Using *Graphviz*as a Library (cgraph version)

Emden R. Gansner

February 28, 2013

Contents

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 A "symmetric" layout algorithm based on stress reduction. This is a variation of multidimensional scaling [KS80, Coh87]. The default implementation uses stress majorization [GKN04]. An alternate implementation uses the Kamada-Kawai algorithm [KK89]

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

sfdp A multiscale force-directed layout using a spring-electrical model [Hu05].

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].

patchwork An implementation of squarified treemaps [BHvW00].

osage A layout algorithm for clustered graphs based on user specifications.

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 *Graphviz* `cgraph` 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. As with all output formats, 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.

The `cgraph` library provides a parser for graphs represented in *DOT*. Using this, 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 in points. The `width` and `height` attributes are floating point numbers giving the width and height, in inches, of the node's bounding box. If the node has a 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 described above, 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.

`_draw_`	General drawing operations
`_ldraw_`	Label drawing operations
`_hdraw_`	Head arrowhead
`_tdraw_`	Tail arrowhead
`_hldraw_`	Head label
`_tldraw_`	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. The color, font name, and style values supplied in the C, c, F, and S operations have the same format and interpretation as the `color`, `fontname`, and `style` attributes in the source graph.

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 graphics operations are only used in the non-label attributes. If, however, a node has `shape="record"` or an HTML-like label

`E` $x_0\ y_0\ w\ h$	Filled ellipse with equation $((x-x_0)/w)^2+((y-y_0)/h)^2=1$
`e` $x_0\ y_0\ w\ h$	Unfilled ellipse with equation $((x-x_0)/w)^2+((y-y_0)/h)^2=1$
`P` $n\ x_1\ y_1\ \ldots\ x_n\ y_n$	Filled polygon with the given n vertices
`p` $n\ x_1\ y_1\ \ldots\ x_n\ y_n$	Unfilled polygon with the given n vertices
`L` $n\ x_1\ y_1\ \ldots\ x_n\ y_n$	Polyline with the given n vertices
`B` $n\ x_1\ y_1\ \ldots\ x_n\ y_n$	B-spline with the given n control points. $n \equiv 1 mod 3$ and $n \geq 4$
`b` $n\ x_1\ y_1\ \ldots\ x_n\ y_n$	Filled B-spline with the given n control points. $n \equiv 1 mod 3$ and $n \geq 4$
`T` $x\ y\ j\ w\ n\ -c_1c_2\cdots c_n$	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.
`C` $n\ -c_1c_2\cdots c_n$	Set color used to fill closed regions. The color is specified by the n characters following `'-'`.
`c` $n\ -c_1c_2\cdots c_n$	Set pen color, the color used for text and line drawing. The color is specified by the n characters following `'-'`.
`F` $s\ n\ -c_1c_2\cdots c_n$	Set font. The font size is s points. The font name is specified by the n characters following `'-'`.
`S` $n\ -c_1c_2\cdots c_n$	Set style attribute. The style value is specified by the n characters following `'-'`.

Table 2: `xdot` drawing operations

is involved, a label attribute may also contain various graphics operations. In addition, if the `decorate` attribute is set on an edge, its `label` attribute will also contain a polyline operation.

All coordinates and sizes are in points. If an edge or node is invisible, no drawing operations are attached to it.

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 `cgraph` library. The price for this simplicity is that the format encodes very little detailed layout information beyond basic position information. If an application needs more than what is supplied in the format, it should use the `dot` or `xdot` format.

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* $x_1\ y_1$ ... $x_n\ y_n$ *[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 *n*`:`*p*.

1.1.5 GXL and GML

The GXL [Win02] dialect of XML and GML [Him] are a widely used standards for representing attributed graphs as text, especially in the graph drawing and software engineering communities. There are many tools available for parsing and analyzing graphs represented in these formats. And, as GXL is based on XML, it is amenable to the panoply of XML tools.

Various graph drawing and manipulation packages either use GXL or GML as their main graph language, or provide a translator. In this, *Graphviz* is no different. We supply the programs `gv2gxl`, `gxl2gv`, `gv2gml` and `gml2gv` for converting between the DOT and these 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 manual 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 2.3 explains how an application can invoke the *Graphviz* renderers, thereby generating a drawing of a graph in a concrete graphics format such as *png* 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 7 discusses an auxiliary library for dealing with graphs containing multiple connected components.

N.B. Using *Graphviz* as a library is not thread-safe.

2 Basic graph drawing

Figure 1 gives a template for the basic library use of *Graphviz*, in this instance using the *dot* hierarchical layout. (Appendix B provides the listing of the complete program.) Basically, the program creates a graph using the `cgraph` library, setting node and edge attributes to affect how the graph is to be drawn; calls the layout code; and then uses the position information attached to the nodes and edges to render the graph. The remainder of this section explores these steps in more detail.

```
Agraph_t* G;
GVC_t* gvc;

gvc = gvContext();        /* library function */
G = createGraph ();
gvLayout (gvc, G, "dot"); /* library function */
drawGraph (G);
gvFreeLayout(gvc, g);     /* library function */
agclose (G);              /* library function */
gvFreeContext(gvc);
```

Figure 1: Basic use

Here, we just note the `gvc` parameter. This is a handle to a *Graphviz context*, which contains drawing and rendering information independent of the properties pertaining to a particular graph as well as various state information. For the present, we view this an abstract parameter required for various *Graphviz* functions. We will discuss it further in Section 4.

