes a layout that represents the hierarchy, i.e., all nodes on the same level have the same y-coordinate.

Input and Output Parameters

input parameter:	graph	$in\_graph = 1$
output parameter:	ranking	$out\_ranking = 1$

The integer constants *in_graph* and *out_ranking* can be used to denote the positions of the input and output parameters.

3. Initialization

RankAssignment R; initializes a rank assignment module.

4. Operations

bool R.check(*leda_graph* G, *leda_string*& *message*)

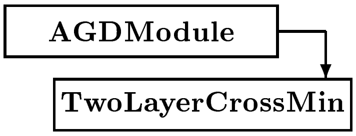
returns *true* if G satisfies the precondition of R. Otherwise, *false* is returned, and *message* contains the error message.

void R.call(*leda_graph* G, *leda_node_array*<*int*>& *rank*) PURE VIRTUAL

implements the module call. Returns the rank for each node $v \in G$ in $rank[v]$.

4.8 Two Layer Crossing Minimization (TwoLayerCrossMin)

1. Base Classes



2. Definition

Modules of type *TwoLayerCrossMin* perform a two layer crossing minimization in a node hierarchy H, that is, the nodes of a graph G on a layer L, called the changeable layer, are reordered according to a neighbored layer of L, called the fixed layer, in order to reduce the number of edge crossings between these two layers.

Modules of type *TwoLayerCrossMin* are used in a process, which iteratively performs two layer crossing minimization between two neighbored layers in order to reduce the total number of edge crossings in the hierarchy. For efficiency reasons, the class *TwoLayerCrossMin* contains two notification methods init and cleanup. A user must first call the init method with the graph G and the node hierarchy H. Then, the call method can be called several times with layers belonging to H. Finally, the cleanup method should be called to free resources.

Implementing Two Layer Crossing Minimization Modules

An implementation of a two layer crossing minimization module must override the method call(G,L), which gets as input a graph G and a layer L. The class *Layer* provides methods to access the fixed and changeable layer, and to manipulate the order of the nodes on the changeable layer. The method init can be overridden to allocate or initialize internal data structures, and the method cleanup can be overridden to free resources allocated by init.

Input and Output Parameters

input parameter:	graph	$in\_graph = 1$
	layer	$in\_layer = 2$
output parameter:	layer	$out\_layer = 1$

The integer constants *in_graph*, *in_layer* and *out_layer* can be used to denote the positions of the input and output parameters.

3. Initialization

TwoLayerCrossMin M; initializes a two layer crossing minimization module.

4. Operations

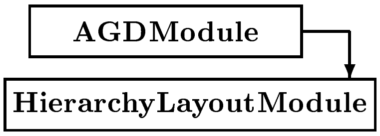
void M.init(*leda_graph* G, *Hierarchy* H)

called before the crossing minimization.

void	M.call(*leda_graph* G, *Layer*& L)	PURE VIRTUAL
		implements the module call for graph G. Reorders the nodes on layer L.
void	M.cleanup( )	called after the crossing minimization.

4.9 Hierarchy Layout Modules (HierarchyLayoutModule)

1. Base Classes



2. Definition

Modules of type *HierarchyLayoutModule* compute a layout of a node hierarchy H. H is a hierarchy of a graph GC, which is a copy of a graph G. GC may contain dummy nodes that are not copies of nodes in G. These dummy nodes have degree two and represent edge bends in the final drawing.

Implementing Hierarchy Layout Modules

An implementation of a hierarchy layout module must override the method call(GC,H,$bbox$,A), which gets as input a graph copy GC and a node hierarchy H of GC. It returns the layout in A and the bounding box of the computed layout in $bbox$.

Input and Output Parameters

input parameter:	graph	$in\_graph = 1$
	hierarchy	$in\_hierarchy = 2$
output parameter:	layout	$out\_layout = 1$

The integer constants *in_graph*, *in_layer* and *out_layer* can be used to denote the positions of the input and output parameters.

3. Initialization

HierarchyLayoutModule L; initializes a hierarchy layout module.

4. Operations

PURE VIRTUAL

void L.call(*GraphCopy* GC, *Hierarchy* H, *LayoutInterface*& A)

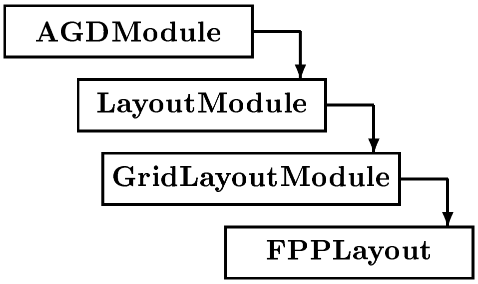
implements the module call for graph GC and hierarchy H. Assigns the layout to A.

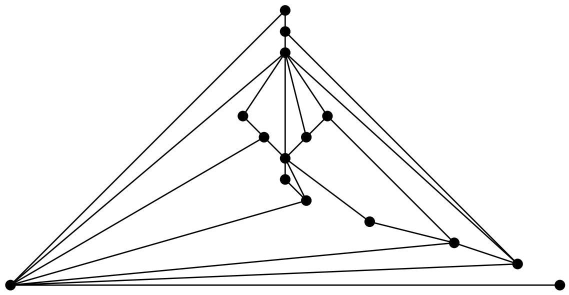
Chapter 5

Layout Algorithms

5.1 The de Fraysseix, Pach, Pollack Algorithm (FPPLayout)

1. Base Classes





2. Definition

The class *FPPLayout* represents the layout algorithm by de Fraysseix, Pach, Pollack [2].

This algorithm draws a planar graph G straight-line without crossings. G may not contain self-loops or mulitple edges. The grid layout size is at most $(2n-4) \times (n-2)$ for a graph with n nodes ($n \geq 3$).

Common Informations

Algorithm		**Implementation**	
name	**de Fraysseix et al.**	author	*Carsten Gutwenger*
long name	**de Fraysseix, Pach, Pollack**	date	*february 1997*
author	*de Fraysseix, Pach, Pollack*	version	*1.0*

Pre- and Postcondition

precondition	=	$\{\, planar, simple, no\ self\ loops \,\}$
postcondition(PRE)	=	$\{\, straight\text{-}line, no\ crossings \,\}$

---

3. Creation

FPPLayout L; creates an instance L of type *FPPLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool L.check(*graph* G, *string*& *message*)

void L.call(*graph* G, *LayoutInterface*& A)

void L.call(*graph* G, *face* f, *LayoutInterface*& A)

void L.call(*PlaneGraphCopy*& *PG*, *face* f, *LayoutInterface*& A)

void L.call_grid(*graph* G, *GridLayout*& *gl*)

void L.call_grid(*graph* G, *face* f, *GridLayout*& *gl*)

void L.call_grid(*PlaneGraphCopy*& *PG*, *face* f, *GridLayout*& *gl*)

IRect L.get_grid_area( )

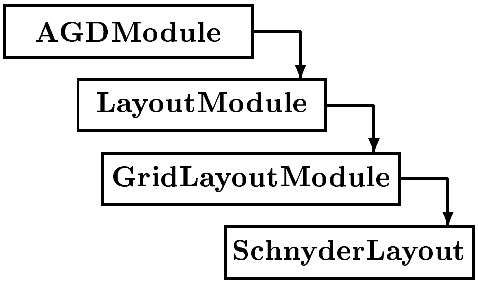
void L.set_coordinate_mapper(*GridCoordinateMapper* *gcm*)

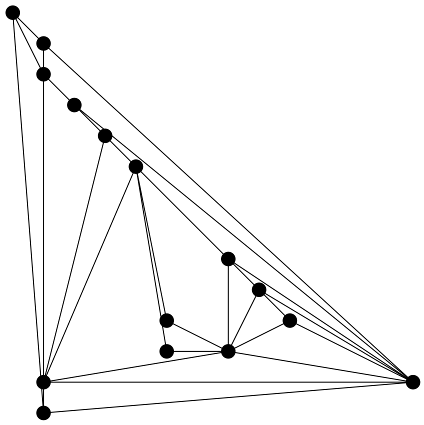
5. Implementation

The computation of the layout takes time $O(n)$ where n is the number of nodes of the drawn graph.

5.2 The Schnyder Algorithm (SchnyderLayout)

1. Base Classes





2. Definition

The class *SchnyderLayout* represents the layout algorithm by Schnyder [18].

This algorithm draws a planar graph G straight-line without crossings. G may not contain self-loops or mulitple edges. The grid layout size is $(n-2) \times (n-2)$ for a graph with n nodes $(n \geq 3)$.

Common Informations

Algorithm		**Implementation**	
name	**Schnyder**	author	*Dirk Ambras, David Alberts*
long name	**Schnyder**	date	*1995/1997*
author	*W. Schnyder*	version	

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, planar, simple, no\ \ self\ \ loops \,\} \\ \text{postcondition}(PRE) & = & \{\, straight\text{-}line, no\ \ crossings \,\} \end{array}$$

---

3. Creation

SchnyderLayout L; creates an instance L of type *SchnyderLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool L.check(*graph* G, *string*& *message*)

void L.call(*graph* G, *LayoutInterface*& A)

void L.call(*graph* G, *face* f, *LayoutInterface*& A)

void L.call(*PlaneGraphCopy*& PG, *face* f, *LayoutInterface*& A)

void L.call_grid(*graph* G, *GridLayout*& *gl*)

void L.call_grid(*graph* G, *face* f, *GridLayout*& *gl*)

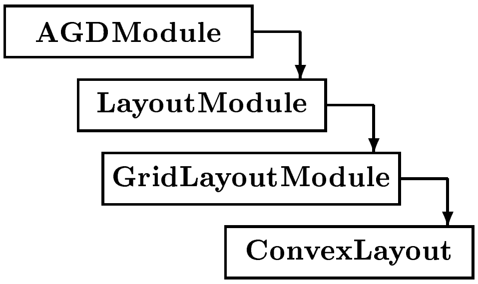
void L.call_grid(*PlaneGraphCopy*& PG, *face* f, *GridLayout*& *gl*)

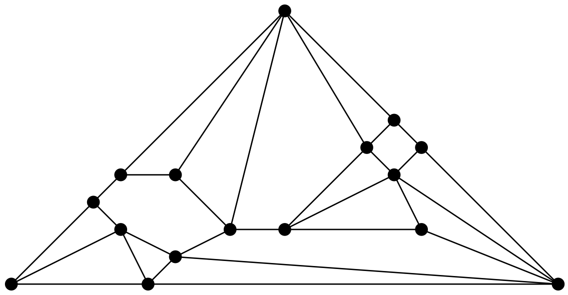
IRect L.get_grid_area( )

void L.set_coordinate_mapper(*GridCoordinateMapper gcm*)

5.3 The Convex Algorithm (ConvexLayout)

1. Base Classes





2. Definition

The class *ConvexLayout* represents the layout algorithm Convex by Goos Kant [11].

This algorithm draws a planar, triconnected graph G straight-line without crossings. G may not contain self-loops or mulitple edges. All faces are drawn convex, the grid layout size is at most $(2n-4) \times (n-2)$ for a graph with n nodes ($n \geq 3$).

Common Informations

Algorithm		**Implementation**	
name	**Convex**	author	*Carsten Gutwenger*
long name	**Convex (Kant)**	date	*march 1997*
author	*Goos Kant*	version	*2.0*

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, \textit{planar}, \textit{triconnected}, \textit{simple}, \textit{no self loops} \,\} \\ \text{postcondition}(\textit{PRE}) & = & \{\, \textit{straight-line}, \textit{no crossings}, \textit{convex} \,\} \end{array}$$

Optional Parameters

Instances of *ConvexLayout* provide the following optional parameters:

- *bool* **size_opt** = *true*
  a boolean value determining if the grid size of the layout is optimized compared to the original Convex algorithm. This option reduces the grid size especially for graphs with few edges. If the graph is (almost) triangulated, the grid size will be (almost) the same.

- *double* **base_ratio** = 0.33
  a double value determining, how many nodes are drawn on the base line. If *base_ratio* is r and the outerface is F, the number of nodes on the base line is $\min(2, \max(\ell, r|\,F\,|))$, where $\ell = \max\{k \mid v_1, \ldots, v_k$ is a path on F_{out} with $(v_i, v_j) \notin E$ for all $j \neq i+1\}$.

---

3. Creation

ConvexLayout L; creates an instance L of type *ConvexLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool L.check(*graph* G, *leda_string&* *message*)

void L.call(*graph* G, *LayoutInterface&* A)

void L.call(*graph* G, *face* f, *LayoutInterface&* A)

void L.call(*PlaneGraphCopy&* PG, *face* f, *LayoutInterface&* A)

void L.call_grid(*graph* G, *GridLayout&* *gl*)

void L.call_grid(*graph* G, *face* f, *GridLayout&* *gl*)

void L.call_grid(*PlaneGraphCopy&* PG, *face* f, *GridLayout&* *gl*)

IRect L.get_grid_area( )

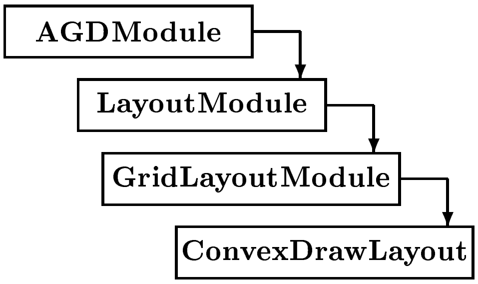
void L.set_coordinate_mapper(*GridCoordinateMapper* *gcm*)

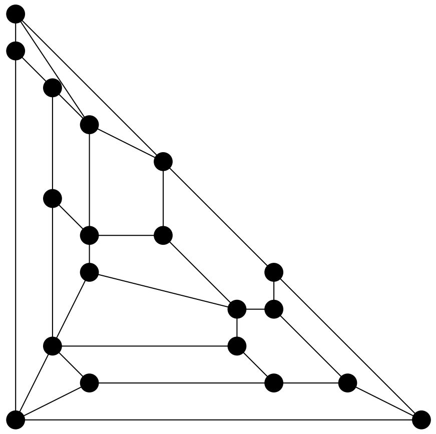
5. Implementation

The computation of the layout takes time $O(n)$ where n is the number of nodes of the drawn graph.

5.4 The ConvexDraw Algorithm (ConvexDrawLayout)

1. Base Classes





2. Definition

The class *ConvexDrawLayout* represents the layout algorithm ConvexDraw by Marek Chrobak and Goos Kant [1].

This algorithm draws a planar, triconnected graph G straight-line without crossings. G may not contain self-loops or mulitple edges. All faces are drawn convex, the grid layout size is at most $(n-1) \times (n-1)$ for a graph with n nodes ($n \geq 3$).

Common Informations

Algorithm		**Implementation**	
name	**ConvexDraw**	author	*Carsten Gutwenger*
long name	**ConvexDraw (Kant)**	date	*march 1997*
author	*Marek Chrobak, Goos Kant*	version	*2.0*

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, planar, triconnected, simple, no\ \ self\ \ loops \,\} \\ \text{postcondition}(PRE) & = & \{\, straight\text{-}line, no\ \ crossings, convex \,\} \end{array}$$

Optional Parameters

Instances of *ConvexDrawLayout* provide the following optional parameters:

- *bool* **size_opt** = *true*
  a boolean value determining if the grid size of the layout is optimized compared to the original Convex algorithm. This option reduces the grid size especially for graphs with few edges. If the graph is (almost) triangulated, the grid size will be (almost) the same.

- *bool* **side_opt** = *false*
  a boolean value determining if it is allowed to change the shape of the outerface (from a triangle) in order to improve the grid size of the layout.

- *double* **base_ratio** $= 0.33$
  a double value determining, how many nodes are drawn on the base line. If *base_ratio* is r and the outerface is F, the number of nodes on the base line is $\min(2, \max(\ell, r|F|))$, where $\ell = \max\{k \mid v_1, \ldots, v_k$ is a path on F_{out} with $(v_i, v_j) \notin E$ for all $j \neq i + 1\}$.

---

3. Creation

ConvexDrawLayout L; creates an instance L of type *ConvexDrawLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool L.check(*graph* G, *string*& *message*)

void L.call(*graph* G, *LayoutInterface*& A)

void L.call(*graph* G, *face* f, *LayoutInterface*& A)

void L.call(*PlaneGraphCopy*& PG, *face* f, *LayoutInterface*& A)

void L.call_grid(*graph* G, *GridLayout*& *gl*)

void L.call_grid(*graph* G, *face* f, *GridLayout*& *gl*)

void L.call_grid(*PlaneGraphCopy*& PG, *face* f, *GridLayout*& *gl*)

IRect L.get_grid_area( )

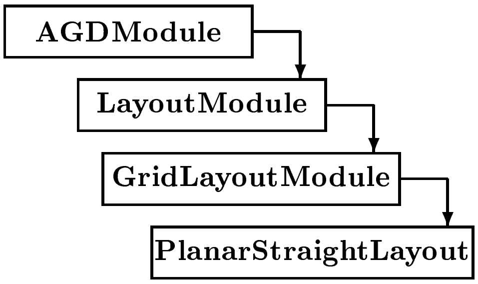
void L.set_coordinate_mapper(*GridCoordinateMapper* *gcm*)

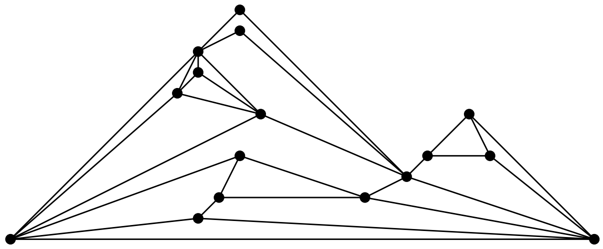
5. Implementation

The computation of the layout takes time $O(n)$ where n is the number of nodes of the drawn graph.

