

# Drawing graphs with *Graphviz*

Emden R. Gansner

30 August 2004

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	String-based layouts . . . . .	3
1.1.1	dot . . . . .	3
1.1.2	xdot . . . . .	4
1.1.3	plain . . . . .	5
1.1.4	plain-ext . . . . .	5
1.1.5	GXL . . . . .	6
1.2	<i>Graphviz</i> as a library . . . . .	6
<b>2</b>	<b>Basic graph drawing</b>	<b>6</b>
2.1	Creating the graph . . . . .	7
2.1.1	Attributes . . . . .	8
2.1.2	Cleaning up a graph . . . . .	13
2.2	Laying out the graph . . . . .	14
2.3	Getting layout information . . . . .	15
2.4	Drawing nodes and edges . . . . .	16
<b>3</b>	<b>Inside the layouts</b>	<b>16</b>
3.1	<i>dot</i> . . . . .	17
3.2	<i>neato</i> . . . . .	18
3.3	<i>fdp</i> . . . . .	20
3.4	<i>twopi</i> . . . . .	20
3.5	<i>circo</i> . . . . .	21
<b>4</b>	<b>Using the output drivers</b>	<b>21</b>
<b>5</b>	<b>Graphics code generators</b>	<b>22</b>
5.1	Style information . . . . .	27
5.2	Color information . . . . .	27
5.3	Using <code>codegen_t</code> to draw nodes and edges . . . . .	28
<b>6</b>	<b>Unconnected graphs</b>	<b>28</b>
<b>A</b>	<b>Compiling and linking</b>	<b>34</b>
A.1	Finer-grained usage . . . . .	35
A.2	Application-specific data . . . . .	36
<b>B</b>	<b>A sample program: <code>simple.c</code></b>	<b>36</b>
<b>C</b>	<b>A sample program: <code>dot.c</code></b>	<b>37</b>
<b>D</b>	<b>A sample program: <code>demo.c</code></b>	<b>38</b>
<b>E</b>	<b>String representations of types</b>	<b>39</b>

## 1 Introduction

The *Graphviz* package consists of a variety of software for drawing attributed graphs. It implements a handful of common graph layout algorithms. These are:

**dot** A Sugiyama-style hierarchical layout[STT81, GKNV93].

**neato** An implementation of the Kamada-Kawai algorithm[KK89] for “symmetric” layouts. This is a variation of multidimensional scaling[KS80, Coh87].

**fdp** An implementation of the Fruchterman-Reingold algorithm[FR91] for “symmetric” layouts. This layout is similar to neato, but there are performance and feature differences.

**twopi** A radial layout as described by Wills[Wil97].

**circo** A circular layout combining aspects of the work of Six and Tollis[ST99, ST00] and Kaufmann and Wiese[KW].

In addition, *Graphviz* provides an assortment of more general-purpose graph algorithms, such as transitive reduction, which have proven useful in the context of graph drawing.

The package was designed[GN00] to rely on the “program-as-filter” model of software, in which distinct graph operations or transformations are embodied as programs. Graph drawing and manipulation are achieved by using the output of one filter as the input of another, with each filter recognizing a common, text-based graph format. One thus has an algebra of graphs, using a scripting language to provide the base language with variables and function application and composition.

Despite the simplicity and utility of this approach, some applications need or desire to use the software as a library with bindings in a non-scripting language, rather than as primitives composed using a scripting language. The *Graphviz* software provides a variety of ways to achieve this, running a spectrum from very simple but somewhat inflexible to fairly complex but offering a good deal of application control.

### 1.1 String-based layouts

The simplest mechanism for doing this consists of using the filter approach in disguise. The application, perhaps using the `libgraph` or `libagraph` library, constructs a representation of a graph in the *DOT* language. The application can then invoke the desired layout program, e.g., using `system` or `popen` on a Unix system, passing the graph using an intermediate file or a pipe. The layout program computes position information for the graph, attaches this as attributes, and delivers the graph back to the application through another file or pipe. The application can then read in the graph, and apply the geometric information as necessary. This is the approach used by many applications, e.g., `dotty`[KN94] and `grappa`[LBM97], which rely on *Graphviz*.

There are several *Graphviz* output formats which can be used in this approach. They are specified by using a `-T` flag when invoking the layout program. The input to the programs must always be in the *DOT* language.

#### 1.1.1 dot

This format relies on the *DOT* language to describe the graphs, with attributes attached as name-value pairs.

Both the `libgraph` and `libagraph` libraries provide parsers for graphs represented in *DOT*. Using these, it is easy to read the graphs and query the desired attributes using `agget` or `agxget`. For more information on these functions, see Section 2.1.1. The string representations of the various types referred to are described in Appendix E.

On output, the graph will have a `bb` attribute of type `rectangle`, specifying the bounding box of the drawing. If the graph has a label, its position is specified by the `lp` attribute of type `point`.

Each node gets `pos`, `width` and `height` attributes. The first has type `point`, and indicates the center of the node. The `width` and `height` attributes are floating point numbers giving the node's width and height in inches. If the node has record shape, the record rectangles are given in the `rects` attribute. This has the format of a space-separated list of rectangles. If the node is a polygon (including ellipses) and the `vertices` attribute is defined for nodes, this attribute will contain the vertices of the node, in inches, as a space-separated list of `pointf` values. For ellipses, the curve is sampled, the number of points used being controlled by the `samplepoints` attribute. The points are given relative to the center of the node. Note also that the points only give the node's basic shape; they do not reflect any internal structure. If the node has `peripheries` greater than one, or a shape like "Msquare", the `vertices` attribute does not represent the extra curves or lines.

Every edge is assigned a `pos` attribute having `splineType` type. If the edge has a label, the label position is given in the `lp` of type `point`.

### 1.1.2 xdot

The `xdot` format is a strict extension of the `dot` format, in that it provides the same attributes as `dot` as well as additional drawing attributes. These additional attributes specify how to draw each component of the graph using primitive graphics operations. This can be particularly helpful in dealing with node shapes and edge arrowheads. Unlike the information provided by the `vertices` attribute, the extra attributes in `xdot` provide all geometric drawing information, including the various types of arrowheads and multiline labels with variations in alignment. In addition, all the parameters use the same units.

There are six new attributes, listed in Table 1. These drawing attributes are only attached to nodes and edges. Clearly, the last four attributes are only attached to edges.

<code>_draw_</code>	General drawing operations
<code>_ldraw_</code>	Label drawing operations
<code>_hdraw_</code>	Head arrowhead
<code>_tdraw_</code>	Tail arrowhead
<code>_hldraw_</code>	Head label
<code>_tldraw_</code>	Tail label

Table 1: `xdot` drawing attributes

The value of these attributes are strings consisting of the concatenation of some (multi-)set of the 7 drawing operations listed in Table 2.

In handling alignment, the application may want to recompute the string width using its own font drawing primitives.

The text operation is only used in the `label` attributes. Normally, the non-text operations are only used in the non-label attributes; if, however, the `decorate` attribute is set on an edge, its `label` attribute will also contain a `polyline` operation.

All coordinates and sizes are in points. Additional information such a line style, fill color, font name or size must be gleaned from the corresponding attributes of the component. Note, though, that if an edge or node is invisible, no drawing operations are attached to it.

At it stands, the `xdot` format cannot describe some of the smaller scale attributes allowed in HTML-like labels (e.g., setting the color of an individual cell). We intend to add additional fine control, especially concerning fonts. This will necessitate extending the `xdot` format to describe such drawings.

<b>E</b> $x_0 y_0 w h$	Filled ellipse with equation $((x - x_0)/w)^2 + ((y - y_0)/h)^2 = 1$
<b>e</b> $x_0 y_0 w h$	Unfilled ellipse with equation $((x - x_0)/w)^2 + ((y - y_0)/h)^2 = 1$
<b>P</b> $n x_1 y_1 \dots x_n y_n$	Filled polygon with the given $n$ vertices
<b>p</b> $n x_1 y_1 \dots x_n y_n$	Unfilled polygon with the given $n$ vertices
<b>L</b> $n x_1 y_1 \dots x_n y_n$	Polyline with the given $n$ vertices
<b>B</b> $n x_1 y_1 \dots x_n y_n$	B-spline with the given $n$ control points. $n \equiv 1 \pmod{3}$ and $n \geq 4$
<b>T</b> $x y j w n -c_1 c_2 \dots c_n$	Text drawn using the baseline point $(x, y)$ . The text consists of the $n$ bytes following ' - '. The text should be left-aligned (centered, right-aligned) on the point if $j$ is -1 (0, 1), respectively. The value $w$ gives the width of the text as computed by the library.

Table 2: xdot drawing operations

### 1.1.3 plain

The `plain` format is line-based and very simple to parse. This works well for applications which need or wish to avoid using the `libgraph` and `libagraph` libraries.

There are four types of lines: `graph`, `node`, `edge` and `stop`. The output consists of a single `graph` line; a sequence of `node` lines, one for each node; a sequence of `edge` lines, one for each edge; and a single terminating `stop` line. All units are in inches, represented by a floating point number.

As noted, the statements have very simple formats.

```
graph scale width height
node name x y width height label style shape color fillcolor
edge tail head n x1 y1 ... xn yn [label xl yl] style color
stop
```

We now describe the statements in more detail.

**graph** The *width* and *height* values give the width and height of the drawing. The lower left corner of the drawing is at the origin. The *scale* value indicates how the drawing should be scaled if a *size* attribute was given and the drawing needs to be scaled to conform to that size. If no scaling is necessary, it will be set to 1.0. Note that all `graph`, `node` and `edge` coordinates and lengths are given unscaled.

**node** The *name* value is the name of the node, and *x* and *y* give the node's position. The *width* and *height* are the width and height of the node. The *label*, *style*, *shape*, *color* and *fillcolor* values give the node's label, style, shape, color and fillcolor, respectively, using default attribute values where necessary. If the node does not have a *style* attribute, "solid" is used.

**edge** The *tail* and *head* values give the names of the head and tail nodes. *n* is the number of control points defining the B-spline forming the edge. This is followed by  $2 * n$  numbers giving the x and y coordinates of the control points in order from tail to head. If the edge has a *label* attribute, this comes next, followed by the x and y coordinates of the label's position. The edge description is completed by the edge's style and color. As with nodes, if a style is not defined, "solid" is used.

### 1.1.4 plain-ext

The `plain-ext` format is identical with the `plain` format, except that port names are attached to the node names in an edge, when applicable. It uses the usual *DOT* representation, where port *p* of node *n* is given as *p*:*n*.

### 1.1.5 GXL

The GXL [Win02] dialect of XML is a widely accepted standard for representing attributed graphs as text, especially in the graph drawing and software engineering communities. As an XML dialect, there are many tools available for parsing and analyzing graphs represented in this format. Other graph drawing and manipulation packages either use GXL as their main graph language, or provide a translator. In this, *Graphviz* is no different. We supply the programs `dot2gxl` and `gxl2dot` for converting between the DOT and GXL formats. Thus, if an application is XML-based, to use the *Graphviz* tools, it needs to insert these filters as appropriate between its I/O and the *Graphviz* layout programs.

