Contents

12

GraphWin

The *GraphWin* data type combines the *graph* and the *window* data type. An object of type *GraphWin* (short: a GraphWin) is a window, a graph, and a drawing of the graph, all at once. The graph and its drawing can be modified either by mouse operations or by running a graph algorithm on the graph. The GraphWin data type can be used to:

- construct and display graphs,
- visualize graphs and the results of graph algorithms,
- write interactive demos for graph algorithms,
- animate graph algorithms.

All demos and animations of graph algorithms in LEDA are based on GraphWin, many of the drawings in this book have been made with GraphWin, and many of the geometry demos in LEDA have a GraphWin button that allows us to view the graph structure underlying a geometric object.

In this chapter we discuss GraphWins and teach the reader the use of GraphWin. We give an overview and discuss the interactive interface of GraphWins. Next we discuss the node and edge attributes and the global parameters that control how graphs are displayed. In the remaining section we discuss the programming interface of GraphWins and show how to write demos using GraphWins. You will see that it is surprisingly simple to write nice demos of graph algorithms.

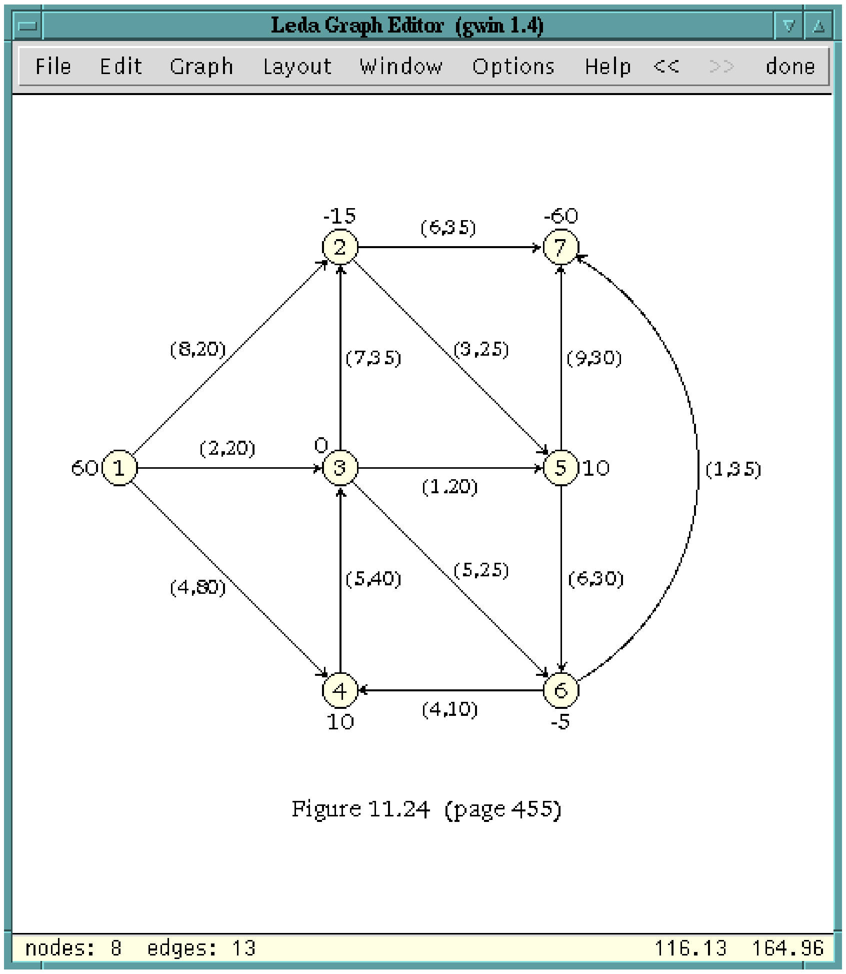


Figure 12.1 A GraphWin. The display part of the window shows a graph *G* and the panel part of the window features the default menu of a GraphWin. We discuss the default menu in Section 12.1. *G* can be edited interactively, e.g., nodes and edges can be added, deleted, and moved around. It is also possible to run graph algorithms on *G* and to display their result or to animate their execution.

12.1 **Overview**

Figure 12.1 shows a GraphWin. We advise that you open a GraphWin before reading on, e.g., by starting the program gwin in xlman. A window as shown in Figure 12.1 will pop up, but with an empty display region. Press the Help button to learn about the interactive use of GraphWin and then construct a graph.

Most of the interaction is with the *left mouse button*. A *single click on the background*

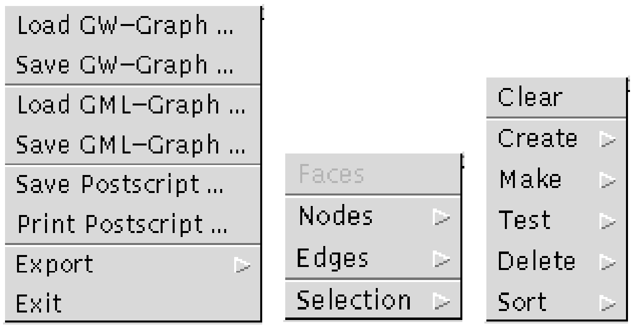


Figure 12.2 GraphWin: File, Edit, and Graph menus.

creates a new node. A *single click on a node* selects the node as the source of a new edge. The next click defines the target of the edge which is either an existing node or a new node (if clicked on the window background). Before defining the target node, bends may be introduced using the middle button. The creation of the new edge can be canceled by clicking the right button.

Nodes can be moved by dragging. Select the node with the left mouse button, hold the button down, and drag the object by moving the cursor. Simultaneously pressing a SHIFT key will move the connected component containing the node. The entire graph can be moved by selecting the background. Of course, when a node is moved all edges incident to it will move with it.

A node is *resized* by clicking on its boundary and dragging the border line of the node.

A *double click* on a node or edge opens a dialog box for setting or changing its attributes. We will discuss the geometric and visual attributes of nodes and edges in Section 12.2.

For the functionality of the middle and the right mouse button we refer the reader to the help menu of GraphWin. Please construct and edit a graph before reading on.

Let us next have a look at the default menu of a GraphWin . We have menu buttons “File”, “Edit”, “Graph”, “Layout”,“Window”, “Options”, “Help”, “undo ($\ll$)”, “redo ($\gg$)”, and “done”. The first six buttons give access to sub-menus as shown in Figures 12.2 and 12.3. We next briefly discuss all buttons and the associated menus.

File: A menu that offers file I/O operations for graphs in either of two formats, allows one to export drawings of graphs, and contains the *exit* button (see Section 12.3 for the effect of the *exit* button).

Edit: A menu with panels for setting the (default) attributes of nodes and edges.

Graph: A menu that offers graph generators, modifiers, and checkers. The generators allow us to construct random, planar, complete, bipartite, grid graphs, ... The modifiers change the current graph (e.g., by removing or adding edges) to make it connected, biconnected, bidirected, ... The checkers can be used to check graph properties, like con-

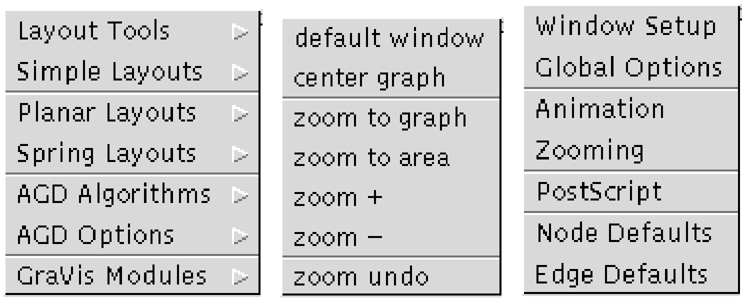


Figure 12.3 GraphWin: Layout, Window and Option menus.

nectedness, biconnectedness, and planarity. Figure 12.4 shows the output of the planarity test for a graph that is non-planar. Many of the checkers can be asked for a proof by clicking the *proof* button. In the case of the planarity test this will either generate a planar drawing or highlight a Kuratowski subgraph as shown in Figure 12.5.

Layout: A menu that gives access to tools for simple layout manipulations (e.g., removing all edge bends or fitting the graph into a box or window) and a collection of graph drawing algorithms. If the graph drawing systems AGD [JMN] or GraVis [LK] are installed, their layout algorithms are included into the menu as shown in Figure 12.3.

Window: A menu with (zoom) operations for changing the user space of the drawing window, e.g., the *zoom graph* button adjusts the window coordinates to the bounding box of the current graph.

Options:A menu with various sub-panels for editing the various window and editor parameters.

undo (≪):A button to undo the last update operation.

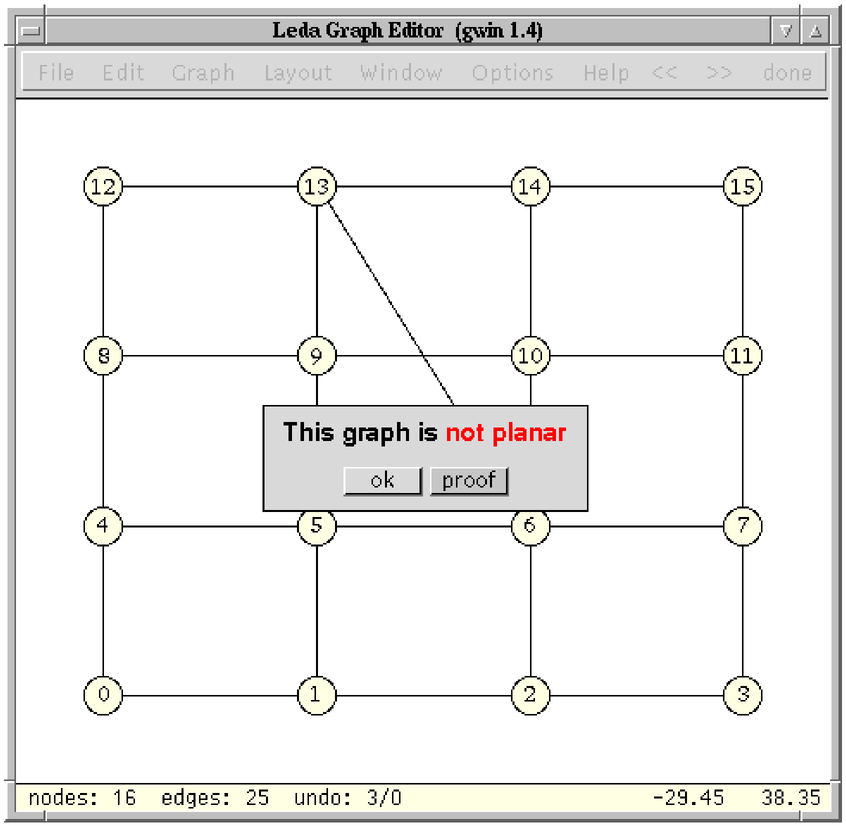
redo (≫): A button to undo the undo.

done: The done button, see Section 12.3.

The drawing of a graph in a GraphWin is controlled by node and edge attributes and by global parameters. We discuss attributes and parameters in the next section.

Exercises for 12.1

1 Call a GraphWin, construct a graph, and test whether it is biconnected.
2 Construct a graph and then change all node shapes from circular to rectangular.
3 Construct the dependency graph for the chapters of this book as shown in the preface. Apply some of the layout algorithms to the graph.


Leda Graph Editor (gwin 1.4)
File Edit Graph Layout Window Options Help << >> done
12
13
14
15
8
9
10
11
4
5
6
7
0
1
2
3
ok
proof

This graph is not planar

nodes: 16 edges: 25 undo: 3/0 -29.45 38.35

Figure 12.4 An outcome of a planarity test.

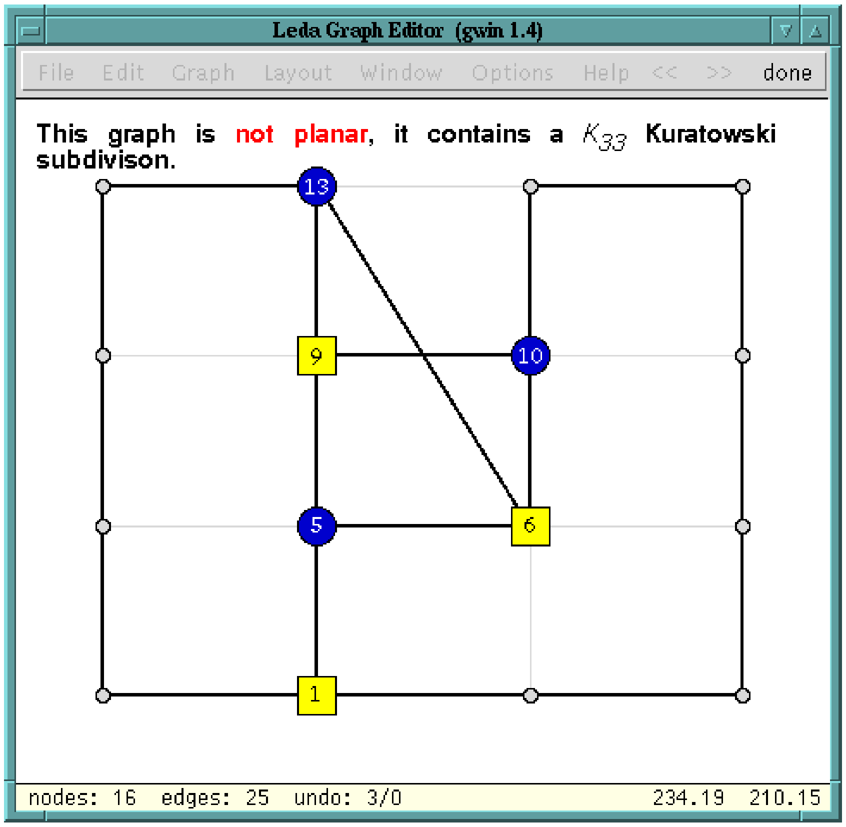
12.2 Attributes and Parameters

In this section we discuss global parameters and node and edge attributes. The node and edge attributes control how nodes and edges are drawn and the global parameters control the general behavior of a GraphWin. Attributes and parameters can be changed either by setup panels (as shown in Figures 12.6 and 12.7) or by operations of the programming interface as discussed in Section 12.3.