5.5 The PlanarStraight Algorithm (PlanarStraightLayout)

1. Base Classes





2. Definition

The class *PlanarStraightLayout* represents the layout algorithm PlanarStraight (Gutwenger and Mutzel [6]), which is a generalization of the algorithm Convex (Kant [11]).

This algorithm draws a planar graph G straight-line without crossings. G may not contain self-loops or mulitple edges. The grid layout size is at most $(2n-4) \times (n-2)$ for a graph with n nodes ($n \geq 3$).

Common Informations

Algorithm

name	**PlanarStraight**
long name	**PlanarStraight (Gutwenger/Mutzel)**
author	*Carsten Gutwenger, Petra Mutzel*

Implementation

author	*Carsten Gutwenger*
date	*march 1997*
version	*2.0*

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, planar, simple, no\ self\ loops \,\} \\ \text{postcondition}(PRE) & = & \{\, straight\text{-}line, no\ crossings, convex \,\} \end{array}$$

Exchangeable Modules

Instances of type *PlanarStraightLayout* provide the following module options:

- **augmenter**
  an *AugmentModule* option used for planar biconnected augmentation.

guaranteed precondition:	*planar*, *simple*, *no self loops*, *directed*
required postcondition:	*planar*, *biconnected*, *simple*, *no self loops*
initial module:	*PlanAug*

Optional Parameters

Instances of *PlanarStraightLayout* provide the following optional parameters:

- *bool* **size_opt** = *true*
  a boolean value determining if the grid size of the layout is optimized compared to the original Convex algorithm. This option reduces the grid size especially for graphs with few edges. If the graph is (almost) triangulated, the grid size will be (almost) the same.

- *bool* **max_outer** = *true*
  a boolean value determining if the algorithm chooses the face of G with maximal degree as outerface (*true*), or the first face in the list of faces.

- *double* **base_ratio** = 0.33
  a double value determining, how many nodes are drawn on the base line. If *base_ratio* is r and the outerface is F, the number of nodes on the base line is $\min(2, \max(\ell, r|F|))$, where $\ell = \max\{k \mid v_1, \ldots, v_k$ is a path on F_{out} with $(v_i, v_j) \notin E$ for all $j \neq i+1\}$.

---

3. Creation

PlanarStraightLayout L; creates an instance L of type *PlanarStraightLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool L.check(*graph G*, *string& message*)

void L.call(*graph G*, *LayoutInterface& A*)

void L.call(*graph G*, *face f*, *LayoutInterface& A*)

void L.call(*PlaneGraphCopy& PG*, *face f*, *LayoutInterface& A*)

void L.call_grid(*graph G*, *GridLayout& gl*)

void L.call_grid(*graph G*, *face f*, *GridLayout& gl*)

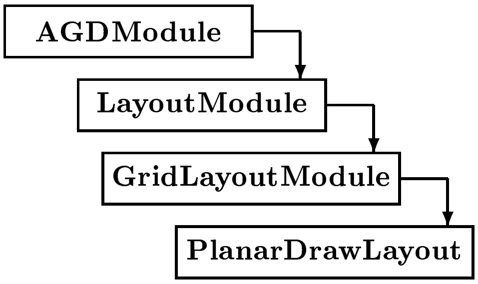
void L.call_grid(*PlaneGraphCopy& PG*, *face f*, *GridLayout& gl*)

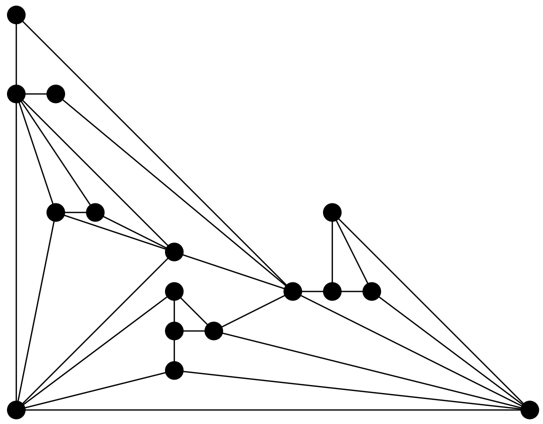
IRect L.get_grid_area( )

void L.set_coordinate_mapper(*GridCoordinateMapper gcm*)

5.6 The PlanarDraw Algorithm (PlanarDrawLayout)

1. Base Classes





2. Definition

The class *PlanarDrawLayout* represents the layout algorithm PlanarDraw (Gutwenger and Mutzel [6]), which is a generalization of the algorithm ConvexDraw (Chrobak and Kant [1]).

This algorithm draws a planar graph G straight-line without crossings. G may not contain self-loops or mulitple edges. The grid layout size is at most $(n-1) \times (n-1)$ for a graph with n nodes $(n \geq 3)$.

Common Informations

Algorithm		**Implementation**	
name	**PlanarDraw**	author	*Carsten Gutwenger*
long name	**PlanarDraw (Gutwenger/Mutzel)**	date	*march 1997*
author	*Carsten Gutwenger, Petra Mutzel*	version	*1.0*

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, \mathit{planar}, \mathit{simple}, \mathit{no\ self\ loops} \,\} \\ \text{postcondition}(\mathit{PRE}) & = & \{\, \mathit{straight\text{-}line}, \mathit{no\ crossings}, \mathit{convex} \,\} \end{array}$$

Exchangeable Modules

Instances of type *PlanarDrawLayout* provide the following module options:

- **augmenter**
  an *AugmentModule* option used for planar biconnected augmentation.

guaranteed precondition:	*planar*, *simple*, *no self loops*, *directed*
required postcondition:	*planar*, *biconnected*, *simple*, *no self loops*
initial module:	*PlanAug*

Optional Parameters

Instances of *PlanarDrawLayout* provide the following optional parameters:

- *bool* **size_opt** = *true*
  a boolean value determining if the grid size of the layout is optimized compared to the original Convex algorithm. This option reduces the grid size especially for graphs with few edges. If the graph is (almost) triangulated, the grid size will be (almost) the same.

- *bool* **side_opt** = *false*
  a boolean value determining if it is allowed to have slopes other than -1, 0, +1 on the outerface.

- *bool* **max_outer** = *true*
  a boolean value determining if the algorithm chooses the face of G with maximal degree as outerface (*true*), or the first face in the list of faces.

- *double* **base_ratio** = 0.33
  a double value determining, how many nodes are drawn on the base line. If *base_ratio* is r and the outerface is F, the number of nodes on the base line is $\min(2, \max(\ell, r|F|))$, where $\ell = \max\{k \mid v_1, \ldots, v_k$ is a path on F_{out} with $(v_i, v_j) \notin E$ for all $j \neq i+1\}$.

---

3. Creation

PlanarDrawLayout *L*; creates an instance *L* of type *PlanarDrawLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool L.check(*graph G*, *string& message*)

void L.call(*graph G*, *LayoutInterface& A*)

void L.call(*graph G*, *face f*, *LayoutInterface& A*)

void L.call(*PlaneGraphCopy& PG*, *face f*, *LayoutInterface& A*)

void L.call_grid(*graph G*, *GridLayout& gl*)

void L.call_grid(*graph G*, *face f*, *GridLayout& gl*)

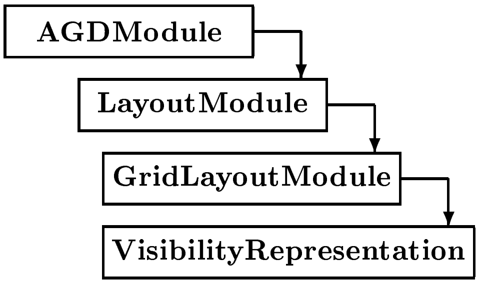
void L.call_grid(*PlaneGraphCopy& PG*, *face f*, *GridLayout& gl*)

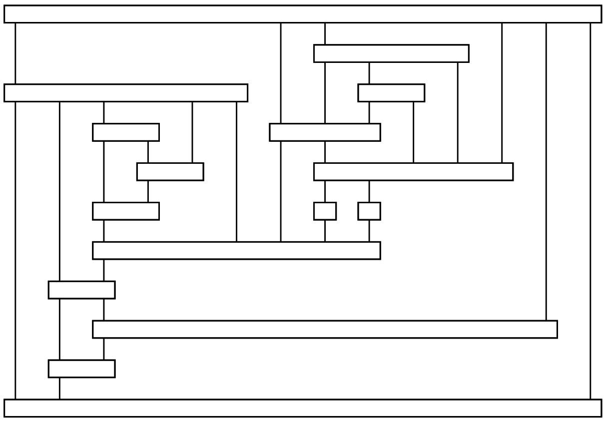
IRect *L*.get_grid_area( )

void *L*.set_coordinate_mapper(*GridCoordinateMapper gcm*)

5.7 Visibility Representation (VisibilityRepresentation)

1. Base Classes





2. Definition

The class *VisibilityRepresentation* represents the visibility representation algorithm by Rosenstiehl and Tarjan [17].

This algorithm draws a planar, biconnected graph G such that each node is represented by a horizontal segment and each edge is represented by a vertical segment, and incidence between nodes and edges translates to intersection of horizontal and vertical segments.

The grid size is at most $(2n-4) \times (n-1)$ for a graph with n nodes. G may not contain self-loops or multiple edges.

Common Informations

Algorithm		**Implementation**	
name	**Visibility Representation**	author	*David Alberts*
long name	**Weak Visibility Representation**	date	*1996/1997*
author	*Rosenstiehl, Tarjan*	version	

Pre- and Postcondition

precondition	=	$\{$ *planar*, *biconnected*, *simple*, *no self loops* $\}$
postcondition(*PRE*)	=	$\{$ *straight-line*, *no crossings* $\}$

---

3. Creation

VisibilityRepresentation L;

creates an instance L of type *VisibilityRepresentation*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

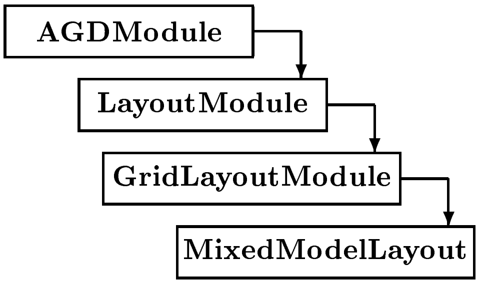
bool	*L*.check(*graph G*, *string*& *message*)
void	*L*.call(*graph G*, *LayoutInterface*& *A*)
void	*L*.call_grid(*graph G*, *GridLayout*& *gl*)
IRect	*L*.get_grid_area( )
void	*L*.set_coordinate_mapper(*GridCoordinateMapper gcm*)

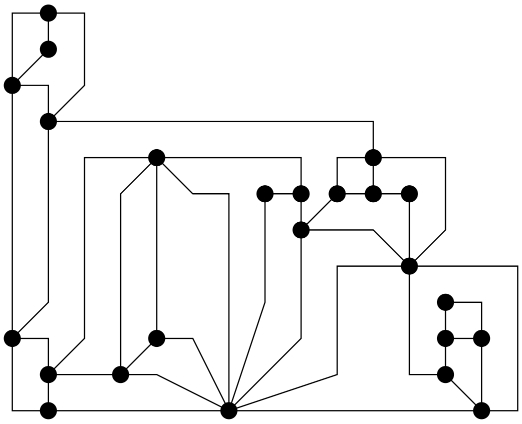
5. Implementation

The computation of the layout takes time $O(n)$ where n is the number of nodes of the drawn graph.

5.8 The Mixed-Model Algorithm (MixedModelLayout)

1. Base Classes





2. Definition

The class *MixedModelLayout* represents the layout algorithm Mixed-Model (Kant [11]) extented by a generalization to connected planar graphs (Gutwenger and Mutzel [6]).

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. G may not contain self-loops or mulitple edges. G must be triconnected if the option *co_type* is *co_triconnected*, and connected if it is *co_biconnected*. The grid size is at most $(2n-6) \times (3n-9)$, the number of bends is at most $5n-15$, and every edge has length $O(n)$ if G has n nodes.

Common Informations

Algorithm		**Implementation**	
name	**Mixed-Model**	author	*Carsten Gutwenger*
long name	**Mixed-Model (Kant,...)**	date	*march 1997*
author	*Goos Kant, ...*	version	*2.1*

Pre- and Postcondition

$$\text{precondition} = \begin{cases} \{\,planar, simple, no\ \ self\ \ loops\,\} & \\ \qquad \textbf{if } co\_type = co\_biconnected & \\ \{\,planar, triconnected, simple, no\ \ self\ \ loops\,\} & \\ \qquad \textbf{if } co\_type = co\_triconnected & \end{cases}$$

$$\text{postcondition}(PRE) = \{\,no\ \ crossings\,\}$$

Exchangeable Modules

Instances of type *MixedModelLayout* provide the following module options:

- **augmenter**
  an *AugmentModule* option used for planar biconnected augmentation.

guaranteed precondition:	*planar*, *simple*, *no self loops*, *directed*
required postcondition:	*planar*, *biconnected*, *simple*, *no self loops*
initial module:	*PlanAug*

Optional Parameters

Instances of *MixedModelLayout* provide the following optional parameters:

- *double* **base_ratio** = 0.33
  a double value determining, how many nodes are drawn on the base line. If *base_ratio* is r and the outerface is F, the number of nodes on the base line is $\min(2, \max(\ell, r|F|))$, where $\ell = \max\{k \mid v_1, \ldots, v_k$ is a path on F_{out} with $(v_i, v_j) \notin E$ for all $j \neq i+1\}$.

- *bool* **do_postprocessing** = *true*
  a boolean value determining if a postprocessing step is used improving the positions of outpoints.

- *CanonicalOrderType* **co_type** = *co_biconnected*
  an enumeration value of type *CanonicalOrderType* determining the canonical ordering used in the algorithm (see *CanonicalOrder*).

---

3. Creation

MixedModelLayout L; creates an instance L of type *MixedModelLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool L.check(*graph* G, *string*& *message*)

void L.call(*graph* G, *LayoutInterface*& A)

void L.call(*graph* G, *face* f, *LayoutInterface*& A)

void L.call(*PlaneGraphCopy*& PG, *face* f, *LayoutInterface*& A)

void L.call_grid(*graph* G, *GridLayout*& *gl*)

void L.call_grid(*graph* G, *face* f, *GridLayout*& *gl*)

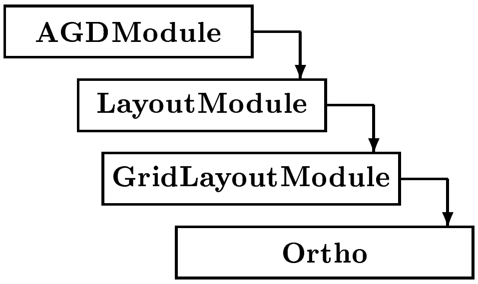
void L.call_grid(*PlaneGraphCopy*& PG, *face* f, *GridLayout*& *gl*)

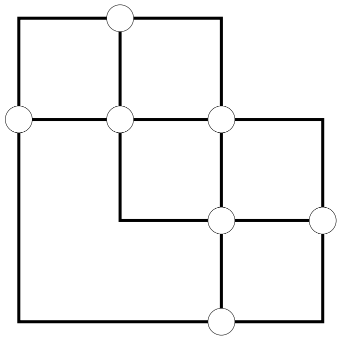
IRect L.get_grid_area( )

void L.set_coordinate_mapper(*GridCoordinateMapper gcm*)

5.9 The Orthogonal Drawing Algorithm (Ortho)

1. Base Classes





2. Definition

The class *Ortho* represents Tamassia's algorithm [20] for constructing orthogonal grid embeddings of 4–planar graphs. This class of graphs contains planar graphs G whose maximal vertex degrees $\Delta(G)$ do not exceed 4. An input graph G will be drawn on a grid such that the edges run along grid lines, vertices are placed on different grid points, and no crossings occur. The number of bends will be minimal with respect to the given embedding. G must be a planar graph without multiple edges and self loops.

Common Informations

Algorithm		**Implementation**	
name	**Ortho**	author	*Gunnar W. Klau*
long name	**Ortho (Tamassia)**	date	*June 1998*
author	*Roberto Tamassia*	version	*1.4*

Pre- and Postcondition

precondition = { *planar, simple, no self loops, 4–graph, connected* }
postcondition(PRE) = { *no crossings, orthogonal* }

Optional Parameters

Instances of *Ortho* provide the following optional parameters:

- *bool* **dissect** = *true*
  Truth value indicating if the faces should be dissected in rectangles during the compaction phase. This is the original method proposed in [20]. Switching this off may lead to violated embeddings in the current version.

- *bool* **check** = *true*
  If set, the checking routines are activated. Especially the orthogonal representation is

verified, which is helpful for debugging and certificates the correctness of the computation.

---

3. Creation

Ortho *L*; creates an instance *L* of type *Ortho*.

4. Operations

bool	*L*.dissect( )	returns the optional parameter *dissect*
void	*L*.dissect(*bool d*)	sets the parameter *dissect* to *d*
bool	*L*.optimally( )	returns the optional parameter *optimally*
void	*L*.optimally(*bool o*)	sets the parameter *optimally* to *d*
bool	*L*.verify( )	returns the verify status
void	*L*.verify(*bool c*)	sets the verify status to *c*

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool *L*.check(*graph G*, *string*& *message*)

void *L*.call(*graph G*, *LayoutInterface*& *A*)

void *L*.call(*graph G*, *face f*, *LayoutInterface*& *A*)

void *L*.call(*PlaneGraphCopy*& *PG*, *face f*, *LayoutInterface*& *A*)

void *L*.call_grid(*graph G*, *GridLayout*& *gl*)

void *L*.call_grid(*graph G*, *face f*, *GridLayout*& *gl*)

void *L*.call_grid(*PlaneGraphCopy*& *PG*, *face f*, *GridLayout*& *gl*)

IRect *L*.get_grid_area( )

void *L*.set_coordinate_mapper(*GridCoordinateMapper gcm*)

5. Implementation

The computation of the layout takes time $O(n^2 \log n \log n)$ where n is the number of nodes of the graph to be drawn. This is due to the flow algorithm, we chose `LEDA`'s `MIN_COST_FLOW` because of its simplicity and stability. A different choice would influence the running time.