## 1.2 *Graphviz* as a library

The role of this document is to describe how an application can use the *Graphviz* software as a library rather than as a set of programs. It will describe the intended API at various levels, concentrating on the purpose of the functions from an application standpoint, and the way the library functions should be used together, e.g., that one has to call function A before function B. The intention is not to provide detailed man pages, partly because most of the functions have a high-level interface, often just taking a graph pointer as the sole argument. The real semantic details are embedded in the attributes of the graph, which are described elsewhere.

The remainder of this manual describes how to build an application using *Graphviz* as a library in the usual sense. The next section presents the basic technique for using the *Graphviz* code. Since the other approaches are merely ramifications and extensions of the basic approach, the section also serves as an overview for all uses. Section 3 breaks each layout algorithm apart into its individual steps. With this information, the application has the option of eliminating certain of the steps. For example, all of the layout algorithms can layout edges as splines. If the application intends to draw all edges as line segments, it would probably wish to avoid the spline computation, especially as it is moderately expensive in terms of time. Section 4 explains how an application can invoke the *Graphviz* code generators, thereby generating a drawing of a graph in a concrete graphics format such as *jpeg* or *PostScript*. For an application intending to do its own rendering, Section 5 recommends a technique which allows the *Graphviz* library to handle all of the bookkeeping details related to data structures and machine-dependent representations while the application need only supply a few basic graphics functions. Section 6 discusses an auxiliary library for dealing with graphs containing multiple connected components.

We note that the interface presented here will be evolving in the future. The libraries are described “as-is”. Since the software was not built with a public library interface in mind, there are various warts which will become apparent in the description. With time, however, we hope to reform the code to provide a cleaner and more consistent interface. In addition, we are now pursuing two major design changes: evolving the `libgraph` library towards the `libagraph` library, and moving all coordinates to floating-point values using a single unit. These will also cause changes to the library interface as described here, though minimal in comparison to actual changes which will occur internally.

## 2 Basic graph drawing

Figure 1 gives a template for the basic library use of *Graphviz*, in this instance using the *dot* hierarchical layout. Basically, the program creates a graph using the `libgraph` library<sup>1</sup>, setting node and edge attributes to affect how the graph is to be drawn; calls the layout code; and then uses the position information

---

<sup>1</sup>As noted in the previous section, we intend to move the layout algorithms to the `libagraph` library, at which time, an application will use that library for manipulating graphs. To a large extent, `libagraph` is upward-compatible with `libgraph` so the discussion that follows will be largely unchanged

attached to the nodes and edges to render the graph. The remainder of this section explores these steps in more detail.

```
Agraph_t* G;

G = createGraph ();

dot_layout (G);    /* library function */
drawGraph (G);
dot_cleanup (G);  /* library function */
agclose (G);      /* library function */
```

Figure 1: Basic use

## 2.1 Creating the graph

The first step in drawing a graph is to create it. To use the *Graphviz* layout software, the graph must be created using the `libgraph` library. Before any function in `libgraph` is called, an application must call the library initialization function `aginit`<sup>2</sup>. We can create a graph in one of two ways, using `agread` or `agopen`. The former function takes a `FILE*` pointer to a file open for reading. It is assumed the file contains the description of graphs using the *DOT* language. The `agread` function parses one graph at a time, returning a pointer to an attributed graph generated from the input, or `NULL` if there are not more graphs or an error occurred.

The alternative technique is to call `agopen`.

```
char*      name;
int        type;
Agraph_t* G = agopen(name, type);
```

The first argument is the name of the graph; the second argument describes the type of graph to be created. A graph can be directed or undirected. In addition, a graph can be strict, i.e., have at most one edge between any pair of nodes, or non-strict, allowing an arbitrary number of edges between two nodes. If the graph is directed, the pair of nodes is ordered, so the graph can have edges from node A to node B as well as edges from B to A. These four combinations are specified by the values in Table 3. The return value is a new graph, with no nodes or edges.

Graph Type	Graph
AGRAPH	Non-strict, undirected graph
AGRAPHSTRICT	Strict, undirected graph
ADIGRAPH	Non-strict, directed graph
ADIGRAPHSTRICT	Strict, directed graph

Table 3: Graph types

Nodes and edges are created by the functions `agnode` and `agedge`, respectively.

<sup>2</sup>This function is called by `dotneato_initialize`, so if this latter routine is used, no additional call to `aginit` is necessary. Also, it is safe to make multiple calls to `aginit`.

```
Agnode_t *agnode(Agraph_t*, char*);
Agedge_t *agedge(Agraph_t*, Agnode_t*, Agnode_t*);
```

The first argument is the graph containing the node or edge. Note that if this is a subgraph, the node or edge will also belong to all containing graphs. The second argument to `agnode` is the node's name. This is a key for the node within the graph. If `agnode` is called twice with the same name, the second invocation will not create a new node but simply return a pointer to the previously created node with the given name. For directed graphs, the first and second node arguments to `agedge` are taken to be the tail and head nodes, respectively.

As suggested above, a graph can also contain subgraphs. These are created using `agsubg`:

```
Agraph_t *agsubg(Agraph_t*, char*);
```

The first argument is the immediate parent graph; the second argument is the name of the subgraph.

Subgraphs play three roles in *Graphviz*. First, a subgraph can be used to represent graph structure, indicating that certain nodes and edges should be grouped together. This is the usual role for subgraphs and typically specifies semantic information about the graph components. In this generality, the drawing software makes no use of subgraphs, but maintains the structure for use elsewhere within an application.

In the second role, a subgraph can provide a context for setting attribute. In *Graphviz*, these are often attributes used by the layout and rendering functions. For example, the application could specify that `blue` is the default color for nodes. Then, every node within the subgraph will have color `blue`. In the context of graph drawing, a more interesting example is:

```
subgraph {
    rank = same; A; B; C;
}
```

This (anonymous) subgraph specifies that the nodes `A`, `B` and `C` should all be placed on the same rank if drawn using *dot*.

The third role for subgraphs combines the previous two. If the name of the subgraph begins with `"cluster"`, *Graphviz* identifies the subgraph as a special *cluster* subgraph. The drawing software<sup>3</sup> will do the layout of the graph so that the nodes belonging to the cluster are drawn together, with the entire drawing of the cluster contained within a bounding rectangle.

### 2.1.1 Attributes

In addition to the abstract graph structure provided by nodes, edges and subgraphs, the *Graphviz* libraries also support graph attributes. These are simply string-valued name/value pairs. Attributes are used to specify any additional information which cannot be encoded in the abstract graph. In particular, the attributes are heavily used by the drawing software to tailor the various geometric and visual aspects of the drawing.

Reading attributes is easily done. The function `agget` takes a pointer to a graph component (node, edge or graph) and an attribute name, and returns the value of the attribute for the given component. Note that the function may return either `NULL` or a pointer to the empty string. The first value indicates that the given attribute has not been defined for any component in the graph of the given kind. Thus, if `abc` is a pointer to a node and `agget(abc, "color")` returns `NULL`, then no node in the root graph has a `color` attribute. If the function returns the empty string, this usually indicates that the attribute has been defined but the attribute value associated with the specified object is the default for the application. So, if `agget(abc, "color")` now returns `" "`, the node is taken to have the default color. In practical terms,

---

<sup>3</sup>if supported

these two cases appear very similar. Using our example, whether the attribute value is `NULL` or `" "`, the drawing code will still need to pick a color for drawing and will probably use the default in both cases.

Setting attributes is a bit more complex. Before attaching an attribute to a graph component, the code must first set up the default case. This is accomplished by a call to `agraphattr`, `agnodeattr` or `agedgeattr` for graph, node or edge attributes, respectively. The types of the 3 functions are identical. They all take a graph and two strings as arguments, and return a representation of the attribute. The first string gives the name of the attribute; the second supplies the default value, which must not be `NULL`. The graph must be the root graph.

Once the attribute has been initialized, the attribute can be set for a specific component by calling `agset` with a pointer to the component, the name of the attribute and the value to which it should be set. The attribute value must not be `NULL`.

When an attribute is assigned a value, the graph library replicates the string. This means the application can use a temporary string as the argument; it does not have to keep the string throughout the application. Each node, edge, and graph maintains its own attribute values. Obviously, many of these are the same strings, so to save memory, the graph library uses a reference counting mechanism to share strings. An application can employ this mechanism by using the `agstrdup()` function. If it does, it must also use the `agstrfree()` function if it wishes to release the string. *Graphviz* supports HTML-like tables as labels. To allow these to be handled transparently, the library uses a special version of reference counted strings. To create one of these, one uses `agstrdup_html()` rather than `agstrdup()`. The `agstrfree()` is still used to release the string.

Note that some attributes are replicated in the graph, appearing once as the usual string-valued attribute, and also in an internal machine format such as `int`, `double` or some more structured type. In general, an application should only set attributes using strings and `agset`. The implementation of the layout algorithm may change the machine-level representation or may change when it does the conversion from a string value. Hence, the low-level interface cannot be relied on by the application. Also note that there is not a one-to-one correspondence between a string-valued attributes and internal attributes. A given string attribute might be encoded as part of some data structure, might be represented via multiple fields, or may have no internal representation at all.

In order to expedite the reading and writing of attributes for large graphs, *Graphviz* provides a lower-level mechanism for manipulating attributes which can avoid hashing a string. Attributes have a representation of type `Agsym_t`. This is basically the value returned by the initialization functions `agraphattr`, etc. It can also be obtained by a call to `agfindattr`, which takes a graph component and an attribute name. If the attribute has been defined, the function returns the corresponding `Agsym_t` value. This can be used to directly access the corresponding attribute value, using the functions `agxget` and `agxset`. These are identical to `agget` and `agset`, respectively, except that instead of taking the attribute name as the second argument, they use the `index` field of the `Agsym_t` value to extract the attribute value from an array.

Due to the nature of the implementation of attributes in *Graphviz*, if possible, an application should attempt to define and initialize all attributes before creating nodes and edges.

The drawing algorithms in *Graphviz* use a large collection of attributes, giving the application a great deal of control over the appearance of the drawing. For more detailed information on what the attributes mean, the reader should consult the manual **Drawing graphs with *dot***.

We can divide the attributes into those that affect the placement of nodes, edges and clusters in the layout and those, such as color, which do not. Table 4 gives the node attributes which have the potential to change the layout. This is followed by Tables 5, 6 and 7, which do the same for edges, graphs, and clusters.