Node Attributes: The node attributes are:

position:An attribute of type *point* (default value: $(0, 0)$) defining the position of the center of the node in the user coordinate system of the window.

shape:An attribute of type *gw_node_shape* (default value: *circle_node*) defining the shape of the node. Possible values are *circle_node*, *ellipse_node*, *square_node*, and *rectangle_node*. The size of a node is determined by its width and its height. Width and height are measured


Leda Graph Editor (gwin 1.4)
File Edit Graph Layout Window Options Help << >> done
13
9
10
5
6
1

This graph is not planar, it contains a K_{33} Kuratowski subdivison.

nodes: 16 edges: 25 undo: 3/0 234.19 210.15

Figure 12.5 The effect of clicking the proof button in Figure 12.4.

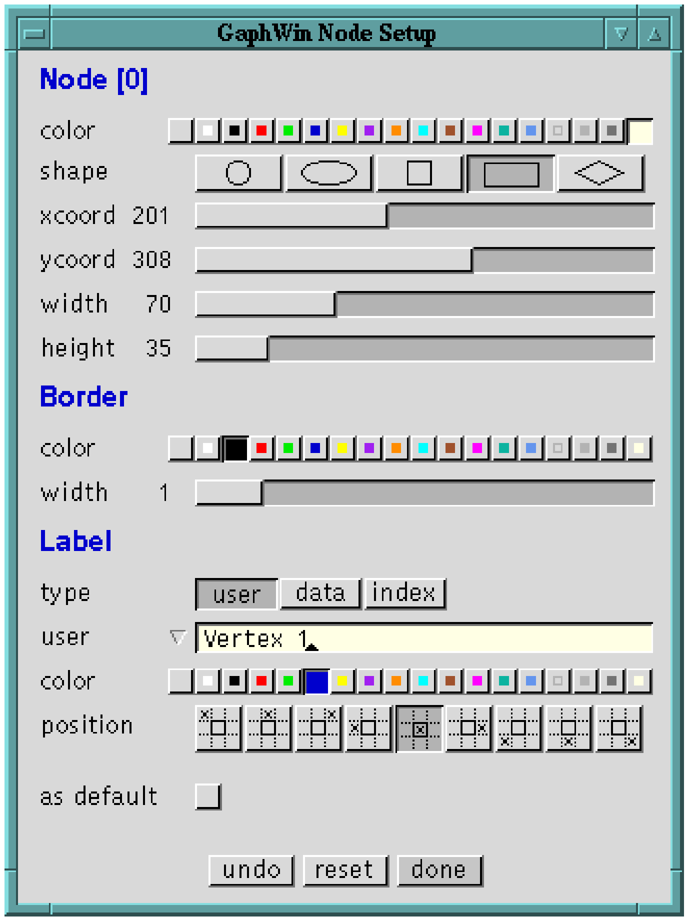
in pixels. The horizontal and vertical dimension can also be measured in user space; we use *radius1* and *radius2* for the dimensions in user space.

width:An attribute of type *int* (default value: 20) defining the width of the node in pixels. The horizontal dimension of a node is also available as an attribute with name *radius1* that gives the horizontal dimension of the node in user space. Any change of one of these two attributes also changes the other, maintaining the relation $radius1 = W.pix\_to\_real(width)/2$.

height:An attribute of type *int* (default value: 20) defining the height of the node in pixels. As for the *width* attribute the vertical dimension of a node can be accessed or changed through a *radius2* attribute giving the vertical dimension of the node in user space.

color:An attribute of type *color* (default value: *ivory*) defining the color used to fill the interior of the node.

pixmap:An attribute of type *char*∗ (default value: *NULL*) defining a pixrect used to fill the interior of the node.



Node [0]

color

xcoord 201

ycoord 308

width 70

height 35

Border

color

width 1

Label

type user

user Vertex 1

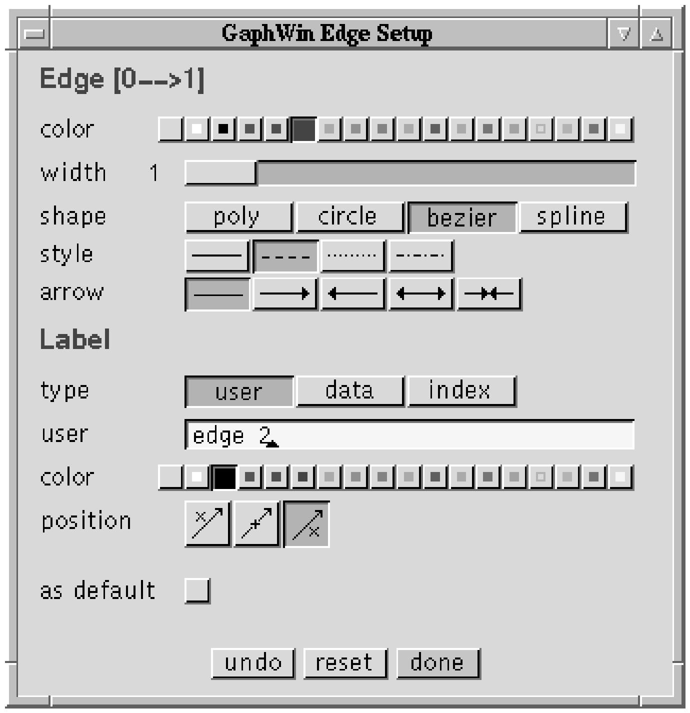
color

Figure 12.6 The node setup panel.

border color:An attribute of type *color* (default value: black) defining the color used to draw the boundary line of the node.

border width: An attribute of type *int* (default value: 1) defining the line width in pixels used to draw the border line of the node. We also have a user space variant of this attribute called *border thickness*: *border_width* and *border_thickness* are related through the equation $border\_thickness = W.pix\_to\_real(border\_width)$.

label type:An attribute of type *gw_label_type* (default value: *index_label*) specifying which label of a node is displayed. Possible values are *no_label*, *user_label*, *data_label*, and *index_label*. Every node of a GraphWin has three labels associated with it: an index label generated automatically from the internal numbering of the nodes, a user label (of type *string*), and a data label that is used to represent the node data of parameterized graphs.



GaphWin Edge Setup

Edge [0-->1]

color

width 1

shape: poly circle [✓ bezier] spline

style

arrow

Label

type: [✓ user] data index

user: edge 2

color

position

as default

undo reset done

Figure 12.7 The edge setup panel.

user label: An attribute of type *string* defining the user label of the node. The default value is the empty string.

label position: An attribute of type *gw_position* (default value: *central_pos*) defining the position of the label. Possible values are *central_pos*, *northwest_pos*, *north_pos*, *northeast_pos*, *east_pos*, *southeast_pos*, *south_pos*, *southwest_pos*, and *west_pos*. Each value defines one of the eight neighboring cells in a rectangular grid of appropriate dimension or the node position itself as the position of the label.

label color: An attribute of type *color* (default value: *black*) defining the color used to draw the label of the node.

Edge Attributes: Edges have the following attributes:

shape:An attribute of type *gw_edge_shape* (default value: *poly_edge*) defining the shape of the edge. Possible values are *poly_edge* (polygonal edges), *circle_edge* (circular arcs), *bezier_edge* (Bezier curves), *spline_edge* (spline curves).

bends:An attribute of type *list<point>* (default value: empty list) defining the sequence of bends of the edge. The interpretation of the bends depends on the shape of the edge. For *poly_edge* this list defines the sequence of bends of the poly-line. For *circle_edge* only the first point p of the sequence is used. Together with the two terminal node positions it defines a circular arc starting at the source position, passing through p and ending in the target position of the edge. For *bezier_edge* and *spline_edge* edges the list gives the sequence of control points that define the corresponding Bezier or spline curve.

direction:An attribute of type *gw_edge_dir* defining whether the edge is drawn as a directed or an undirected edge. Possible values are *undirected_edge* (the edge is drawn undirected), *directed_edge* (the edge is drawn directed from source to target), *redirected_edge* (the edge is drawn directed from target to source), and *bidirected_edge* (the edge is drawn bidirected).

width:An attribute of type *int* (default value: 1) defining the width of the edge in pixels. The width of an edge can also be specified by an attribute called *thickness* that gives the line width of the edge in user coordinates; *thickness* and *width* are related through $thickness = W.pix\_to\_real(width)$.

color:An attribute of type *color* (default value: *black*) defining the color of the edge.

style:An attribute of type *gw_edge_style* (default value: *solid*) defining the line style of the edge. Possible values are *solid*, *dashed*, *dotted*, and *dashed_dotted*.

label type:An attribute of type *gw_label_type* (default value: *no_label*) defining the type of the label of the edge. Possible values are *no_label*, *user_label*, *data_label*, and *index_label* (see the corresponding attribute for nodes for an explanation).

user label: An attribute of type *string* defining the user label of the edge. The default value is the empty string.

label position: An attribute of type *gw_position* (default value: *west_pos*) defining the position of the label. Possible values are *central_pos* (the label is placed centered on the edge), *east_pos* (the label is placed to the right of the edge), and *west_pos* (the edge is placed to the left of the edge).

label color: An attribute of type *color* (default value: *black*) defining the color of the edge label.

slider positions:Every edge has three *sliders* associated with it. They are only visible if the corresponding handler (see Section 12.5.1) is defined. For each slider the *slider position* is an attribute of type *double* (default value: 0) defining the relative position of the slider on the (directed) edge. The value of slider position lies between zero and one. Edge sliders can be used to adjust the value of an edge label interactively.

Global Parameters: A *GraphWin* has the *window* parameters background color, background pixmap, grid style, and grid distance, and the following additional parameters.

flush: A parameter of type *bool* (default value: *true*) that controls whether changes of node and edge attributes are shown directly or not. If *flush* is false, changes are invisible up to the next call of the *redraw* operation. In this way, it is possible to hide all intermediate steps of a sequence of operations and to show only the end result.

animation steps: A parameter of type *int* (default value: 16) that defines the number of intermediate drawings used in the animation of layout changes and zoom operations. Setting animation steps to 0 disables all animations.

zoom objects: A parameter of type *bool* (default value: true). If this flag is true, the size of nodes and edges is adjusted automatically during zoom operations. If the flag is false, the pixel width and height of all objects is preserved during zoom operations.

show status:A parameter of type *bool* (default value: true). If this flag is true, some selected parameters, e.g., the number of nodes and edges and the current position of the mouse cursor in user coordinates, is shown in a status line at the bottom of the display region.

12.3 The Programming Interface

So far we have concentrated on the interactive interface of GraphWins, as most LEDA users will become acquainted with GraphWins through their interactive use. We now turn to the programming interface. You must read this section if you want to write programs that use a GraphWin.

The *GraphWin* data type offers a large variety of operations. We discuss the most important one in the remainder of this chapter and refer the reader to the manual for the complete list of operations.

12.3.1 *Creating and Opening a Graph Window*

A GraphWin has an associated graph and an associated window. Either one of them may or may not be specified in the constructor.

```
GraphWin gw;
```

creates a graph window *gw* that uses its own (private) graph G and window W. G is initialized with the empty graph. Three optional arguments may be passed to initialize W: a *label* of type *string*, the initial *width*, and the initial *height* both of type *int*.

```
GraphWin gw(graph& G);
```

creates a graph window *gw* and associates the graph G with it. G may also be a parameterized graph of some type *GRAPH*<*vtype*, *etype*>. In this case, every node v has an additional *data label* attribute that contains a string representation of the *vtype* value $G[v]$ associated with v. This representation is constructed using the stream output operator (*operator* $\ll$ (*ostream*&, *const vtype*&)). In the same way, every edge e has a data label representing $G[e]$. In Section 12.6 we give a program that uses *GraphWin* to display a graph of type *GRAPH*<*point*, *int*> representing a Delaunay triangulation.

```
GraphWin gw(window& W);
GraphWin gw(graph& G, window& W);
```

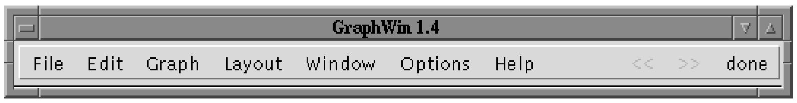


Figure 12.8 A GraphWin panel for editing a graph in a different window.

do not create their own window but use the supplied window *W* for displaying the graph. In this case, the display operation opens a small panel window (see Figure 12.8) containing only the standard menu.