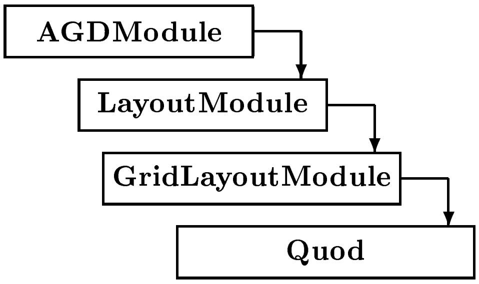
6. Example

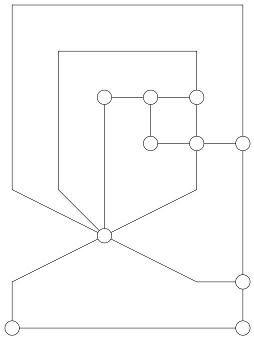
The following program shows how to use the orthogonal layout algorithm in combination with a `TopologyModule`. The user can draw or load a graph in a `GraphWin`, pressing the "done" button initiates the layout process. If the input drawing divides the plane into faces, the same faces will emerge in the output drawing (i.e. the topological representation and the external face will be preserved during the computations).

```
#include <AGD/TopologyModule.h>
#include <AGD/Ortho.h>
#include <AGD/GraphWinInterface.h>
#include <LEDA/graphwin.h>

main() {

  GraphWin W; W.display();

  while (W.edit()) {

    GraphWinInterface I(W);
    graph& G = W.get_graph(); face f_0; string msg;
    TopologyModule T; Ortho O;
    T.call(G, f_0, I);
    if (O.check(G, msg)) O.call(G, f_0, I);
    else W.acknowledge(msg);
    W.fill_window();

  }

}
```

5.10 The Quasi Orthogonal Drawing Algorithm (Quod)

1. Base Classes





2. Definition

The class *Quod* is an implementation of an algorithm extending Tamassia's bend minimizing algorithm to the class of planar graphs with arbitrary degree. Its implementation is very similar to the one for the `Ortho` class, and also the operations it provides are identical. The only difference to the user is the extended applicability and the slightly different output style.

Common Informations

Algorithm		**Implementation**	
name	**Quod**	author	*Gunnar W. Klau*
long name	**Quasi Orthogonal Drawing Algorithm**	date	*June 1998*
author	*Gunnar W. Klau*	version	*0.3*

Pre- and Postcondition

precondition	=	{ *planar, simple, no self loops* }
postcondition(*PRE*)	=	{ *no crossings* }

Optional Parameters

Instances of *Quod* provide the following optional parameters:

- *bool* **dissect** = *true*
  Truth value indicating if the faces should be dissected in rectangles during the compaction phase. This is the original method proposed in [20]. Switching this off may lead to violated embeddings in the current version.

- *bool* **check** = *true*
  If set, the checking routines are activated. Especially the orthogonal representation is verified, which is helpful for debugging and certificates the correctness of the computation.

---

3. Creation

Quod L; creates an instance L of type *Quod*.

4. Operations

bool L.dissect( ) returns the optional parameter *dissect*.

void L.dissect(*bool* d)
sets the parameter *dissect* to d.

bool L.verify( ) returns the verify status.

void L.verify(*bool* c)
sets the verify status to c.

bool L.variant( ) ...

void L.variant(*bool* d)
...

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool L.check(*graph* G, *string*& *message*)

void L.call(*graph* G, *LayoutInterface*& A)

void L.call(*graph* G, *face* f, *LayoutInterface*& A)

void L.call(*PlaneGraphCopy*& *PG*, *face* f, *LayoutInterface*& A)

void L.call_grid(*graph* G, *GridLayout*& *gl*)

void L.call_grid(*graph* G, *face* f, *GridLayout*& *gl*)

void L.call_grid(*PlaneGraphCopy*& *PG*, *face* f, *GridLayout*& *gl*)

IRect L.get_grid_area( )

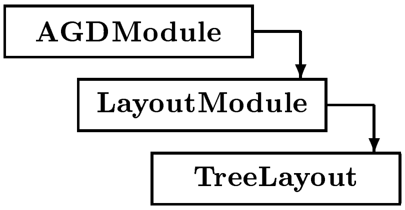
void L.set_coordinate_mapper(*GridCoordinateMapper* *gcm*)

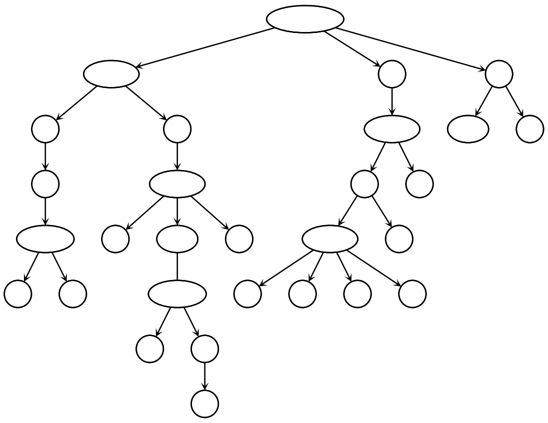
5. Implementation

The computation of a quasi orthogonal grid embedding by algorithm `Quod` takes time $O(n^2 \log n \log n)$ where n is the number of nodes of the graph to be drawn. The running time is asymptotically equal to the time to compute an `Ortho` layout.

5.11 Walker's Algorithm (TreeLayout)

1. Base Classes





2. Definition

The class *TreeLayout* represents the layout algorithm by Walker II (Reingold and Tilford, Walker [16, 22, 13]).

This algorithm draws binary and general rooted trees. The graph has to be directed, connected, and has to have exactly one source. Furthermore the equality $|V| - 1 = |E|$ must hold.

The drawing is straight-line. Nodes of the same level are placed on a horizontal line.

Common Informations

Algorithm		**Implementation**	
name	**TreeLayout**	author	*Sebastian Leipert*
long name	**Layout algorithm for general trees**	date	*may 1997*
author	*John Q. Walker II*	version	*0.1*

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, \textit{tree}, \textit{single source} \,\} \\ \text{postcondition}(PRE) & = & \{\, \textit{straight-line}, \textit{no crossings}, \textit{upward planar} \,\} \end{array}$$

Optional Parameters

Instances of *TreeLayout* provide the following optional parameters:

- *double* **sib_separation**
  The spacing between two sibling nodes.
- *double* **subtree_separation**
  The spacing between two adjacent subtrees.
- *double* **level_separation**
  The spacing between the nodes of two adjacent levels.

- *double* **left_margin**
  Additional spacing at the left side of the drawing.

- *double* **upper_margin**
  Additional spacing at the upper side of the drawing.

---

3. Creation

TreeLayout L; creates an instance L of type *TreeLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base class (*LayoutModule*).

bool L.check(*graph* G, *string*& *message*)

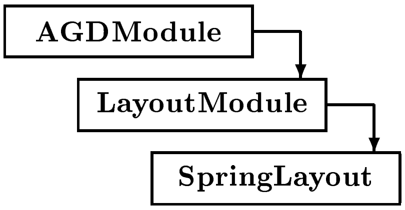
void L.call(*graph* G, *LayoutInterface*& A)

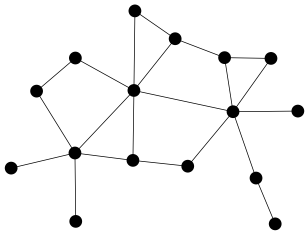
5. Implementation

The running time of the algorithm is $O(|V|)$.

5.12 Spring Embedder (SpringLayout)

1. Base Classes





2. Definition

The class *SpringLayout* represents the spring embedder layout algorithm by Fruchtermann and Rheingold [5].

This algorithm draws a general graph G straight-line. The drawing of a planar graph may contain crossings.

Common Informations

Algorithm

name	**Spring Embedder**
long name	**Spring Embedder (Fruchtermann/Rheingold)**
author	*Fruchtermann, Rheingold*

Implementation

author	*Stefan Naeher, David Alberts*
date	*April 1997*
version	

Pre- and Postcondition

precondition	=	{ }
postcondition(*PRE*)	=	{ *straight-line* }

Optional Parameters

Instances of *SpringLayout* provide the following optional parameters:

- *int* **iterations** = 400
  the number of main steps that the algorithm performs.
- *bool* **noise** = *true*
  determines if the algorithm performs random perturbations.

---

3. Creation

SpringLayout L; creates an instance L of type *SpringLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base class (*LayoutModule*).

bool L.check(*graph* G, *string*& *message*)

void L.call(*graph* G, *LayoutInterface*& A)

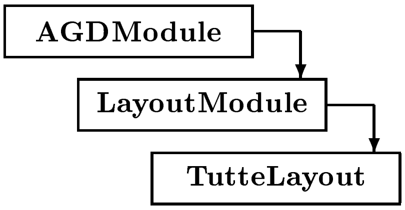
bool L.start_animation(*graph* G, *LayoutInterface*& A)

bool L.next_step( )

void L.stop_animation( )

5.13 Tutte's Algorithm (TutteLayout)

1. Base Classes



2. Definition

The class *TutteLayout* represents the layout algorithm by Tutte[21].

This algorithm draws a planar graph G straight-line without crossings. It can also draw non-planar graphs.

Common Informations

Algorithm		**Implementation**	
name	**Tutte**	author	*David Alberts*
long name	**Tutte**	date	*April 1997*
author	*Tutte*	version	

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, connected, no\ \ self\ \ loops \,\} \\ \text{postcondition}(PRE) & = & \{\, straight\text{-}line \,\} \end{array}$$

Optional Parameters

Instances of *TutteLayout* provide the following optional parameters:

- *DRect* **bbox** = (0.0, 250.0, 0.0, 250.0)
  a double rectangle determining the bounding box of the layout.

---

3. Creation

TutteLayout L; creates an instance L of type *TutteLayout*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base class (*LayoutModule*).

bool *L*.check(*graph G*, *string*& *message*)

void *L*.call(*graph G*, *LayoutInterface*& *A*)

c) Special Functionality

bool *L*.do_tutte(*leda_graph G*, *LayoutInterface*& *a*, *leda_list*<*leda_node*>& *fixed_nodes*, *leda_list*<*DPoint*>& *fixed_positions*)

does the actual computation. *fixed_nodes* and *fixed_positions* contain the nodes with fixed positions.
Precondition: All nodes in *fixed_nodes* are distinct and belong to G. The sizes of *fixed_nodes* and *fuxed_positions* match.

5. Implementation

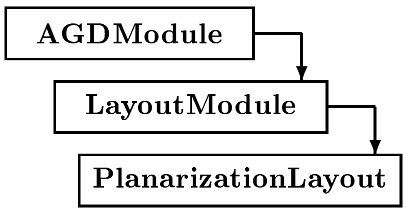
The algorithm solves a system of linear equations and takes $O(n^3)$ time.

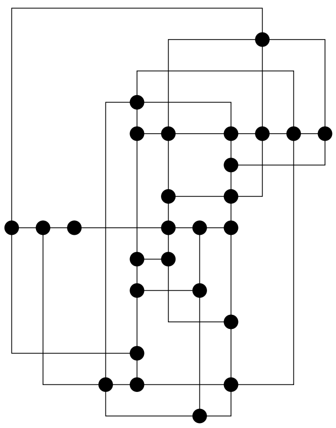
Chapter 6

Generic Layout Algorithms

6.1 The Planarization Method (PlanarizationLayout)

1. Base Classes





2. Definition

The class *PlanarizationLayout* represents a general framework for drawing with planarization. It transforms a general graph into a planar graph using a planarization module, and draws the resulting planar graph using a layout algorithm for planar graphs. The planarization and planar layout algorithm are exchangeable modules allowing to use *PlanarizationLayout* with different methods for planarization and planar drawing.

Common Informations

Algorithm		**Implementation**	
name	**Planarization**	author	*Carsten Gutwenger*
long name	**Planarization**	date	*decembre 1997*
author		version	*1.0*

Pre- and Postcondition

precondition	$=$	$\{\, simple, no\ \ self\ \ loops \,\}$
postcondition(PRE)	$=$	$\emptyset$

Exchangeable Modules

Instances of type *PlanarizationLayout* provide the following module options:

- **planarizer**
  a *PlanarizerModule* option used to compute the planarization of the input graph.

guaranteed precondition:	$\{\, simple, no\ \ self\ \ loops, directed \,\}$
required postcondition:	$\{\, planar, simple, no\ \ self\ \ loops \,\}$
initial module:	*PlanarSubgraph*

- **planar_layouter**
  a *LayoutModule* option used to draw the planarized graph.

guaranteed precondition:	$\{\, simple, no\ \ self\ \ loops, planar \,\}$
required postcondition:	$\{planar\ \ layout\}, \{\, no\ \ crossings \,\}$
initial module:	*Quod*

---

3. Operations

Generic Interface (Inherited Methods)

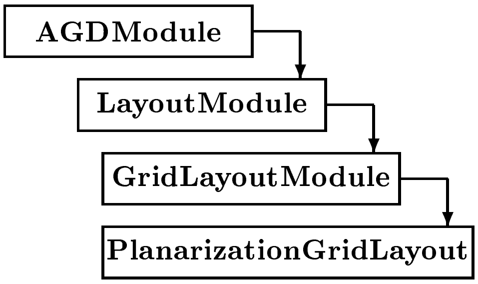
The detailed description of these methods can be found in the manual entries of the base class (*LayoutModule*).

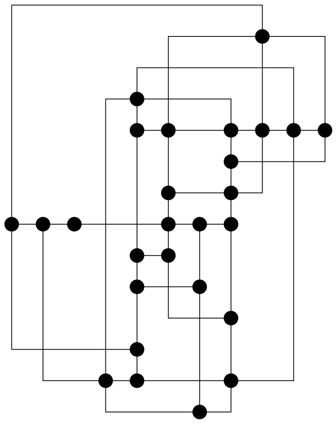
bool L.check(*graph* G, *string*& *message*)

void L.call(*graph* G, *LayoutInterface*& A)

6.2 Planarization on the Grid (PlanarizationGridLayout)

1. Base Classes





2. Definition

The class *PlanarizationGridLayout* represents a general framework for drawing with planarization on the grid. It transforms a general graph into a planar graph using a planarization module, and draws the resulting planar graph using a grid layout algorithm for planar graphs. The planarization and planar layout algorithm are exchangeable modules allowing to use *PlanarizationGridLayout* with different methods for planarization and planar drawing.

Common Informations

Algorithm		**Implementation**	
name	**Planarization**	author	*Carsten Gutwenger*
long name	**Grid Planarization**	date	*may 1998*
author		version	*1.0*

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, simple, no\ \ self\ \ loops \,\} \\ \text{postcondition}(PRE) & = & \emptyset \end{array}$$

Exchangeable Modules

Instances of type *PlanarizationGridLayout* provide the following module options:

- **planarizer**
  a *PlanarizerModule* option used to compute the planarization of the input graph.

guaranteed precondition:	$\{\, simple, no\ \ self\ \ loops, directed \,\}$
required postcondition:	$\{\, planar, simple, no\ \ self\ \ loops \,\}$
initial module:	*PlanarSubgraph*

- **planar_layouter**
  a *GridLayoutModule* option used to draw the planarized graph.

guaranteed precondition:	$\{\, simple, no\ \ self\ \ loops, planar\, \}$
required postcondition:	$\{planar\ \ layout\}, \{\, no\ \ crossings\, \}$
initial module:	*Quod*

---

3. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base classes (*LayoutModule*, *GridLayoutModule*).

bool *L*.check(*graph G*, *string*& *message*)

void *L*.call(*graph G*, *LayoutInterface*& *A*)

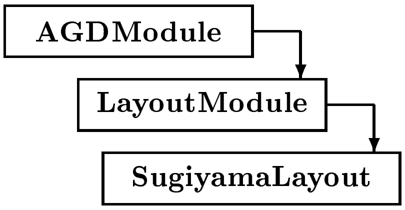
void *L*.call_grid(*graph G*, *GridLayout*& *gl*)

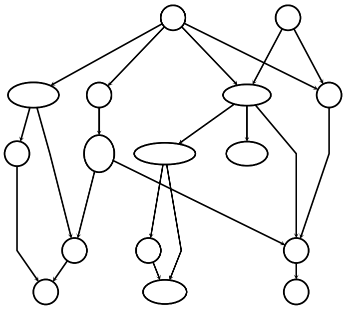
IRect *L*.get_grid_area( )

void *L*.set_coordinate_mapper(*GridCoordinateMapper gcm*)

6.3 Sugiyama's Algorithm (SugiyamaLayout)

1. Base Classes





2. Definition

The class *SugiyamaLayout* represents a general framework for drawing graphs with the hierarchical drawing method suggested by Sugiyama, Tagawa, and Toda [19]. This method works in three phases:

1. We partition the nodes into levels using a *RankAssignment* module. Then, we build a proper hierarchy, that is, we reverse and split edges until each edge is directed downwards and connects two nodes on neighbored layers. The newly introduced nodes are called *dummy nodes* and represent edge bends in the final drawing.

2. We reorder the nodes on each layer in order to reduce the number of edge crossings. Therefore, we iteratively call a *TwoLayerCrossMin* module.

3. We call a *HierarchyLayoutModule* to compute the final layout of the hierarchy. This module assigns coordinates to nodes and bend points to edges.