Note that in some cases, the effect is indirect. An example of this is the `nslimit` attribute, which potentially reduces the effort spent on network simplex algorithms to position nodes, thereby changing the

Name	Default	Use
bottomlabel		auxiliary label for nodes of shape M*
distortion	0.0	node distortion for shape=polygon
fixedsize	false	label text has no affect on node size
fontname	Times-Roman	font family
fontsize	14	point size of label
group		name of node's group
height	.5	height in inches
label	node name	any string
margin	0.11,0.055	space between node label and boundary
orientation	0.0	node rotation angle
peripheries	shape-dependent	number of node boundaries
pin	false	fix node at its pos attribute
regular	false	force polygon to be regular
root		indicates node should be used as root of a layout
shape	ellipse	node shape
shapefile		† external EPSF or SVG custom shape file
sides	4	number of sides for shape=polygon
skew	0.0	skewing of node for shape=polygon
toplabel		auxiliary label for nodes of shape M*
width	.75	width in inches
z	0.0	† z coordinate for VRML output

Table 4: Geometric node attributes

Name	Default	Use
constraint	true	use edge to affect node ranking
fontname	Times-Roman	font family
fontsize	14	point size of label
headclip	true	clip head end to node boundary
headport	center	position where edge attaches to head node
label		edge label
len	1.0	preferred edge length
lhead		name of cluster to use as head of edge
ltail		name of cluster to use as tail of edge
minlen	1	minimum rank distance between head and tail
samehead		tag for head node; edge heads with the same tag are merged onto the same port
sametail		tag for tail node; edge tails with the same tag are merged onto the same port
tailport	center	position where edge attaches to tail node
weight	1	importance of edge

Table 5: Geometric edge attributes

Name	Default	Use
center	false	† center drawing on page
clusterrank	local	may be global or none
compound	false	allow edges between clusters
concentrate	false	enables edge concentrators
defaultdist	$1 + (\sum_{e \in E} len) /  E  \sqrt{ V }$	separation between nodes in different components
dim	2	dimension of layout
dpi	96/0	dimension of layout
epsilon	.0001 V  or .0001	termination condition
fontname	Times-Roman	font family
fontpath		list of directories to such for fonts
fontsize	14	point size of label
label		† any string
margin		† space placed around drawing
mclimit	1.0	scale factor for mincross iterations
mindist	1.0	minimum distance between nodes
mode	major	variation of layout
model	shortpath	model used for distance matrix
nodesep	.25	separation between nodes, in inches
nslimit		if set to <i>f</i> , bounds network simplex iterations by <i>(f)(number of nodes)</i> when setting x-coordinates
ordering		specify out or in edge ordering
orientation	portrait	† use landscape orientation if rotate is not used and the value is landscape
pack		do components separately, then pack
packmode	node	granularity of packing
page		† unit of pagination, e.g. "8.5, 11"
quantum		if quantum > 0.0, node label dimensions will be rounded to integral multiples of quantum
rank		same, min, max, source or sink
rankdir	TB	sense of layout, i.e, top to bottom, left to right, etc.
ranksep	.75	separation between ranks, in inches.
ratio		approximate aspect ratio desired, fill or auto
remincross		If true and there are multiple clusters, re-run crossing minimization
root		specifies node to be used as root of a layout
rotate		† If 90, set orientation to landscape
searchsize	30	maximum edges with negative cut values to check when looking for a minimum one during network simplex
sep	0.01	factor to increase nodes when removing overlap
size		maximum drawing size, in inches
splines		render edges using splines
start	random	manner of initial node placement
voron_margin	0.05	factor to increase bounding box when more space is necessary during Voronoi adjustment
viewport		†Clipping window

Table 6: Geometric graph attributes

Name	Default	Use
fontname	Times-Roman	font family
fontsize	14	point size of label
label		edge label
peripheries	1	number of cluster boundaries

Table 7: Geometric cluster attributes

layout. Some of these attributes affect the initial layout of the graph in universal coordinates. Others only play a role if the application uses the *Graphviz* code generators (cf. Section 4), which map the drawing into device-specific coordinates related to a concrete output format. For example, *Graphviz* only uses the `center` attribute, which specifies that the graph drawing should be centered within its page, when the library generates a concrete representation. The tables distinguish these device-specific attributes by a † symbol at the start of the Use column.

Tables 8, 9, 10 and 11 list the node, edge, graph and cluster attributes, respectively, that do not effect the placement of components. Obviously, the values of these attributes are not reflected in the position information of the graph after layout. If the application handles the actual drawing of the graph, it must decide if it wishes to use these attributes or not.

Name	Default	Use
<code>color</code>	black	node shape color
<code>fillcolor</code>	lightgrey	node fill color
<code>fontcolor</code>	black	text color
<code>layer</code>	overlay range	all, <i>id</i> or <i>id:id</i>
<code>nojustify</code>	false	context for justifying multiple lines of text
<code>style</code>		style options, e.g. <code>bold</code> , <code>dotted</code> , <code>filled</code> ;

Table 8: Decorative node attributes

Name	Default	Use
<code>arrowhead</code>	normal	style of arrowhead at head end
<code>arrowsize</code>	1.0	scaling factor for arrowheads
<code>arrowtail</code>	normal	style of arrowhead at tail end
<code>color</code>	black	edge stroke color
<code>decorate</code>		if set, draws a line connecting labels with their edges
<code>dir</code>	forward	forward, back, both, or none
<code>fontcolor</code>	black	type face color
<code>headlabel</code>		label placed near head of edge
<code>labelangle</code>	-25.0	angle in degrees which head or tail label is rotated off edge
<code>labeldistance</code>	1.0	scaling factor for distance of head or tail label from node
<code>labelfloat</code>	false	lessen constraints on edge label placement
<code>labelfontcolor</code>	black	type face color for head and tail labels
<code>labelfontname</code>	Times-Roman	font family for head and tail labels
<code>labelfontsize</code>	14	point size for head and tail labels
<code>layer</code>	overlay range	all, <i>id</i> or <i>id:id</i>
<code>nojustify</code>	false	context for justifying multiple lines of text
<code>style</code>		drawing attributes such as <code>bold</code> , <code>dotted</code> , or <code>filled</code>
<code>taillabel</code>		label placed near tail of edge

Table 9: Decorative edge attributes

Among these attributes, some are used more frequently than others. A graph drawing typically needs to encode various application-dependent properties in the representations of the nodes. This can be done with text, using the `label`, `fontname` and `fontsize` attributes; with color, using the `color`, `fontcolor`, `fillcolor` and `bgcolor` attributes; or with shapes, the most common attributes being `shape`, `height`, `width`, `style`, `fixedsize`, `peripheries` and `regular`,

Edges often display additional semantic information with the `color` and `style` attributes. If the edge is directed, the `arrowhead`, `arrowsize`, `arrowtail` and `dir` attributes can play a role. Using splines

Name	Default	Use
bgcolor		background color for drawing, plus initial fill color
fontcolor	black	type face color
labeljust	left-justified	"r" for right-justified cluster labels
labelloc	top	"r" for right-justified cluster labels
layers		names for output layers
layersep	" :~"	separator characters used in layer specification
nojustify	false	context for justifying multiple lines of text
pagedir	BL	traversal order of pages
samplepoints	8	number of points used to represent ellipses and circles on output
stylesheet		XML stylesheet

Table 10: Decorative graph attributes

Name	Default	Use
bgcolor		background color for cluster
color	black	cluster boundary color
fillcolor	black	cluster fill color
fontcolor	black	text color
labeljust	left-justified	"r" for right-justified cluster labels
labelloc	top	"r" for right-justified cluster labels
nojustify	false	context for justifying multiple lines of text
pencolor	black	cluster boundary color
style		style options, e.g. <code>bold</code> , <code>dotted</code> , <code>filled</code> ;

Table 11: Decorative cluster attributes

rather than line segments for edges, as determined by the `splines` attribute, is done for aesthetics or clarity rather than to convey more information.

There are also a number of frequently used attributes which affect the layout geometry of the nodes and edges. These include `compound`, `len`, `lhead`, `ltail`, `minlen`, `nodesep`, `pin`, `pos`, `rank`, `rankdir`, `ranksep` and `weight`. Within this category, we should also mention the `pack` and `overlap` attributes, though they have a somewhat different flavor.

The attributes described thus far are used as input to the layout algorithms. There is a collection of attributes, displayed in Table 12, which, by convention, *Graphviz* uses to specify the geometry of a layout. After an application has used *Graphviz* to determine position information, if it wants to write out the graph

Name	Use
bb	bounding box of drawing or cluster
lp	position of graph, cluster or edge label
pos	position of node or edge control points
rects	rectangles used in records
vertices	points defining node's boundary

Table 12: Output position attributes

in *DOT* with this information, it should use the same attributes.

In addition to the attributes described above which have visual effect, there is a collection of attributes used to supply identification information or web actions. Table 13 lists these.

### 2.1.2 Cleaning up a graph

Once all layout information is obtained from the graph, the resources should be reclaimed. This is a two-step process. First, the application should call the cleanup routine associated with the layout algorithm used

Name	Use
URL	hyperlink associated with node, edge, graph or cluster
comment	comments inserted into output
headURL	URL attached to head label
headhref	synonym for headURL
headtarget	browser window associated with headURL
headtooltip	tooltip associated with headURL
href	synonym for URL
tailURL	URL attached to tail label
tailhref	synonym for tailURL
tailtarget	browser window associated with tailURL
tailtooltip	tooltip associated with tailURL
target	browser window associated with URL
tooltip	tooltip associated with URL

Table 13: Miscellaneous attributes

to draw the graph. The names all have the same form. Thus, the cleanup routine for *dot* is `dot_cleanup`. Second, the generic graph resources are reclaimed by closing the graph using `agclose`.

The example of Figure 1 handles the case where the application is drawing a single graph. The example given in Appendix C shows how cleanup might be done when processing multiple graphs.

The application can determine best when it should clean up. The example in the appendix performs this just before a new graph is drawn, but the application could have done this much earlier, for example, immediately after the graph is drawn using `drawGraph`. Note, though, that layout information will be destroyed during cleanup. If the application needs to reuse this data, for example, to refresh the display, it should avoid calling the cleanup function, or arrange to copy the layout data elsewhere. Also, in the simplest case where the application just draws one graph and exits, there is no need to do cleanup at all.

If a given graph is to be laid out multiple times, there is no need to reconstruct the graph. The application, however, should call the layout's cleanup function before invoking the same or a new layout function.

## 2.2 Laying out the graph

Once the graph exists and the attributes are set, the application can pass the graph to one of the *Graphviz* layout functions:

- `dot_layout`
- `neato_layout`
- `fdp_layout`
- `twopi_layout`
- `circo_layout`

These will assign node positions, represent edges as splines<sup>4</sup>, handle the special case of an unconnected graph, plus deal with various technical features such as preventing node overlaps.

For some applications, however, even these functions do more than what is desired. In that case, the application will want to consider each step in a layout and decide whether it is appropriate. Section 3 describes in detail the steps used by each algorithm.

<sup>4</sup>Line segments are represented as degenerate splines.

## 2.3 Getting layout information

Once the layout is done, the graph data structures contain the position information for drawing the graph. An application-supplied drawing routine such as `drawGraph` can then read this information, map it to display coordinates, and call the device-specific routines to render the drawing. If the application chose to avoid the high-level *Graphviz* layout functions, and applied the steps individually, certain position information might be missing.

In this section, we describe in reasonable detail various data structures used by *Graphviz* to specify how to draw the graph. An application can use these values directly to guide its drawing. In some cases, for example, with arrowheads attached to `bezier` values, a complete description of how to interpret the data is beyond the current scope of this manual. For this reason as well as simplicity, if an application wishes to provide all of the graphics features while avoiding the low-level details of the data structures, we suggest using the approach described in Sections 5.3.