References to the graph and window of a *GraphWin gw* can be retrieved by

```
window& W = gw.get_window();
```

and

```
graph&  G = gw.get_graph();
```

respectively.

A graph window is opened and displayed by calling one of the two following *display* operations:

```
gw.display()
```

opens *gw* and displays it at the default position of data type *window* and

```
gw.display(x,y)
```

opens *gw* and displays it with its left upper corner at the position with pixel coordinates (x, y). As for windows the special coordinate *window*::*center* can be used to center the graph window in either coordinate on the screen.

The interactive interface is started by the *edit* operation.

```
bool gw.edit();
```

puts *gw* into *edit mode* (also called *interactive mode*). The buttons of *gw* are now enabled; in particular the graph associated with *gw* may now be changed interactively. The edit session is terminated when either the *done* button is pressed or *exit* is selected from the file menu. The edit operation returns *true* in the first case and *false* in the second case.

We are now ready for the first example program. We declare and display a graph window *gw*, and then start an edit loop (*while* (*gw.edit*( ))) that lets the user construct or modify the graph *G* associated with *gw*. If *edit* is terminated by pressing the *done* button, the graph is tested for planarity, the outcome of the planarity test is written to standard output, and *gw* is again put into edit mode. If the editor is left by pressing the *exit* button in the *file* menu, the loop and the program terminate.

⟨*gw.c*⟩≡

```
#include <LEDA/graphwin.h>
#include <LEDA/graph_alg.h>
main()
{
  GraphWin gw("Leda Graph Editor");
  graph& G = gw.get_graph();
  gw.display(window::center,window::center);
  while ( gw.edit() )
  { if (PLANAR(G))
      cout << "This graph is planar." << endl;
    else
      cout << "This graph is non-planar." << endl;
   }
  return 0;
}
```

The structure of the program above is generic for many simple interactive demos of graph algorithms. The program runs in a loop. In each iteration the graph is edited and the graph algorithm is run. We call this scheme the *edit-and-run paradigm* for interactive demos. We will see a more elaborate use of the paradigm in Section 12.4.

12.3.2 *Graph Operations*

A GraphWin has an associated graph. There are two methods to update this graph through the programming interface.

The first method uses the update operations offered by GraphWin. For example,

```
node gw.new_node(const point& p);
```

creates a new node v with default attributes. The position of v is set to p.

```
void gw.del_node(node v);
```

removes v from the graph,

```
edge gw.new_edge(node v, node w);
```

creates a new edge $e = (v, w)$ with default attributes,

```
void gw.del_edge(edge e);
```

removes e from the graph, and

```
void gw.clear_graph();
```

makes the graph empty.

The second method reuses the update operations for graphs. We obtain a reference to the graph associated with *gw* by calling *gw.get_graph*( ) and then apply graph update operations to it.

```
graph& G = gw.get_graph();
// some update operations on G
G.new_node();
G.del_edge(e);
gw.update_graph();                  // CRUCIAL
```

Observe the *gw.update_graph*( ) statement at the end of the sequence. This statement informs *gw* about the fact that its graph was modified and allows it to update its internal data structures. Without the statement the graph *G* and the internal data structures of *gw* will go out of sync and disasters may occur.

We illustrate the use of *update_graph* operation by giving an implementation of the *new_node* operation of GraphWin; the actual implementation is different and more efficient.

```
node gw_new_node(GraphWin& gw, const point& p)
{ graph& G = gw.get_graph();
  node v = G.new_node();
  gw.update_graph();
  gw.set_position(v,p);
  return v;
}
```

12.3.3 *Attribute and Parameter Operations*

Attributes of nodes and edges and global parameters are manipulated by *get* and *set* operations. In the case of attributes we distinguish between the individual attributes of existing nodes and edges and the *default attributes* which are used to initialize the attributes of new nodes and edges.

Individual Attributes of Nodes and Edges: The attributes of existing nodes and edges can be retrieved or changed by the following operations. We use *object* for either *node* or *edge* and *attrib* (of type *attrib_type*) for an arbitrary attribute.

```
attrib_type  gw.get_attrib(object x);
```

returns the current value of attribute *attrib* of object x.

```
attrib_type  gw.set_attrib(object x, attrib_type a);
```

sets the attribute *attrib* of object x to a and returns the previous value of the attribute.

```
void  gw.set_attrib(list<object>& L, attrib_type a);
```

sets attribute *attrib* for all objects in L to a.

```
void gw.reset_attributes();
```

resets the attributes of all objects to their default values.

The current attributes of all nodes and edges may be saved and restored later to the saved values by the following functions.

```
void gw.save_node_attributes();
void gw.save_edge_attributes();
void gw.restore_node_attributes();
void gw.restore_edge_attributes();
```

These functions are very useful if the appearance of the graph has to be changed temporarily, e.g., to highlight a substructure of the graph.

We give an example. We replace all nodes of elliptic shape by yellow rectangular nodes and all blue edges by black dashed edges, wait five seconds, and then restore all attributes to their original values.

```
graph& G = gw.get_graph();

void gw.save_node_attributes();
void gw.save_edge_attributes();

node v;
forall_nodes(v,G) {
 if (gw.get_shape(v) == ellipse_node)
 { gw.set_shape(v,rectangle_node);
   gw.set_color(v,yellow);
  }
}

edge e
forall_edge(e,G) {
 if (gw.get_color(e) == blue)
 { gw.set_style(e,dashed);
   gw.set_color(e,black);
  }
}

gw.redraw();

leda_wait(5);

void gw.restore_node_attributes();
void gw.restore_edge_attributes();
```

Default Attribute Values: Every attribute has a default value which is used to initialize the attributes of new objects. The default attribute values can be changed by the following operations. Note that changing a default attribute also affects all existing objects, unless the optional boolean flag *apply* in the corresponding *set_node_attrib* operation is set to *false*.

```
attrib_type  get_node_attrib();
attrib_type  set_node_attrib(attrib_type x, bool apply=true);
```

reads or sets the default value of node attribute *attrib*. If *apply* is true, the *attrib* attribute of all existing nodes is changed in the same way.

```
attrib_type  get_edge_attrib();
attrib_type  set_edge_attrib(attrib_type x, bool apply=true);
```

reads or sets the default value of edge attribute *attrib*. If *apply* is true, the *attrib* attribute of all existing edges is changed in the same way.

The current default values of all attributes can be saved to a file and later reloaded by the following operations.

```
void gw.save_defaults(string fname);
void gw.read_defaults(string fname);
```

We close with an example. We declare a GraphWin *gw*, change the default values of some attributes, open the window associated with *gw*, and put *gw* into edit mode.

⟨*gw_attributes.c*⟩≡

```
#include <LEDA/graphwin.h>
main()
{
  GraphWin gw;
  // default attributes of nodes
  gw.set_node_shape(rectangle_node);
  gw.set_node_color(yellow);
  // default attributes of edges
  gw.set_edge_width(2);
  gw.set_edge_color(blue);
  gw.set_edge_direction(undirected_edge);
  gw.display();
  gw.edit();
}
```

Almost every program using a GraphWin starts with a small preamble that changes default attributes to settings that are appropriate for the application.

Global Parameters: Global parameters can be retrieved or changed by a collection of *get*- and *set*-operations. We use *param_type* for the type and *param* for the value of the corresponding parameter.

There is a *get* and *set* operation for each global parameter *param*.

```
param_type  gw.get_param();
param_type  gw.set_param(param_type x);
```

The set operation returns the previous value of the corresponding parameter.

In the following example we set the *flush* parameter to *false* before changing the individual attributes of some nodes. Then we redraw the graph to display the changes and reset the *flush* parameter to its previous value;

```
gw.set_animation_steps(12);
bool fl = gw.set_flush(false);
forall(v,L) {
 gw.set_color(v,blue);
```

```
  gw.set_shape(v,rhombus_node);
 }
 gw.redraw();
 gw.set_flush(fl);
```

12.3.4 *I/O Operations*

GraphWin supports two file formats for the permanant storage of graphs and their attributes, the (native) gw-format and the GML-format [Him97] of Himsolt. It also allows to generate a Postscript representation of the current drawing that can easily be included into LaTeX documents. Many of the figures of this book have been produced in this way. The operations in this section are available in the file-menu.

The read operations

```
int  gw.read_gw(istream& istr);
int  gw.read_gw(string fname);
int  gw.read_gml(istream& istr);
int  gw.read_gml(string fname);
```

clear the current graph and read a new graph and its attributes from the input stream *istr* or file *fname*, respectively. The operations return 0 on success and a special error code if something goes wrong (see the manual for details). The write operations

```
int gw.save_gw(ostream& ostr);
int gw.save_gw(string fname);
int gw.save_gml(ostream& ostr);
int gw.save_gml(string fname);
```

write the current graph and its layout to output stream *ostr* or to file *fname*, respectively. The operations return 0 on success and a non-zero error code if something goes wrong.

Postscript representations of drawings are generated by

```
bool gw.save_ps(ostream& ostr);
bool gw.save_ps(string fname);
```

which write the current drawing as a Postscript file to output stream *ostr* or to file *fname*, respectively.

12.3.5 *Layout Operations*

We discuss operations for manipulating the *layout* of the graph associated with a GraphWin, i.e., the positions of the nodes and the sequence of bends of the edges. The operations are, for example, used to realize the functions in the layout-menu.

The arguments of the layout operations specify new node positions and/or new sequences of bends. The layout operation moves the nodes and changes the drawings of edges accordingly. The animation of the layout operations (and also of the zooming operations) is controlled by the *animation_steps* parameter. *GraphWin* animates changes in the layout by linear interpolation. It shows a sequence of *animation_steps* intermediate layouts, where each node and edge moves a fraction of $1/$*animation_steps* of its total movement in each step. If *animation_steps* is set to zero, the layout change is performed instantaneously.

In most layout operations the new node position can be specified either as *points* or as pairs of *doubles*. We list both versions for the first layout function and only one for the others. The operations

```
void gw.set_position(const node_array<double>& xpos,
                     const node_array<double>& ypos);
void gw.set_position(const node_array<point>& pos);
```

move every node v of *gw* from its old position to position $(xpos[v], ypos[v])$ or $pos[v]$, respectively, and leave the bends of all edges unchanged,

```
void gw.set_layout(const node_array<point>& pos,
                   const edge_array<list<point> >& bends);
```

moves every node v to position $pos[v]$ and sets the bend sequence of every edge e to $bends[e]$,

```
void gw.set_layout(const node_array<point>& pos);
```

moves every node v of the graph to position $pos[v]$ and removes all edge bends from the layout,

```
void gw.remove_bends();
```

removes all bends from the layout and leaves the node positions unchanged,

```
void gw.place_into_box(double x0, double y0, double x1, double y1);
```

moves the graph into the rectangular box $(x0, y0, x1, y1)$ by scaling and translating the layout, and

```
void gw.place_into_win();
```

moves the graph into the drawing window by scaling and translating.

Layout coordinate computations: Consider the following situation. We have a graph window *gw* and its associated graph G. We have computed a new layout for G, but the new layout does not conform to the coordinate space of *gw*. We want to adjust the layout data before applying it. Section 12.4 gives an application.

The operations in this section are very helpful in this situation. They apply the transformations *place_into_box* and *place_into_win* to the layout data supplied separately in node and edge arrays.

```
void gw.adjust_coords_to_box(node_array<double>& xpos,
                             node_array<double>& ypos,
                             edge_array<list<double> >& xbends,
                             edge_array<list<double> >& ybends,
                             double x0, double y0, double x1, double y1);
```

transforms the layout given by *xpos*, *ypos*, *xbends*, and *ybends* in the same way as a call *place_into_box*(*x0*, *y0*, *x1*, *y1*) would do. However, the actual layout of the current graph is not changed by this operation.

```
void gw.adjust_coords_to_box(node_array<double>& xpos,
                             node_array<double>& ypos,
                             double x0, double y0, double x1, double y1);
```

transforms the layout given by *xpos*, *ypos* as *gw.place_into_box*(*x0*, *y0*, *x1*, *y1*) would do. It ignores any edge bends. The actual layout of the current graph is not changed by this operation.

```
void gw.adjust_coords_to_win(node_array<double>& xpos,
                             node_array<double>& ypos,
                             edge_array<list<double> >& xbends,
                             edge_array<list<double> >& ybends);
```

calls *adjust_coords_to_box*(*xpos*, *ypos*, *xbends*, *ybends*, *wx0*, *wy0*, *wx1*, *wy1*) with the current window rectangle (*wx0*, *wy0*, *wx1*, *wy1*). Finally,

```
void gw.adjust_coords_to_win(node_array<double>& xpos,
                             node_array<double>& ypos);
```

calls *adjust_coords_to_box*(*xpos*, *ypos*, *wx0*, *wy0*, *wx1*, *wy1*), where as in the preceding operation (*wx0*, *wy0*, *wx1*, *wy1*) is the current window rectangle .