Common Informations

Algorithm		**Implementation**	
name	**Sugiyama**	author	*Carsten Gutwenger, Michael Krüger*
long name	**Sugiyama**	date	*march 1998*
author	*Sugiyama, Tagawa, Toda*	version	*2.0*

Pre- and Postcondition

$$\begin{array}{lcl} \text{precondition} & = & \{\, simple, no\ self\ loops, no\ reversals\,\} \\ \text{postcondition}(PRE) & = & \emptyset \end{array}$$

Exchangeable Modules

Instances of type *SugiyamaLayout* provide the following module options:

- **ranking**
  a *RankAssignment* option used to compute the ranking.

guaranteed precondition:	$\{$ *simple*, *no self loops*, *no reversals*, *directed* $\}$
required postcondition:	$\emptyset$
initial module:	*HierarchicalRanking*

- **cross_min**
  a *TwoLayerCrossMin* option used in the crossing reduction phase.

guaranteed precondition:	$\{$ *simple*, *no self loops*, *no reversals*, *directed* $\}, \emptyset$
required postcondition:	$\emptyset$
initial module:	*BarycenterHeuristic*

- **comp_coord**
  a *HierarchyLayoutModule* option used to compute the final layout.

guaranteed precondition:	$\{$ *simple*, *no self loops*, *no reversals*, *directed* $\}, \emptyset$
required postcondition:	$\emptyset$
initial module:	*FastHierarchyLayout*

Optional Parameters

Instances of *SugiyamaLayout* provide the following optional parameters:

- *int* **runs** $= 3$
  determines, how many times the crossing minimization is repeated. Each repetition (except for the first) starts with randomly permuted nodes on each layer. Deterministic behaviour can be achieved by setting *runs* to 1.

---

3. Creation

SugiyamaLayout L; creates an instance *L* of type *SugiyamaLayout*.

4. Operations

void *L*.ext_call(*leda_graph G*, *leda_node_array*<*int*>& *rank*, *LayoutInterface*& *A*)
calls *L* for graph *G* with layout interface *A*. Uses the level assignment defined by *rank*.

void *L*.reduce_crossings(*GraphCopy*& *GC*, *Hierarchy*& *H*)
calls only the crossing reduction step for graph copy *GC* and hierarchy *H*.

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base class (*LayoutModule*).

bool *L*.check(*graph G*, *string*& *message*) *void* $\qquad$ *L*.call(*graph G*, *LayoutInterface& A*)

5. Example

Example 1: We use only the crossing reduction phase of the Sugiyama algorithm with the Split heuristic and our own level assignment.

```
#include <LEDA/graphwin.h>
#include <AGD/SugiyamaLayout.h>
#include <AGD/SplitHeuristic.h>
#include <AGD/DfsRanking.h>
#include <AGD/Hierarchy.h>
#include <AGD/GraphCopy.h>

void reduce crossings(graph& G)
{
  node_array<int> rank(G);

  // assign a non-negative level to each node in G
  // we simply use a rank assignment module
  DfsRanking().call(G,rank);

  // build the hierarchy
  GraphCopy GC(G);
  Hierarchy H(GC,rank);

  // call crossing minimization
  SugiyamaLayout sugiyama;
  sugiyama.set_cross_min(SplitHeuristic());
  sugiyama.reduce_crossings(GC,H);

  // the positions of the nodes (in GC) on each layer
  // can now be accessed via H
}
```

Example 2: We compute a layout in a *GraphWin* with our own level assignment.

```
#include <LEDA/graphwin.h>
#include <AGD/SugiyamaLayout.h>
#include <AGD/DfsRanking.h>

void call_with_ranking(GraphWin& GW)
{
  graph& G = GW.get_graph();
  node_array<int> rank(G);

  // assign a non-negative level to each node in G
  // we simply use a rank assignment module
  DfsRanking().call(G,rank);

  // compute layout in GW
  GraphWinInterface A(GW);
  SugiyamaLayout sugiyama;
```

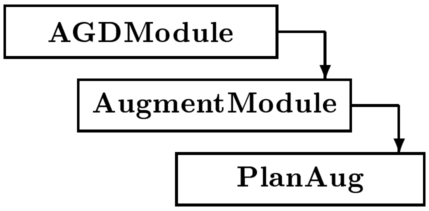
```
  sugiyama.ext_call(G,rank,A);
}
```

Chapter 7

Augmentation Modules

7.1 Planar Biconnected (PlanAug)

1. Base Classes



2. Definition

The *PlanAug* module augments a planar graph to a planar biconnected graph ([4], [15]).

Common Informations

Algorithm		**Implementation**	
name	**Planar Biconnected**	author	*Sergey Fialko*
long name	**Planar Biconnected**	date	*octobre 1997*
author		version	

Pre- and Postcondition

$$\begin{aligned} &\text{precondition} &&= \{\,\textit{planar}\,\} \\ &\text{postcondition}(\textit{PRE}) &&= \{\,\textit{planar}, \textit{biconnected}\,\} \cup \Big(\{\,\textit{simple}, \textit{no self loops}\,\} \cap \textit{PRE}\Big) \end{aligned}$$

Optional Parameters

Instances of *PlanAug* provide the following optional parameters:

- *bool* **streng_cond** = *true*
  ...

---

3. Creation

PlanAug A; creates an instance A of type *PlanAug*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base class (*AugmentModule*).

bool A.check(*graph* G, *string*& *message*)

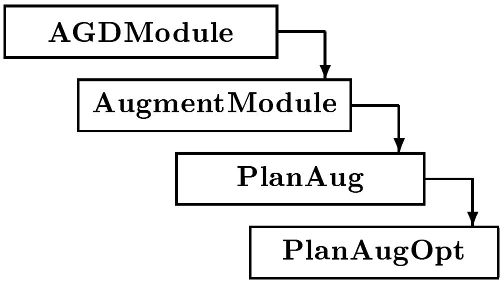
bool A.call(*graph*& G)

bool A.call(*graph*& G, *list*<*edge*>& L)

7.2 Optimal Planar Biconnected (PlanAugOpt)

ABACUS & CPLEX

1. Base Classes



2. Definition

The *PlanAugOpt* module augments a planar graph to a planar biconnected graph by adding the minimal number of edges.

Common Informations

Algorithm		**Implementation**	
name	**Optimal Planar Biconnected**	author	*Sergey Fialko*
long name	**Optimal Planar Biconnected**	date	*octobre 1997*
author		version	

Pre- and Postcondition

$$\begin{aligned} \text{precondition} &= \{\, planar \,\} \\ \text{postcondition}(PRE) &= \{\, planar, biconnected \,\} \cup \Big(\{\, simple, no\ \ self\ \ loops \,\} \cap PRE\Big) \end{aligned}$$

Optional Parameters

Instances of *PlanAugOpt* provide the following optional parameters:

- *bool* **streng_cond** = *true*
  ...

---

3. Creation

PlanAugOpt A; creates an instance A of type *PlanAugOpt*.

4. Operations

b) Generic Interface (Inherited Methods)

bool A.check(*graph* G, *string*& *message*)

returns *true* if G satisfies the precondition of A. Otherwise, *false* is returned, and *message* contains the error message.

bool A.call(*graph*& G)

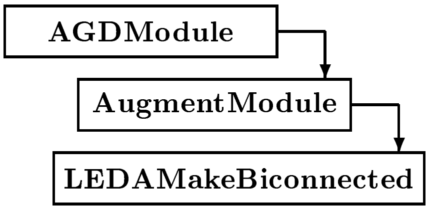
calls the *PlanAugOpt* module for graph G.

bool A.call(*graph*& G, *list*<*edge*>& L)

calls the *PlanAugOpt* module for graph G. Returns the list of added edges in L.

7.3 LEDA's Make Biconnected (LEDAMakeBiconnected)

1. Base Classes



2. Definition

The *LEDAMakeBiconnected* module is a wrapper for LEDA's Make_Biconnected function. It augments a graph G to a biconnected graph G'. If G was planar, the output graph G' is also planar.

Common Informations

Algorithm		**Implementation**	
name	**Make Biconnected**	author	*Stefan Naeher*
long name	**LEDA's Make Biconnected**	date	
author		version	

Pre- and Postcondition

$$\text{precondition} = \emptyset$$
$$\text{postcondition}(PRE) = \{\, biconnected \,\} \cup \Big(\{\, planar, simple, no\ self\ \ loops \,\} \cap PRE \Big)$$

---

3. Creation

LEDAMakeBiconnected A; creates an instance A of type *LEDAMakeBiconnected*.

4. Operations

Generic Interface (Inherited Methods)

The detailed description of these methods can be found in the manual entries of the base class (*AugmentModule*).

bool A.check(*graph* G, *string*& *message*)

bool A.call(*graph*& G)

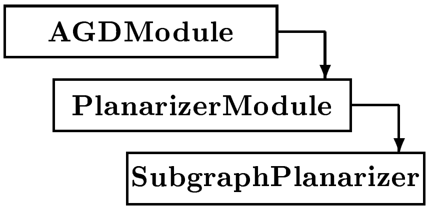
bool A.call(*graph*& G, *list*<*edge*>& L)

Chapter 8

Modules for Planarization

8.1 Planarization with Planar Subgraph (SubgraphPlanarizer)

1. Base Classes



2. Definition

...

Common Informations

Algorithm		**Implementation**	
name	**Subgraph Planarizer**	author	*Carsten Gutwenger*
long name	**Planarization with Planar Subgraph**	date	*august 1997*
author		version	

Pre- and Postcondition

precondition $= \{\, simple, no\ self\ loops \,\}$
postcondition(PRE) $= \{\, planar, simple, no\ self\ loops \,\}$

Exchangeable Modules

Instances of type *SubgraphPlanarizer* provide the following module options:

- **subgraph**
  a *SubgraphModule* option used to compute a planar subgraph of the input graph.

guaranteed precondition:	{ *simple*, *no self loops*, *no reversals* }
required postcondition:	{ *planar*, *simple*, *no self loops*, *no reversals* }
initial module:	*SubgraphPlanarizer*

- **inserter**
  an *EdgeInsertModule* option used to reinsert the deleted edges.

guaranteed precondition:	$\emptyset$
required postcondition:	$\emptyset$
initial module:	*ShortestPathInserter*

---

3. Creation

SubgraphPlanarizer P; creates an instance P of type *SubgraphPlanarizer*.

4. Operations

Generic Interface (Inherited Methods)

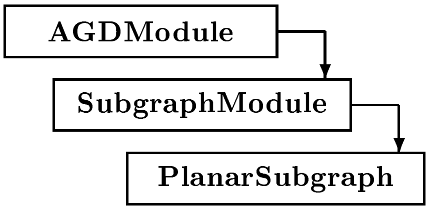
The detailed description of these methods can be found in the manual entries of the base class (*PlanarizerModule*).

bool P.check(*GraphCopy* G, *string*& *message*)

bool P.call(*GraphCopy*& G, *edge_array*<*list*<*edge*> >& *chain*)

8.2 Planar Subgraphs (PlanarSubgraph)

1. Base Classes



2. Definition

The module *PlanarSubgraph* computes a planar subgraph of an input graph G ([8]).

Common Informations

Algorithm		**Implementation**	
name	**Planar Subgraph**	author	*Sebastian Leipert*
long name	**Planar Subgraph**	date	
author		version	*novembre 1997*

Pre- and Postcondition

$$\begin{aligned} \text{precondition} &= \{\,\} \\ \text{postcondition}(PRE) &= \{\, planar \,\} \cup \Big(\{\, simple, no\ self\ loops, no\ reversals \,\} \cap PRE \Big) \end{aligned}$$

---

3. Creation

PlanarSubgraph S; creates an instance S of type *PlanarSubgraph*.

4. Operations

Generic Interface (Inherited Methods)

bool S.check(*graph* G, *string*& *message*)

returns *true* if G satisfies the precondition of S. Otherwise, *false* is returned, and *message* contains the error message.

bool S.call(*graph* G, *list*<*edge*>& L)

calls S for graph G. Returns in L the list of edges $e_1, \ldots, e_k$.

bool S.call(*graph*& G) calls S for graph G, and deletes all edges $e_1, \ldots, e_k$ in G.

bool	S.call(*GraphCopy*& G, *list*<*edge*>& L)	calls S for graph copy G. Deletes all edges $e_1, \ldots, e_k$ in G, and returns in L the list of corresponding edges $original(e_1), \ldots, original(e_k)$ in $original(G)$.
int	S.num_del_edges( )	returns the number k of edges in $E \cap E'$.

8.3 Maximum Planar Subgraphs (MaximumPlanarSubgraph)

ABACUS & CPLEX

1. Base Classes

AGDModule → **SubgraphModule** → **MaximumPlanarSubgraph**

2. Definition

The module *MaximumPlanarSubgraph* computes a maximum planar subgraph of an input graph G ([9]).

Common Informations

Algorithm		**Implementation**	
name	**Maximum Planar Subgraph**	author	*Sergey Fialko*
long name	**Maximum Planar Subgraph**	date	*march 1998*
author	*Petra Mutzel*	version	

Pre- and Postcondition

$$\begin{aligned} \text{precondition} &= \emptyset \\ \text{postcondition}(PRE) &= \{\, planar \,\} \cup \Big(\{\, simple, no\ self\ loops, no\ reversals \,\} \cap PRE \Big) \end{aligned}$$

---

3. Creation

MaximumPlanarSubgraph S; creates an instance S of type *MaximumPlanarSubgraph*.

4. Operations

Generic Interface (Inherited Methods)

bool S.check(*graph* G, *string*& *message*)

returns *true* if G satisfies the precondition of S. Otherwise, *false* is returned, and *message* contains the error message.

bool S.call(*graph* G, *list*<*edge*>& L)

calls S for graph G. Returns in L the list of edges $e_1, \ldots, e_k$.

bool S.call(*graph*& G) calls S for graph G, and deletes all edges $e_1, \ldots, e_k$ in G.

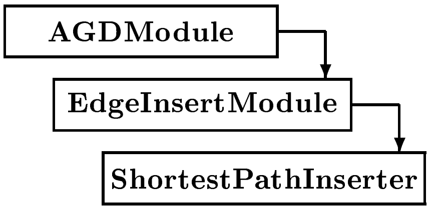
bool S.call(*GraphCopy*& G, *list*<*edge*>& L)

calls S for graph copy G. Deletes all edges $e_1, \ldots, e_k$ in G, and returns in L the list of corresponding edges $original(e_1), \ldots, original(e_k)$ in $original(G)$.

int	S.num_del_edges( )	returns the number k of edges in $E \cap E'$.

8.4 Shortest Path Edge Inserter (ShortestPathInserter)

1. Base Classes



2. Definition

...

Common Informations

Algorithm		**Implementation**	
name	**Shortest Path Inserter**	author	*Thomas Ziegler*
long name	**Shortest Path Edge Inserter**	date	*decembre 1997*
author		version	

Pre- and Postcondition

precondition $= \{ \}$
postcondition(PRE) $= \{ \}$

---

3. Creation

ShortestPathInserter I;

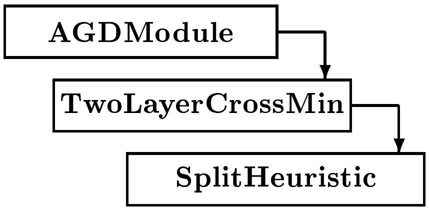
creates an instance I of type *ShortestPathInserter*.

Chapter 9

Two Layer Crossing Minimization

9.1 Split Heuristic (SplitHeuristic)

1. Base Classes



2. Definition

3. Creation

SplitHeuristic M; creates an instance M of type *SplitHeuristic*.

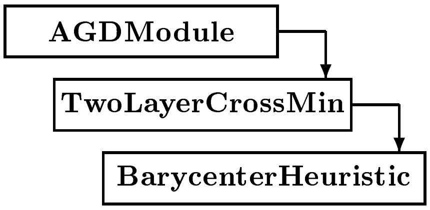
4. Operations

void M.call(*leda_graph* G, *Layer&* L)

calls M for graph G and layer L. Reorders the nodes on layer L.

9.2 Barycenter Heuristic (BarycenterHeuristic)

1. Base Classes



2. Definition

The class *BarycenterHeuristic* implements the two layer crossing minimization heuristic *Barycenter*. It reorders the nodes on the changeable layer according to the barycenter weight [19]

$$Bary(v) = \begin{cases} \dfrac{\sum_{w \in adj(v)} pos(w)}{\mid adj(v) \mid} & \text{if } adj(v) \neq \emptyset \\ 0.0 & \text{otherwise} \end{cases}$$

where v is a node on the changeable layer, $adj(v)$ is the set of neighbors of v on the fixed layer, and $pos(w)$ is the position of w on the fixed layer.

3. Creation

BarycenterHeuristic M; creates an instance M of type *BarycenterHeuristic*.

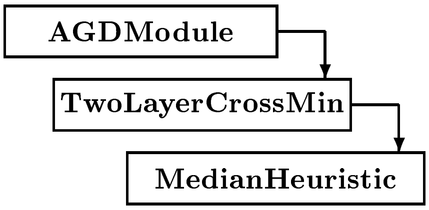
4. Operations

void M.call(*leda_graph* G, *Layer&* L)

calls M for graph G and layer L. Reorders the nodes on layer L.

9.3 Median Heuristic (MedianHeuristic)

1. Base Classes



2. Definition

The class *MedianHeuristic* implements the two layer crossing minimization heuristic *Median*. It reorders the nodes on the changeable layer according to the median weight [3]

$$Median(v) = \begin{cases} \dfrac{pos(w_{\lfloor \frac{n}{2} \rfloor}) + pos(w_{\lceil \frac{n}{2} \rceil})}{2} & \text{if } n \geq 1 \\ 0.0 & \text{otherwise} \end{cases}$$

where v is a node on the changeable layer, and $v_1, \ldots, v_n$ is the sequence of neighbors of v on the fixed layer ordered according to the position pos.