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 `cgraph` library.

We can create a graph in one of two main ways, using `agread` or `agopen`. The former function takes a `FILE*` pointer to a file open for reading.

```
FILE* fp;
Agraph_t* G = agread(fp, 0);
```

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 no more graphs or an error occurred.

The *Graphviz* library provides several specialized variations of `agread`. If the *DOT* representation of the graph is stored in memory at `char* cp`, then

```
Agraph_t* G = agmemread(cp);
```

can be used to parse the representation. By default, the `agread` function relies on the standard `FILE` structure and `fgets` function of the stdio library. You can supply your own data source `dp` coupled with your own discipline `disc` for reading the data to read a graph using

```
Agraph_t* G = agread(dp, &disc);
```

Further details on using `agread` and disciplines can be found in the `cgraph` library manual.

The alternative technique is to call `agopen`.

```
Agraph_t* G = agopen(name, type, &disc);
```

The first argument is a `char*` giving the name of the graph; the second argument is an `Agdesc_t` value describing 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. So, to open a graph named

Graph Type	Graph
Agundirected	Non-strict, undirected graph
Agstrictundirected	Strict, undirected graph
Agdirected	Non-strict, directed graph
Agstrictdirected	Strict, directed graph

Table 3: Graph types

`"network"` that is directed but not strict, one would use

```
Agraph_t* G = agopen("network", Agdirected, 0);
```

The third argument is a pointer to a discipline of functions used for reading, memory, etc. If the value of 0 or NULL is used, the library uses a default discipline.

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

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

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. The third argument specifies whether or not a node of the given name should be created if it does not already exist.

Edges are created using `agedge` by passing in the edge's two nodes. If the graph is not strict, additional calls to `agedge` with the same arguments will create additional edges between the two nodes. The string argument allows you to supply a further name to distinguish between edges with the same head and tail. If the graph is strict, extra calls will simply return the already existing edge. For directed graphs, the first and second node arguments are taken to be the tail and head nodes, respectively. For undirected graph, they still play this role for the functions `agfstout` and `agfstin`, but when checking if an edge exists with `agedge` or `agfindedge`, the order is irrelevant. As with `agnode`, the final argument specifies whether or not the edge should be created if it does not yet exist.

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

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

The first argument is the immediate parent graph; the second argument is the name of the subgraph; the final argument indicates if the subgraph should be created.

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 attributes. 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[1] 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.

We note here some important fields used in nodes, edges and graphs. If `np`, `ep` and `gp` are pointers to a node, edge and graph, respectively, `agnameof(np)` and `agraphof(np)` give the name of the node and the root graph containing it, `agtail(ep)` and `aghead(ep)` give the tail and head nodes of the edge, and `agroot(gp)` gives the root graph containing the subgraph. For the root graph, this field will point to itself.

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, these two cases are 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 `agattr`. It takes a graph, an object type (`AGRAPH, AGNODE, AGEDGE`), and two strings as arguments, and return a representation of the

[1] if supported

attribute. The first string gives the name of the attribute; the second supplies the default value. 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 (void*, char*, char*)
```

with a pointer to the component, the name of the attribute and the value to which it should be set. For example, the call

```
agset (np, "color", "blue");
```

sets the color of node `np` to `"blue"`. The attribute value must not be `NULL`.

For simplicity, the `cgraph` library provides the function

```
agsafeset(void*, char*, char*, char*)
```

the first three arguments being the same as those of `agset`. This function first checks that the named attribute has been declared for the given graph component. If it has not, it declares the attribute, using its last argument as the required default value. It then sets the attribute value for the specific component.

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 an `int`, `double` or some more structured type. An application should only set attributes using strings and `agset`. The implementation of the layout algorithm may change the machine-level representation at any time. Hence, the low-level interface cannot be relied on by the application to supply the desired input values. Also note that there is not a one-to-one correspondence between 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 function `agattr`. (Passing `NULL` as the default value will cause `agattr` to return the `Agsym_t` if it exists, and `NULL` otherwise.) An attribute can also be obtained by a call to `agattrsym`, which takes a graph component and an attribute name. If the attribute has been defined, the function returns a pointer to 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 `Agsym_t` value to access the attribute value from an array.

Due to the nature of the implementation of attributes in *Graphviz*, an application should, if possible, 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 and complete information on what the attributes mean, the reader should consult the page `http://www.graphviz.org/content/attrs`.

Here, we consider some of the more commonly used attributes. 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

Name	Default	Use
`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`
`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`	1.0/0.3	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
`tailclip`	true	clip tail end to node boundary
`tailport`	center	position where edge attaches to tail node
`weight`	`1`	importance of edge

Table 5: Geometric edge attributes

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 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* renderers (cf. Section 2.3), 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.

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
maxiter	*layout dependent*	bound on iterations in layout
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 *f*, bounds network simplex iterations by *(f)(number of nodes)* 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
overlap	true	specify if and how to remove node overlaps
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
resolution		synonym for dpi
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	0.1	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
voro_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

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
color	black	node shape color
fillcolor	lightgrey	node fill color
fontcolor	black	text color
layer	overlay range	all, *id* or *id:id*
nojustify	false	context for justifying multiple lines of text
style		style options, e.g. bold, dotted, filled

Table 8: Decorative node attributes

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

Name	Default	Use
bgcolor		background color for drawing, plus initial fill color
charset	UTF-8	character encoding for text
fontcolor	black	type face color
labeljust	centered	left, right or center alignment for graph labels
labelloc	bottom	top or bottom location for graph labels
layers		names for output layers
layersep	":"	separator characters used in layer specification
nojustify	false	context for justifying multiple lines of text
outputorder	breadthfirst	order in which to emit nodes and edges
pagedir	BL	traversal order of pages
samplepoints	8	number of points used to represent ellipses and circles on output
stylesheet		XML stylesheet
truecolor		determines truecolor or color map model for bitmap output

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	centered	left, right or center alignment for cluster labels
labelloc	top	top or bottom location for cluster labels
nojustify	false	context for justifying multiple lines of text
pencolor	black	cluster boundary color
style		style options, e.g. bold, dotted, filled;

Table 11: Decorative cluster attributes

is directed, the `arrowhead`, `arrowsize`, `arrowtail` and `dir` attributes can play a role. Using splines 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, if requested

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.

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

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 by a call to `gvLayout`. As arguments, this function takes a pointer to a `GVC_t`, a pointer to the graph to be laid out, and the name of the desired layout algorithm. The algorithm names are the same as those of the layout programs listed in Section 1. Thus, `"dot"` is used to invoke `dot`, etc.[2]

The layout algorithm will do everything that the corresponding program would do, given the graph and its attributes. This includes assigning node positions, representing edges as splines[3], handling the special case of an unconnected graph, plus dealing with various technical features such as preventing node overlaps.

[2]Usually, all of these algorithms are available. It is possible, however, that an application can arrange to have only a subset made available.

[3]Line segments are represented as degenerate splines.

There are two special layout engines available in the library: "nop" and "nop2". These correspond to running the *neato* command with the flags -n and -n2, respectively. That is, they assume the input graph already has position information stored for nodes and, in the latter case, some edges. They can be used to route edges in the graph or perform other adjustments. Note that they expect the position information to be stored as pos attributes in the nodes and edges. The application can do this itself, or use the dot renderer.

For example, if one wants to position the nodes of a graph using a *dot* layout, but wants edges drawn as line segments, one could use the following code shown in Figure 2. The first call to gvLayout lays out the