12.3.6 ***Zoom Operations***

Zoom operations change the coordinate system of the window but do not change the layout of the graph. A zoom operation is a combination of a stretch or shrink transformation (changing the scaling factor of the window) with a translation of the window in user space. The *animation step* parameter specifies the number of intermediate window positions to be shown in the animation of the zoom operation; if the parameter is zero the zoom is performed instantaneously.

```
void gw.zoom(double f)
```

zooms the window by the factor f; this multiplies the scaling factor by f and leaves the coordinates of the center of the window unchanged.

```
void gw.zoom_area(double x0, double y0, double x1, double y1)
```

zooms the window to rectangle $(x0, y0, x1, y1)$. More precisely, if the aspect ratio of the zoom rectangle $r = (y1 - y0)/(x1 - x0)$ is equal to the aspect ratio wr of the current window, the window coordinates are set to $(x0, y0, x1, y1)$. Otherwise, if r is smaller than wr the new window coordinates are $(x0, y0, x1, y')$ with $y' = y0 + wr * (x1 - x0)$ and if r is greater than wr the new coordinates are $(x0, x', y0, y1)$ with $x' = x0 + (y1 - y0)/wr$.

```
void gw.center_graph()
```

performs a zoom operation that does not change the scaling of the window and moves the center of the bounding box of the current graph layout to the center of the window.

```
void gw.zoom_graph();
```

calls *gw.zoom_area*($x0, y0, x1, y1$) such that $x0$, $x1$, and $y0$ are the left, right and lower coordinates of the bounding box of the current layout of the graph.

12.3.7 ***Miscellaneous Operations***

We close our discussion of the programming interface with a list of small, but useful functions.

```
void gw.message(string msg);
```

displays *msg* at the top of the window. If *msg* is the empty string, the previous message is deleted.

```
bool gw.wait(const msg);
```

displays *msg* and waits until the done-button is pressed or exit is selected from the file menu. The result of the operation is *true* in the first case and *false* in the second case.

```
int gw.open_panel(panel& P)
```

displays panel P centered on *gw* and returns the result of P.*open*( ). During the execution of P.*open*( ) all menus of *gw* are disabled.

```
node gw.ask_node();
```

asks the user to select a node by clicking with the left mouse button on it. The selected node is returned; *nil* is returned if the click does not hit a node.

```
edge gw.ask_edge();
```

asks the user to select an edge by clicking with the left mouse button on it. The selected edge is returned; *nil* is returned if the click does not hit an edge.

```
void gw.get_bounding_box(double& x0, double& y0, double& x1, double& y1);
```

computes the coordinates ($x0, y0, x1, y1$) of a minimal area rectangular bounding box containing the current layout of the graph.

12.4 **Edit and Run: A Simple Recipe for Interactive Demos**

We implement a simple demo that illustrates planarity testing based on the edit-and-run paradigm for interactive demos of graph algorithms. The demo illustrates many of the functions discussed in the preceding sections.

We define a GraphWin *gw* with frame label “Planarity Test Demo” and open it. We then enter the edit-loop. After each edit operation, we run the graph algorithm on the graph G associated with *gw* and display the result.

⟨*gw_plandemo.c*⟩≡

```
  #include <LEDA/graphwin.h>
  #include <LEDA/graph_alg.h>
  ⟨plandemo: highlight⟩
  int main()
  {
    GraphWin gw("Planarity Test Demo");
    gw.display(window::center,window::center);
    while (gw.edit())
    {
      graph& G = gw.get_graph();
      ⟨run graph algorithm and display result⟩
    }
    return 0;
  }
```

So far the program is generic (except for the frame label). We now come to the part specific to the planarity demo.

We test G for planarity. If G is planar and has at least three nodes (otherwise the current drawing is already without crossings), we compute a straight line embedding and display it. The computation of the straight line embedding returns the coordinates of a straight line embedding in some coordinate system. We adjust the coordinates to the coordinate space of *gw* by calling *adjust_coords_to_win*. Finally, we display the straight line embedding by calling *gw.set_layout*(...).

If the graph is non-planar, we compute a Kuratowski subdivision $K = (V_k, E_k)$ and display it by calling the *high_light* function. We wait until the user clicks done and then restore the old drawing. The function KURATOWSKI computes the set of nodes and edges of the subdivision and for each node of G the degree of the node in the subdivision. For all $v \in V$ the degree $deg[v]$ is equal to 2 for subdivision points, 4 for all other nodes if K is a K_5, and -3 ($+3$) for the nodes of the left (right) side if K is a $K_{3,3}$.

⟨*run graph algorithm and display result*⟩≡

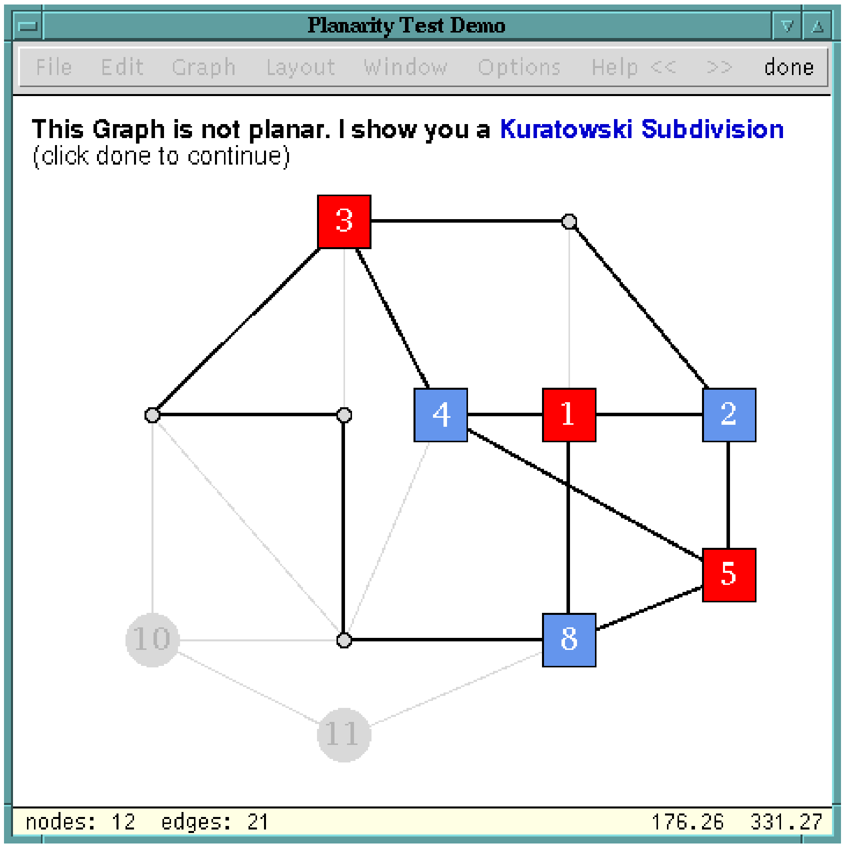
```
  if (PLANAR(G))
  { if (G.number_of_nodes() < 3) continue;
    node_array<double> xcoord(G);
    node_array<double> ycoord(G);
    STRAIGHT_LINE_EMBEDDING(G,xcoord,ycoord);
    gw.adjust_coords_to_win(xcoord,ycoord);     // !!!
    gw.set_layout(xcoord,ycoord);
  }
  else
  { list<node> V_k;
    list<edge> E_k;
    node_array<int> kind(G);
    KURATOWSKI(G,V_k,E_k,kind);
    gw.save_all_attributes();
    highlight(gw,V_k,E_k,kind);
    gw.wait("This Graph is not planar. I show you a\
             Kuratowski Subdivision (click done).");
    gw.restore_all_attributes();
  }
```

We still have to define the function *highlight* that highlights the Kuratowski subgraph. We set *flush* to false at the beginning of *highlight* and call *redraw* and restore the old value of *flush* at the end. This ensures that all changes made by *highlight* will become effective at the same time.

We highlight the Kuratowski subgraph by drawing its edges with width two and black (all other edges are drawn grey and with width one) and by using color and shape codes to highlight its nodes. Figure 12.9 shows an example.

⟨*plandemo: highlight*⟩≡

```
  void highlight(GraphWin& gw, list<node> V, list<edge> E,
                               node_array<int>& kind)
  {
    const graph& G = gw.get_graph();
    bool flush0 = gw.set_flush(false);
    node v;
    forall_nodes(v,G) {
      switch (kind[v]) {
        case  0: gw.set_color(v,grey1);
                 gw.set_border_color(v,grey1);
                 gw.set_label_color(v,grey2);
                 break;
        case  2: gw.set_color(v,grey1);
                 gw.set_label_type(v,no_label);
                 gw.set_width(v,8);
                 gw.set_height(v,8);
                 break;
        case  3:
        case  4: gw.set_shape(v,rectangle_node);
                 gw.set_color(v,red);
                 break;
        case -3: gw.set_shape(v,rectangle_node);
                 gw.set_color(v,blue2);
                 break;
      }
    }
    edge e;
    forall_edges(e,G) gw.set_color(e,grey1);
    forall(e,E)
    { gw.set_color(e,black);
```


Planarity Test Demo
File Edit Graph Layout Window Options Help << >> done
3
4
1
2
5
8
10
11
nodes: 12 edges: 21 176.26 331.27

This Graph is not planar. I show you a Kuratowski Subdivision
(click done to continue)

Figure 12.9 The planarity test demo: Highlighting a Kuratowski subdivision.

```
      gw.set_width(e,2);
    }
  gw.redraw();
  gw.set_flush(flush0);
}
```

Exercises for 12.4

1 Write a program that animates quicksort. Have a graph with one node for each input and no edges. Change the layout of the graph as the sort progresses.

2 Write a program that animates heapsort.

3 Write a program that always shows a DFS-structure of the currently edited graph by drawing the different edge types (tree, backward, forward, cross) in different colors or styles.

12.5 Customizing the Interactive Interface

We describe three ways for customizing the interactive interface:

- Call-back functions,
- Extended and/or additional menus, and
- Redefined edit actions.

Each method will allow us to write nicer demos.

12.5.1 *Call-Back Functions*

Call-back or handler functions can be used to associate arbitrary functionality with the edit operations of *GraphWin*. Two handlers can be defined for every operation. The first one, the so-called *pre-handler*, is called immediately before the corresponding edit operation. The second one, the so-called *post-handler*, is called at the end of the operation. For move operations of nodes and sliders, there is a third handler, the so-called *move-handler* which is called for all intermediate positions.

The pre-handlers have a boolean return value which tells *GraphWin* whether the corresponding edit operation is to be executed or not. This provides a simple way of disallowing edit operations under certain conditions. In general, pre- and post-handler also have different parameter lists.

The null-handler (*NULL*) can be used to remove a pre- or post-handler from an edit operation.

We give a list of the most important handlers and the corresponding *set* operations. There are two versions of each *set_handler*, one each for defining the pre- and post-handler. The functions have the same name and differ in the type of the function pointer argument: functions for setting pre-handlers take an argument of type *bool* $(*func)(GraphWin\&, ...)$ and functions for setting post-handlers take an argument of type *void* $(*func)(GraphWin\&, ...)$.

```
void gw.set_new_node_handler(bool (*f)(GraphWin&,point));
```

sets the pre-handler of the new-node operation to f, i.e., $f(gw, p)$ is called before a node is created at position p.

```
void gw.set_new_node_handler(void (*f)(GraphWin&,node));
```

sets the post-handler of the new-node operation to f, i.e., $f(gw, v)$ is called after a new node v has been created.

```
void gw.set_new_edge_handler(bool (*f)(GraphWin&,node,node));
```

sets the pre-handler of the new-edge operation to f, i.e., $f(gw, v, w)$ is called before a new edge (v, w) is created.

```
void gw.set_new_edge_handler(void (*f)(GraphWin&,edge));
```

sets the post-handler of the new-edge operation to f, i.e., $f(gw, e)$ is called after a new edge e has been created.

```
void gw.set_del_node_handler(bool (*f)(GraphWin&,node));
```

sets the pre-handler of the del-node operation to f, i.e., $f(gw, v)$ is called each time before a node v is deleted.

```
void gw.set_del_node_handler(void (*f)(GraphWin&));
```

sets the post-handler of the del-node operation to f, i.e., $f(gw)$ is called each time a node has been deleted.

```
void gw.set_del_edge_handler(bool (*f)(GraphWin&,edge));
```

sets the pre-handler of the del-edge operation to f, i.e., $f(gw, e)$ is called each time before an edge e is deleted.

```
void gw.set_del_edge_handler(void (*f)(GraphWin&));
```

sets the post-handler of the del-edge operation to f, i.e., $f(gw)$ is called each time an edge has been deleted.

```
void gw.set_init_graph_handler(bool (*f)(GraphWin&));
```

sets the pre-handler of the init-graph operation to f, i.e., $f(gw)$ is called every time before any global update of the graph, e.g., in a clear, generate, or load operation.

```
gw.set_init_graph_handler(void (*f)(GraphWin&));
```

sets the post-handler of the init-graph operation to f, i.e., f is called after each global update of the graph.