3. Creation

MedianHeuristic M; creates an instance M of type *MedianHeuristic*.

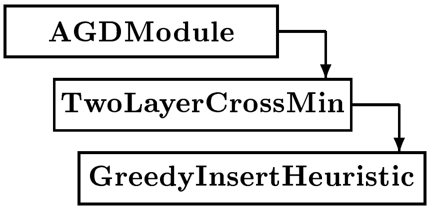
4. Operations

void M.call(*leda_graph* G, *Layer*& L)

calls M for graph G and layer L. Reorders the nodes on layer L.

9.4 Greedy Insert (GreedyInsertHeuristic)

1. Base Classes



2. Definition

3. Creation

GreedyInsertHeuristic M; creates an instance M of type *GreedyInsertHeuristic*.

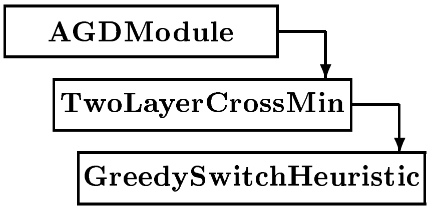
4. Operations

void M.call(*leda_graph* G, *Layer*& L)

calls M for graph G and layer L. Reorders the nodes on layer L.

9.5 Greedy Switch (GreedySwitchHeuristic)

1. Base Classes



2. Definition

3. Creation

GreedySwitchHeuristic M; creates an instance M of type *GreedySwitchHeuristic*.

4. Operations

void M.call(*leda_graph* G, *Layer*& L)

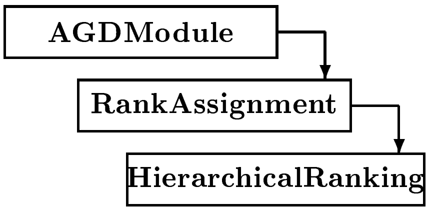
calls M for graph G and layer L. Reorders the nodes on layer L.

Chapter 10

Other Modules

10.1 Hierarchical Ranking (HierarchicalRanking)

1. Base Classes



2. Definition

The *HierarchicalRanking* module computes the following ranking of a graph $G = (V, E)$: First, an acyclic graph $G = (V, E')$ is computed such that G and G' have the same underlying undirected graph. If G is acyclic, then G' equals G. Then, we compute a ranking satisfying

- If v, w are two nodes in G' with v is a source, w is a sink, and there is a path from v to w in G', then exists also a path $v = u_1, \ldots, u_p = w$ with $rank(u_{i+1}) = 1 + rank(u_i)$ for $1 \leq i < p$.

- $\min\{rank(v) \mid v \in V\} = 0$

Optional Parameters

Instances of *HierarchicalRanking* provide the following optional parameters:

- *bool* **separate_deg0_layer** = *true*
  a boolean value determining if the nodes with degree 0 are placed on a separate layer, i.e., they get the highest rank.

3. Creation

HierarchicalRanking $R(bool\ sdl\ =\ true)$;

creates an instance R of type *HierarchicalRanking*. Sets the option *separate_deg0_layer* to *sdl*.

4. Operations

void R.call(*leda_graph* G, *leda_node_array*<*int*>& *rank*)

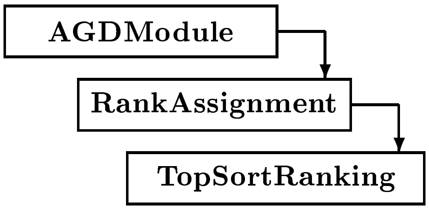
calls R for graph G. Returns the rank for each node $v \in G$ in $rank[v]$.

5. Implementation

The computation of the ranking takes time $O(|V| + |E|)$.

10.2 Ranking by Topological Sorting (TopSortRanking)

1. Base Classes



2. Definition

The *TopSortRanking* module computes the following ranking of a graph $G = (V, E)$: First, an acyclic graph $G = (V, E')$ is computed such that G and G' have the same underlying undirected graph. If G is acyclic, then G' equals G. We set

$$rank(v) = \begin{cases} 1 + \max\{rank(w) \mid w \in pred(v)\} & \text{if } pred(v) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

where $pred(v) = \{w \in V \mid (w, v) \in E'\}$.

3. Creation

TopSortRanking R; creates an instance R of type *TopSortRanking*.

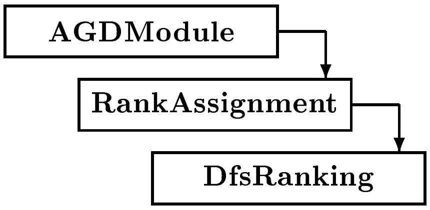
4. Operations

void *R*.call(*leda_graph* G, *leda_node_array*<*int*>& *rank*)

calls R for graph G. Returns the rank for each node $v \in G$ in $rank[v]$.

10.3 Ranking by Depth First Search (DfsRanking)

1. Base Classes



2. Definition

The *DfsRanking* module computes the following ranking of a graph G: We build a DFS forest T of the underlying undirected graph of G and set

$$rank(v) = \begin{cases} 0 & \text{if } v \text{ is a root of } T \\ rank(pred(v)) + 1 & \text{otherwise} \end{cases}$$

where $pred(v)$ is the predecessor of v in T.

3. Creation

DfsRanking R; creates an instance R of type *DfsRanking*.

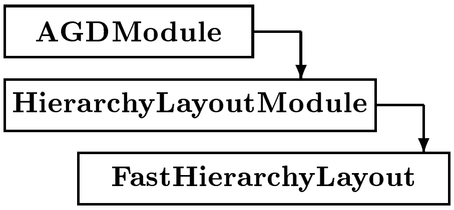
4. Operations

void R.call(*leda_graph* G, *leda_node_array*<*int*>& *rank*)

calls R for graph G. Returns the rank for each node $v \in G$ in $rank[v]$.

10.4 Fast Layout of Hierarchies (FastHierarchyLayout)

1. Base Classes



2. Definition

The class *FastHierarchyLayout* represents a simple layout algorithm for hierarchies with given order of nodes on each layer. Thus it can be used as a third phase for the Sugiyama algorithm.

In the layout, all edges will have at most two bends. Additionally, for each edge having exactly two bends, the section between them is drawn vertically. This applies in particular to the long edges arising in the first phase of the Sugiyama algorithm.

Common Informations

Algorithm		**Implementation**	
name	**Phase 3**	author	*Christoph Buchheim*
long name	**Sugiyama Algorithm, Phase 3**	date	*march 1998*
author		version	

Pre- and Postcondition

$$\text{precondition} = \emptyset$$
$$\text{postcondition}(PRE) = \emptyset$$

Optional Parameters

Instances of *FastHierarchyLayout* provide the following optional parameters:

- *double* **node_distance**
  determines the minimal horizontal distance between two nodes on the same layer.
- *double* **layer_distance**
  determines the minimal vertical distance between two nodes on neighbored layers.
- *bool* **fixed_layer_distance**
  if set to *true*, the distance between neighbored layers is fixed.

---

3. Creation

FastHierarchyLayout L; creates an instance L of type *FastHierarchyLayout*.

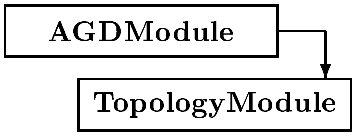
4. Operations

void L.call(*GraphCopy*, *Hierarchy*, *LayoutInterface*&)

calls the *FastHierarchyLayout* module for graph GC and hierarchy H. Assigns the layout to A.

10.5 Topology Modules (TopologyModule)

1. Base Classes



2. Definition

The purpose of class *TopologyModule* is to relate the geometrical and topological structure of a graph. An instance of type *TopologyModule* transports the information from the actual drawing in the combinatorial structure of the underlying graph G. The output is a *planar representation* for G, i.e., a combinatorial embedding defining the faces and an external face f_0.

Input and Output Parameters

input parameter:	graph	$in\_graph = 1$
	layout	$in\_layout = 2$
output parameter:	planar representation	$out\_plan\_rep = 1$

The integer constants *in_graph*, *in_layout*, and *out_plan_rep* denote the positions in the input and output parameter lists. The values can be used to reference the desired parameter when setting the pre– and postcondition.

3. Initialization

TopologyModule T; initializes a topology module.

4. Operations

Standard Interface

bool *T*.check(*leda_graph G*, *leda_string& message*)

returns *true* if G satisfies the precondition of T. Otherwise, *false* is returned, and *message* contains the error message.

leda_list<leda_edge> *T*.call(*leda_graph& G*, *leda_face& f_0*, *LayoutInterface& L*)

calls the topology module for graph G and layout interface L. The method changes the order of edges in G such that the topological structure of the drawing is reflected. If the imposed combinatorial embedding is proper, the parameter f_0 contains the external face, otherwise *NULL*.

leda_list<leda_edge> *T*.call(*leda_graph& G*, *LayoutInterface& L*)

calls the topology module but only changes the order of edges in G according to the layout given by L. The external face is not determined.

Example

An example program showing how to use a `TopologyModule` in combination with a planar layout algorithm can be found in Section 5.9 (p. 73).

Chapter 11

Graph Generators

void random_tric_graph(*leda_graph*& G, *int* n, *int* m)

creates a random triconnected planar graph G with n nodes and m edges. The number of edges can differ slightly from m. Note: For a triconnected planar graph with n nodes ($n \geq 4$) and m edges always holds $\left\lceil \frac{3}{2}n \right\rceil \leq m \leq 3n - 6$.

void random_hierarchy(*leda_graph*& G, *leda_node_array*<*int*>& *rank*, *leda_node_array*<*int*>& *pos*, *int* n, *int* m, *bool planar* = *false*, *bool singleSource* = *false*)

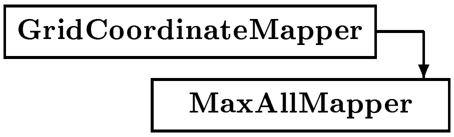
creates a random hierarchical graph G with n nodes and m edges. Assigns for each node v the layer of v to $rank[v]$ and the position of v on the layer to $pos[v]$. The hierarchy is planar, if *planar* is set to *true*. It contains a single source, if *singleSource* is set to *true*.

Chapter 12

Grid Coordinate Mapper

12.1 The Grid Coordinate Mapper MaxAll (MaxAllMapper)

1. Base Classes



2. Definition

The MaxAll mapper maps the grid coordinates such that each row and each column has the same size $size$. This is given by

$$size = \max\{width(v), height(v) \mid v \in V\}$$

where $G = (V, E)$ is the given graph, and $width(v)$, $height(v)$ are the width and the height of the node graphics, respectively.

Rows and columns are separated by an additional distance $separation$. Furthermore, a left margin $left\_margin$ and a right margin $right\_margin$ can be set. The mapping is then defined as

$$\varphi_x(column) = (size + separation) * (column - xmin) + left\_margin + \frac{size}{2}$$
$$\varphi_y(row) = (size + separation) * (ymax - row) + upper\_margin + \frac{size}{2}$$

The size of the node graphics is not changed, that means

$$\left.\begin{array}{rcl} width'(v) &=& width(v) \\ height'(v) &=& height(v) \end{array}\right\} \text{for all } v \in V$$

3. Creation

MaxAllMapper $M(double\ sep,\ double\ lm,\ double\ um);$

creates an instance M of type *MaxAllMapper*, and sets $separation$ to sep, $left\_margin$ to lm, and $upper\_margin$ to um.

4. Operations

a) Initialization

void	M.init( )	computes the mapping for the grid layout given by the graph G, the node coordinates *xpos*, *ypos*, the bendpoints *bends* of the edge polylines, the layout interface a, and the bounding box of the grid layout *grid_bbox*.

b) Mapping of coordinates

double	M.map_x(*int column*)	returns $\varphi_x(column)$.
double	M.map_y(*int row*)	returns $\varphi_y(row)$.
void	M.get_layout_size(*DRect& bbox*)	returns the bounding box of the mapped layout.

c) Further operations

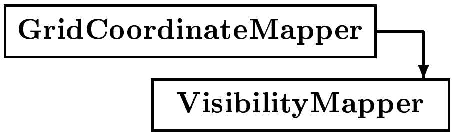
GridCoordinateMapper $*$	M.clone( )	returns a pointer to a new *MaxAllMapper* object that behaves like M, i.e. *clone* only copies options, not computed informations.
double	M.separation( )	returns the current setting of *separation*.
void	M.separation(*double sep*)	sets *separation* to *sep*.
double	M.left_margin( )	returns the current setting of *left_margin*.
void	M.left_margin(*double lm*)	sets *left_margin* to *lm*.
double	M.upper_margin( )	returns the current setting of *upper_margin*.
void	M.upper_margin(*double um*)	sets *upper_margin* to *um*.

5. Implementation

The initialization of the mapping (*init*) takes time $O(n)$ where n is the number of nodes of the given graph. All other operations take constant time.

12.2 The Grid Coordinate Mapper Visibility (VisibilityMapper)

1. Base Classes



2. Definition

The Visibility mapper is a grid coordinate mapper that can be used for visibility representations.

A visibility representation represents each node v as a horizontal segment. The visibility mapper chooses the width of v such that this segment overlaps all horizontal edge segments outgoing from v.

We define the length $len(v)$ of a node v as follows:

$$len(v) = \max Segment(v) - \min Segment(v) + 1$$

where the set $Segment$(v) is defined as

$$\begin{aligned} Segment(v) = \{column(v)\} \quad & \cup \quad \{col_2^e \mid e = (v,w) \text{ and } row_1^e = row_2^e\} \\ & \cup \quad \left\{col_{k(e)-1}^e \mid e = (w,v) \text{ and } row_{k(e)}^e = row_{k(e)-1}^e\right\} \end{aligned}$$

The mapping is done in such a way, that each column has size $size_x$, and each row has size $size_y$. These are given by

$$\begin{aligned} size_x &= \max\left\{ \frac{width(v) - (len(v)-1) * sep_x}{len(v)} \,\middle|\, v \in V \right\} \cup \{min\_size_x\} \\ size_y &= \max\{height(v) \mid v \in V\} \end{aligned}$$

We can now define the mapping by

$$\varphi_x(column) = (size_x + sep_x) * (column - xmin) + left\_margin + \frac{size_x}{2}$$

$$\varphi_y(row) = (size_y + sep_y) * (ymax - row) + upper\_margin + \frac{size_y}{2}$$

Since the mapper changes the width of nodes, the x-coordinate must be adjusted:

$$\begin{aligned} x(v) &= \widetilde{\varphi}_x\left(\frac{\min Segment(v) + \max Segment(v)}{2}\right) \\ width'(v) &= (len(v)-1) * sep_x + len(v) * size_x \end{aligned}$$

where $\widetilde{\varphi}_x$ is the extension of φ_x to $\mathbb{R} \to \mathbb{R}$.

3. Creation

VisibilityMapper M(*double sepx*, *double sepy*, *double msx*, *double lm*, *double um*);

creates an instance M of type *VisibilityMapper*, and sets sep_x to *sepx*, sep_y to *sepy*, min_size_x to *msx*, *left_margin* to *lm*, and *upper_margin* to *um*.

4. Operations

a) Initialization

void	M.init( )	computes the mapping for the grid layout given by the graph G, the node coordinates *xpos*, *ypos*, the bend-points *bends* of the edge polylines, the layout interface a, and the bounding box of the grid layout *grid_bbox*.

b) Mapping of coordinates

double	M.map_x(*int column*)	returns $\varphi_x(column)$.
double	M.map_y(*int row*)	returns $\varphi_y(row)$.
DPoint	M.map_node(*leda_node v*)	returns $\varphi(ip)$.
void	M.map_edge(*leda_edge e*, *DPolyline& points*, *DPoint& s_a*, *DPoint& t_a*)	
void	M.get_layout_size(*DRect& bbox*)	returns the bounding box of the mapped layout.

c) Further operations

void	M.cleanup( )	frees resources allocated by *init*.
GridCoordinateMapper ∗	M.clone( )	returns a pointer to a new *VisibilityMapper* object that behaves like M, i.e. *clone* only copies options, not computed informations.
double	M.sep_x( )	returns the current setting of sep_x.
void	M.sep_x(*double sep*)	sets sep_x to *sep*.
double	M.sep_y( )	returns the current setting of sep_y.
void	M.sep_y(*double sep*)	sets sep_y to *sep*.
double	M.min_size_x( )	returns the current setting of min_size_x.

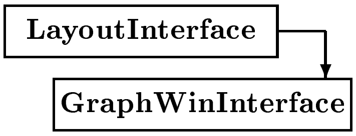
void	*M*.min_size_x(*double msx*)	
		sets min_size_x to *msx*.
double	*M*.left_margin( )	returns the current setting of *left_margin*.
void	*M*.left_margin(*double lm*)	
		sets *left_margin* to *lm*.
double	*M*.upper_margin( )	returns the current setting of *upper_margin*.
void	*M*.upper_margin(*double um*)	
		sets *upper_margin* to *um*.

Chapter 13

Layout Interfaces

13.1 Interface to GraphWin (GraphWinInterface)

1. Base Classes



2. Definition

An instance A of type *GraphWinInterface* is the interface to the graphical attributes of a *GraphWin* object.

3. Creation

GraphWinInterface $A(GraphWin\&\ GW)$;

creates an instance A of type *GraphWinInterface* initialized to an interface to GW.