```
Agraph_t* G;
GVC_t* gvc;

/*
 * Create gvc and graph
 */

gvLayout (gvc, G, "dot");
gvRender (gvc, G, "dot", NULL);
gvFreeLayout(gvc, G);
gvLayout (gvc, G, "nop");
gvRender (gvc, G, "png", stdout);
gvFreeLayout(gvc, G);
agclose (G);
```

Figure 2: Basic use

graph using dot; the first call to gvRender attaches the computed position information to the nodes and edges. The second call to gvLayout adds straight-line edges to the already positioned nodes; the second call to gvRender outputs the graph in png for on stdout.

2.3 Rendering the graph

Once the layout is done, the graph data structures contain the position information for drawing the graph. The application needs to decide how to use this information.

To use the renderers supplied with the *Graphviz* software, the application can call one of the library functions

```
gvRender (GVC_t *gvc, Agraph_t* g, char *format, FILE *out);
gvRenderFilename (GVC_t *gvc, Agraph_t* g, char *format, char *filename);
```

The first and second arguments are a graphviz context handle and a pointer to the graph to be rendered. The final argument gives, respecitively, a file stream open for writing or the name of a file to which the graph should be written. The third argument names the renderer to be used, such as "ps", "png" or "dot". The allowed strings are the same ones used with the -T flag when the layout program is invoked from a command shell.

After a graph has been laid out using gvLayout, an application can perform multiple calls to the rendering functions. A typical instance might be

```
gvLayout (gvc, g, "dot");
gvRenderFilename (gvc, g, "png", "out.png");
gvRenderFilename (gvc, g, "cmap", "out.map");
```

in which the graph is laid out using the *dot* algorithm, followed by PNG bitmap output and a corresponding map file which can be used in a web browser.

As with reading, *Graphviz* provides some specialized functions for rendering. Of note is