Node moving and edge slider moving operations may have three different handlers. The first is called before the moving starts, the second is called for every intermediate position, and the third one is called at the final position of the node after the moving has been finished. The handlers are set by:

```
gw.set_start_move_node_handler(bool (*f)(GraphWin&,node));
gw.set_move_node_handler(bool (*f)(GraphWin&,node,point));
gw.set_end_move_node_handler(void (*f)(GraphWin&,node));
gw.set_start_edge_slider_handler(
                   void (*f)(GraphWin& gw,edge,double),int i);
gw.set_edge_slider_handler(
                   void (*f)(GraphWin& gw,edge,double),int i);
gw.set_end_edge_slider_handler(
                   void (*f)(GraphWin& gw,edge,double),int i);
```

Recall that each edge has three sliders associated with it. The integer argument i in the last three functions selects the slider, $0 \le i \le 2$.

12.5.2 ***A Recipe for On-line Demos of Graph Algorithms***

The edit-and-run paradigm for demos of graph algorithms requires an explicit user action, namely a click on the done-button, to start the graph algorithm to be demonstrated. Call-back or handler functions allow us to write on-line demos which show the result of a graph algorithm while the graph is edited and not only after editing.

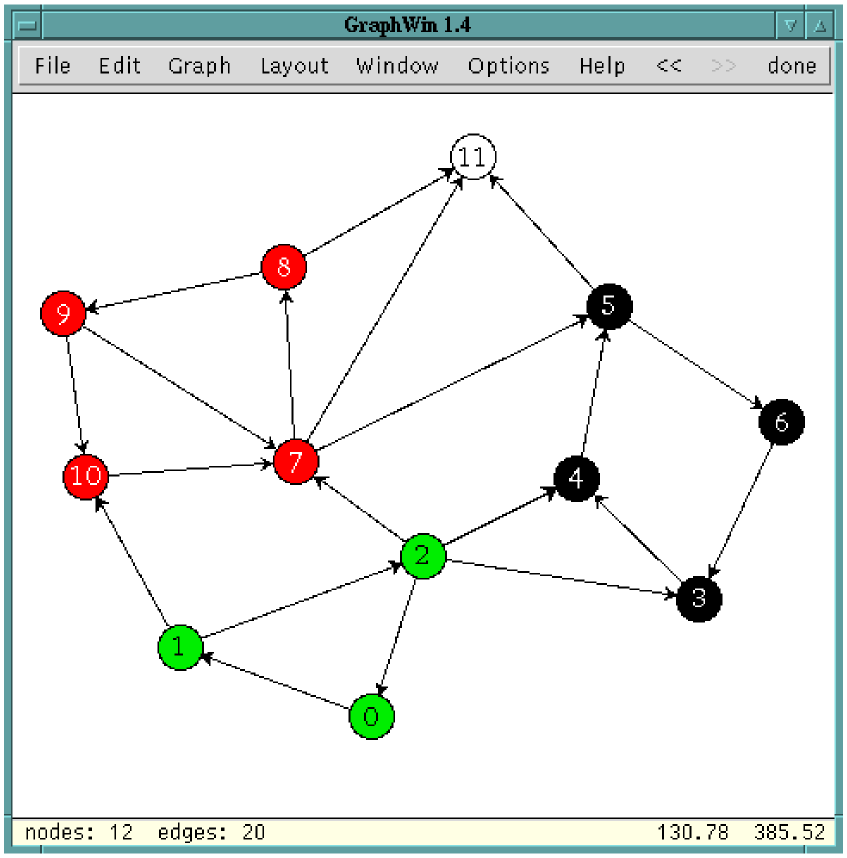
We give the generic structure of a demo that calls a graph algorithm after every addition or deletion of a node or edge and after the initialization of the graph (for example, by reading it from a file). We define a function *run_and_display* that runs the graph algorithm on the graph associated with *gw* and updates the display. We then define post-handlers for the *new_node*, *new_edge*, *del_node*, *del_edge*, and *init_graph* operations; each handler simply calls *run_and_display*(*gw*). In the main program we tell *GraphWin* which handlers to use by calling the corresponding *set_handler* functions, display the window, and call *gw.edit*( ). That's all.

⟨*gw_handler.c*⟩≡

```
#include <LEDA/graph_alg.h>
#include <LEDA/graphwin.h>

void run_and_display(GraphWin& gw)
{ ⟨run algorithm and update display⟩ }

void new_node_handler(GraphWin& gw, node) { run_and_display(gw); }
void new_edge_handler(GraphWin& gw, edge) { run_and_display(gw); }
void del_edge_handler(GraphWin& gw)       { run_and_display(gw); }
void del_node_handler(GraphWin& gw)       { run_and_display(gw); }
void init_graph_handler(GraphWin& gw)     { run_and_display(gw); }

int main()
{
  GraphWin gw;

  gw.set_init_graph_handler(init_graph_handler);
  gw.set_new_edge_handler(new_edge_handler);
  gw.set_del_edge_handler(del_edge_handler);
  gw.set_new_node_handler(new_node_handler);
  gw.set_del_node_handler(del_node_handler);

  gw.display();
  gw.edit();

  return 0;
}
```

We will next derive a specific demo from this framework by instantiating the *run_and_display* function. We illustrate the strongly connected components of the graph associated with *gw*; all nodes belonging to the same component should be colored the same and nodes in different components should be colored differently.

The "work horse" of our demo is a function *void run_and_display*(*GraphWin*&) that uses the graph algorithm *STRONG_COMPONENTS* to compute a numbering *comp_num* of the nodes of the current graph, such that all nodes of a strongly connected component receive the same number. Each node is painted with the number of its component.



nodes: 12 edges: 20 130.78 385.52

Figure 12.10 An screen shot of an on-line demo for the strongly connected components of a graph.

⟨*run algorithm and update display*⟩≡

```
graph& G = gw.get_graph();
node_array<int> comp_num(G);
STRONG_COMPONENTS(G,comp_num);
node v;
forall_nodes(v,G) gw.set_color(v,color(comp_num[v]));
```

Figure 12.10 shows a screen shot of the program after a few editing operations.

12.5.3 *Defining and Changing Menus*

The menus of *GraphWin* are not fixed. New sub-menus and buttons can be added to the main window and any sub-menu, in this way extending the set of functions and algorithms that can be applied to the current graph by a mouse click. Furthermore, the set of default menus in the main window's menu bar can be changed by removing standard menus. All operations for changing menus have to be called before the window is displayed for the first time.

Changing the Standard Main Menu: The default menus in *GraphWin*'s menu bar are determined by a bit mask that is the bitwise-or of an arbitrary subset of the predefined constants *M_FILE*, *M_EDIT*, *M_GRAPH*, *M_LAYOUT*, *M_WINDOW*, *M_OPTIONS*, *M_HELP*, and *M_DONE*. Each of these constants represents the corresponding standard menu discussed in Section 12.1. The value *M_COMPLETE* is defined as the bitwise-or of all constants above, i.e., it specifies a menu bar containing all standard menus. The operation

```
long gw.set_default_menu(long mask);
```

defines the set of standard menus, where *mask* is the bitwise-or of an arbitrary subset of the predefined constants listed above. The operation

```
void gw.del_menu(long mask);
```

removes all menus corresponding to 1-bits in *mask* from the menu bar.

Adding New Menus: New sub-menus can be added to an existing menu (or the main menu bar) by calling the *add_menu* operation. Each menu is represented by an integer (*menu_id*) from an internal numbering of all menus. The main menu bar has *menu_id* zero.

```
int  gw.add_menu(GraphWin& gw, string label, int menu_id = 0),
```

creates a sub-menu in menu with id *menu_id*. The corresponding button is labeled with *label*. The operation returns the menu id of the new menu. The menu id of a standard menu can be obtained by calling *get_menu*(*string*) with the name of the menu, e.g.,

```
get_menu("Help");
```

returns the menu id of the help menu.

Adding Simple Functions: We call functions of type *void func*(*GraphWin*& *gw*) *simple*. The *add_simple_call* operation of *GraphWin* can be used to add (buttons for starting) simple functions to an existing menu or the main menu bar.

```
void  gw.add_simple_call(void (*func)(GraphWin&),
                         string label, int menu_id = 0);
```

adds a new button with label *label* to the menu with menu id *menu_id*. Whenever this button is pressed during edit mode *func*(*gw*) is called.

We give an example. Assume we want to add a button to the main menu that runs a DFS algorithm of type

```
void dfs(graph& G, node s, node_array<bool>& reached)
```

on the current graph. We write a simple function *void* (*run_dfs*)(*GraphWin*&) that tells *GraphWin* how to call *dfs* and how to display its result.

```
void run_dfs(GraphWin& gw)
{
  // provide arguments
  graph& G = gw.get_graph();
  node s = gw.ask_node();
  node_array<bool> reached(G,false);
  // call function
  dfs(G,s,reached);
  // display result
  node v;
  forall_nodes(v,G) if (reached[v]) gw.set_color(v,red);
}
```

and add the function to the main menu by calling

```
gw.add_simple_call(run_dfs,"dfs");
```

The string argument “dfs” will be used as the label of the new menu button. We may also want to extend the help menu. We define a simple function *about_dfs* that opens a panel and displays a help string

```
void about_dfs(GraphWin& gw)
{ window& W = gw.get_window();

  panel P;
  P.set_panel_bg_color(win_p->mono() ? white : ivory);
  P.text_item("The dfs-button runs dfs on the current graph.");
  P.button("OK");
  W.disable_panel();
  P.open(W);
  W.enable_panel();
}
```

and add it to the help menu.

```
int h_menu = gw.get_menu("Help");
gw.add_simple_call(about_dfs, "About DFS",h_menu);
```

Adding GraphWin Member Functions: Not every operation of the programming interface of *GraphWin* is available in the interactive interface. However, there is an easy way of adding operations of type *void GraphWin*::*func*( ), i.e., member functions without parameters and without a result. The operation

```
gw.add_member(void (*GraphWin::func)(), string label, int menu_id = 0);
```

adds a new button with label *label* to the menu with menu id *menu_id*. Whenever this button is pressed during edit mode *gw.func*( ) is called.

As an example, we add a ”redraw” button, that calls the *gw.redraw*( ) operation, to the main panel.

```
gw.add_member_call(&GraphWin::redraw,"redraw");
```

Adding Families of Functions: Sometimes, one wants to add an entire group of functions, all with the same interface, to a menu. In this case it would be tedious to write a wrapper for each of these functions. It is more convenient to write only a single *caller* function that can deal with all functions of the group. The caller takes a reference to a *GraphWin* and a pointer to the function to be called as arguments. More precisely, if the function to be called is of type *function_t*, the caller has type *void* $(*caller)(GraphWin\&, function\_t)$.

The *gw_add_call* function template adds a function together with its caller to a menu. This operation should better be realized by a member function template. However, only a few compilers currently support this feature of C++.

```
template <class function_t>
void  gw_add_call(GraphWin& gw, function_t func,
                  void (*caller)(GraphWin&, function_t),
                  string label, int menu_id=0);
```

adds a new button with label *label* to the menu with menu id *menu_id*. Whenever this button is pressed in edit mode, the function *caller* is called with arguments *gw* and *func*.

We use a family of graph drawing functions as an example. Assume we have a library of graph drawing algorithms (e.g., the AGD library [JMN]) and want to build a *graph_draw* menu which makes all functions in the library available on a mouse click. We assume that all graph drawing algorithms take a graph G and compute for every node v of G a position $(xcoord[v], ycoord[v])$.

```
void draw_alg1(const graph& G, node_array<double> xcoord,
                               node_array<double> ycoord);
void draw_alg2(const graph& G, node_array<double> xcoord,
                               node_array<double> ycoord);
...
```

A generic caller function for this type of graph algorithm is as follows:

```
typedef void (*draw_alg)(graph&, node_array<double>&,
                                 node_array<double>&);
void call_draw_alg(GraphWin& gw, draw_alg draw)
{
  // provide arguments
  graph& G = gw.get_graph();
  node_array<double> xcoord(G);
  node_array<double> ycoord(G);
  // call function
  draw(G,xcoord,ycoord);
  // display result
  gw.adjust_coords_to_win(xcoord,ycoord);
  gw.set_layout(xcoord,ycoord);
  if (!gw.get_flush()) gw.redraw();
}
```

The new menu is now easily created.

```
int draw_menu = gw.add_menu("graph drawing");
gw_add_call(gw,draw_alg1,call_draw_alg,"draw_alg1",draw_menu)
gw_add_call(gw,draw_alg2,call_draw_alg,"draw_alg2",draw_menu)
...
```

A Complete Example: We give a complete example that illustrates the possibilities to extend and modify menus. We will write a demo that illustrates dfs, spanning trees, connected components, and strongly connected components.

For dfs and spanning trees we use simple functions.