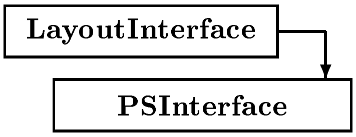
4. Operations

double	A.get_x(*leda_node v*)	returns the x-coordinate of the position of node v.
void	A.set_x(*leda_node v*, *double new_x*)	sets the x-coordinate of the position of node v.
double	A.get_y(*leda_node v*)	returns the y-coordinate of the position of node v.
void	A.set_y(*leda_node v*, *double new_y*)	sets the y-coordinate of the position of node v.
DPoint	A.get_position(*leda_node v*)	returns the position of node v.

void	A.set_position(*leda_node* v, *DPoint* p)	
		sets the position of node v.
DPolyline	A.get_bend_points(*leda_edge* e)	
		returns the bend points of edge e.
void	A.set_bend_points(*leda_edge* e, *DPolyline* l)	
		sets the bend points of edge e.
double	A.get_width(*leda_node* v)	
		returns the width of the graphics of node v.
void	A.set_width(*leda_node* v, *double* *new_w*)	
		sets the width of the graphics of node v.
double	A.get_height(*leda_node* v)	
		returns the height of the graphics of node v.
void	A.set_height(*leda_node* v, *double* *new_h*)	
		sets the height of the graphics of node v.
DPoint	A.get_source_anchor(*leda_edge* e)	
		returns the source anchor position of edge e.
void	A.set_source_anchor(*leda_edge* e, *DPoint* *pos*)	
		sets the source anchor position of edge e to *pos*.
DPoint	A.get_target_anchor(*leda_edge* e)	
		returns the target anchor position of edge e.
void	A.set_target_anchor(*leda_edge* e, *DPoint* *pos*)	
		sets the target anchor position of edge e to *pos*.

13.2 Interface to PostScript (PSInterface)

1. Base Classes



2. Definition

An instance A of type *PSInterface* is the interface to the graphical attributes in a PostScript^{TM} file. When a layout algorithm is called with an instance of *PSInterface* as a parameter, then the drawing is saved to a file in PostScript^{TM} format. The file name can be specified in the constructor of *PSInterface*. The generated file can be used as Encapsulated PostScript^{TM}, too.

3. Creation

PSInterface A(*leda_graph G*, *leda_string name* = "*unnamed.ps*");

creates an instance A of type *PSInterface* initialized to an interface to *ps*.

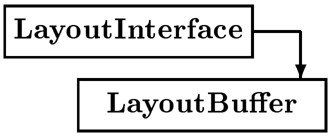
4. Operations

double	A.get_x(*leda_node v*)	returns the x-coordinate of the position of node v.
void	A.set_x(*leda_node v*, *double new_x*)	sets the x-coordinate of the position of node v.
double	A.get_y(*leda_node v*)	returns the y-coordinate of the position of node v.
void	A.set_y(*leda_node v*, *double new_y*)	sets the y-coordinate of the position of node v.
DPolyline	A.get_bend_points(*leda_edge e*)	returns the bend points of edge e.
void	A.set_bend_points(*leda_edge e*, *DPolyline l*)	sets the bend points of edge e.
double	A.get_width(*leda_node v*)	returns the width of the graphics of node v.
void	A.set_width(*leda_node v*, *double new_w*)	sets the width of the graphics of node v.
double	A.get_height(*leda_node v*)	returns the height of the graphics of node v.

void	*A*.set_height(*leda_node v*, *double new_h*)	
		sets the height of the graphics of node v.
DPoint	*A*.get_source_anchor(*leda_edge e*)	
		returns the source anchor position of edge e.
void	*A*.set_source_anchor(*leda_edge e*, *DPoint pos*)	
		sets the source anchor position of edge e to *pos*.
DPoint	*A*.get_target_anchor(*leda_edge e*)	
		returns the target anchor position of edge e.
void	*A*.set_target_anchor(*leda_edge e*, *DPoint pos*)	
		sets the target anchor position of edge e to *pos*.

13.3 Layout Buffers (LayoutBuffer)

1. Base Classes



2. Definition

A layout buffer for graph G is a layout interface that stores the following graphical attributes for graph G:

- the position of each node.
- the bend points of each edge.
- the source and target anchor of each edge.

The class *LayoutBuffer* overrides the respective pure virtual methods of *LayoutInterface* for setting and getting these attributes. The methods set_width, get_width, set_height, and get_height must be overridden by derived classes.

3. Initialization

LayoutBuffer A(*leda_graph* G); initializes a layout buffer for graph G.

4. Operations

double	*A*.get_x(*leda_node* v)	returns the x-coordinate of the position of node v.
void	*A*.set_x(*leda_node* v, *double* *new_x*)	sets the x-coordinate of the position of node v to *new_x*.
double	*A*.get_y(*leda_node* v)	returns the y-coordinate of the position of node v.
void	*A*.set_y(*leda_node* v, *double* *new_y*)	sets the y-coordinate of the position of node v to *new_y*.
DPoint	*A*.get_position(*leda_node* v)	returns the position of node v.
void	*A*.set_position(*leda_node* v, *DPoint* *pos*)	sets the position of node v to *pos*.
DPolyline	*A*.get_bend_points(*leda_edge* e)	returns the bend points of edge e.

void A.set_bend_points(*leda_edge e*, *DPolyline l*)
sets the bend points of edge e to l.

DPoint A.get_source_anchor(*leda_edge e*)
returns the source anchor position of edge e.

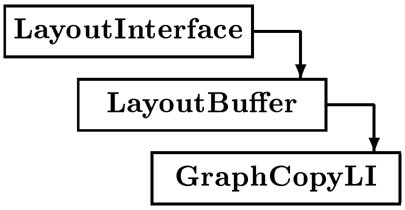
void A.set_source_anchor(*leda_edge e*, *DPoint pos*)
sets the source anchor position of edge e to pos.

DPoint A.get_target_anchor(*leda_edge e*)
returns the target anchor position of edge e.

void A.set_target_anchor(*leda_edge e*, *DPoint pos*)
sets the target anchor position of edge e to pos.

13.4 Layout Interface for Graph Copies (GraphCopyLI)

1. Base Classes



2. Definition

An instance A of type *GraphCopyLI* is a temporary buffer for storing graphical attributes of a planarized graph PG that is a copy of a graph G. The width and height of each node are accessed through a layout interface A for G. The update method assigns the computed layout to A by transforming dummy nodes in PG into bend points.

3. Creation

GraphCopyLI A(*GraphCopy*& GC, *LayoutInterface*& A);

creates an instance A of type *GraphCopyLI* initialized to a layout buffer for planarized graph PG with layout interface A.

4. Operations

double A.get_width(*leda_node* v)

returns the width of the graphics of node $original(v)$ if $original(v) \neq nil$, 0 otherwise.

void A.set_width(*leda_node* v, *double* *new_w*)

sets the width of the graphics of node $original(v)$ to *new_w* if $original(v) \neq nil$.

double A.get_height(*leda_node* v)

returns the height of the graphics of node $original(v)$ if $original(v) \neq nil$, 0 otherwise.

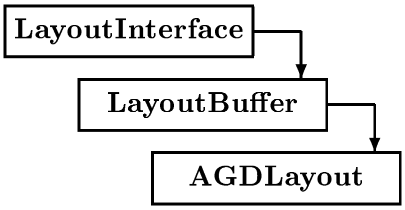
void A.set_height(*leda_node* v, *double* *new_h*)

sets the height of the graphics of node $original(v)$ to *new_h* *new_w* if $original(v) \neq nil$..

void A.update( ) assigns the stored layout to A and calls the update method of A, where A is the layout interface for the original graph G.

13.5 Layouts (AGDLayout)

1. Base Classes



2. Definition

An instance A of type *AGDLayout* is a layout interface that stores the graphical attributes of a particular graph G. It can be used, e.g., if the layout of G shall not be visualized immediately.

3. Creation

AGDLayout A(*leda_graph* G); creates an instances A of type *AGDLayout* initialized to a layout for graph G.

4. Operations

double A.get_width(*leda_node* v)
returns the width of the graphics of node v.

void A.set_width(*leda_node* v, *double* *new_w*)
sets the width of the graphics of node v to *new_w*.

double A.get_height(*leda_node* v)
returns the height of the graphics of node v.

void A.set_height(*leda_node* v, *double* *new_h*)
sets the height of the graphics of node v to *new_h*.

Chapter 14

Canonical Orderings

14.1 Sets of a Canonical Ordering (OrderSet)

1. Definition

An instance V of type *OrderSet* represents one set of a canonical ordering C of a graph $G = (V, E)$. It contains a set $\{v_, \ldots, v_p\}$ of nodes of G ordered from left to right, a leftvertex c_ℓ, a rightvertex c_r, and additional information *has_left* and *has_right*, where $has\_left = true \iff (v_1, c_\ell) \in E$, and $has\_right = true \iff (v_p, c_r) \in E$.

2. Creation

OrderSet V;	creates an instance V of type *OrderSet* representing an empty set.

OrderSet V(*int* l, *leda_edge* el = *nil*, *leda_edge* er = *nil*);

creates an instance V of type *OrderSet* representing a set of length l with left edge el and right edge er.

3. Operations

leda_node	V.left( )	returns the leftvertex c_ℓ of V.
leda_node	V.right( )	returns the rightvertex c_r of V.
leda_edge	V.left_edge( )	returns the edge (z_1, c_ℓ) or *nil*.
leda_edge	V.right_edge( )	returns the edge (z_p, c_r) or *nil*.
bool	V.has_left( )	returns the information *has_left* of V.
bool	V.has_right( )	returns the information *has_right* of V.
void	V.left(*leda_node* cl)	sets the leftvertex of V to cl.
void	V.right(*leda_node* cr)	sets the rightvertex of V to cr.

void	V.left_edge(*leda_edge el*)	sets the edge (z_1, c_ℓ).
void	V.right_edge(*leda_edge er*)	sets the edge (z_p, c_r).
int	V.len()	returns the length of V, i.e. the number of nodes in V.
leda_node&	V[*const int i*]	returns the i-th node of V from left (the leftmost node has index 1).

4. Implementation

Order sets are implemented by Arrays. All operations take constant time.

14.2 The Canonical Ordering of a Graph (CanonicalOrder)

1. Definition

An instance C of type *CanonicalOrder* ia canonical ordering of a plane graph $G = (V, E)$. It represents an ordered partition $V_1, \ldots, V_m$ of V. We provide a canonical ordering for triconnected planar graphs (Goos Kant), and another one for biconnected planar graphs:

Definition (canonical ordering for triconnected planar graphs): Let $G = (V, E)$ be a triconnected plane graph with $|V| \geq 3$, outerface F_{out}, and $v_1, \ldots, v_q$ a path on F_{out} with $q < |F_{out}|$. Let $C = V_1, \ldots, V_m$ be an ordered partition of V, that is, $V_1 \cup \ldots \cup V_m = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. Define G_k to be the subgraph of G induced by $V_1 \cup \ldots \cup V_k$, and denote by C_k the outerface of G_k. C is a **canonical ordering** (for triconnected planar graphs) of G with base $v_1, \ldots, v_q$ if:

- $V_1 = \{v_1, \ldots, v_q\}$
- $V_m = \{z_0\}$, where z_0 lies on the outerface of G.
- Each C_k $(k > 1)$ is a cycle containing $v_1, \ldots, v_q$.
- Each G_k is biconnected and internally triconnected, that is, removing two internal vertices of G_k does not disconnect it.
- For each k in $\{2, \ldots, m-1\}$, one of the following conditions holds:
  (a) $V_k = \{z\}$, where z belongs to C_k and has at least one neighbour in $G \setminus G_k$.
  (b) V_k is a chain, $(z_1, \ldots, z_p)$, where each z_i has at least one neighbour in $G \setminus G_k$, and where z_1 and z_p each have one neighbour on C_{k-1}, and these are the only two neighbours of V_k in G_{k-1}.

This is the definition of the canonical ordering for triconnected planar graphs by Goos Kant extended by the possibility to have more than two nodes in V_1.

Definition (canonical ordering for biconnected planar graphs): Let $G = (V, E)$ be a biconnected plane graph with $|V| \geq 2$, outerface F_{out} and $v_1, \ldots, v_q$ a path on F_{out} with $(v_i, v_j) \notin E$ for all $j \neq i+1$. Let $C = (V_1, \ldots, V_m)$ be an ordered partition of V, that is, $V_1 \cup \ldots \cup V_m = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. Define G_k to be the subgraph of G induced by $V_1 \cup V_2 \ldots \cup V_k$, and denote by C_k the outerface of G_k. C is a **canonical ordering** (for biconnected planar graphs) of G with base $v_1, \ldots, v_q$ if:

- $V_1 = \{v_1, \ldots, v_q\}$
- Each C_k $(k > 1)$ is a cycle containing $v_1, \ldots, v_q$.
- Each G_k is connected and internally biconnected, that is, removing one internal vertex of G_k does not disconnect it.
- For each k in $\{2, \ldots, m-1\}$ holds:
  Each $z \in V_k$ belongs to C_k, and one of the following conditions holds, where $C_{k-1} = (v_1 = c_1, \ldots, c_q = v_2)$:
  (1) $V_k = \{z\}$ and z has at least two neighbours in G_{k-1}.
  (2) V_k is a chain $(z_1, \ldots z_p)$ with $p \geq 1$ of vertices with degree ≤ 2 in G_k, and there are two vertices c_ℓ and c_r $(1 \leq \ell < r \leq q)$ on C_{k-1}, and the following three conditions hold:
    (a) For $\ell < i < r$ holds: c_i has no neighbours in $G \setminus G_k$.
    (b) If $(c_\ell, z_1) \in E$, then z_1 immediatelly follows $c_{\ell+1}$ in the counter-clockwise circular ordering around c_ℓ in G.
    (c) If $(z_p, c_r) \in E$, then z_p immediatelly precedes c_{r-1} in the counter-clockwise circular ordering around c_r in G.

The computation of the canonical ordering (call of the constructor) allows the following options:

- **co_type={*co_triconnected*, *co_biconnected* }:**
  selects the type of the canonical ordering.
- **max_outer={*true*, *false* }:**
  determines which face of the graph is used as outerface. Selects the face with maximal degree if *max_outer* is *true*, or the first face in the list of faces otherwise.
- **prefer_nodes={*false*, *true* }:**
  only used for the order type *co_triconnected*. If in a step of the computation of the ordering, it is possible to select a vertex or a chain as next order set, a vertex is selected if *prefer_nodes* is *true*, a chain otherwise.
- **base_ratio:**
  a double value determining the length of the base, that is the number of nodes in V_1. If *base_ratio* is r and the outerface is F, the length of the base is $\min(2, \max(\ell, r|F|))$, where $\ell = \max\{k \mid v_1, \ldots, v_k$ is a path on F_{out} with $(v_i, v_j) \notin E$ for all $j \neq i+1\}$.

2. Creation

CanonicalOrder C(*leda_graph*& G, *CanonicalOrderType co_type* = *co_triconnected*, *leda_face* F = *nil*, *bool prefer_nodes* = *false*, *double base_ratio* = 0.0);

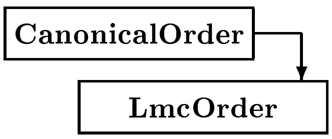
creates an instance C of type *CanonicalOrder* initialized to a canonical ordering for triconnected, planar graphs (*co_triconnected*), or for biconnected, planar graphs (*co_biconnected*). *Precondition*: G is a directed graph representing a planar map.

3. Operations

int	C.length( )	returns the number m of sets in the partition.
int	C.len(*int* i)	returns the length of the i-th set V_i.
leda_node	C(*int* i, *int* j)	returns the j-th node of the i-th set V_i.
const OrderSet&	C[*int* i]	returns a constant reference to the i-th set V_i.
leda_node	C.left(*int* i)	returns the leftvertex of the i-th set V_i.
leda_node	C.right(*int* i)	returns the rightvertex of the i-th set V_i.
int	C.rank(*leda_node* v)	returns the rank of node v, i.e. $rank(v) = i \iff v \in V_i$.

14.3 The Leftmost Canonical Order of a Graph (LmcOrder)

1. Base Classes



2. Definition

An instance *Lmc* of type *LmcOrder* is a leftmost canonical ordering of a graph G. We define a leftmost canonical ordering as follows:

Definition: *A canonical ordering is a leftmost canonical (lmc-)ordering if we can add in any step k a vertex V_k with leftvertex c_l or a vertex set $V_{k'}$ with leftvertex $c_{l'}$, and $l < l'$ holds, then $k < k'$.*

3. Creation

LmcOrder *Lmc*(*leda_graph*& *g*, *CanonicalOrderType co_type* = *co_triconnected*, *leda_face F* = *nil*, *bool prefer_nodes* = *false*, *double base_ratio* = 0.0);

creates an instance *Lmc* of type *LmcOrder* initialized to a leftmost canonical order for G. *Precondition*: G is a directed graph representing a planar map.

4. Implementation

LmcOrder derives from *CanonicalOrder*. The transformation into a leftmost ordering takes time $O(n)$, where n is the number of nodes of G.

Chapter 15

Handling Pre- and Postconditions

15.1 Key Structs (AGD_Key)

1. Definition

An instance K of type AGD_Key is a pair (s, i) of a string s and an index i. We define the type agd_key to be a pointer to an instance of type AGD_Key and call such a pointer a key.

2. Creation

$AGD\_Key \;\; K(leda\_string\ s,\ int\ id);$

creates an instances K of type AGD_Key initialized to the pair (s, id).

3. Operations

leda_string	K.info( )	returns the string s if $K = (s, i)$.
int	K.id( )	returns the index i if $K = (s, i)$.
ostream&	*ostream& os* $\ll$ *agd_key k*	writes s to the output stream os if k points to (s, i).

15.2 Key Mapper (Keymapper)

1. Definition

An instance M of type *Keymapper* is an injective mapping from strings to keys. We use $dom(M)$ to denote the set of used strings, and $M(s)$ to denote the value of M for string s.

2. Creation

Keymapper M;	creates an instances M of type *Keymapper*. Sets $dom(M)$ to the empty set.

3. Operations

int	M.number_of_keys( )	returns the number of defined keys.
agd_key	M.ask_key(*leda_string* s)	returns $M(s)$ if $s \in dom(M)$, *nil* otherwise.
agd_key	M.get_key(*leda_string* s)	returns $M(s)$. Creates a new key and adds s to $dom(M)$ if $s \notin dom(M)$.
leda_string	M.get_string(*agd_key* k)	returns the string s with $M(s) = k$.