In general, the `libgraph` library allows an application to define specific data fields which are compiled into the node, edge and graph structures. These have the names

- `Agnodeinfo_t`
- `Agedgeinfo_t`
- `Agraphinfo_t`

respectively. The *Graphviz* layout algorithms rely on a specific set of fields to record position and drawing information, and therefore provide definitions for the three fields. Thus, the definitions of the information fields are fixed by the layout library and cannot be altered by the application.<sup>5</sup>

These information structures occur as the field named `u` in the node, edge and graph structure. The definition of the information structures as defined by the layout code is given in `types.h`, along with various auxiliary types such as `point` or `bezier`. This file also provides macro expressions for accessing these fields. Thus, if `np` is a node pointer, the `width` field can be read using `np->u.width` or `ND_width(np)`. Edge and graph attributes follow the same convention, with prefixes `ED_` and `GD_`, respectively. We strongly deprecate the former access method, especially in light of the transition of the drawing algorithms to an extension of `libgraph` by using the macro access, source code will not be affected by this change.

Each node has `ND_coord`, `ND_width` and `ND_height` attributes. The value of `ND_coord` gives the position of the center of the node, in points. The `ND_width` and `ND_height` attributes specify the size of the bounding box of the node, in inches.

Edges, even if a line segment, are represented as B-splines or piecewise Bezier curves. The `spl` attribute of the edge stores this spline information. It has a pointer to an array of 1 or more `bezier` structures. Each of these describes a single piecewise Bezier curve as well as associated arrowhead information. Normally, a single `bezier` structure is sufficient to represent an edge. If, however, the `concentrate` attribute is set, whereby mostly parallel edges are represented by a shared spline, the edge may need multiple `bezier` parts. Of course, the application always has the possibility of drawing a line segment connecting the centers of edge's nodes.

---

<sup>5</sup>This is a limitation of the `libgraph` library. We plan to remove this restriction by moving to a mechanism similar to the one used in `libagraph` which allows arbitrary dynamic extensions to the node, edge and graph structures. Meanwhile, if the application requires the addition of extra fields, it can define its own structures, which should be extensions of the components or the information types, with the additional fields attached at the end. Then, instead of calling `aginit()`, it can use the more general `aginitlib()`, and supply the sizes of its nodes, edges and graphs. This will ensure that these components will have the correct sizes and alignments. The application can then cast the generic `libgraph` types to the types it defined, and access the additional fields.

If a subgraph is specified as a cluster, the nodes of the cluster will be drawn together and the entire subgraph is contained within a rectangle containing no other nodes. The rectangle is specified by the `bb` attribute of the subgraph, the coordinates in points in the global coordinate system.

## 2.4 Drawing nodes and edges

With the position and size information described above, an application can draw the nodes and edges of a graph. It could just use rectangles or circles for nodes, and represent edges as line segments or splines. However, nodes and edges typically have a variety of other attributes, such as color or line style, which an application can read from the appropriate fields in `Agnodeinfo_t` and `Agedgeinfo_t` and use in its rendering.

Additional drawing information about the node depends mostly on the shape of the node. For record-type nodes, where `ND_shape(n)->name` is "record" or "Mrecord", the node consists of a packed collection of rectangles. In this case, `ND_shape_info(n)` can be cast to `field_t*`, which describes the recursive partition of the node into rectangles. The value `b` of `field_t` gives the bounding rectangle of the field, in points in the coordinate system of the node, i.e., where the center of the node is at the origin.

If `ND_shape(n)->usershape` is true, the shape is specified by the user. Typically, this is format dependent, e.g., the node might be specified by a GIF image, and we ignore this case for the present.

The final node class are those with polygonal shape<sup>6</sup>, which includes the limiting cases of circles and ellipses. In this case, `ND_shape_info(n)` can be cast to `polygon_t*`, which specifies the many parameters (number of sides, skew and distortions, etc.) used to describe polygons, as well as the points used as vertices. Note that the vertices are in inches, and are in the coordinate system of the node.

To honor a node's shape, an application has three basic choices. It can implement the geometry for each of the different shapes. Thus, it could see that `ND_shape(n)->name` is "box", and use the `ND_coord`, `ND_width` and `ND_height` attributes to draw rectangle at the given position with the given width and height. Another second would be to use the specification of the shape as stored internally in the `shape_info` field of the node. For example, given a polygonal node, its `find_shape_info(n)` field contains a `vertices` field, mentioned above, which is an ordered list of all the vertices used to draw the appropriate polygon, taking into account multiple peripheries. Finally, there is a third possibility available in *Graphviz*. Given a node pointer `n`, it is possible for an application to draw this node by calling the library function `emit_node`. This will automatically handle all the geometric details as well the additional node attributes such as color and fonts. This technique also works for edges, using `emit_edge`. The details of the approach are described in Section 5.3.

The label field (`ND_label(n)`, `ED_label(e)`, `GD_label(g)`) encodes any text label associated with a graph object. Edges, graphs and clusters will occasionally have labels; nodes almost always have a label, since the default label is the node's name. The basic label string is stored in the `text` field, while the `fontname`, `fontcolor` and `fontsize` fields describe the basic font characteristics. In many cases, the basic label string is further parsed, either into multiple, justified text lines, or as a nested box structure for HTML-like labels or nodes of record shape.

## 3 Inside the layouts

For graph layout within an application, it is usually adequate to invoke the top-level entry point, such as `dot_layout`. In certain cases, though, an application may wish to impose finer control on the layout process, or avoid certain steps which are irrelevant to its presentation. For these reasons, this section describes the individual passes used within each algorithm.

---

<sup>6</sup>This is not quite true but close enough for now.

Here, we will assume that the graph is connected. All of the layouts handle unconnected graphs. Sometimes, though, an application may not want to use the built-in technique. For these cases, *Graphviz* provides tools for decomposing a graph, and then combining multiple layouts. This is described in Section 6.

In all of algorithms, the first step is to call a layout-specific `init_graph` function. For example, in the case of *dot*, we call `dot_init_graph`). These functions initialize the graph for the particular algorithm. This will first call common routines to set up basic data structures, especially those related to the final layout results and code generation. In particular, the size and shape of nodes will have been analyzed and set at this point, which the application can access via the `ND_width`, `ND_height`, `ND_ht`, `ND_lw`, `ND_rw`, `ND_shape`, `ND_shape_info` and `ND_label` attributes. Initialization will then establish the data structures specific to the given algorithm. Both the generic and specific layout resources are released when the corresponding cleanup function (e.g., `dot_cleanup`) is called (cf. Section 2.1.2). As a rule, a layout's `init_graph` routine should always be paired with its `cleanup` function. The former guarantees correctly allocated and initialized data structures, while the latter releases the allocated space, preventing memory leaks.

A layout algorithm will typically set low-level parameters in its `init_graph` function, converting the string-valued parameters specified in the graph. Thus, if an application insists on setting the field in a data structure directly, it should only do this after the graph is initialized. If done earlier, there is a chance the value will be overridden by the initialization routine.

Almost always, the penultimate step is to generate the edges. As this may be expensive to compute and irrelevant to an application, this is one call an application may decide to avoid. Obviously, if this function is not called, the application should not expect any edge position information. On the other hand, it should be safe to call `dotneato_postprocess` or the *Graphviz* renderers.

The *dot* algorithm has its own special function `dot_splines` for handling edge generation. Its algorithm allows it to integrate edge placement with node layout. Most of the other layouts make a call to `spline_edges`, which uses the *Graphviz* path planning library to generate spline for edges if required. The only exception is *fdp*, which calls `spline_edges0` instead. This avoids some initial calculations in `spline_edges` which are unnecessary in *fdp*.

The algorithms all end with `dotneato_postprocess`. The role of this function is to do some final tinkering with the layout, still in layout coordinates. Specifically, the function rotates the layout for *dot* if `rankdir` is set, attaches the root graph's label, if any, and normalizes the drawing so that the lower left corner of its bounding box is at the origin. In addition to the graph, the function takes an algorithm-specific function used for setting node sizes. Each algorithm has defined one of these.

### 3.1 *dot*

The *dot* algorithm produces a ranked layout of a graph honoring edge directions if possible. It is particularly appropriate for displaying hierarchies or directed acyclic graphs. The basic layout scheme is attributed to Sugiyama et al.[STT81] The specific algorithm used by *dot* follows the steps described by Gansner et al.[GKNV93]

```
dot_init_graph(g);
dot_rank(g);
dot_mincross(g);
dot_position(g);
dot_sameports(g);
dot_splines(g);
dot_compoundEdges (g);
dotneato_postprocess(g, dot_nodesize);
```

After graph initialization (`dot_init_graph`), the algorithm assigns each node to a discrete rank (`dot_rank`) using an integer program to minimize the sum of the (discrete) edge lengths. The next step (`dot_mincross`) rearranges nodes within ranks to reduce edge crossings. This is followed by the assignment (`dot_position`) of actual coordinates to the nodes, using another integer program to compact the graph and straighten edges. At this point, all nodes will have a position set in the `coord` attribute. In addition, the bounding box `bb` attribute of all clusters are set.

The `dot_sameports` step is an addition to the basic layout. It implements the feature, based on the edge attributes "samehead" and "sametail", by which certain edges sharing a node all connect to the node at the same point.

To generate edge representations, the application can call `dot_splines`. At present, *dot* draws all edges as B-splines, though some edges will actually be the degenerate case of a line segment.

Although *dot* supports the notion of cluster subgraphs, its model does not correspond to general compound graphs. In particular, a graph cannot have edges connecting two clusters, or a cluster and a node. The layout can emulate this feature. Basically, if the head and tail nodes of an edge lie in different, non-nested clusters, the edge can specify these clusters as a logical head or logical tail using the `lhead` or `ltail` attribute. The spline generated in `dot_splines` for the edge can then be clipped to the bounding box of the specified clusters.

### 3.2 *neato*

The layout computed by *neato* is specified by a virtual physical model, i.e., one in which nodes are treated as physical objects influenced by forces, some of which arise from the edges in the graph. The layout is then derived by finding positions of the nodes which minimize the forces or total energy within the system. The forces need not correspond to true physical forces, and typically the solution represents some local minimum. Such layouts are sometimes referred to as symmetric, as the principal aesthetics of such layouts tend to be the visualization of geometric symmetries within the graph. To further enhance the display of symmetries, such drawings tend to use line segments for edges.

The model used by *neato* comes from Kamada and Kawai[KK89], though it was first introduced by Kruskal and Seely[KS80] in a different format. The model assumes there is a spring between every pair of vertices, each with an ideal length. The ideal lengths are a function of the graph edges. The layout attempts to minimize the energy in this system.

```
neato_init_graph(g);
neatoLayout (g, layoutMode, model);
adjustNodes(g);
spline_edges(g);
dotneato_postprocess(g, neato_nodesize);
```

As usual, we start with a call to `neato_init_graph`. former function is that the dimension of the layout *Ndim* is determined from the graph's "dim" attribute, with the default being a two-dimensional layout.

The call to actually perform the layout requires 2 additional parameters to specify both the mode and the distance model used. If one uses `layoutMode = MODE_MAJOR`, which is what *neato* uses by default, the optimization is performed by stress majorization[GKN04]. If one uses `layoutMode = MODE_KK`, the algorithm employs the solution technique proposed by Kamada and Kawai[KK89]. The latter mode is typically slower than the former, and introduces the possibility of cycling. It is maintained solely for backward compatibility.