```
gvRenderData (GVC_t *gvc, Agraph_t* g, char *format, char **result,
  unsigned int *length)
```

which writes the output of the rendering onto an allocated character buffer. A pointer to this buffer is returned in `result` and the number of bytes written is stored in `length`. After using the buffer, the memory should be freed by the application.

Sometimes, an application will decide to do its own rendering. An application-supplied drawing routine, such as `drawGraph` in Figure 1 can then read this information, map it to display coordinates, and call routines to render the drawing.

One simple way to do this is to use the position and drawing information as supplied by the `dot` or `xdot` format (see Sections 1.1.1 and 1.1.2). To get this, the application can call the appropriate renderer, passing a NULL stream pointer to `gvRender`[4] as in Figure 2. This will attach the information as string attributes. The application can then use `agget` to read the attributes.

On the other hand, an application may desire to read the primitive data structures used by the algorithms to record the layout information. In the remainder of this section, we describe in reasonable detail these data structures. An application can use these values directly to guide its drawing. In some cases, for example, with arrowheads attached to `bezier` values or HTML-like labels, it would be onerous for an application to fully interpret the data. For this reason, if an application wishes to provide all of the graphics features while avoiding the low-level details of the data structures, we suggest either using `xdot` approach, described above, or supplying its own renderer plug-in as described in Section 5.

The *Graphviz* layout algorithms rely on a specific set of fields to record position and drawing information. Thus, the definitions of the information fields are fixed by the layout library and cannot be altered by the application.[5]

The fields should only be accessed using macro expressions provided for this purpose. Thus, if `np` is a node pointer, the width field should be read using `ND_width(np)`. Edge and graph attributes follow the same convention, with prefixes `ED_` and `GD_`, respectively. A complete list of these macros is given in `types.h`, along with various auxiliary types such as `pointf` or `bezier` [6].

We now consider the principal fields providing position information.

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.[7] The `ND_width` and `ND_height` attributes specify the size of the bounding box of the node, in inches. Note that the `width` and `height` attributes provide in the input graph are minimum values, so that the values stored in `ND_width` and `ND_height` may be larger.

[4]This convention only works, and only makes sense, with the `dot` and `xdot` renderers. For other renders, a NULL stream will cause output to be written on `stdout`.

[5]This is a limitation of the `cgraph` library. We plan to remove this restriction by moving to a mechanism 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 of 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 `cgraph` types to the types it defined, and access the additional fields.

[6]We strongly deprecate accessing the fields directly, for the usual reason of good programming style. By using the macros, source code will not be affected by any changes to the how the value is provided

[7]The *neato* and *fdp* layouts allow the graph to specify fixed positions for nodes. Unfortunately, some post-processing done in *Graphviz* translates the layout so that its lower-left corner is at the origin. To recover the original coordinates, the application will need to translate all positions by the vector $p_0 - p$, where p_0 and p are the input position and the final position of some node whose position was fixed.

Edges, even if a line segment, are represented as cubic B-splines or piecewise Bezier curves. The `ED_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. In some cases, however, the edge may need multiple `bezier` parts, as when the `concentrate` attribute is set, whereby mostly parallel edges are represented by a shared spline. Of course, the application always has the possibility of drawing a line segment connecting the centers of the edge's nodes.

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 `GD_bb` attribute of the subgraph, the coordinates in points in the global coordinate system.

2.3.1 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 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 consists of those with polygonal shape[8], which includes the limiting cases of circles, ellipses, and none. 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, with the origin at the center of the node.

To handle a node's shape, an application has two 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. A second approach 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 `ND_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. Again, if an application desires to be fully faithful in the rendering, it may be preferable to use the `xdot` information or to supply its own renderer plugin.

For edges, each `bezier` structure has a `list` field pointing to an array containing the control points and a `size` field giving the number of points in `list`, which will always have the form $(3 * n + 1)$. In addition, there are fields for specifying arrowheads. If `bp` points to a `bezier` structure and the `bp->sflag` field is true, there should be an arrowhead attached to the beginning of the bezier. The field `bp->sp` gives the point where the nominal tip of the arrowhead would touch the tail node. (If there is no arrowhead, `bp->list[0]` will touch the node.) Thus, the length and direction of the arrowhead is determined by the vector going from `bp->list[0]` to `bp->sp`. The actual shape and width of the arrowhead is determined by the `arrowtail` and `arrowsize` attributes. Analogously, an arrowhead at the head node is specified

[8] This is not quite true but close enough for now.

by `bp->eflag` and the vector from `bp->list[bp->size-1]` to `bp->ep`.

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. This information is available in other fields.

2.4 Cleaning up a graph

Once all layout information is obtained from the graph, the resources should be reclaimed. To do this, the application should call the cleanup routine associated with the layout algorithm used to draw the graph. This is done by a call to `gvFreeLayout`.

The example of Figure 1 demonstrates 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 best determine 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 `gvRender`. Note, though, that layout information is destroyed during cleanup. If the application needs to reuse this data, for example, to refresh the display, it should delay 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, though this is sometimes considered poor programming style.

A given graph can be laid out multiple times. The application, however, must clean up the earlier layout's information with a call to `gvFreeLayout` before invoking a new layout function. An example of this was given in Figure 2.

Once the application is totally done with a graph, it should call `agclose` to close the graph and reclaim the remaining resources associated with it.

3 Inside the layouts

Each *Graphviz* layout algorithm consists of multiple steps, some of which are optional. As the only entry point in the *Graphviz* library for laying out a graph is the function `gvLayout`, the control of which steps are used is determined by graph attributes, in the same way this is controlled when passing a graph to one of the layout programs. In this section, we provide a high-level description of the layout steps, and note the relevant attributes.

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 7.

In all of the algorithms, the first step is to call a layout-specific initialization function. 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 is called in `gvFreeLayout` (cf. Section 2.4).

By default, the layout algorithms position the edges as well as the nodes of the graph. As this may be expensive to compute and irrelevant to an application, an application may decide to avoid this. This can be achieved by setting the graph's `splines` attribute to the empty string `""`.

The algorithms all end with a postprocessing step. The role of this 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.

Except for dot, the algorithms also provide a node's position, in inches, in the array give by `ND_pos`.

3.1 *dot*

The *dot* algorithm produces a ranked layout of a graph respecting 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]

The steps in the *dot* layout are:

```
initialize
rank
mincross
position
sameports
splines
compoundEdges
```

After initialization, the algorithm assigns each node to a discrete rank (`rank`) using an integer program to minimize the sum of the (discrete) edge lengths. The next step (`mincross`) rearranges nodes within ranks to reduce edge crossings. This is followed by the assignment (`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 `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.

Edge representations are generated in the `splines` step. 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 `splines` for the edge can then be clipped to the bounding box of the specified clusters. This is accomplished in the `compoundEdges` step.

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.