Iteration

forall_keys(k, M) { "the keys defined in M are successively assigned to k" }

15.3 Key Sets (KeySet)

1. Definition

An instance S of the data type *KeySet* is a collection of elements of the data type *agd_key* with an associated key mapper.

2. Creation

KeySet S;	creates an instances S of type *KeySet* initialized to the empty set. Sets the associated key mapper to *AGD*'s default key mapper.
KeySet S(*Keymapper*& *mapper*);	creates an instances S of type *KeySet* initialized to the empty set. Sets the associated key mapper to *mapper*.
KeySet S(*KeySet* x);	creates an instances S of type *KeySet* initialized to a copy of x.

3. Operations

a) Common Set Operations

void	S.insert(*agd_key* k)	adds k to S.
void	S.insert(*leda_string* s)	adds <s> to S.
void	S.del(*agd_key* k)	deletes k from S.
void	S.del(*leda_string* s)	deletes <s> from S.
bool	S.member(*agd_key* k)	returns *true* iff $k \in S$.
KeySet	S.join(*KeySet* T)	returns $S \cup T$.
KeySet	S.diff(*KeySet* T)	returns $S \setminus T$.
KeySet	S.intersect(*KeySet* T)	returns $S \cap T$.
KeySet	S.symdiff(*KeySet* T)	returns $(S \cup T) \setminus (S \cap T)$.
bool	S.empty()	returns *true* iff $S = \emptyset$.
void	S.clear()	makes S the empty set.

b) Operators

The following operators return a new key set instead of changing S.

KeySet	$S + T$	returns S.join(T).

KeySet	$S - T$	returns S.diff(T).
KeySet	S & T	returns S.intersect(T).
KeySet	S % T	returns S.symdiff(T).

The following operators manipulate S directly.

KeySet&	S += T	assigns S.join(T) to S and returns S.
KeySet&	S −= T	assigns S.diff(T) to S and returns S.
KeySet&	S &= T	assigns S.intersect(T) to S and returns S.
KeySet&	S %= T	assigns S.symdiff(T) to S and returns S.
KeySet&	$S \ll$ *agd_key* k	inserts k into S and returns S.

Relational and assignment operators:

bool	$S \leq T$	returns *true* iff $S \subseteq T$.
bool	$S \geq T$	returns *true* iff $S \supseteq T$.
KeySet&	$S = T$	assigns T to S. Returns S.

c) Input and Output

void S.read(*istream*& *is*, *int delim* = (*char*)*EOF*)

reads a sequence of strings $s_1, \ldots, s_k$ terminated by *delim* from the input stream *is*, and assigns the set $\{$<s_1>$, \ldots,$ <s_k>$\}$ to S.

void S.read(*char delim* = '\n')

calls S.read(*cin*, *delim*) to read S from the standard input stream *cin*.

void S.print(*ostream*& *os*, *char space* = ' ')

prints the string representations of the elements of S to the output stream *os*. The elements are separated by *space*.

void S.print(*char space* = ' ')

calls S.print(*cout*, *space*) to print S to the standard output stream *cout*.

istream&	*istream& is* $\gg$ *KeySet& S*	reads the key set S from the input stream *is*.
ostream&	*ostream& os* $\ll$ S	writes the key set S to the output stream *os*.

Iteration

forall(k, S) { "the elements of S are successively assigned to k" }

4. Implementation

Key sets are implemented by bit vectors.

15.4 Tagged Rule Literals (tr_literal)

1. Definition

An instance l of the data type *tr_literal* is a pair (k, i) of a key k and an index i. Tagged rule literals are used by tagged rules.

2. Creation

tr_literal l; creates an instances l of type *tr_literal* initialized to $(nil, 1)$.

tr_literal $l(agd\_key\ k)$;

creates an instances l of type *tr_literal* initialized to $(k, 1)$.

tr_literal $l(agd\_key\ k,\ int\ i)$;

creates an instances l of type *tr_literal* initialized to (k, i).

15.5 Tagged Rules (TaggedRule)

1. Definition

An instance r of the data type *TaggedRule* is a rule $(p_1, i_1) \wedge \ldots \wedge (p_k, i_k) \to p$, where (p_j, i_j) $(1 \le j \le k)$ are tagged rule literals, and p is a key. Tagged rules are used by modules to define postcondition rules.

2. Creation

TaggedRule r; creates an instances r of type *TaggedRule* initialized to $\to nil$.

TaggedRule r(*agd_key* p);
creates an instances r of type *TaggedRule* initialized to $\to p$.

TaggedRule r(*leda_list*<*tr_literal*> L, *agd_key* p);
creates an instances r of type *TaggedRule* initialized to $l_1 \wedge \ldots \wedge l_k \to p$ if $L = l_1, \ldots, l_k$.

3. Operations

agd_key	r.right()	returns the key p.
leda_list<*tr_literal*>	r.left()	returns the list of tagged literals $(p_1, i_1), \ldots, (p_k, i_k)$.
int	r.length()	returns the number k of literals on the left side of r.
TaggedRule&	r.and(*agd_key* q, *int* j = 1)	extends r to $(p_1, i_1) \wedge \ldots \wedge (p_k, i_k) \wedge (q, j) \to p$. Returns r.
ostream&	*ostream*& $os \ll r$	writes the tagged rule r to the output stream os.

4. Example

We create a tagged rule $(planar, 1) \wedge (triconnected, 1) \wedge (simple, 1) \wedge (no\_self\_loops, 1) \to convex$ and set it as postcondition rule of a layout algorithm.

```
SampleLayout::SampleLayout() : LayoutModule (true)
{
  TaggedRule r(convex);

  r.and(planar).and(triconnected).and(simple).and(no_self_loops);

  add_post_rule(r);

    ....
}
```

Chapter 16

Hierarchies

16.1 Layers of Hierarchies (Layer)

1. Definition

An instance L of type *Layer* is an ordered collection $v_0, \ldots, v_{m-1}$ of nodes with an associated node hierarchy H (see 16.2) to which L belongs.

The changeable and the fixed layer of H are called *neighbored*.

2. Creation

Layer L(*Hierarchy*& H, *int* i, *leda_list*<*leda_node*> *nodes*);

creates an instance L of type *Layer* initialized to a layer of hierarchy H with nodes *nodes*.

3. Operations

Hierarchy	L.hierarchy( )	returns a reference to the associated node hierarchy.
Layer	L.neighbored_layer( )	returns a reference to the neighbored layer. *Precondition*: L is currently the fixed or changeable layer.
int	L.length( )	returns the number of nodes on L.
leda_node	L[*int* i]	returns the node at position i. *Precondition*: $0 \leq i \leq m-1$.
void	L.swap(*int* i, *int* j)	swaps the ith and the jth nodes on L.

leda_list<*leda_node*> L.adj_nodes(*leda_node* v)

returns a reference to the list of nodes on the neighbored layer that are adjacent to v.

leda_list<*leda_edge*> L.adj_edges(*leda_node* v)

returns a reference to the list of edges leading to the neighbored layer that are adjacent to v.

void	L.quicksort(*leda_node_array*<*double*> *weight*)	sorts the nodes on L according to *weight* using a randomized quicksort. The sort is stable, i.e., if two nodes have the same weight, they keep in order of their previous positions.
void	L.set_pos(*leda_node_map*<*int*> *position*)	orders the nodes on L according to *position*. *Precondition*: $\{position[v] \mid v \in L\} = \{0, \ldots, m-1\}$
void	L.shuffle( )	permutes the nodes on L randomly.

16.2 Node Hierarchies (Hierarchy)

1. Definition

An instance H of type *Hierarchy* is an ordered collection $L_0, \ldots, L_{n-1}$ of layers. The nodes on each layer are numbered $0, 1, \ldots$

We call the index of the layer to which a node v of GC belongs the rank of v. H always represents a proper hierarchy, i.e, rank(v) =rank(w) + 1 holds for every edge (v, w) of GC.

Node hierarchies are used by Sugiyama's layout algorithm, which performs iteratively a two layer crossing minimization between two neighbored layers, the fixed and the changeable layer. The crossing minimization proceeds alternating downward and upward, and reorders the nodes on the changeable layer depending on the positions of the nodes on the fixed layer.

The class *Hierarchy* provides a data type *Direction*, which can be *downward* or *upward*, and maintains the current direction of the crossing minimization and the current changeable layer. If L_i is the changeable layer, then L_{i-1} is the fixed layer if the direction is *downward* (and $i \geq 1$), and L_{i+1} if the direction is *upward* (and $i \leq n-1$).

2. Creation

Hierarchy H(*GraphCopy*& GC, *leda_node_array*<*int*> R);

creates an instance H of type *Hierarchy* initialized to a proper hierarchy of GC. Places node v on layer R[original(v)]. Changes GC by reversing and splitting edges.

3. Operations

int	H.number_of_layers( )	returns the number of layers of H.
int	H.length(*int* i)	returns the number of nodes on layer L_i.
int	H.rank(*leda_node* v)	returns the index of the layer to which v belongs.
int	H.pos(*leda_node* v)	returns the position of v on its layer.
Layer&	H[*int* i]	returns a reference to layer L_i. *Precondition*: $0 \leq i \leq n-1$
Direction	H.direction( )	returns the current direction of crossing minimization.
void	H.direction(*Direction* *dir*)	sets the current direction of crossing minimization to *dir*.
void	H.set_changeable_layer(*int* i)	sets the current changeable layer to L_i.

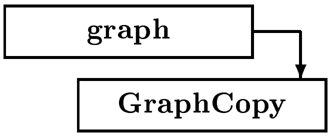
bool	H.is_dummy(*leda_node* v)	returns true iff v is a dummy node. *Precondition*: v is a node of GC.
int	H.calc_crossings(*int* i)	calculates the number of crossings between layer L_i and layer L_{i+1}. *Precondition*: $0 \leq i \leq n-1$
int	H.calc_crossings( )	calculates the total number of crossings.
void	H.store_pos(*leda_node_map*<*int*>& *old_pos*)	stores the current positions of nodes in *old_pos*.
void	H.restore_pos(*leda_node_map*<*int*> *new_pos*)	sets the positions of all nodes v to *new_pos*$[v]$. *Precondition*: $\{new\_pos[v] \mid v \in L_i\} = \{0, \ldots, \text{length}(i)-1\}$ holds for $0 \leq i \leq n-1$.
void	H.shuffle( )	permutes the nodes on each layer randomly.

Chapter 17

Graphs

17.1 Copy of a Graph (GraphCopy)

1. Base Classes



2. Definition

An instance GC of type *GraphCopy* is a copy of an original graph G. For each node v' of GC, *GraphCopy* stores the corresponding node *original*(v') in the original graph G, for each node v in G the newly created copy *copy*(v) in GC, for each edge e' in GC the corresponding edge *original*(e') in G, and for each edge e in G a list of dummy edges *chain*(e) in GC.

For nodes or edges which are created afterwards with new_node or new_edge, the corresponding node or edge is set to *nil*.

3. Creation

GraphCopy GC; — creates an instance GC of type *GraphCopy* initialized to an empty graph without original graph.

GraphCopy GC(*leda_graph* G); — creates an instance GC of type *GraphCopy* initialized to a copy of G.

4. Operations

leda_graph GC.original( ) — returns the original graph G.

leda_node GC.original(*leda_node* v)

returns the node of the original graph G that corresponds to v, *nil* if v was not in G, e.g. v was created afterwards.

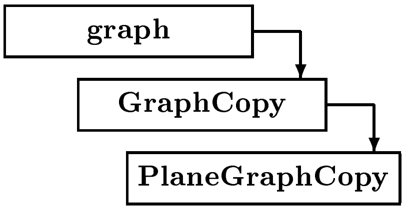
leda_edge	GC.original(*leda_edge* e)	returns the edge of the original graph G that corresponds to e, *nil* if e was not in G, e.g. e was created afterwards.
leda_node	GC.copy(*leda_node* v)	returns $copy(v)$. *Precondition*: $v \in G$.
leda_list<*leda_edge*>	GC.chain(*leda_edge* e)	returns the list of dummy edges for edge e. *Precondition*: e is an edge in G.
bool	GC.is_dummy(*leda_node* v)	returns *true* iff v has a corresponding node in the original graph.
bool	GC.is_dummy(*leda_edge* e)	returns *true* iff e has a corresponding edge in the original graph.
bool	GC.is_reversed(*leda_edge* e)	returns *true* iff $e \in G$ has been reversed in GC.
leda_edge	GC.split(*leda_edge* e)	splits edge $e = (v, w)$ into edges (v, u), (u, w) by introducing a new noe u.Returns the edge (u, w).
bool	GC.is_map( )	virtual method returning *false*.

5. Implementation

All operations for graphs are available for instances of type *GraphCopy*, too. All additional operations take constant time.

17.2 Plane Graph Copies (PlaneGraphCopy)

1. Base Classes



2. Definition

Plane graph copies represent copies of planar graphs or planarized graphs as planar maps.

3. Creation

PlaneGraphCopy PG;

creates an instance PG of type *PlaneGraphCopy* initialized to an empty graph wothout original graph.

PlaneGraphCopy PG(*leda_graph* G);

creates an instance PG of type *PlaneGraphCopy* initialized to a copy of G.

PlaneGraphCopy PG(*leda_graph* G, *leda_list<leda_edge>&* E);

creates an instance PG of type *PlaneGraphCopy* initialized to a copy of G without the edges in E. *Precondition*: Each $e \in E$ is an edge in G.

4. Operations

leda_face PG.copy_of_face(*leda_face* f)

returns a face of PG that corresponds to F. *Precondition*: F is a face in G.

leda_edge PG.split(*leda_edge* e)

split edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges (v, u), (u, w) and their reversals by introducing a new node u. Returns the edge (u, w).

void PG.new_edge_chain(*leda_edge* e, *leda_edge* *from*, *leda_edge* *to*, *leda_list<leda_edge>* *crossings*)

Inserts a chain of edges for edge e into PG. Let $crossings = e_1, \ldots, e_k$ for a $k \geq 0$, $e_{k+1} = to$, and $e = (v, w)$. Then, we split each e_i $(1 \leq i \leq k)$ with a node u_i, and insert the edges $(source(from), u_1)$,(u_i, u_{i+1}),$(u_k, source(to))$ into PG. The inserted edges are appended to $chain(e)$. *Precondition*: e is an edge in G, all other edges are edges in PG, $source(from) = copy(v)$, $source(to) = copy(w)$, $face(from) = face(e_1)$, and $face(reversal(e_i)) = face(e_{i+1})$.

leda_face PG.expand(*leda_node* v)

replaces node v by a face f. Let $v_1, \ldots, v_k$ be the adjacent nodes of v. Then $f = w_1, \ldots, w_k$, where $\deg(w_i) = 3$ and w_i is connected to v_i. Returns f. *Precondition*: $\deg(v) \geq 3$.

leda_node PG.collapse(*leda_face* f, *leda_node* *v_orig*)

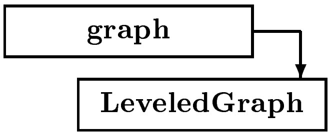
collapses face f to a single node v. Sets the original node of v to *v_orig*. Serves as undo for expand(v). Returns v. *Precondition*: Each node on f has degree 3.

leda_face PG.max_face( ) returns a face of PG with maximal degree.

bool PG.is_map( ) returns *true*.

17.3 Leveled Graphs (LeveledGraph)

1. Base Classes



2. Definition

An instance LG of type *LeveledGraph* is a graph $G = (V, E)$ with an injective mapping $level : V \to \{0, 1, 2, \ldots\}$. We call the set of nodes $L_i := \{v \mid level(v) = i\}$ the ith level of LG. The number of levels of LG is $\max\{i \mid L_i \neq \emptyset\} + 1$ if $V \neq \emptyset$, and 0 otherwise.

3. Creation

LeveledGraph LG;	creates an instance LG of type *LeveledGraph* initialized to an empty graph.

4. Operations

int	LG.number_of_levels( )	returns the number of levels of LG.
int	LG.level(*leda_node* v)	returns the index i of the level L_i to which v belongs.
void	LG.level(*leda_node* v, *int* i)	changes the level to which v belongs to L_i.
leda_list<leda_node>	LG.nodes(*int* i)	returns the list of nodes on level L_i.
leda_node	LG.new_node(*int* i)	adds a new node on level L_i to LG and returns it.

5. Implementation

All operations take expected time $O(1)$.

Chapter 18

Auxiliary Classes

18.1 Grid Layouts (GridLayout)

1. Definition

A grid layout *gl* of a graph *G* stores for each node of *G* an integer x- and y-coordinate, and for each edge of *G* a list of bend points of type *IPolyline*.

2. Creation

GridLayout *gl*; creates an instance *gl* of type *GridLayout* initialized to the empty layout with no associated graph.

GridLayout *gl*(*leda_graph G*);

creates an instance *gl* of type *GridLayout* initialized to a grid layout for graph *G*. All x- and y-coordinates are initially 0, and all bend point lists are empty.