The model indicates how the ideal distances are computed between all pairs of nodes. Usually, *neato* uses a shortest path model (`model = MODEL_SHORTPATH`), so that the length of the spring between

nodes  $p$  and  $q$  is the length of the shortest path between them in the graph. Note that the shortest path calculation takes into account the lengths of edges as specified by the "len" attribute, with one inch being the default.

If `MODE_KK` is used and the graph attribute `pack` is false, *neato* sets the distance between nodes in separate connected components to  $1.0 + L_{avg} \cdot \sqrt{|V|}$ , where  $L_{avg}$  is the average edge length and  $|V|$  is the number of nodes in the graph. This supplies sufficient separation between components so that they do not overlap. Typically, the larger components will be centrally located, while smaller components will form a ring around the outside.

In some cases, an application may decide to use circuit model (`model = MODEL_CIRCUIT`), a model based on electrical circuits as first proposed by Cohen[Coh87]. In this model, the spring length is derived from resistances using Kirchoff's law. This means that the more paths between  $p$  and  $q$  in the graph, the smaller the spring length. This has the effect of pulling clusters closer together. We note that this approach only works if the graph is connected. If the graph is not connected, the layout automatically reverts to the shortest path model.

The third model is the subset model (`model = MODEL_SUBSET`). This sets the length of each edge to be the number of nodes that are neighbors of exactly one of the end points, and then calculates remaining distances using shortest paths. This helps to separate nodes with high degree.

The library provides two utility functions

```
int neatoMode (Agraph_t* g);
int neatoModel (Agraph_t* g);
```

which can be used to query the graph attributes `mode` and `model`, respectively, and return the corresponding `mode` and `model` integer values.

The basic algorithm used by *neato* performs the layout assuming point nodes. Since in many cases, the final drawing uses text labels and various node shapes, the drawing ends up with many nodes overlapping each other. For certain uses, the effect is desirable. If not, the application can use `adjustNodes` to reposition the nodes to eliminate overlaps. There are 2 methods available, depending on the graph attribute "overlap". One[GN99] uses a Voronoi-diagram based approach, which requires the least additional space, but can greatly distort the original layout. At the other extreme, `adjustNodes` allows scaling[MSTH]. This exactly preserves the shape of the layout but at the expense of much space. We note the obvious fact that, in both methods, the nodes sizes are preserved; only the node positions are altered.

With nodes positioned, the algorithm can proceed to draw the edges using the `spline_edges` function. By default, edges are drawn as line segments. If, however, the "splines" graph attribute is set to true, `spline_edges` will construct the edges as splines[DGKN97], routing them around the nodes. Topologically, the spline follows the shortest path between two nodes while avoiding all others. Clearly, for this to work, there can be no node overlaps. If overlaps exist, edge creation reverts back to line segments. When this function returns, the positions of the nodes will be recorded in their `coords` attribute, in points.

The programmer should be aware of certain limitations and problems with the *neato* algorithm. First, as noted above, if `layoutMode = MODE_KK`, it is possible for the minimization technique used by *neato* to cycle, never finishing. At present, there is no way for the library to detect this, though once identified, it can easily be fixed by simply picking another initial position. Second, although multiedges affect the layout, the spline router `spline_edges` does not handle them. Thus, two edges between the same nodes will receive the same spline. Finally, *neato* provides no mechanism for drawing clusters. If clusters are required, one should use the *fdp* algorithm, which belongs to the same family as *neato* and is described next.

### 3.3 *fdp*

The *fdp* layout is similar in appearance to *neato* and also relies on a virtual physical model, this time proposed by Fruchterman and Reingold[FR91]. This model uses springs only between nodes connected with an edge, and an electrical repulsive force between all pairs of nodes. Also, it achieves a layout by minimizing the forces rather than energy of the system.

Unlike *neato*, *fdp* supports cluster subgraphs. In addition, it allows edges between clusters and nodes, and between cluster and clusters. At present, an edge from a cluster cannot connect to a node or cluster with the cluster.

```
fdp_init_graph (g);
fdpLayout (g);
spline_edges0(g);
dotneato_postprocess(g, neato_nodesize);
```

The layout scheme is fairly simple: initialization; layout; a call to route the edges; and postprocessing. In *fdp*, because it is necessary to keep clusters separate, the removal of overlaps is (usually) obligatory and there is no explicit call to `adjustNodes`.

### 3.4 *twopi*

The radial layout algorithm represented by *twopi* is conceptually the simplest in *Graphviz*. Following an algorithm described by Wills[Wil97], it takes a node specified as the center of the layout and the root of the generated spanning tree. The remaining nodes are placed on a series of concentric circles about the center, the circle used corresponding to the graph-theoretic distance from the node to the center. Thus, for example, all of the neighbors of the center node are placed on the first circle around the center. The algorithm allocates angular slices to each branch of the induced spanning tree to guarantee enough space for the tree on each ring.

It should be obvious from the description that the basic version of the *twopi* algorithm relies on the graph being connected. If this is not the case, the application can use the technique described in Section 6. At present, the algorithm does not attempt to visualize clusters.

```
Agnode_t*  ctr;          /* center node of layout */

twopi_init_graph(g);
circleLayout (g,ctr);
adjustNodes (g);
spline_edges(g);
dotneato_postprocess(g, twopi_nodesize);
```

As usual, the layout commences by initializing the graph (`twopi_init_graph`). The entire layout is handled by a call to `circleLayout`. Note that it is up to the application to supply a center node. If `ctr == NULL`, the algorithm will select some “most central” node, i.e., one whose minimum distance from a leaf node is maximal. Upon the function’s return, the x and y coordinates of the node can be found, in inches, in the attributes `pos[0]` and `pos[1]`, respectively.

As with *neato*, if the application invokes `adjustNodes`, the library will, if specified, adjust the layout to avoid node-node overlaps. Again as with *neato*, a call to `spline_edges` will compute drawing information for edges. See Section 3.2 for more details.

### 3.5 circo

The *circo* algorithm is based on the work of Six and Tollis[ST99, ST00], as modified by Kaufmann and Wiese[KW]. The nodes in each biconnected component are placed on a circle, with some attempt to minimize edge crossings. Then, by considering each component as a single node, the derived tree is laid out in a similar fashion to *twopi*, with some component considered as the root node.

```

circo_init_graph(g);
circoLayout(g);
spline_edges(g);
dotneato_postprocess(g, circo_nodysize);

```

As with *fdp*, the scheme is very simple. By construction, the *circo* layout avoids node overlaps, so no use `adjustNodes` is necessary.

## 4 Using the output drivers

If an application decides to avail itself of the code generators in the *Graphviz* libraries, the simplest way is to pass a graph which has been laid out to the function `dotneato_write`. This will generate and write the appropriate graphics instructions.

The *Graphviz* architecture uses a data structure `GVC_t` to define a rendering environment or context. Among other parameters, this specifies what output formats are created and in which files. A single `GVC_t` value can be used with multiple graphs. An instance of an environment can be created by a call to

```
extern GVC_t *gvNEWcontext(char **info, char *user);
```

The first argument is an array of 3 character pointers providing version information; see Appendix A.2 for a description of this data. The second argument is a string giving a name for the user. If desired, the application can call the library function `username` to obtain this value.

For convenience, the *Graphviz* library provides a simple way to create an environment:

```
extern GVC_t *gvContext();
```

This uses the `Info` created when *Graphviz* was built, plus an empty string for the user's name.

To initialize the environment, the application should call the function `dotneato_initialize`:

```
extern void dotneato_initialize(GVC_t* gvc, int argc, char* argv[]);
```

This function takes the environment value, plus an array of strings. It calls `aginit` to initialize the `libgraph` library, and then uses the values in `argv` to set the appropriate values in `gvc`. These strings should be the flags commonly accepted by the various *Graphviz* programs such as `-T` and `-o`.

For example, the application can use a synthetic argument list

```

GVC_t* gvc = gvContext();
char* args[] = {
    "command",
    "-Tgif",          /* gif output */
    "-oabc.gif"     /* output to file abc.gif */
};
dotneato_initialize (gvc, sizeof(args)/sizeof(char*), args);

```

to specify GIF output written to the file `abc.gif`. Another approach is to use the program's actual argument list, after removing flags not handled by *Graphviz*.

Most of the information is stored in `gvc` for use later. However, if the `argv` array contains non-flag arguments, these are taken to be input files and are stored in `char* Files[]`. This is a NULL-terminated array of all the non-flag values.

Sometime before calling `dotneato_write`, the application needs to bind the environment and graph together. For this, it uses the function:

```
extern void gvBindContext(GVC_t* gvc, Agraph_t* g);
```

With this setup, and after a call to a layout algorithm, the application can generate output by calling `dotneato_write`:

```
extern void dotneato_write (GVC_t *gvc);
```

This function relies on a variety of implicit parameters which are used to control the concrete representation. Most of these are specified as graph attributes. These are noted in Tables 4, 5 and 6.

If an application uses `dotneato_write`, it should afterwards call `dotneato_eof`. This tells the code generator that output is finished, so it can emit any needed trailers. If appropriate, this can be achieved by a call to `dotneato_terminate`, which invokes `dotneato_eof` and then exits with an exit value reflecting any errors that arose in using the *Graphviz* libraries.

The Appendices B, C, and D show how all of these pieces fit together.

## 5 Graphics code generators

Except for the simple text output formats (`dot`, `xdot`, `plain` and `plain-ext`), all graph output done in *Graphviz* goes through a driver specified by the type `codegen_t`. In addition to the drivers built into the library, an application can provide its own, allowing it to specialize or control the output as necessary. If an application has defined a value of type `codegen_t`, it can set `gvc->codegen` to point to it and then call

```
emit_graph (g, flags);
```

where `g` is the root graph. This will traverse the graph structure, emitting instructions for drawing nodes, edges and clusters using the operations supplied in `gvc->codegen`. The `flags` parameter can be used to control in what order certain aspects of the drawing are done. It is the bitwise-or of the flags

**EMIT\_SORTED** If set, first all nodes are drawn, then all edges.

**EMIT\_EDGE\_SORTED** If set, first all edges are drawn, then all nodes.

**EMIT\_COLORS** If set, before any drawing starts, the generator registers all colors used in the graph by calls to `set_fillcolor` and `set_pencolor`.

**EMIT\_CLUSTERS\_LAST** If set, the generator emits clusters (i.e., the cluster's bounding rectangle and optional label) only after all nodes and edges are drawn. By default, clusters are emitted before any of the nodes or edges.

The flags take into account the various idiosyncracies of some graphics formats. Obviously, only one of `EMIT_SORTED` or `EMIT_EDGE_SORTED` should be set. If neither is set, the output intermingles the drawing of nodes and edges using a breadth-first search.