```
initialize
position
adjust
splines
```

As usual, the layout starts with an initialization step. The actual layout is parameterized by the `mode` and `model` attributes. The mode attribute determines how the optimization problem is solved, either by the default, stress majorization[GKN04] mode, (`mode="major"`), or the gradient descent technique proposed by Kamada and Kawai[KK89] (`mode="KK"`). 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. By default, *neato* uses a shortest path 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"` and the graph attribute `pack` is false, *neato* sets the distance between nodes in separate connected components to $1.0 + L_{avg} \cdot \sqrt{|\mathtt{V}|}$, where L_{avg} is the average edge length and $|\mathtt{V}|$ 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 the circuit 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="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 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 the `adjust` step to reposition the nodes to eliminate overlaps. This is controlled by the graph attribute `"overlap"`.

With nodes positioned, the algorithm proceeds to draw the edges using its `splines` function. By default, edges are drawn as line segments. If, however, the `"splines"` graph attribute is set to true, the edges will be constructed 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 `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 does not yet 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.

```
initialize
position
splines
```

The layout scheme is fairly simple: initialization; layout; and a call to route the edges. In *fdp*, because it is necessary to keep clusters separate, the removal of overlaps is (usually) obligatory.

3.4 *sfdp*

The *sfdp* layout is similar to *fdp* except it uses a refined multilevel approach that enables it to handle very large graphs. The algorithm is due to Hu[Hu05].

Unlike *fdp*, *sfdp* does not support cluster subgraphs. It also does not model edge lengths or weights.

```
initialize
position
adjust
splines
```

The layout scheme is fairly simple: initialization; layout; node overlap removal; and a call to route the edges.

3.5 *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. At present, the algorithm does not attempt to visualize clusters.

```
initialize
position
adjust
splines
```

As usual, the layout commences by initializing the graph. This is followed by the `position` step, which is parameterized by the central node, specified by the graph's `root` attribute. If unspecified, the algorithm will select some "most central" node, i.e., one whose minimum distance from a leaf node is maximal.

As with *neato*, the layout allows an `adjust` step to eliminate node-node overlaps. Again as with *neato*, the call to `splines` computes drawing information for edges. See Section 3.2 for more details.

3.6 *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.

```
initialize
position
splines
```

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

4 The *Graphviz* context

Up to now, we have used a *Graphviz* context `GVC_t` without considering its purpose. As suggested earlier, this value is used to store various layout information that is independent of a particular graph and its attributes. It holds the data associated with plugins, parsed-command lines, script engines, and anything else with a scope potentially larger than one graph, up to the scope of the application. In addition, it maintains lists of the available layout algorithms and renderers; it also records the most recent layout algorithm applied to a graph. It can be used to specify multiple renderings of a given graph layout into different associated files. It is also used to store various global information used during rendering.

There should be just one `GVC_t` created for the entire duration of an application. A single `GVC_t` value can be used with multiple graphs, though with only one graph at a time. In addition, if `gvLayout()` was invoked for a graph and `GVC_t`, then `gvFreeLayout()` should be called before using `gvLayout()` again, even on the same graph.

Normally, one creates a `GVC_t` by a call to:

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

which is what we have used in the examples shown here.

One can initialize a `GVC_t` to record a list of graphs, layout algorithms and renderers. To do this, the application should call the function `gvParseArgs`:

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

This function takes the context value, plus an array of strings using the same conventions as the parameters to `main` function in a C program. In particular, `argc` should be the number of values in `argv`. If `argv[0]` is the name of one of the layout algorithms, this will be bound to the `GVC_t` value and used at layout time. The remaining `argv` values, if any, are interpreted exactly like the allowed command line flags for any *Graphviz* program. Thus, `"-T"` can be used to set the output type, and `"-o"` can be used to specify the output files.

For example, the application can use a synthetic argument list

```
GVC_t* gvc = gcContext();
char* args[] = {
    "dot",
    "-Tgif",        /* gif output */
    "-oabc.gif"     /* output to file abc.gif */
};
gvParseArgs (gvc, sizeof(args)/sizeof(char*), args);
```

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

Most of the information is stored in a `GVC_t` value for use during rendering. However, if the `argv` array contains non-flag arguments, i.e., strings after the first not beginning with `"-"`, these are taken to be input files defining a stream of graphs to be drawn. These graphs can be accessed by calls to `gvNextInputGraph`.

Once the `GVC_t` has been initialized this way, the application can call `gvNextInputGraph` to get each input graph in sequence, and then invoke `gvLayoutJobs` and `gvRenderJobs` to do the specified layouts and renderings. See Appendix C for a typical example of this approach.

We note that `gvLayout` basically attaches the graph and layout algorithm to the `GVC_t`, as would be done by `gvParseArgs`, and then invokes `gvLayoutJobs`. A similar remark holds for `gvRender` and `gvRenderJobs`.

4.1 Version-specific data

When the `GVC_t` is created, it stores version and build date information that can be used by renderers to identify which version of *Graphviz* produced the output. It is also what is printed when a layout program is given the `-V` flag. This information is stored as an array of three `char*`, giving the name, version number and build date, respectively. These can be accessed via the functions:

```
extern char **gvcInfo(GVC_t*);
extern char *gvcVersion(GVC_t*);
extern char *gvcBuildDate(GVC_t*);
```

5 Graphics renderers

All graph output done in *Graphviz* goes through a renderer with the type `gvrender_engine_t`, used in the call to `gvRender`. In addition to the renderers which are part of the library, an application can provide its own, allowing it to specialize or control the output as necessary. See Section 6.1 for further details.

As in the layout phase invoked by `gvLayout`, all control over aspects of rendering are handled via graph attributes. For example, the attribute `outputorder` determines whether all edges are drawn before any nodes, or all nodes are drawn before any edges.

Before describing the renderer functions in detail, it may be helpful to give an overview of how output is done. Output can be viewed as a hierarchy of document 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 as determined by a graph's `layers` attribute, if any. Each layer may be divided into a 2-dimensional array of pages. A page will then contain nodes, edges, and clusters. Each of these may contain an HTML anchor. During rendering, 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.

Figure 3 lists the names and type signatures of the fields of `gv_render_engine_t`, which are used to emit the components described above.[9] All of the functions take a `GVJ_t*` value, which contains various information about the current rendering, such as the output stream, if any, or the device size and resolution. Section 5.1 describes this data structure.

Most of the functions handle the nested graph structure. All graphics output is handled by the `textpara`, `ellipse`, `polygon`, `beziercurve`, and `polyline` functions. The relevant drawing information such as color and pen style is available through the `obj` field of the `GVJ_t*` parameter. This is described in Section 5.2. Font information is passed with the text.

We note that, in *Graphviz*, each node, edge or cluster in a graph has a unique `id` field, which can be used as a key for storing and accessing the object.

```
void (*begin_job) (GVJ_t*);
void (*end_job) (GVJ_t*);
void (*begin_graph) (GVJ_t*);
void (*end_graph) (GVJ_t*);
void (*begin_layer) (GVJ_t*, char*, int, int);
void (*end_layer) (GVJ_t*);
void (*begin_page) (GVJ_t*);
void (*end_page) (GVJ_t*);
void (*begin_cluster) (GVJ_t*, char*, long);
void (*end_cluster) (GVJ_t*);
void (*begin_nodes) (GVJ_t*);
void (*end_nodes) (GVJ_t*);
void (*begin_edges) (GVJ_t*);
void (*end_edges) (GVJ_t*);
void (*begin_node) (GVJ_t*, char*, long);
void (*end_node) (GVJ_t*);
void (*begin_edge) (GVJ_t*, char*, bool, char*, long);
void (*end_edge) (GVJ_t*);
void (*begin_anchor) (GVJ_t*, char*, char*, char*);
void (*end_anchor) (GVJ_t*);
void (*textpara) (GVJ_t*, pointf, textpara_t*);
void (*resolve_color) (GVJ_t*, gvcolor_t*);
void (*ellipse) (GVJ_t*, pointf*, int);
void (*polygon) (GVJ_t*, pointf*, int, int);
void (*beziercurve) (GVJ_t*, pointf*, int, int, int, int);
void (*polyline) (GVJ_t*, pointf*, int);
void (*comment) (GVJ_t*, char*);
```

Figure 3: Interface for a renderer

In the following, we describe the functions in more detail, though many are self-explanatory. All positions and sizes are in points.

`begin_job(job)` Called at the beginning of all graphics output for a graph, which may entail drawing multiple layers and multiple pages.

[9] Any types mentioned in this section are either described in this section or in Appendix E.

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

`begin_graph(job)` Called at the beginning of drawing a graph. The actual graph is available as `job->obj->u.g`.

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

`begin_layer(job,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`, starting from 0.

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

`begin_page(job)` Called at the beginning of a new output page. A page will contain a rectangular portion of the drawing of the graph. The value `job->pageOffset` gives the lower left corner of the rectangle in layout coordinates. The point `job->pagesArrayElem` is the index of the page in the array of pages, with the page in the lower left corner indexed by (0,0). The value `job->zoom` provides a scale factor by which the drawing should be scaled. The value `job->rotation`, if non-zero, indicates that the output should be rotated by $90°$ counterclockwise.

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

`begin_cluster(job)` Called at the beginning of drawing a cluster subgraph. The actual cluster is available as `job->obj->u.sg`.

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

`begin_nodes(job)` Called at the beginning of drawing the nodes on the current page. Only called if the graph attribute `outputorder` was set to a non-default value.

`end_nodes(job)` Called when all nodes on a page have been drawn. Only called if the graph attribute `outputorder` was set to a non-default value.

`begin_edges(job)` Called at the beginning of drawing the edges on the current page. Only called if the graph attribute `outputorder` was set to a non-default value.

`end_edges()` Called when all edges on the current page are drawn. Only called if the graph attribute `outputorder` was set to a non-default value.

`begin_node(job)` Called at the start of drawing a node. The actual node is available as `job->obj->u.n`.

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

`begin_edge(job)` Called at the start of drawing an edge. The actual edge is available as `job->obj->u.e`.

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

`begin_anchor(job,href,tooltip,target)` Called at the start of an anchor context associated with the current node, edge, or graph, or its label, assuming the graph object or its label 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`.

If the anchor information is attached to a graph object, the `begin_anchor` and `end_anchor` calls enclose the `begin_...` and `end_...` calls on the object. If the anchor information is attached to part of an object's label, the `begin_anchor` and `end_anchor` calls enclose the rendering of that part of the label plus any subparts.

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

`textpara(job, p, txt)` Draw text at point `p` using the specified font and fontsize and color. The `txt` argument provides the text string `txt->str`, stored in UTF-8, a calculated width of the string `txt->width` and the horizontal alignment `txt->just` of the string in relation to `p`. The values `txt->fontname` and `txt->fontname` give the desired font name and font size, the latter in points.

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.

`txt->just`	`p.x`
`'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 renderer 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.

`resolve_color(job, color)` Resolve a color. The `color` parameter points to a color representation of some particular type. The renderer can use this information to resolve the color to a representation appropriate for it. See Section 5.3 for more details.

`ellipse(job, ps, filled)` Draw an ellipse with center at `ps[0]`, with horizontal and vertical half-axes `ps[1].x - ps[0].x` and `ps[1].y - ps[0].y` using the current pen color and line style. If `filled` is non-zero, the ellipse should be filled with the current fill color.

`polygon(job, 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 polygon should be filled with the current fill color.

`beziercurve(job, A, n, arrow_at_start, arrow_at_end, filled)` 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 the renderer has specified that it does not want to do its own arrowheads (cf. Section 6.1), 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`. If `filled` is non-zero, the bezier should be filled with the current fill color.

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

`comment(job, text)` Emit text comments related to a graph object. For nodes, calls will pass the node's name and any `comment` attribute attached to the node. For edges, calls will pass a string description of the edge and any `comment` attribute attached to the edge. For graphs and clusters, a call will pass a any `comment` attribute attached to the object.

Although access to the graph object being drawn is available through the `GVJ_t` value, a renderer can often perform its role by just implementing the basic graphics operations. It need have no information about graphs or the related *Graphviz* data structures. Indeed, a particular renderer need not define any particular rendering function, since a given entry point will only be called if non-NULL.

5.1 The `GVJ_t` data structure

We now describe some of the more important fields in the `GVJ_t` structure, concentrating on those regarding output. There are additional fields relevant to input and GUIs.

`common` This points to various information valid throughout the duration of the application using *Graphviz*. In particular, the `common->info` contains *Graphviz* version information, as described in Section 4.1.

`output_file` The `FILE*` value for an open stream on which the output should be written, if relevant.

`pagesArraySize` The size of the array of pages in which the graph will be output, given as a `point`. If `pagesArraySize.x` or `pagesArraySize.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. 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.

`bb` The bounding box of the layout in the universal space in points. It has type `boxf`.

`boundingBox` The bounding box of the layout in the device space in device coordinates. It has type `box`.

`layerNum` The current layer number.

`numLayers` The total number of layers.

`pagesArrayElem` The row and column of the current page.

`pageOffset` The origin of the current page in the universal space in points.

`zoom` Factor by which the output should be scaled.

`rotation` Indicates whether or not the rendering should be rotated.

`obj` Information related to the current object being rendered. This is a pointer of a value of type `obj_state_t`. See Section 5.2 for more details.

5.2 Inside the `obj_state_t` data structure

A value of type `obj_state_t` encapsulates various information pertaining to the current object being rendered. In particular, it provides access to the current object, and provides the style information for any rendering operation. Figure 4 notes some of the more useful fields in the structure.