3. Operations

void	*gl*.init( )	reinitializes *gl* to the empty layout with no associated graph.
void	*gl*.init(*leda_graph G*)	reinitializes *gl* to a grid layout for graph *G*. All x- and y-coordinates are set to 0, and all bend point lists are empty.
int&	*gl*.x(*leda_node v*)	returns a reference to the x-coordinate of node *v*.
int&	*gl*.y(*leda_node v*)	returns a reference to the y-coordinate of node *v*.
IPolyline&	*gl*.bends(*leda_edge e*)	returns a reference to the bend point list of edge *e*.
leda_node_map<int>&	*gl*.x( )	returns a reference to the node map storing the x-coordinates of the nodes.
leda_node_map<int>&	*gl*.y( )	returns a reference to the node map storing the y-coordinates of the nodes.

leda_edge_map<*IPolyline*>& *gl*.bends( )		
		returns a reference to the edge map storing the bend point lists of the edges.
IRect	*gl*.area( )	returns a tight bounding box of the grid layout.
int	*gl*.number_of_bends( )	returns the total number of bend points.
IPolyline	*gl*.get_compact_bends(*leda_edge e*)	
		returns the bend point list for edge e without unnecessary bends.
IPolyline	*gl*.get_compact_bends(*GraphCopy GC*, *leda_edge e*)	
		returns the bend point list for edge e of graph copy GC without unnecessary bends. *Precondition*: *gl* is a grid layout for GC and e is an edge in *original*(e).
void	*gl*.compact_all_bends( )	removes all unnecessary bends.

18.2 Integer Points (IPoint)

1. Definition

An instance ip of type $IPoint$ is a point (a, b), where a and b are integers. We call a the x-coordinate of ip and b the y-coordinate of ip.

2. Creation

$IPoint\ \ ip;$	creates an instance ip of type $IPoint$ initialized to the point $(0, 0)$.
$IPoint\ \ ip(int\ x,\ int\ y);$	creates an instance ip of type $IPoint$ initialized to the point (x, y).
$IPoint\ \ ip(IPoint\ p);$	creates an instance ip of type $IPoint$ initialized to the point p.

3. Operations

int	ip.xcoord( )	returns the x-coordinate of ip.
int	ip.ycoord( )	returns the y-coordinate of ip.
void	ip.xcoord($int\ x$)	sets the x-coordinate of ip to x.
void	ip.ycoord($int\ y$)	sets the y-coordinate of ip to y.
IPoint&	ip.inc_x($int\ dx$)	increments the x-coordinate of ip by dx. Returns a reference to ip.
IPoint&	ip.inc_y($int\ dy$)	increments the y-coordinate of ip by dy. Returns a reference to ip.
bool	$ip == p$	Test for equality.
bool	$ip\ != p$	Test for inequality.
IPoint	$ip + ip1$	returns the addition (of coordinates).
IPoint	$ip - ip1$	returns the subtraction (of coordinates).
IPoint&	$ip\ += ip1$	adds $ip1$ to ip. Returns a reference to ip.
IPoint&	$ip\ -= ip1$	subtracts $ip1$ from ip. Returns a reference to ip.
ostream&	$ostream\&\ O \ll ip$	writes ip to output stream O.
istream&	$istream\&\ I \gg IPoint\&\ ip$	reads the coordinates of ip (two *int* numbers) from input stream I.

18.3 Integer Line Segments (ISegment)

1. Definition

An instance *is* of type *ISegment* is a line segment $[p, q]$, where p and q are integer points. We call p the source point and q the target point of the segment.

2. Creation

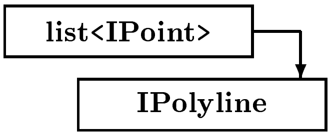
ISegment is;	creates an instance *is* of type *ISegment* initialized to the segment $[(0,0),(0,0)]$.
ISegment is(*IPoint p*, *IPoint q*);	creates an instance *is* of type *ISegment* initialized to the segment $[p, q]$.

3. Operations

IPoint	*is*.source( )	returns a reference to the source point of *is*.
IPoint	*is*.target( )	returns a reference to the target point of *is*.
bool	*is*.horizontal( )	returns *true* iff *is* is horizontal, i.e the y-coordinate of the source point equals the y-coordinate of the target point.
bool	*is*.vertical( )	returns *true* iff *is* is vertical, i.e the x-coordinate of the source point equals the x-coordinate of the target point.

18.4 Integer Polylines (IPolyline)

1. Base Classes



2. Definition

An instance *ipl* of type *IPolyline* is a list $ip_1, \ldots, ip_n$ of integer points ip_i.

3. Creation

IPolyline	*ipl*;	creates an instance *ipl* of type *IPolyline* initialized to an empty list.

4. Operations

IPoint&	*ipl*.get_point(*int* i)	returns a reference to ip_i if *ipl* is $ip_1, \ldots, ip_n$. *Precondition*: $1 \le i \le n$
IPoint&	*ipl*.get_rev_point(*int* i)	returns a reference to ip_{n-i+1} if *ipl* is $ip_1, \ldots, ip_n$. *Precondition*: $1 \le i \le n$
ISegment	*ipl*.get_segment(*int* i)	returns the integer segment $[ip_i, ip_{i+1}]$ if *ipl* is $ip_1, \ldots, ip_n$. *Precondition*: $1 \le i < n$
ISegment	*ipl*.get_rev_segment(*int* i)	returns the integer segment $[ip_{n-i+1}, ip_{n-i}]$ if *ipl* is $ip_1, \ldots, ip_n$. *Precondition*: $1 \le i < n$
void	*ipl*.reverse( )	reverses the list of integer points.

5. Implementation

All operations for *list*<*IPoint*> are also available for *IPolyline*.

18.5 Integer Rectangles (IRect)

1. Definition

An instance *ir* of type *IRect* is a rectangle $(left, right, top, bottom)$ in a grid coordinate system. That means $left \leq right$ and $bottom \leq top$, because positive x is to the right, and positive y is up.

2. Creation

IRect *ir*; creates an instance *ir* of type *IRect* initialized to the rectangle $(0, 0, 0, 0)$.

IRect *ir*(*int xmin*, *int xmax*, *int ymax*, *int ymin*);

creates an instance *ir* of type *IRect* initialized to the rectangle $(xmin, xmax, ymin, ymax)$.

IRect *ir*(*IPoint ll*, *IPoint ur*);

creates an instance *ir* of type *IRect* initialized to the rectangle with lower left corner *ll* and upper right corner *ur*.

3. Operations

IPoint	*ir*.lower_left()	returns the lower left corner of *ir*.
IPoint	*ir*.upper_right()	returns the upper right corner of *ir*.
int&	*ir*.left()	returns the x-coordinate of the lower left corner of *ir*.
int&	*ir*.right()	returns the x-coordinate of the upper right corner of *ir*.
int&	*ir*.top()	returns the y-coordinate of the upper right corner of *ir*.
int&	*ir*.bottom()	returns the y-coordinate of the lower left corner of *ir*.
int	*ir*.width()	returns the width $(= right - left)$ of *ir*.
int	*ir*.height()	returns the height $(= bottom - top)$ of *ir*.

18.6 Double Points (DPoint)

1. Definition

An instance dp of type $DPoint$ is a point (a, b), where a and b are doubles. We call a the x-coordinate of dp and b the y-coordinate of dp.

2. Creation

$DPoint\ \ dp;$ creates an instance dp of type $DPoint$ initialized to the point $(0, 0)$.

$DPoint\ \ dp(double\ x,\ double\ y);$

creates an instance dp of type $DPoint$ initialized to the point (x, y).

$DPoint\ \ dp(DPoint\ p);$

creates an instance dp of type $DPoint$ initialized to the point dp.

3. Operations

double	dp.xcoord( )	returns the x-coordinate of dp.
double	dp.ycoord( )	returns the y-coordinate of dp.
void	dp.xcoord($double\ x$)	sets the x-coordinate of dp to x.
void	dp.ycoord($double\ y$)	sets the y-coordinate of dp to y.
DPoint&	dp.inc_x($double\ dx$)	increments the x-coordinate of dp by dx. Returns a reference to dp.
DPoint&	dp.inc_y($double\ dy$)	increments the y-coordinate of dp by dy. Returns a reference to dp.
bool	$dp == p$	Test for equality.
bool	$dp\ != p$	Test for inequality.
DPoint	$dp + dp1$	returns the addition (of coordinates).
DPoint	$dp - dp1$	returns the subtraction (of coordinates).
DPoint&	$dp += dp1$	adds $dp1$ to dp. Returns a reference to dp.
DPoint&	$dp -= dp1$	subtracts $dp1$ from dp. Returns a reference to dp.
ostream&	$ostream\&\ O \ll dp$	writes dp to output stream O.

istream& $\quad$ *istream*& $I \gg DPoint$& *dp*

reads the coordinates of *dp* (two *double* numbers) from input stream I.

18.7 Double Line Segments (DSegment)

1. Definition

An instance ds of type *DSegment* is a line segment $[p, q]$, where p and q are double points. We call p the source point and q the target point of the segment.

2. Creation

DSegment ds; creates an instance ds of type *DSegment* initialized to the segment $[(0,0),(0,0)]$.

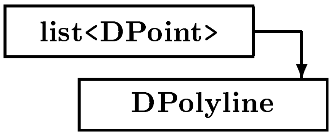
DSegment ds(*DPoint* p, *DPoint* q);
creates an instance ds of type *DSegment* initialized to the segment $[p, q]$.

3. Operations

DPoint	ds.source( )	returns a reference to the source point of ds.
DPoint	ds.target( )	returns a reference to the target point of ds.
bool	ds.horizontal( )	returns *true* iff ds is horizontal, i.e the y-coordinate of the source point equals the y-coordinate of the target point.
bool	ds.vertical( )	returns *true* iff ds is vertical, i.e the x-coordinate of the source point equals the x-coordinate of the target point.

18.8 Double Polylines (DPolyline)

1. Base Classes



2. Definition

An instance *dpl* of type *DPolyline* is a list $dp_1, \ldots, dp_n$ of double points dp_i.

3. Creation

DPolyline dpl; creates an instance *dpl* of type *DPolyline* initialized to an empty list.

4. Operations

DPoint&	*dpl*.get_point(*int i*)	returns a reference to dp_i if *dpl* is $dp_1, \ldots, dp_n$. *Precondition*: $1 \le i \le n$
DPoint&	*dpl*.get_rev_point(*int i*)	returns a reference to dp_{n-i+1} if *dpl* is $dp_1, \ldots, dp_n$. *Precondition*: $1 \le i \le n$
DSegment	*dpl*.get_segment(*int i*)	returns the double segment $[dp_i, dp_{i+1}]$ if *dpl* is $dp_1, \ldots, dp_n$. *Precondition*: $1 \le i < n$
DSegment	*dpl*.get_rev_segment(*int i*)	returns the double segment $[dp_{n-i+1}, dp_{n-i}]$ if *dpl* is $dp_1, \ldots, dp_n$. *Precondition*: $1 \le i < n$
void	*dpl*.reverse( )	reverses the list of double points.

5. Implementation

All operations for *list<DPoint>* are also available for *DPolyline*.

18.9 Double Rectangles (DRect)

1. Definition

An instance *dr* of type *DRect* is a rectangle $(left, right, top, bottom)$ in the standard coordinate system. That means $left \leq right$ and $top \leq bottom$, because positive x is to the right, and positive y is down.

2. Creation

DRect dr; creates an instance *dr* of type *DRect* initialized to the rectangle $(0, 0, 0, 0)$.

DRect dr(double xmin, double xmax, double ymin, double ymax);

creates an instance *dr* of type *DRect* initialized to the rectangle $(xmin, xmax, ymin, ymax)$.

DRect dr(DPoint ll, DPoint ur);

creates an instance *dr* of type *DRect* initialized to the rectangle with lower left corner *ll* and upper right corner *ur*.

3. Operations

DPoint	*dr*.lower_left( )	returns the lower left corner of *dr*.
DPoint	*dr*.upper_right( )	returns the upper right corner of *dr*.
double&	*dr*.left( )	returns the x-coordinate of the lower left corner of *dr*.
double&	*dr*.right( )	returns the x-coordinate of the upper right corner of *dr*.
double&	*dr*.top( )	returns the y-coordinate of the upper right corner of *dr*.
double&	*dr*.bottom( )	returns the y-coordinate of the lower left corner of *dr*.
double	*dr*.width( )	returns the width $(= right - left)$ of *dr*.
double	*dr*.height( )	returns the height $(= bottom - top)$ of *dr*.
DPoint	*dr*.center( )	returns the center point of *dr*.

Appendix A

Class Overview

Class	Header	Library	Page
AGD	AGD.h	libAGD	15
AGD_Key	AGD_key.h	libAGD	141
AGDLayout	AGDLayout.h	libAGD	134
AGDModule	AGDModule.h	libAGD	19
AGDModuleOption	AGDModuleOption.h	libAGD	24
AugmentModule	AugmentModule.h	libAGD	41
BarycenterHeuristic	BarycenterHeuristic.h	libAGD	106
CanonicalOrder	CanonicalOrder.h	libAGD	136
ConvexDrawLayout	ConvexDrawLayout.h	libAGD	59
ConvexLayout	ConvexLayout.h	libAGD	57
DfsRanking	DfsRanking.h	libAGD	114
DPoint	DPoint.h	libAGD	165
DPolyline	DPolyline.h	libAGD	168
DRect	DRect.h	libAGD	169
DSegment	DSegment.h	libAGD	167
EdgeInsertModule	EdgeInsertModule.h	libAGD	47
fastHierarchyLayout	fastHierarchyLayout.h	libAGD	115
FPPLayout	FPPLayout.h	libAGD	53
GraphCopy	GraphCopy.h	libAGD	153
GraphCopyLI	GraphCopyLI.h	libAGD	133
GraphWinInterface	GraphWinInterface.h	libAGD	127
GreedyInsertHeuristic	GreedyInsertHeuristic.h	libAGD	108
GreedySwitchHeuristic	GreedySwitchHeuristic.h	libAGD	109
GridCoordinateMapper	GridCoordinateMapper.h	libAGD	26
GridLayout	GridLayout.h	libAGD	159
GridLayoutModule	GridLayoutModule.h	libAGD	37
HierarchicalRanking	HierarchicalRanking.h	libAGD	111
Hierarchy	Hierarchy.h	libAGD	151
HierarchyLayoutModule	HierarchyLayoutModule.h	libAGD	52
IPoint	IPoint.h	libAGD	161
IPolyline	IPolyline.h	libAGD	163
IRect	IRect.h	libAGD	164
ISegment	ISegment.h	libAGD	162
Keymapper	Keymapper.h	libAGD	142
KeySet	KeySet.h	libAGD	143

Class	Header	Library	Page
Layer	Layer.h	libAGD	149
LayoutBuffer	LayoutBuffer.h	libAGD	131
LayoutInterface	LayoutInterface.h	libAGD	29
LayoutModule	LayoutModule.h	libAGD	33
LEDAMakeBiconnected	LEDAMakeBiconnected.h	libAGD	95
LeveledGraph	LeveledGraph.h	libAGD	157
LmcOrder	CanonicalOrder.h	libAGD	138
MaxAllMapper	MaxAllMapper.h	libAGD	121
MaximumPlanarSubgraph	MaximumPlanarSubgraph.h	libAGDopt	101
MedianHeuristic	MedianHeuristic.h	libAGD	107
MixedModelLayout	MixedModelLayout.h	libAGD	68
OrderSet	CanonicalOrder.h	libAGD	135
Ortho	Ortho.h	libAGD	71
PlanarDrawLayout	PlanarDrawLayout.h	libAGD	63
PlanarizationGridLayout	PlanarizationGridLayout.h	libAGD	85
PlanarizationLayout	PlanarizationLayout.h	libAGD	83
PlanarizerModule	PlanarizerModule.h	libAGD	43
PlanarStraightLayout	PlanarStraightLayout.h	libAGD	61
PlanarSubgraph	PlanarSubgraph.h	libAGD	99
PlanAug	PlanAug.h	libAGD	91
PlanAugOpt	PlanAugOpt.h	libAGDopt	93
PlaneGraphCopy	PlaneGraphCopy.h	libAGD	155
PSInterface	PSInterface.h	libAGD	129
Quod	Quod.h	libAGD	74
RankAssignment	RankAssignment.h	libAGD	49
SchnyderLayout	SchnyderLayout.h	libAGD	55
ShortestPathInserter	ShortestPathInserter.h	libAGD	103
SplitHeuristic	SplitHeuristic.h	libAGD	105
SpringLayout	SpringLayout.h	libAGD	79
SubgraphModule	SubgraphModule.h	libAGD	45
SubgraphPlanarizer	SubgraphPlanarizer.h	libAGD	97
SugiyamaLayout	SugiyamaLayout.h	libAGD	87
TaggedRule	TaggedRule.h	libAGD	147
TopSortRanking	TopSortRanking.h	libAGD	113
tr_literal	tr_literal.h	libAGD	146
TreeLayout	TreeLayout.h	libAGD	77
TutteLayout	TutteLayout.h	libAGD	81
TwoLayerCrossMin	TwoLayerCrossMin.h	libAGD	50
VisibilityMapper	VisibilityMapper.h	libAGD	123
VisibilityRepresentation	VisibilityRepresentation.h	libAGD	66

Bibliography

[1] 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.

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

[3] P. Eades and N. Wormald. The median heuristic for drawing 2-layers networks. Technical Report 69, Department of Computer Science, University of Queensland, 1986.

[4] S. Fialko. Das planare Augmentierungsproblem. Diplomarbeit (Mehlhorn), Universität des Saarlandes, Saarbrücken, 1997.

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

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

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

[8] M. Jünger, S. Leipert, and P. Mutzel. On computing a maximal planar subgraph using PQ–trees. Technical Report 96.227, Institut für Informatik, Universität zu Köln, 1996.

[9] M. Jünger and P. Mutzel. The polyhedral approach to the maximum planar subgraph problem: New chances for related problems. In *DIMACS Graph Drawing '94*, volume 894 of *LNCS*, pages 119–130. Springer-Verlag, 1994.

[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] 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.

[15] P. Mutzel. A polyhedral approach to planar augmentation and related problems. In Paul Spirakis, editor, *Annual European Symposium on Algorithms (ESA-3) : Corfu, Greece, September 2 5-27, 1995; proceedings*, volume LNCS 979, pages 494–507, Berlin, 1995. Springer.

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

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

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

[19] 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.

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

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

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