Before describing the code generator functions in detail, it may be helpful to give an overview of how output is done. Output can be viewed as a hierarchy of components. At the highest level is the job, representing an output format and target. Bound to a job might be multiple graphs, each embedded in some universal space. Each graph may be partitioned into multiple layers. Each layer is divided into a 2-dimensional array of pages. A page will then contain nodes, edges, and clusters. Each of these may contain an anchor. They will all construct a local graphics context before doing a rendering. During code generation, each component is reflected in paired calls to its corresponding `begin...` and `end...` functions. The layer and anchor components are omitted if there is only a single layer or the enclosing component has no browser information.

Rendering is defined by a small collection of graphics primitives and a few functions for setting graphics attributes.

Table 14 lists the names and type signatures of the fields of `codegen_t`, which are used to emit the components described above. In the following, we describe the functions in more detail, though many are self-explanatory. All positions and sizes are in points.

`reset` Reinitialize code generator; called between jobs.

`begin_job(ofp, g, lib, user, info, pages)` Called at the beginning of all graphics output for a graph, following a call to `emit_graph` or `dotneato_write`. This is needed in addition to `begin_graph` because the graph may be drawn using multiple pages. The code generator should use the open file pointer `ofp` for output. Note that this parameter is not passed as an argument to any other functions, so the generator must store this value. The parameter `g` indicates the graph to be drawn. The `user` argument gives identifying information, usually including the login ID, of the owner of the process. The `info` argument gives name and version information about the drawing program (cf. Section A.2).

The graph will be printed in an array of `size` pages. If `pages.x` or `pages.y` is greater than one, this indicates that a page size was set and the graph drawing is too large to be printed on a single page.

The `lib` argument is a null-terminated array of strings which can pass user or application code or information to the code generator. The use of `lib` is output-specific. For example, the PostScript generator uses it to extend or override the standard PostScript functions with ones supplied by the user.

`end_job()` Called at the end of all graphics output for graph. The output stream is still open, so the code generator can append any final information to the output.

`begin_graph(g, bb, pb)` Called at the beginning of drawing graph `g`. The device or format specific output space is the rectangle whose corners are the origin and the point `pb`. Within this space, the parameter `bb` gives the bounding box of the drawing. The origin in layout space should map to `bb.LL` in format or device space.

`end_graph()` Called when the drawing of a graph is complete.

`begin_page(g, page, scale, rot, offset)` Called at the beginning of a new output page. The page will contain part of the drawing of graph `g`. The point `page` is the index of the page in the array of pages. Thus, page (0,0) is the page containing the bottom, lefthand corner of the graph drawing; page (1,0) will contain that part of the graph drawing to the right of page (0,0); etc.

The remaining parameters instruct the code generator in additional coordinate transformations it is responsible for. `rot` gives the angle, in degrees, through which the drawing should be rotated. At present, `rot` is either 0 or 90, representing portrait or landscape modes. The parameters `scale` and

```
void reset()
void begin_job(FILE*, graph_t*, char**, char*, char**, point)
void end_job()
void begin_graph(graph_t*, box, point)
void end_graph()
void begin_page(graph_t*, point, double, int, point)
void end_page();
void begin_layer(char*, int, int)
void end_layer()
void begin_cluster(graph_t*)
void end_cluster()
void begin_nodes()
void end_nodes()
void begin_edges()
void end_edges()
void begin_node(node_t*)
void end_node()
void begin_edge(edge_t*)
void end_edge()
void begin_anchor(char*, char*, char*)
void end_anchor()
void begin_context()
void end_context()
void set_font(char*, double)
void textline(point, textline_t*)
void set_pencolor(char*)
void set_fillcolor(char*)
void set_style(char**)
void ellipse(point, int, int, int)
void polygon(point*, int, int)
void beziercurve(point*, int, int, int)
void polyline(point*,int)
boolean bezier_has_arrows
void user_shape(char*, point*, int, int)
void comment(void*, attrsym_t*)
point textsize(char*, char*, double)
point usershapesize(node_t*, char*)
```

Table 14: Interface for a code generator

`offset` indicate how layout coordinates should be scaled and translated. Scaling is always uniform in `x` and `y`, with `scale == 1.0` indicating no scaling.

`end_page()` Called when the drawing of a current page is complete.

`begin_layer(layerName, n, nLayers)` Called at the beginning of each layer, only if `nLayers > 0`. The `layerName` parameter is the logical layer name given in the `layers` attribute. The layer has index `n` out of `nLayers`.

`end_layer()` Called at the end of drawing the current layer.

`begin_cluster(g)` Called at the beginning of drawing cluster subgraph `g`.

`end_cluster()` Called at the end of drawing the current subgraph.

`begin_nodes()` Called at the beginning of drawing the nodes on the current page. Only called if the `flags` parameter to `emit_graph` has `EMIT_SORTED` or `EMIT_EDGE_SORTED` set.

`end_nodes()` Called when all nodes on a page have been drawn. Only called if the `flags` parameter to `emit_graph` has `EMIT_SORTED` or `EMIT_EDGE_SORTED` set.

`begin_edges()` Called at the beginning of drawing the edges on the current page. Only called if the `flags` parameter to `emit_graph` has `EMIT_SORTED` or `EMIT_EDGE_SORTED` set.

`end_edges()` Called when all edges on the current page are drawn. Only called if the `flags` parameter to `emit_graph` has `EMIT_SORTED` or `EMIT_EDGE_SORTED` set.

`begin_node(n)` Called at the start of drawing node `n`.

`end_node()` Called at the end of drawing the current node.

`begin_edge(e)` Called at the start of drawing edge `e`.

`end_edge()` Called at the end of drawing the current edge.

`begin_anchor(href, tooltip, target)` Called at the start of an anchor context associated with the current node, edge, or graph, assuming the graph object has a `URL` or `href` attribute. The `href` parameter gives the associated href, while `tooltip` and `target` supply any tooltip or target information. If the object has no tooltip, its label will be used. If the object has no target attribute, this parameter will be `NULL`.

`end_anchor()` Called at the end of the current anchor context.

`begin_context()` Called at the start of a new graphics context or state. The context encapsulates the graphics attributes drawing or pen color, fill color, font (face and size), and supported styles. The context is defined by the various set routines below. Contexts should be kept in a stack. If an attribute has not been set in the current context, its value should be looked up further down the stack.

`end_context()` Called at the end of the current graphics context. The context should be removed, and the previous context now becomes the current one.

`set_font(fontname, fontsize)` Set the context so that text will be drawn using font `fontname` of size `fontsize` points. The `fontname` parameter is usually a short, common name representing a font face such as "Times-Roman", which is the default font in *Graphviz*. The interpretation of the font name, whether mapped to a font predefined in the format or to a file, is format specific.

`set_pencolor(name)` Set the color for line or text drawing to `name`. See Section 5.2 for the possible values and meanings for `name`.

`set_fillcolor(name)` Set the color for area filling to `name`. See Section 5.2 for the possible values and meanings for `name`.

`set_style(s)` Set the style of the node or edge. The argument `s` is a null-terminated array of strings specifying style features such as "invis", "dashed" or "filled". Although some style attributes are almost universally supported, styles are format-dependent. See Section 5.1 for information on how to parse the array.

`textsize(str, fontname, fontsz)` Returns the size, in points, of the bounding rectangle for the text string `str` if drawn using font `fontname` with point size `fontsz`. If the function is not defined, the library will attempt to estimate the size.

`textline(p, txt)` Draw text at point `p` using the current font and fontsize and color. The `txt` argument provides the text string `txt.str`, a calculated width of the string `txt.width` and the horizontal alignment `txt.just` of the string in relation to `p`.

The base line of the text is given by `p.y`. The interpretation of `p.x` depends upon the value of `txt.just`. Basically, `p.x` provides the anchor point for the alignment.

<code>txt.just</code>	<code>p.x</code>
'n'	Center of text
'l'	Left edge of text
'r'	Right edge of text

The leftmost `x` coordinate of the text, the parameter most graphics systems use for text placement, is given by `p.x + j * txt.width`, where `j` is 0.0 (-0.5,-1.0) if `txt.just` is 'l' ('n', 'r'), respectively. This representation allows the code generator to accurately compute the point for text placement that is appropriate for its format, as well as use its own mechanism for computing the width of the string.

`ellipse(p, rx, ry, filled)` Draw an ellipse with center at `p`, with horizontal and vertical half-axes `rx` and `ry` using the current pen color and line style. If `filled` is non-zero, the ellipse should be filled with the current fill color.

`polygon(A, n, filled)` Draw a polygon with the `n` vertices given in the array `A`, using the current pen color and line style. If `filled` is non-zero, the ellipse should be filled with the current fill color.

`beziercurve(A, n, arrow_at_start, arrow_at_end)` Draw a B-spline with the `n` control points given in `A`. This will consist of  $(n-1)/3$  cubic Bezier curves. The spline should be drawn using the current pen color and line style. If `bezier_has_arrows` is false, the parameters `arrow_at_start` and `arrow_at_end` will both be 0. Otherwise, if `arrow_at_start` (`arrow_at_end`) is true, the function should draw an arrowhead at the first (last) point of `A`.

`polyline(A, n)` Draw a polyline with the `n` vertices given in the array `A`, using the current pen color and line style.

`bezier_has_arrows` This should be set to true if the function `beziercurve` will generate the appropriate arrowheads. If false, the *Graphviz* drawing routines will emit graphics code to draw the arrowhead.

`comment(obj, sym)` Unused

`user_shape(name, A, n, filled)` Called by the library to draw a node of shape `name` whose contents are user-defined using a format-dependent mechanism. Some formats require the node's `shapefile` attribute to define a file or URL containing the node's content. Note that if the generator needs additional information or attributes from the node being drawn, it should store the node pointer when its `begin_node` function is called.

The region allocated for the node is a polygon with `n` sides, whose vertices are given by the array `A`. (At present, the polygon is always a rectangle.) If `filled` is non-zero, the node should be filled with the current fill color.

This function should always attempt to honor the area specified by `A`, perhaps scaling the node contents to fit the region, even if a `usershapesize` function is available to provide the library with the exact size. The reason for this is that the node might have its `fixedsize` attribute set to true.

`usershapesize(n, name)` If defined, called by the library to ascertain the size of a user-defined shape. If the shape of a node is not recognized, the library assumes this is a user-defined shape and will use this function to determine its size. The parameter `n` is the node involved, and `name` is the value of the node's `shape` attribute. A user-defined node is always taken to be rectangular.

A code generator need not supply a `usershapesize` function to support user-defined nodes. If the function is not available, the library will set the node size using the usual mechanism, taking into account the `label`, `width` and `height` attributes of the node.

## 5.1 Style information

One of the functions in the `codegen_t` interface is the `set_style` function, which takes a null-terminated array `s` of character pointers, each specifying a style attribute. Usually, a style will consist of a single, null-terminated string of characters, specifying a style operand such as "bold". More generally, the operand may be followed by one or more null-terminated strings, specifying arguments to the operand. The entire style specification is terminated by a null character. For example, `s[0]` might point to the string "setlinewidth\0004\000", which could be used get more flexibility with line thicknesses beyond a simple "bold" style. Note that the character array ends in two null characters, one ending the argument "4", the other terminating the style specification.

As noted above, the interpretation and use of styles depends on the code generator.