⟨*simple functions*⟩≡

```
void dfs_num(GraphWin& gw)
{ graph& G = gw.get_graph();
  node_array<int> dfsnum(G);
  node_array<int> compnum(G);
  DFS_NUM(G,dfsnum,compnum);
  node v;
  forall_nodes(v,G) gw.set_label(v,string("%d|%d",dfsnum[v],compnum[v]));
  if (gw.get_flush() == false) gw.redraw();
}
void span_tree(GraphWin& gw)
{ graph& G = gw.get_graph();
  list<edge> L = SPANNING_TREE(G);
  gw.set_color(L,red);
  gw.set_width(L,2);
  if (gw.get_flush() == false) gw.redraw();
}
```

The LEDA functions to compute components of a graph all have the same interface. They take a graph and compute a node array of *ints*, and return an int. Any such function can be added to a GraphWin using the caller

⟨*components caller*⟩≡

```
// a caller for component algorithms
void call_comp(GraphWin& gw,
               int (*comp)(const graph& G, node_array<int>& compnum) )
{ graph& G = gw.get_graph();
  node_array<int> compnum(G);
  comp(G,compnum);
  node v;
  forall_nodes(v,G)
  { int i = compnum[v];
    gw.set_label(v,string("%d",i));
    gw.set_color(v,(color)(i%16));
   }
  if (gw.get_flush() == false) gw.redraw();
}
```

In the main program we define a GraphWin, delete some of the standard menus (just to illustrate how it is done), add our simple calls, add a reset button, and finally create a sub-menu for the components functions.

⟨*gw_menu.c*⟩≡

```
#include <LEDA/graphwin.h>
#include <LEDA/graph_alg.h>
#include <LEDA/graph_misc.h>
⟨components caller⟩
⟨simple functions⟩
int main()
{
  GraphWin gw;
  // we delete some of the standard menus
  gw.set_default_menu(M_COMPLETE & ~M_LAYOUT & ~M_HELP);
  // add two simple function calls
  gw.add_simple_call(dfs_num,   "dfsnum");
  gw.add_simple_call(span_tree, "spanning");
  // a member call
  gw.add_member_call(&GraphWin::reset,"reset");
  // and a menu with three non-simple functions using
  // a common call function
  int menu1 = gw.add_menu("components");
  gw_add_call(gw,COMPONENTS,       call_comp,"simply connected",  menu1);
  gw_add_call(gw,COMPONENTS1,      call_comp,"simply connected1", menu1);
  gw_add_call(gw,STRONG_COMPONENTS,call_comp,"strongly connected",menu1);
  gw.display();
  gw.edit();
  return 0;
}
```

Figure 12.11 shows a screen shot of this demo.

12.5.4 ***Defining Edit Actions***

Mouse operations in the display region of a *GraphWin* generate events. An event is characterized by its event bit mask *event_mask* (which is the or of elementary masks to be defined below) and the current position *mouse_position* of the mouse pointer. Event masks have associated *edit actions*. All edit actions are functions of type

```
void action(GraphWin& gw, const point& pos);
```

When an event occurs, the associated action function is called with the *GraphWin* object and the current mouse pointer position *mouse_position* as arguments. The object (node or edge) under the current position can be queried by the *get_edit_node* or *get_edit_edge* operation.

Event masks are the bitwise-or of some of the following predefined constants:

A_LEFT, *A_MIDDLE*, *A_RIGHT*: If one of these bits is set, the corresponding mouse button (left, middle, or right) has been clicked.

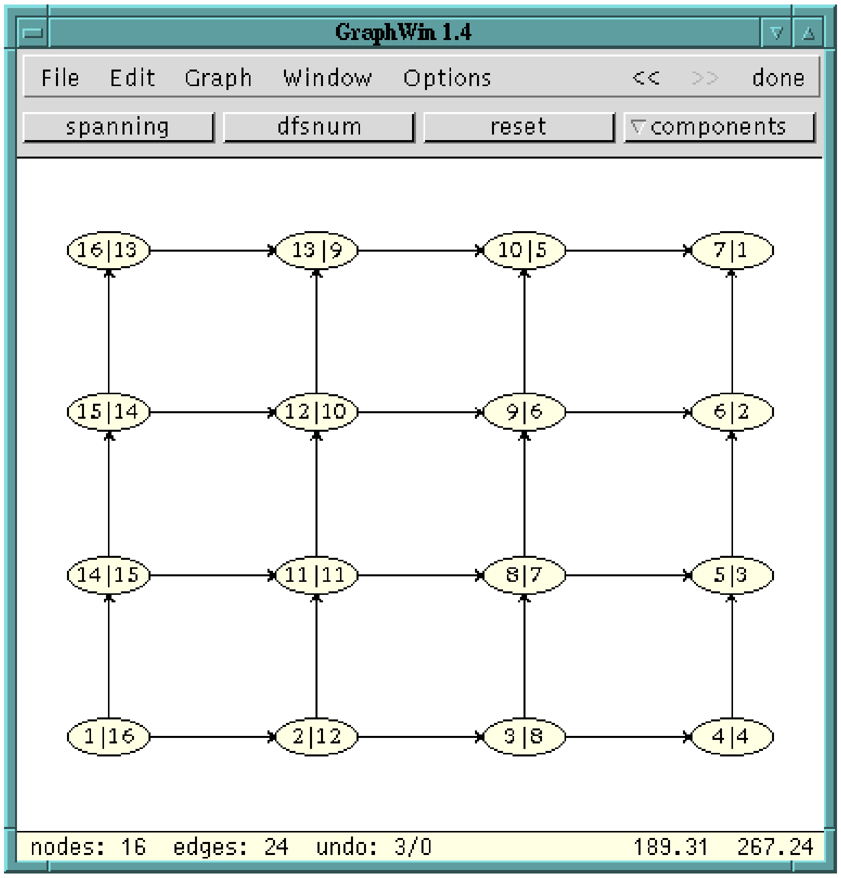


Figure 12.11 Extending the menu: Computing a DFS-numbering.

A_DRAG: This bit indicates that the mouse is moved with one or more buttons (specified by the bits discussed above) held down.

A_DOUBLE: This bit indicates a double click , i.e., the event that a mouse button has been clicked twice.

A_SHIFT, *A_CTRL*, *A_ALT*: If one of these bits is set, the corresponding keyboard control key (Shift,Ctrl,Alt) is pressed.

A_NODE: If this bit is set, the mouse pointer is located over a node and the node can be queried by the *gw.get_edit_node*( ) operation.

A_EDGE: If this bit is set, the mouse pointer is located over an edge and the edge can be queried by the *gw.get_edit_edge*( ) operation.

A_SLIDER: If this bit is set, the mouse pointer is located over a slider of an edge. The corresponding edge can be queried as above and the number of the slider (0,1, or 2) can be obtained by calling the *gw.get_edit_slider*( ) operation.

An event mask is defined by a combination of these bits, for instance

```
( A_LEFT | A_NODE | A_DOUBLE )
```

describes a double click of the left mouse button on a node.

Setting Edit Actions: The following operations can be used to change the action functions associated with events.

```
gw_action gw.set_action(long mask,void (*func)(GraphWin&, const point&));
```

sets the action on condition *mask* to *func* and return the previous action of this condition. After this call *func* is called with the *GraphWin* object and the current edit position as arguments whenever the condition defined by *mask* becomes true.

```
void gw.reset_actions();
```

resets all actions to their default values and

```
void gw.clear_actions();
```

sets all actions to *NULL*.

The following piece of code shows part of the initialization of the default edit actions.

```
// left button (create,move,scroll,zoom)
set_action( A_LEFT                              , gw_new_node);
set_action( A_LEFT |                    A_NODE  , gw_new_edge);
set_action( A_LEFT |           A_DRAG |A_NODE  , gw_move_node);
set_action( A_LEFT |           A_DRAG |A_EDGE  , gw_move_edge);
set_action( A_LEFT |           A_DRAG          , gw_scroll_graph);
set_action( A_LEFT |           A_DRAG |A_SLIDER, gw_move_edge_slider);
set_action( A_LEFT |A_SHIFT  |A_DRAG |A_NODE  , gw_move_component);
set_action( A_LEFT |A_DOUBLE |        A_NODE  , gw_setup_node);
set_action( A_LEFT |A_DOUBLE |        A_EDGE  , gw_setup_edge);
```

An Example Program: The following program redefines some of the default actions, for example, when the left mouse button is clicked over a node with the control key pressed, the node color will be increased by one.

⟨*gw_action.c*⟩≡

```
#include<LEDA/graphwin.h>

void change_node_color(GraphWin& gw, const point&)
{ node v  = gw.get_edit_node();
  int col = (gw.get_color(v) + 1) % 16;
  gw.set_color(v,color(col));
 }

void change_edge_color(GraphWin& gw, const point&)
```

```
{ edge e  = gw.get_edit_edge();
  int col = (gw.get_color(e) + 1) % 16;
  gw.set_color(e,color(col));
 }

void center_node(GraphWin& gw, const point& p)
{ node v  = gw.get_edit_node();
  gw.set_position(v,p);
 }
void delete_node(GraphWin& gw, const point&)
{ node v  = gw.get_edit_node();
  gw.del_node(v);
 }
void zoom_up(GraphWin& gw, const point&)   { gw.zoom(1.5); }
void zoom_down(GraphWin& gw, const point&) { gw.zoom(0.5); }

main()
{
  GraphWin gw;

  gw.set_action(A_LEFT | A_NODE | A_CTRL,  change_node_color);
  gw.set_action(A_LEFT | A_EDGE | A_CTRL,  change_edge_color);
  gw.set_action(A_LEFT | A_NODE | A_SHIFT, center_node);
  gw.set_action(A_RIGHT| A_NODE, delete_node);

  gw.set_action(A_LEFT | A_CTRL, zoom_up);
  gw.set_action(A_RIGHT| A_CTRL, zoom_down);

  gw.display(window::center,window::center);
  gw.edit();
}
```

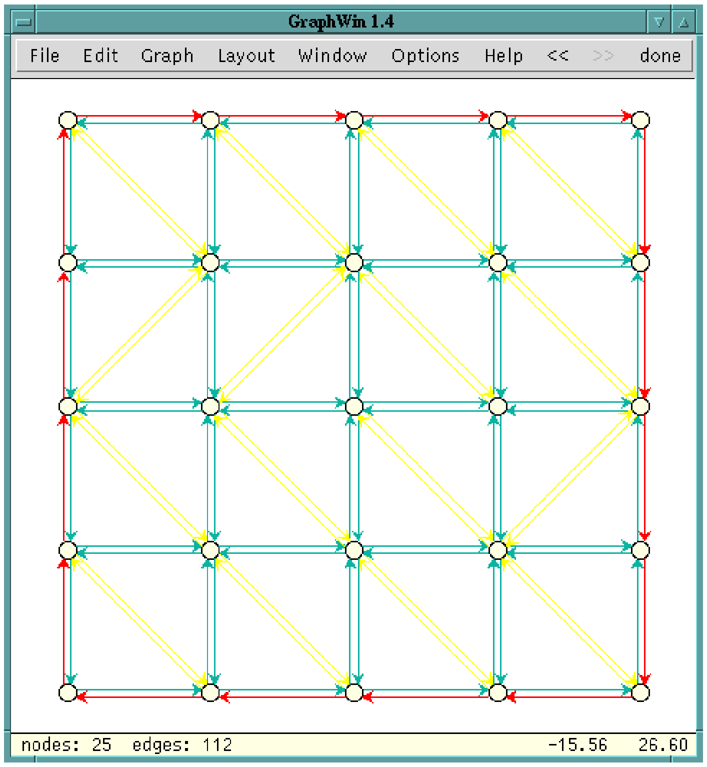
12.6 Visualizing Geometric Structures

Many geometric data structures of LEDA are implemented by labeled graphs, e.g., Delaunay diagrams are represented by graphs of type *GRAPH<point, int>* and Voronoi diagrams are represented as graphs of type *GRAPH<CIRCLE, POINT>*. Many geometry demos have a GraphWin-button for viewing the underlying graph structures.

We sketch how this button is realized. In the demo below we compute the Delaunay triangulation *DT* of a set *L* of twenty-five points on a regular grid. We then declare a GraphWin *gw* for *DT*, tell *gw* that we want each node v to be drawn at position $DT[v]$, as a circle of radius eight pixels, and without label, and that we want each edge to be drawn with a color indicating its label. Start the demo and the graph shown in Figure 12.12 will appear.