```
obj_type type;
union {
        graph_t *g;
        graph_t *sg;
        node_t *n;
        edge_t *e;
} u;
gvcolor_t pencolor;
gvcolor_t fillcolor;
pen_type pen;
double penwidth;
char *url;
char *tailurl;
char *headurl;
char *tooltip;
char *tailtooltip;
char *headtooltip;
char *target;
char *tailtarget;
char *headtarget;
```

Figure 4: Some fields in `obj_state_t`

`type` **and** `u` The `type` field indicates what kind of graph object is currently being rendered. The possible values are `ROOTGRAPH_OBJTYPE`, `CLUSTER_OBJTYPE`, `NODE_OBJTYPE` and `EDGE_OBJTYPE`, indicating the root graph, a cluster subgraph, a node and an edge, respectively. A pointer to the actual object is available via the subfields `u.g`, `u.sg`, `u.n` and `u.e`, respectively, of the union `u`.

`pencolor` The `gvcolor_t` value indicating the color used to draw lines, curves and text.

`pen` The style of pen to be used. The possible values are `PEN_NONE`, `PEN_DOTTED`, `PEN_DASHED` and `PEN_SOLID`.

`penwidth` The size of the pen, in points. Note that, by convention, a value of 0 indicates using the smallest width supported by the output format.

`fillcolor` The `gvcolor_t` value indicating the color used to fill closed regions.

Note that font information is delivered as part of the `textpara_t` value passed to the `textpara` function.

As for the url, tooltip and target fields, these will point to the associated attribute value of the current graph object, assuming it is defined and that the renderer support map, tooltips, and targets, respectively (cf. Section 6.1).

5.3 Color information

There are five ways in which a color can be specified in *Graphviz*: RGB + alpha, HSV + alpha, CYMK, color index, and color name. In addition, the RGB + alpha values can be stored as bytes, words or doubles.

A color value in *Graphviz* has the type `gvcolor_t`, containing two fields: a union `u`, containing the color data, and the `type` field, indicating which color representation is used in the union. Table 14 describes the allowed color types, and the associated union field.

Type	Description	Field
RGBA_BYTE	RGB + alpha format represented as 4 bytes from 0 to 255	u.rgba
RGBA_WORD	RGB + alpha format represented as 4 words from 0 to 65535	u.rrggbbaa
RGBA_DOUBLE	RGB + alpha format represented as 4 doubles from 0 to 1	u.RGBA
HSVA_DOUBLE	HSV + alpha format represented as 4 doubles from 0 to 1	u.HSVA
CYMK_BYTE	CYMK format represented as 4 bytes from 0 to 255	u.cymk
COLOR_STRING	text name	u.string
COLOR_INDEX	integer index	u.index

Table 14: Color type representations

Before a color is used in rendering, *Graphviz* will process a color description provided by the input graph into a form desired by the renderer. This is three step procedure. First, *Graphviz* will see if the color matches the renderer's known colors, if any. If so, the color representation is COLOR_STRING. Otherwise, the library will convert the input color description into the renderer's preferred format. Finally, if the renderer also provides a resolve_color function, *Graphviz* will then call that function, passing a pointer to the current color value. The renderer then has the opportunity to adjust the value, or convert it into another format. In a typical case, if a renderer uses a color map, it may request RGB values as input, and then store an associated color map index using the COLOR_INDEX format. If the renderer does a conversion to another color type, it must reset the type field to indicate this. It is this last representation which will be passed to the renderer's drawing routines. The renderer's known colors and preferred color format are described in Section 6.1 below.

6 Adding Plug-ins

The *Graphviz* framework allows the programmer to use plug-ins to extend the system in several ways. For example, the programmer can add new graph layout engines along with new renderers and their related functions. Table 15 describes the plug-in APIs supported by *Graphviz*. Each plug-in is defined

Kind	Functions	Features	Description
API_render	gvrender_engine_t	gvrender_features_t	Functions for rendering a graph
API_device	gvdevice_engine_t	-	Functions for initializing and terminating a device
API_loadimage	gvloadimage_engine_t	-	Functions for converting from one image format to another
API_layout	gvlayout_engine_t	gvlayout_features_t	Functions for laying out a graph
API_textlayout	gvtextlayout_engine_t	-	Functions for resolving font names and text size

Table 15: Plug-in API types

by an engine structure containing its function entry points, and a features structure specifying features supported by the plug-in. Thus, a renderer is defined by values of type gvrender_engine_t and gvrender_features_t.

Once all of the plug-ins of a given kind are defined, they should be gathered into a 0-terminated array of element type gvplugin_installed_t, whose fields are shown in Figure 5. The fields have the following meanings.

id Identifier for a given plug-in within a given package and with a given API kind. Note that the id need

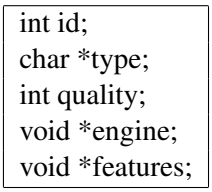

Figure 5: Plug-in fields

only be unique within its plug-in package, as these packages are assumed to be independent.

`type` Name for a given plug-in, used during plug-in lookup.

`quality` An arbitrary integer used for ordering plug-ins with the same `type`. Plug-ins with larger values will be chosen before plug-ins with smaller values.

`engine` Points to the related engine structure.

`features` Points to the related features structure.

As an example, suppose we wish to add various renderers for bitmap output. A collection of these might be combined as follows.