## 5.2 Color information

There are four ways a color can be specified in *Graphviz*: RGB, RGB + alpha, HSV and color name, where color names are those supplied with the X library. Each of these has its own string representation, and the core part of *Graphviz* keeps them as strings. It is up to each code generator to interpret what color is denoted by a string.

In most cases, this can be done using the function

```
void colorxlate(char*, color_t*, color_type);
```

supplied by the library. If invoked as

```
colorxlate(str, color, target_type)
```

the `str` argument is the string representation of a color which would come from an attribute such as `fontcolor`. The `target_type` specifies which machine-level representation is desired. Possible values of `target_type` are:

**HSV\_DOUBLE** HSV format represented as 3 doubles from 0 to 255.0.

**RGBA\_BYTE** RGB + alpha format represented as 4 bytes from 0 to 255.

**CMYK\_BYTE** CMYK format represented as 4 bytes from 0 to 255.

**RGBA\_WORD** RGB + alpha format represented as 4 bytes from 0 to 65535.

The result is stored into the `color_t` value pointed to by `color`. The `color_t` type is simply a union of the four machine-level representations supported.

### 5.3 Using `codegen_t` to draw nodes and edges

An application can use a call `emit_graph` after setting its own `codegen_t` to generate the entire graph using its rendering. In certain cases, especially interactive applications, it may be desirable to have finer control. This can be done by using calls to `emit_node` and `emit_edge` instead of calling `emit_graph`.

The application still needs to define and install its `codegen_t`. For node drawing, the library uses just a subset of the these functions. These are:

- `begin_node`, `begin_anchor`, `begin_context`, `end_context`, `end_anchor`, `end_node`, `set_style`, `set_pencolor`, `set_fillcolor`, `set_font`, `textline ellipse`, `polygon`

For edge drawing, the required functions are:

- `begin_edge`, `begin_anchor`, `begin_context`, `end_context`, `end_anchor`, `end_edge`, `set_style`, `set_pencolor`, `set_font`, `textline`, `beziercurve`, `polyline`

The last function is only used if an edge has its `ornament` attribute set. The functions `set_font` and `textline` are only used if an edge has labels.

## 6 Unconnected graphs

All of the basic layouts provided by *Graphviz* are based on a connected graph. Each is then extended to handle the not uncommon case of having multiple components. Most of the time, the obvious approach is used: draw each component separately and then assemble the drawings into a single layout. The only place this is not done is in *neato* when the mode is `MODE_KK` and `pack` is false (cf. Section 3.2).

For the *dot* algorithm, its layered drawings make the merging simple: the nodes on the highest rank of each component are all put on the same rank. For the other layouts, it is not obvious how to put the components together.

The *Graphviz* software provides the library `libpack` to assist with unconnected graphs, especially by supplying a technique for packing arbitrary graph drawings together quickly, aesthetically and with efficient use of space. The following code indicates how the library can be integrated with the basic algorithm (cf. Section 3.4).

```

Agnode_t*  c = NULL;
Agraph_t** ccs;          /* array of connected components */
Agraph_t*  sg;

```

```

Agnode_t*  c = NULL;
int        ncc;          /* number of connected components */
int        i;

ccs = ccomps (g, &ncc, (char*)0);
if (ncc == 1) {
    circleLayout (g,ctr);
    adjustNodes (g);
    spline_edges(g);
}
else {
    pack_info pinfo;
    pack_mode pmode = getPackMode (g,l_node);

    for (i = 0; i < ncc; i++) {
        sg = ccs[i];
        if (ctr && agcontains (sg, ctr)) c = ctr;
        else c = 0;
        nodeInduce (sg);
        circleLayout (sg,c);
        adjustNodes (sg);
    }
    spline_edges(g);
    pinfo.margin = getPack (g, CL_OFFSET, CL_OFFSET);
    pinfo.doSplines = 1;
    pinfo.mode = pmode;
    pinfo.fixed = 0;
    packSubgraphs (ncc, ccs, g, &pinfo);
}
for (i = 0; i < ncc; i++) {
    agdelete (g, ccs[i]);
}

```

The call to `ccomps` splits the graph into its connected components. `ncc` is set to the number of components. The components are represented by subgraphs of the input graph, and are stored in the returned array. The function gives names to the components in a way that should not conflict with previously existing subgraphs. If desired, the third argument to `ccomps` can be used to designate what the subgraphs should be called. Also, for flexibility, the subgraph components do not contain the associated edges.

Certain layout algorithms, such as *neato*, allow the input graph to fix the position of certain nodes, indicated by `ND_pinned(n)` being non-zero. In this case, all nodes with a fixed position need to be laid out together, so they should all occur in the same “connected” component. The `libpack` library provides `pccomps`, an analogue to `ccomps` for this situation. It has almost the same interface as `ccomps`, but takes a `boolean*` third parameter. The function sets the boolean pointed to to true if the graph has nodes with fixed positions. In this case, the component containing these nodes is the first one in the returned array.

Continuing with the example, if there is only one component, we can revert to the base implementation for simplicity. If there are multiple components, we take one at a time, using `nodeInduce` to create the corresponding node-induced subgraph, laying out the component with `circleLayout`, and removing node overlaps, if necessary, by calling `adjustNodes`. As a technical point, if a center node is provided,

it is only used with the component containing it. A better implementation would allow a center node to be specified on a per component basis.

After the nodes of each component are positioned, with coordinates stored in the attributes `pos[0]` and `pos[1]`, the code calls `spline_edges` to generate the edge representations.

Next, it uses the `libpack` function `packSubgraphs` to reassemble the graph into a single drawing. To position the components, `libpack` uses the polyomino-based approach described by Freivalds et al[FDK02]. The first three arguments to the function are clear. The fourth argument sets various parameters used in the packing. The `pinfo.margin` field specifies the margin, in points, maintained between components. The value `CL_OFFSET` is defined by *Graphviz*; it is the same amount of space used by *dot* around clusters. Here, we employ the auxiliary function `getPack`. This uses the graph attribute "pack" to determine this value, which can take boolean and integer values. If the attribute evaluates to a non-negative integer, this value is returned. If the attribute evaluates to true, the third argument is returned. Otherwise, the function returns the second value.

The `pinfo.doSplines` field, if non-zero, tells the function that edge representations have already been computed for the graph and should be used in determining the packing. Otherwise, the packing will treat edges as line segments connecting the centers of the two endpoints.

The `pinfo.mode` field specifies how the packing should be done. At present, packing uses the single algorithm mentioned above, but allows three varying granularities, represented by the values `l_node`, `l_clust` and `l_graph`. In the first case, packing is done at the node and edge level. This provides the tightest packing, using the least area, but also allows a node of one component to lie between two nodes of another component. The second value, `l_clust`, requires that the packing treat top-level clusters with a set bounding box `GD_bb` value like a large node. Nodes and edges not entirely contained within a cluster are handled as in the previous case. This prevents any components which do not belong to the cluster from intruding within the cluster's bounding box. The last case does the packing at the graph granularity. Each component is treated as one large node, whose size is determined by its bounding box.

In our example, we use another library function, `getPackMode`, to set the mode value. This function uses the graph's "packmode" attribute to determine the value. If this is "node", "cluster" or "graph", the function returns `l_node`, `l_clust` and `l_graph`, respectively. Otherwise, the function returns its second argument.

The last field, `pinfo.fixed`, is used to constrain the placement of certain components. If non-NULL, this field should point to an array of `ncc` booleans, where `pinfo.fixed[i]` is true if component `i` should be left at its current position. If the application specifies fixed components, these are placed first. Then the remaining components are packed into the unoccupied space, respecting the `pinfo.mode` field. It is the application's responsibility to make sure that the fixed components do not overlap each other, if that is desired.

Note that the library automatically computes the bounding box of each of the components. Also, as a side-effect, `packSubgraphs` finishes by recomputing and setting the bounding box attribute `GD_bb` of the graph.

The final step is to free the component subgraphs.

Although *dot* and *neato* have their specialized approaches to unconnected graphs, it should be noted that these are not without their deficiencies. The approach used by *dot*, aligning the drawings of all components along the top, works well until the number of components grows large. When this happens, the aspect ratio of the final drawing can become very bad. *neato*'s handling of an unconnected graph can have two drawbacks. First, there can be a great deal of wasted space. The value chosen to separate components is a simple function of the number of nodes. With a certain edge structure, component drawings may use much less area. This can produce a drawing similar to a classic atom: a large nucleus surrounded by a ring of electrons with a great deal of empty space between them. Second, the *neato* model is essentially quadratic. If the

components are drawn separately, one can see a dramatic decrease in layout time, sometimes several orders of magnitudes. For these reasons, it sometimes makes sense to apply the *twopi* approach for unconnected graphs to the *dot* and *neato* layouts. In fact, as we've noted, `neato_layout` typically uses the `libpack` library by default.

## References

- [Coh87] J. Cohen. Drawing graphs to convey proximity: an incremental arrangement method. *ACM Transactions on Computer-Human Interaction*, 4(11):197–229, 1987.
- [DGKN97] D. Dobkin, E. Gansner, E. Koutsofios, and S. North. Implementing a general-purpose edge router. In G. DiBattista, editor, *Proc. Symp. Graph Drawing GD'97*, volume 1353 of *Lecture Notes in Computer Science*, pages 262–271, 1997.
- [FDK02] K. Freivalds, U. Dogrusoz, and P. Kikusts. Disconnected graph layout and the polyomino packing approach. In P. Mutzel et al., editor, *Proc. Symp. Graph Drawing GD'01*, volume 2265 of *Lecture Notes in Computer Science*, pages 378–391, 2002.
- [FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph Drawing by Force-directed Placement. *Software – Practice and Experience*, 21(11):1129–1164, November 1991.
- [GKN04] E. Gansner, Y. Koren, and S. North. Graph drawing by stress majorization. In *Proc. Symp. Graph Drawing GD'04*, September 2004.
- [GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *IEEE Trans. Software Engineering*, 19(3):214–230, May 1993.
- [GN99] Emden R. Gansner and Stephen North. Improved force-directed layouts. In S. Whitesides, editor, *Proc. Symp. Graph Drawing GD'98*, volume 1547 of *Lecture Notes in Computer Science*, pages 364–373, Montreal, Canada, 1999. Springer-Verlag.
- [GN00] E.R. Gansner and S.C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30:1203–1233, 2000.
- [KK89] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989.
- [KN94] Eleftherios Koutsofios and Steve North. Applications of Graph Visualization. In *Proceedings of Graphics Interface*, pages 235–245, May 1994.
- [KS80] J. Kruskal and J. Seery. Designing network diagrams. In *Proc. First General Conf. on Social Graphics*, pages 22–50, 1980.
- [KW] M. Kaufmann and R. Wiese. Maintaining the mental map for circular drawings. In M. Goodrich, editor, *Proc. Symp. Graph Drawing GD'02*, volume 2528 of *Lecture Notes in Computer Science*, pages 12–22.
- [LBM97] W. Lee, N. Barghouti, and J. Mocenigo. Grappa: A graph package in Java. In G. DiBattista, editor, *Proc. Symp. Graph Drawing GD'97*, volume 1353 of *Lecture Notes in Computer Science*, 1997.
- [MSTH] K. Marriott, P.J. Stuckey, V. Tam, and W. He. Removing node overlapping in graph layout using constrained optimization. *Constraints*.
- [ST99] Janet Six and Ioannis Tollis. Circular drawings of biconnected graphs. In *Proc. ALENEX 99*, pages 57–73, 1999.