⟨*gw_delaunay.c*⟩≡

```
#include <LEDA/plane_alg.h>
#include <LEDA/graphwin.h>

main()
```



nodes: 25 edges: 112 -15.56 26.60

Figure 12.12 GraphWin displaying a Delaunay triangulation.

```
{
  GRAPH<rat_point,int> DT;
  list<rat_point> L;
  lattice_points(25,100,L);
  DELAUNAY_TRIANG(L,DT);
  GraphWin gw(DT);
  node v;
  forall_nodes(v,DT)
  { rat_point p = DT[v];
    gw.set_position(v,p.to_point());
    gw.set_label_type(v,no_label);
    gw.set_width(v,8);
    gw.set_height(v,8);
  }
  edge e;
  forall_edges(e,DT)
  { switch (DT[e]) {
     case DIAGRAM_EDGE:     gw.set_color(e,green2); break;
     case NON_DIAGRAM_EDGE: gw.set_color(e,yellow); break;
```

```
      case HULL_EDGE:        gw.set_color(e,red);    break;
     }
    }
    gw.display();
    gw.zoom_graph();
    gw.edit();
  }
```

12.7 A Recipe for On-line Demos of Network Algorithms

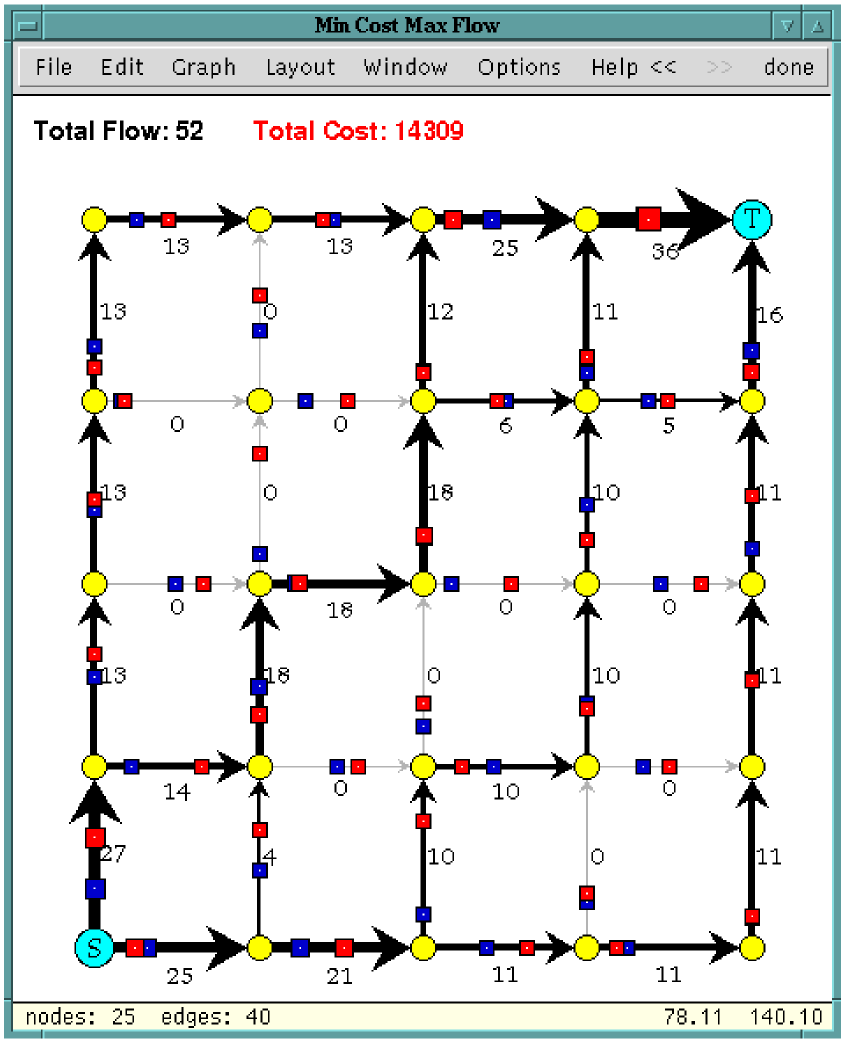
Networks are graphs whose edges (and sometimes nodes) are labeled with numbers, e.g., capacities or costs. On-line demos of network algorithms should allow the user to edit the underlying graph as well as the edge capacities. We have already seen how to react on-line to update operations. In this section we will show how to implement capacity changes by edge sliders. All demos of network algorithms follow the paradigm presented in this section. We use the min cost flow algorithm as our example. All other demos are simpler. Figure 12.13 shows a screenshot.

The global structure of our demo is as follows. We define edge maps *cap* and *cost* in order to make edge capacities and edge costs globally available for the handler functions. We then define a function that runs the min cost flow algorithm and displays the result and we define handlers for edge events and handlers for slider events.

In the main program we generate the grid graph *G* shown in Figure 12.13 and associate the edge maps *cap* and *cost* with it. We define a GraphWin *gw* for *G* and set its header to "Min Cost Max Flow". We disable edge bends since sliders can be used for straight line edges only. We set the node and edge attributes to the colors hinted at in the figure, and we adjust the size of the layout such that it uses about 90% of the window. Finally, we open the window and put it into edit mode.

⟨*gw_mcmflow.c*⟩≡

```
  #include <LEDA/graphwin.h>
  #include <LEDA/graph_alg.h>
  static edge_map<int> cap;
  static edge_map<int> cost;
  ⟨run min cost flow and display result⟩
  ⟨edge handlers⟩
  ⟨capacity and cost sliders⟩
  int main()
  {
    // construct a (grid) graph
    graph G;
    node_array<double> xcoord;
    node_array<double> ycoord;
```



Total Flow: 52 Total Cost: 14309

nodes: 25 edges: 40 78.11 140.10

Figure 12.13 Animation of a min-cost-flow algorithm.

```
grid_graph(G,xcoord,ycoord,5);
// initialize cap and cost maps
cap.init(G);
cost.init(G);
GraphWin gw(G,"Min Cost Max Flow");
// disable edge bends
gw.set_action(A_LEFT | A_DRAG | A_EDGE , NULL);
```

⟨*set handlers*⟩

⟨*set attributes of nodes and edges*⟩

```
    //adjust layout
    gw.adjust_coords_to_win(xcoord,ycoord);
    gw.set_layout(xcoord,ycoord);
    gw.zoom(0.9);

    // open gw
    gw.display();
    gw.edit();

    return 0;
  }
```

Setting the node and edge attributes is routine.

⟨*set attributes of nodes and edges*⟩≡

```
    gw.set_node_color(yellow);
    gw.set_node_shape(circle_node);
    gw.set_node_label_type(no_label);
    gw.set_node_width(14);
    gw.set_node_height(14);

    gw.set_edge_direction(directed_edge);

    node s = G.first_node();
    gw.set_shape(s,rectangle_node);
    gw.set_width(s,22);
    gw.set_height(s,22);
    gw.set_color(s,cyan);
    gw.set_label(s,"S");

    node t = G.last_node();
    gw.set_shape(t,rectangle_node);
    gw.set_width(t,22);
    gw.set_height(t,22);
    gw.set_color(t,cyan);
    gw.set_label(t,"T");
```

The function that runs the min cost flow algorithm and displays its result is similar to the display function in the strongly connected components demo of Section 12.4, but slightly more complex because we are aiming for a more elaborated visualization.

We obtain the graph G from *gw*, we set s and t to the first and last node, respectively, and compute the flow using the global edge maps *cap* and *cost*. We compute the flow value and the cost of the flow and we set the width of every edge proportional to the flow through the edge. Edges with flow zero are faded to grey. We reset flush, write a message containing flow value and cost, and redraw.

⟨*run min cost flow and display result*⟩≡

```
  void run_mcm_flow(GraphWin& gw)
  { bool flush = gw.set_flush(false);

    graph& G = gw.get_graph();
    node   s = G.first_node();
    node   t = G.last_node();

    gw.message("\\bf Computing MinCostMaxFlow");
    edge_array<int> flow(G);
    int F = MIN_COST_MAX_FLOW(G,s,t,cap,cost,flow);
    int C = 0;
    // sum up total cost and indicate flow[e] by the width of e
    edge e;
    forall_edges(e,G)
    { C += flow[e]*cost[e];
      gw.set_label_color(e,black);
      gw.set_label(e,string("%d",flow[e]));
      gw.set_width(e,1+int((flow[e]+4)/5.0));
      if (flow[e] == 0)
        gw.set_color(e,grey2); // 0-flow edges are faded to grey
      else
        gw.set_color(e,black);
     }
    gw.set_flush(flush);
    gw.message(string("\\bf Flow: %d  \\bf Cost: %d",F,C));
    gw.redraw();
  }
```

We come to the edge handlers. We first define an auxiliary function *init_edge* that sets the capacity and the cost of an edge to random values and sets the slider values for the zeroth and the first slider of the edge accordingly. The *init_handler* initializes all edges, computes a min cost flow and displays it. The new edge handler initializes the edge, computes a min cost flow and displays it.

The init handler and the node and edge handlers of *gw* are set in the obvious way.

⟨*edge handlers*⟩≡

```
  void init_edge(GraphWin& gw, edge e)
  { // init capacity and cost to a random value
    cap[e] = rand_int(10,50);
    cost[e] = rand_int(10,75);
    // set sliders accordingly
    gw.set_slider_value(e,cap[e]/100.0,0);  // slider zero
    gw.set_slider_value(e,cost[e]/100.0,1); // slider one
  }
  void init_handler(GraphWin& gw)
  { edge e;
    forall_edges(e,gw.get_graph()) init_edge(gw,e);
    run_mcm_flow(gw);
  }
  void new_edge_handler(GraphWin& gw, edge e)
  { init_edge(gw,e);
    run_mcm_flow(gw);
  }
```

⟨*set handlers*⟩≡

```
  gw.set_init_graph_handler(init_handler);
```

```
gw.set_del_edge_handler(run_mcm_flow);
gw.set_del_node_handler(run_mcm_flow);
gw.set_new_edge_handler(new_edge_handler);
```

We come to the sliders. The cap slider handlers handle the change of capacities. We use the zeroth edge slider for the capacities. When an edge slider is picked up we display an appropriate message. As long as the slider is moved we display the new capacity. When the edge slider is released we recompute the flow and display it.

⟨*capacity and cost sliders*⟩≡

```
// capacity sliders
void start_cap_slider_handler(GraphWin& gw, edge, double)
{ gw.message("\\bf\\blue Change Edge Capacity"); }
void cap_slider_handler(GraphWin& gw,edge e, double f)
{ cap[e] = int(100*f);
  gw.set_label_color(e,blue);
  gw.set_label(e,string("cap = %d",cap[e]));
}
void end_cap_slider_handler(GraphWin& gw, edge, double)
{ run_mcm_flow(gw); }
```

⟨*set handlers*⟩+≡

```
gw.set_start_edge_slider_handler(start_cap_slider_handler,0);
gw.set_edge_slider_handler(cap_slider_handler,0);
gw.set_end_edge_slider_handler(end_cap_slider_handler,0);
gw.set_edge_slider_color(blue,0);
```

Cost sliders are treated completely analogously.

⟨*capacity and cost sliders*⟩+≡

```
// cost sliders
void start_cost_slider_handler(GraphWin& gw, edge, double)
{ gw.message("\\bf\\red Change Edge Cost"); }
void cost_slider_handler(GraphWin& gw, edge e, double f)
{ cost[e] = int(100*f);
  gw.set_label_color(e,red);
  gw.set_label(e,string("cost = %d",cost[e]));
}
void end_cost_slider_handler(GraphWin& gw, edge, double)
{ run_mcm_flow(gw); }
```

⟨*set handlers*⟩+≡

```
  gw.set_start_edge_slider_handler(start_cost_slider_handler,1);
  gw.set_edge_slider_handler(cost_slider_handler,1);
  gw.set_end_edge_slider_handler(end_cost_slider_handler,1);
  gw.set_edge_slider_color(red,1);
```

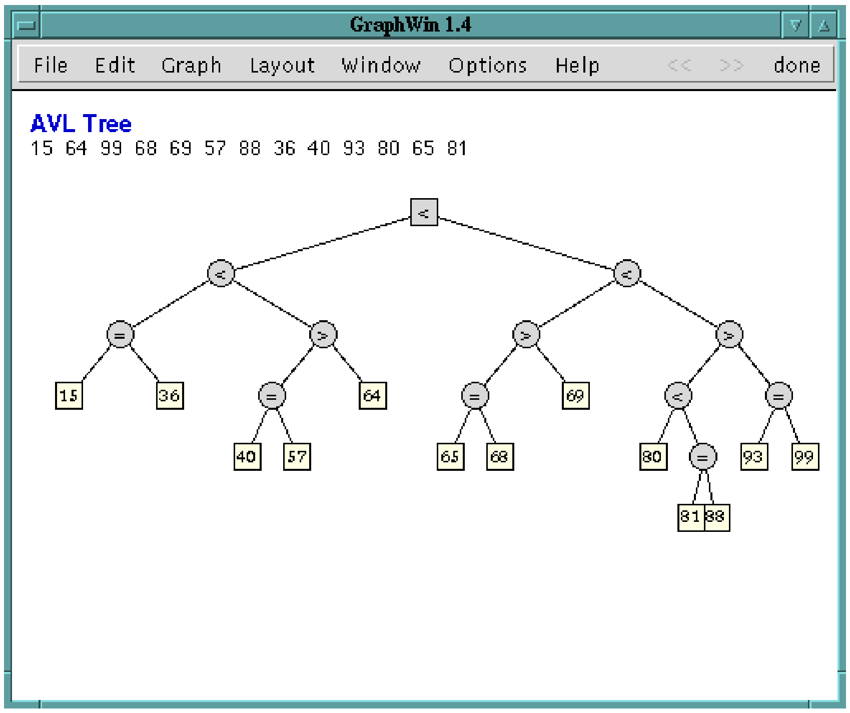


Figure 12.14 Visualization of an AVL tree.