- [ST00] Janet Six and Ioannis Tollis. A framework for circular drawings of networks. In *Proc. Symp. Graph Drawing GD'99*, volume 1731 of *Lecture Notes in Computer Science*, pages 107–116. Springer-Verlag, 2000.
- [STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Trans. Systems, Man and Cybernetics*, SMC-11(2):109–125, February 1981.
- [Wi197] G. Wills. Nicheworks - interactive visualization of very large graphs. In G. DiBattista, editor, *Symposium on Graph Drawing GD'97*, volume 1353 of *Lecture Notes in Computer Science*, pages 403–414, 1997.
- [Win02] A. Winter. Gxl - overview and current status. In *Procs. International Workshop on Graph-Based Tools (GraBaTs)*, October 2002.

## A Compiling and linking

All of the necessary include files and libraries are available in the `include` and `lib` directories where *Graphviz* is installed. At the simplest level, all an application needs to do to use the layout algorithms is to include `dotneato.h`, which declares all of the *Graphviz* types and functions, compile the code, and link the program with the necessary libraries.

By convention, for the Windows environment, the libraries and include files are found in the `lib` and `include` directories, respectively. On Unix systems, they will occur in `graphviz/lib` and `graphviz/include`. Depending on the what settings are used, however, these files may occur elsewhere.

For linking, the application should use the *Graphviz* libraries

- `dotneato`
- `dotgen`
- `neatogen`
- `fdpgen`
- `twopigen`
- `circogen`
- `pack`
- `common`
- `gvrender`
- `pathplan`
- `gd`<sup>7</sup>
- `graph`
- `cdt`

In addition, the following additional libraries should be linked in:

- `freetype2`
- `jpeg`
- `png`
- `z`

For Windows, the *Graphviz* binary package provides these latter libraries as `ft.lib` `libexpat.lib` `libexpatw.lib` `jpeg.lib` `png.lib` `z.lib`. Also, in some environments, it may be necessary to link in the standard C math library.

If *Graphviz* is built and installed with the GNU build tools, there is a `dotneato-config` script created in the `bin` directory which can be used to obtain the include file and library information for a given installation. If GNU `make` is used, a sample `Makefile` for building the programs `dot` and `demo` could have the form:

---

<sup>7</sup>As of version 1.14, the `gd` library provided by *Graphviz* is essentially the same as the standard version. Whether or not the *Graphviz* version is built depends on how the software is configured at build time. Also, the *Graphviz* version may be named `gvgd`.

```

COMPILE=libtool --tag=CC --mode=compile ${CC} -c
LINK=libtool --tag=CC --mode=link ${CC}

CFLAGS='dotneato-config --cflags'
LDFLAGS='dotneato-config --libtool'

all: simple dot demo

simple: simple.lo
    ${LINK} ${LDFLAGS} -o $@ simple.lo

simple.lo: simple.c
    ${COMPILE} ${CFLAGS} -o $@ simple.c

dot: dot.lo
    ${LINK} ${LDFLAGS} -o $@ dot.lo

dot.lo: dot.c
    ${COMPILE} ${CFLAGS} -o $@ dot.c

demo: demo.lo
    ${LINK} ${LDFLAGS} -o $@ demo.lo

demo.lo: demo.c
    ${COMPILE} ${CFLAGS} -o $@ demo.c

clean:
    rm -rf .libs simple dot demo *.o *.lo

```

The code for `simple.c`, `dot.c`, and `demo.c` are given in Appendices B, C, and D.<sup>8</sup> They provide three typical examples of the use of *Graphviz*.

## A.1 Finer-grained usage

On occasion, the programmer may wish to be more precise as to which *Graphviz* include files and libraries are used. For example, the application may need a function from another library whose name conflicts with a function in an unused *Graphviz* library. The complete interface is provided by:

```

#include <render.h>
#include <dotprocs.h>
#include <neatoproc.h>
#include <adjust.h>
#include <circle.h>
#include <circo.h>
#include <fdp.h>
#include <pack.h>

```

---

<sup>8</sup>They can also be found, along with the Makefile, in the `dot.demo` directory of the *Graphviz* source.

The file `render.h` provides the basic types and the common functions, as well as the entry points for format-dependent output. The files `dotprocs.h`, `neatprocs.h`, `circle.h`, `circo.h`, and `fdp.h` define the drawing routines specific to *dot*, *neato*, *twopi*, *circo*, and *fdp*, respectively. The entry points for removing node overlaps is given by `adjust.h`. Finally, `pack.h` declares the functions for splitting graphs into connected components, and packing individual layouts together.

Concerning libraries, the `dotgen`, `neatogen`, `fdpgen`, `circogen`, and `twopigen` libraries provide the functions specific to the corresponding layout algorithms. Note, though, that the *twopi*, *circo*, and *fdp* layouts also require the `neatogen` library. The `pack` library provides the code for handling disconnected graphs. It is not necessary if only the *dot* algorithm is used. The `common` library defines the many shared functions used in all of the algorithms. It requires the `graph` library, which in turn relies on the `cdt` library. The `gvrender` library provides an functions for hooking in the renderers and plug-in libraries. The `dotneato` library provides the array `char* Info[3]`; which identifies the *Graphviz* version and can be used as an argument when creating a `GVC_t` value. Finally, the `pathplan` and `gd` libraries define functions for edge routing and bitmap rendering. At present, these are usually required even if the application does not appear to use these features.

Obviously, depending on what is used in the application, various of the include files and libraries will not be needed. In particular, if an application does not plan to use any of the concrete code generators supplied by *Graphviz*, the software can be built without any of the optional libraries `jpeg`, `png`, `z`, or `freetype`, and these can be avoided in linking. On the other hand, if *Graphviz* was built with some or all of these libraries, linking will have to use the appropriate libraries. The `expat` library is used by `common`, though it is possible to build *Graphviz* without this.

## A.2 Application-specific data

When possible, the code generators in the library log how an image file was created in the output file. To do this, they rely on the application providing an array

```
extern char* Info[3];
```

giving the desired version information. The three strings should be the name of the application, the version of the application, and a build date. For example, *dot* might provide

```
char *Info[] = {
    "dot",           /* Program */
    "1.8.10",       /* Version */
    "6 Dec 2002"    /* Build Date */
};
```

A default definition for `Info` is provided in the auxiliary library `dotneato`, so an application need only load this library if desired. **MOVE FORWARD, also note `gvContext` function**

## B A sample program: `simple.c`

This following code illustrates an application which only uses *Graphviz* to position a graph. The application will either provide its own rendering, perhaps in an interactive setting, or, as in this case, it does not render the graph at all.

```
#include <dotneato.h>
```

```

int main(int argc, char** argv)
{
Agraph_t* g;
Agnode_t* n;
FILE*     fp;
char      buf[BUFSIZ];
point     p;

aginit ();

if (argc > 1)
fp = fopen (argv[1], "r");
else
fp = stdin;
g = agread (fp);

dot_layout(g);

agnodeattr(g, "pos", "");
for (n = agfstnode(g); n; n = agnxtnode(g,n)) {
p = ND_coord_i(n);
sprintf(buf, "%d,%d", p.x, p.y);
agset(n, "pos", buf);
printf ("node %s at position (%s)\n", n->name, buf);
}
agwrite(g, stdout);

dot_cleanup(g);
return 0;
}

```

## C A sample program: dot.c

This example shows how an application might read a stream of input graphs, lay out each, and then use the *Graphviz* renderers to write the drawings to an output file. Indeed, this is precisely how the *dot* program is written, ignoring some signal handling, its specific declaration of the `Info` data (cf. Section A.2), and a few other minor details. If someone desired to write a new layout algorithm, this code could be copied directly, merely changing the algorithm-specific layout and cleanup functions `dot_layout` and `dot_cleanup`. A more detailed and realistic example is provided in Appendix D.

```

#include <dotneato.h>

int main(int argc, char** argv)
{
    Agraph_t *g, *prev=NULL;
    GVC_t *gvc;

```

```

gvc = gvContext();

dotneato_initialize(gvc,argc,argv);
while ((g = next_input_graph())) {
    if (prev) {
        dot_cleanup(prev);
        agclose(prev);
    }
    prev = g;

    gvBindContext(gvc, g);

    dot_layout(g);
    dotneato_write(gvc);
}
dotneato_terminate(gvc);
return 0;
}

```

## D A sample program: demo.c

This example provides a modification of the previous example. Again it relies on the *Graphviz* renderers, but now it creates the graph dynamically rather than reading the graph from a file.

```

#include <dotneato.h>

int main(int argc, char** argv)
{
    Agraph_t *g;
    Agnode_t *n,*m;
    Agedge_t *e;
    Agsym_t *a;
    GVC_t *gvc;

    /* set up renderer context */
    gvc = gvContext();

    /* Accept -T and -o options like dot.
     * Input files are ignored in this demo. */
    dotneato_initialize(gvc, argc, argv);

    /* Create a simple digraph */
    g = agopen("g",AGDIGRAPH);
    n = agnode(g,"n");
    m = agnode(g,"m");
    e = agedge(g,n,m);

    /* Set an attribute - in this case one that affects the visible rendering */

```

```

if (!(a = agfindattr(g->proto->n, "color")))
    a = agnodeattr(g, "color", "");
agxset(n, a->index, "red");

/* bind graph to GV context - currently must be done before layout */
gvBindContext(gvc, g);

/* Compute a layout */
neato_layout(g);

/* Write the graph according to -T and -o options */
dotneato_write(gvc);

/* Clean out layout data */
/* neato_cleanup(g); */

/* Free graph structures */
agclose(g);

/* Clean up output file and errors */
dotneato_terminate(gvc);

return 0;
}

```

## E String representations of types

The following gives the accepted string representations corresponding to values of the given types. Whitespace is ignored when converting these values from strings to their internal representations.

`point "x,y"` where  $(x,y)$  are the integer coordinates of a position in points (72 points = 1 inch).

`pointf "x,y"` where  $(x,y)$  are the floating-point coordinates of a position in inches.

`rectangle "llx, lly, urx, ury"` where  $(llx, lly)$  is the lower left corner of the rectangle and  $(urx, ury)$  is the upper right corner, all in integer points.

`splineType` A semicolon-separated list of `spline` values.

`spline` This type has an optional end point, an optional start point, and a space-separated list of  $N = 3n + 1$  points for some positive integer  $n$ . An end point consists of a `point` preceded by "e, "; a start point consists of a `point` preceded by "s, ". The optional components are separated by spaces.

The terminating list of points  $p_1, p_2, \dots, p_N$  gives the control points of a B-spline. If a start point is given, this indicates the presence of an arrowhead. The start point touches one node of the corresponding edge and the direction of the arrowhead is given by the vector from  $p_1$  to the start point. If the start point is absent, the point  $p_1$  will touch the node. The analogous interpretation holds for an end point and  $p_N$ .