Exercises for 12.7

1 Add menus to the main window for running and displaying the result of the different shortest-path and network flow algorithms of LEDA. Use edge sliders for the input of edge cost and capacities.

2 Design and implement an animation of the vertex addition planarity test algorithm discussed in Chapter 8.

3 Write an animation program of the generic preflow-push algorithm for computing a maximum flow in a network.

12.8 A Binary Tree Animation

We close this chapter with a demo which animates several implementations of balanced binary trees, namely AVL-trees, BB[α]-trees, and red-black trees.

All balanced binary tree implementations use a common base, the classes *bin_tree* and *bin_tree_node*. A *bin_tree* is a collection of *bin_tree_node*s. Each *bin_tree_node* stores pointers to its parent and its children, and a balance of type *int*. The interpretation of the balance of a node depends on the tree structure. In the case of AVL-trees it is the height difference between the left and right subtree, in the case of BB[α]-trees it is the number of nodes in the subtree rooted at the node, and in the case of red-black trees it encodes the color of the node. The access functions

```
int             T.get_bal(bin_tree_node*)
bin_tree_node*  T.parent(bin_tree_node*)
bin_tree_node*  T.l_child(bin_tree_node*)
bin_tree_node*  T.r_child(bin_tree_node*)
```

give access to the fields of a node. One can also ask whether a node is a root or a leaf

```
bool  T.is_root(bin_tree_node*)
bool  T.is_leaf(bin_tree_node*)
```

and one can inquire about the type and name of a tree. The name of a tree is one of "AVL Tree", "BB[alpha] Tree", ..., and the type of a tree is an integer from an enumeration type encoding the same information as the name.

```
int   T.tree_type()
char* T.tree_name()
```

A pointer to a *bin_tree_node* is a *bin_tree_item*.

The overall structure of the demo is as follows. We define the control parameters *n*, the number of insertions, *input*, the choice between random and sorted insertions, and *kind*, the type of tree to be used, we define a panel that allows us to set the control parameters, and we define three *bin_trees* and initialize them to an empty AVL-tree, BB[α]-tree, and red-black tree, respectively. We then enter a loop.

In each iteration of the loop we open the panel and ask the reader to set the control parameters. We then define an object T of class *anim_bin_tree* for the GraphWin *gw* and the tree selected by *kind*. The class *anim_bin_tree* will be discussed below and does the bulk of the work. We perform *n* insertions on T with either random inputs or increasing inputs. Finally, we display the message "Press done to continue" and put *gw* into edit mode such that the user can reply.

⟨*gw_bintree.c*⟩≡

```
#include <LEDA/graphwin.h>
#include <LEDA/impl/bin_tree.h>
#include <LEDA/impl/avl_tree.h>
#include <LEDA/impl/bb_tree.h>
#include <LEDA/impl/rb_tree.h>
#include <LEDA/impl/rs_tree.h>

#include <LEDA/map.h>

⟨class anim_bin_trees⟩

int main()
{
  GraphWin gw(500,400);

  gw.set_node_width(18);
  gw.set_node_height(18);
  gw.set_node_label_type(no_label);
  gw.set_node_label_font(roman_font,10);
  gw.set_edge_direction(undirected_edge);
  gw.set_show_status(false);
  gw.display(window::center,window::center);
  int n = 16;
  int input = 0;
  int kind  = 0;
  // define a panel P to control n, input, and kind
  panel P;
  P.text_item("\\bf\\blue Binary Tree Animation");
  P.text_item("");
  P.choice_item("tree type",kind, "avl-tree","bb-tree","rb-tree");
  P.choice_item("input data",input,"random", "1 2 3 ...");
  P.int_item("# inserts",n,0,64);
  P.button("ok",0);
  P.button("quit",1);
  bin_tree* tree[3];
  tree[0] = new avl_tree;
  tree[1] = new bb_tree;
  tree[2] = new rb_tree;
  while ( gw.open_panel(P) == 0)
  {
    anim_bin_tree T(gw,tree[kind]);
    switch (input) {
    case 0: { // random
              for(int i=0;i<n;i++) T.insert(rand_int(0,99));
              break;
             }
    case 1: { // increasing
              for(int i=0;i<n;i++) T.insert(i);
              break;
             }
   }
   gw.message("Press done to continue.");
   gw.edit();
  }
  delete[] tree;
  return 0;
}
```

It remains to explain the class *bin_tree_anim*. An object of this class consists of a reference T to a *bin_tree* and a reference *gw* to a GraphWin, a *GRAPH<point, int>* G, and a map *NODE* from tree items to graph nodes; T and *gw* are set in the constructor to references of our GraphWin and the selected tree, respectively.

The idea is that G represents a drawing of T and that *NODE* makes the translation from tree nodes to graph nodes. In the constructor we make G the graph of *gw* and set *flush* to false, and in the destructor we reset T to the empty tree. The other functions will be discussed below.

⟨*class anim_bin_trees*⟩≡

```
class anim_bin_tree {
  GraphWin& gw;
  bin_tree& T;
  GRAPH<point,int> G;
  map<bin_tree_item,node>  NODE;
  ⟨functions to compute a drawing of T⟩
public:
anim_bin_tree(GraphWin& gwin, bin_tree* tptr) : gw(gwin), T(*tptr)
{ gw.message(string("\\bf\\blue %s",T.tree_name()));
  //G.clear();
  gw.set_flush(false);
  gw.set_graph(G);
}
~anim_bin_tree() { T.clear(); }
⟨anim_bin_tree:: insert⟩
};
```

We next explain the function *scan_tree* that computes the layout and sets the visual parameters of the nodes by calling *set_node_params* for each item r of T. Setting the node parameters is easy. We draw leaves and the root as rectangles and all other nodes as ellipses. For non-leaves we display the balance of the node in an appropriate form: in the case of AVL-trees we use the labels $<$, $=$, and $>$, in the case of BB[α]-trees we display the balance, and in the case of red-black trees we display the balance as a color.

⟨*functions to compute a drawing of T*⟩≡

```
void set_node_params(bin_tree_item r)
{
  node v = NODE[r];
  if ( T.is_leaf(r) )
  { gw.set_color(v,ivory);
    gw.set_label(v,string("%d",T.key(r)));
    gw.set_shape(v,rectangle_node);
    return;
   }
  if ( T.is_root(r) )
    gw.set_shape(v,rectangle_node);
  else
    gw.set_shape(v,ellipse_node);
  gw.set_color(v,grey1);
  int bal = T.get_bal(r);
  switch ( T.tree_type() ) {
    case  LEDA_AVL_TREE:
```

```
               switch (bal) {
                 case  0: gw.set_label(v,"="); break;
                 case -1: gw.set_label(v,">"); break;
                 case  1: gw.set_label(v,"<"); break;
                }
               break;
      case  LEDA_BB_TREE:
               gw.set_label(v,string("%d",bal));
               break;
      case  LEDA_RB_TREE:
               gw.set_label_type(v,no_label);
               gw.set_color(v,(bal == 0) ? red : grey3);
               break;
     }
  }
```

The function *scan_tree* computes the layout for the subtree rooted at r and also adds the edges in the subtree to G. The subtree is placed in the rectangle with left boundary *x0*, right boundary *x1*, upper boundary y, and vertical displacement *dy* between parents and their children. Such a layout is easily computed. We set the x-coordinate of r to the midpoint of *x0* and *x1* and the y-coordinate to the upper boundary and then place the left subtree in the left half of the rectangle and the right subtree in the right half of the rectangle. In both halves we lower the upper boundary by *dy*.

⟨*functions to compute a drawing of T*⟩+≡

```
  node scan_tree(bin_tree_item r,double x0, double x1, double y, double dy)
  {
    set_node_params(r);
    node   v = NODE[r];
    double x = (x0 + x1)/2;
    G[v] = point(x,y);
    bin_tree_item left  = T.l_child(r);
    bin_tree_item right = T.r_child(r);
    if (left)  G.new_edge(v,scan_tree(left,x0,x,y-dy,dy));
    if (right) G.new_edge(v,scan_tree(right,x,x1,y-dy,dy));
    return v;
  }
```

We finally explain the insertion procedure. We lookup x; our trees store generic pointers of type *void∗* as explained in Chapter 13. We therefore need to convert x to a generic pointer. If x is already in the tree, we do nothing. Otherwise, we insert the pair $(x, 0)$ into T and store the tree item returned in p. If p is the root of T, i.e., the current insertion was the first insertion into T, we add a node to *gw* (and hence G), place it at the origin, and associate it with p. If p is not the root of T and hence the current insertion is not the first, the insertion added two nodes to the tree as shown in Figure 12.15. The node p is a leaf of T and p and $r = T.get\_last\_node(\,)$ are the new nodes of T.

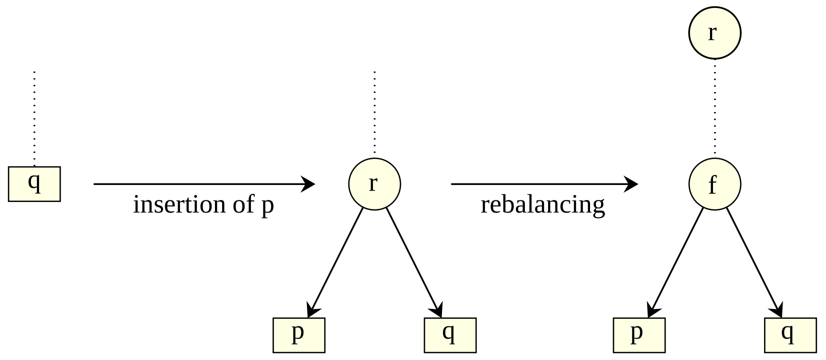


Figure 12.15 Insertion of a new key adds a new leaf p and a new node r. The search for the key of p in the old tree ended in q and the key of p is either smaller or larger than the key of q. In the former case, p will be the left child or r and in the latter case it will be the right child. After the addition of the new leaf the tree is rebalanced and r might move to a different position in the tree. A call of *T.get_last_node*( ) after the insertion returns r. We set the initial positions of p and r to the position of q before the insertion.

We add two new nodes to *gw*, one corresponding to p and the other one corresponding to r. We place both nodes on top of q. We next compute the drawing area and update the drawing. We compute the drawing area as follows. We leave four pixels unused on either side and we divide the y-extension of the window into ten (since our trees will never grow deeper than eight) strips. We leave the two top-most strips unused.

⟨*anim_bin_tree:: insert*⟩≡

```
void insert(int x)
{
  if (T.lookup(GenPtr(x))) return;
  bin_tree_item p = T.insert(GenPtr(x),0);
  if ( T.is_root(p) )
     NODE[p] = gw.new_node(point(0,0));
  else
  { bin_tree_item f = T.parent(p);
    bin_tree_item q = T.l_child(f);
    if (p == q) q = T.r_child(f);
    point pos = gw.get_position(NODE[q]);
    bin_tree_item r = T.get_last_node();
    NODE[p] = gw.new_node(pos);
    NODE[r] = gw.new_node(pos);
   }
  node v = NODE[p];
  // compute drawing area
  double dx = gw.get_window().pix_to_real(4);
  double x0 = gw.get_xmin() + dx;
  double x1 = gw.get_xmax() - dx;
  double y0 = gw.get_ymin();
  double y1 = gw.get_ymax();
```

```
    double dy = (y1-y0)/10;
    ⟨update drawing⟩
  }
```

It remains to explain how we update the drawing. We first remove all edges from G and then call *scan_tree* for the root of T and the entire drawing area. This builds T in G and computes a new layout in the node data of G. We then inform *gw* that G has changed and set the color of the new node to green. We set flush to true so that changes go into effect and change the node positions to the node data of G by the call *gw.set_position*(*G.node_data*( )). Because layout changes are animated this will make the tree move slowly into its new shape. You may change the speed in the options menu. When the tree is in its new form we reset the color of v and set flush back to false.

⟨*update drawing*⟩≡

```
    G.del_all_edges();
    scan_tree(T.root(),x0,x1,y1-2*dy,dy);
    gw.update_graph();
    color col = gw.set_color(v,green2);
    gw.set_flush(true);
    gw.set_position(G.node_data());
    gw.set_color(v,col);
    gw.set_flush(false);
```

Exercise for 12.8

1 Extend the binary tree animation of this chapter to allow deletions of keys by clicking on the corresponding leaves.

Bibliography

[Him97] M. Himsolt. The graphlet system. *Lecture Notes in Computer Science*, 1190:233–??, 1997.

[JMN] M. Jünger, P. Mutzel, and S. Näher. The AGD graph drawing library. search the WEB for AGD or one of the authors.

[LK] Lauer and M. Kaufmann. GraVis. `http://www-pr.informatik.uni-tuebingen.de/Forschung/GraVis`.

Index