```
gvplugin_installed_t render_bitmap_types[] = {
    {0, "jpg", 1, &jpg_engine, &jpg_features},
    {0, "jpeg", 1, &jpg_engine, &jpg_features},
    {1, "png", 1, &png_engine, &png_features},
    {2, "gif", 1, &gif_engine, &gif_features},
    {0, NULL, 0, NULL, NULL}
};
```

Note that this allows `"jpg"` and `"jpeg"` to refer to the same renderers. For the plug-in kinds without a features structure, the feature pointer in its `gvplugin_installed_t` should be NULL.

All of the plug-ins of all API kinds should then be gathered into a 0-terminated array of element type `gvplugin_api_t`. For each element, the first field indicates the kind of API, and the second points to the array of plug-ins described above (`gvplugin_installed_t`).

Continuing our example, if we have supplied, in addition to the bitmap rendering plug-ins, plug-ins to render VRML, and plug-ins to load images, we would define

```
gvplugin_api_t apis[] = {
    {API_render, &render_bitmap_types},
    {API_render, &render_vrml_types},
    {API_loadimage, &loadimage_bitmap_types},
    {0, 0},
};
```

Here `render_vrml_types` and `render_vrml_types` are also 0-terminated arrays of element type `gvplugin_installed_t`. Note that there can be multiple items of the same API kind.

A final definition is used to attach a name to the package of all the plug-ins. This is done using a `gvplugin_library_t` structure. Its first field is a `char*` giving the name of the package. The second field is a `gvplugin_api_t*` pointing to the array described above. The structure itself must be named `gvplugin_`*name*`_LTX_library`, where *name* is the name of the package as defined in the first field.

For example, if we have decided to call our package `"bitmap"`, we could use the following definition:

```
gvplugin_library_t gvplugin_bitmap_LTX_library = { "bitmap", apis };
```

To finish the installation of the package, it is necessary to create a dynamic library containing the `gvplugin_library_t` value and all of the functions and data referred by it, either directly or indirectly. The library must be named `gvplugin_`*name*, where again *name* is the name of the package. The actual filename of the library will be system-dependent. For example, on Linux systems, our library `gvplugin_bitmap` would have filename `libgvplugin_bitmap.so.3`.

In most cases, *Graphviz* is built with a plug-in version number. This number must be included in the library's filename, following any system-dependent conventions. The number is given as the value of `plugins` in the file `libgvc.pc`, which can be found in the directory `lib/pkgconfig` where *Graphviz* was installed. In our example, the "3" in the library's filename gives the version number.

Finally, the library must be installed in the *Graphviz* library directory, and `dot -c` must be run to add the package to the *Graphviz* configuration. Note that both of these steps typically assume that one has installer privileges.[10]

In the remainder of this section, we shall look at the first three types of plug-in APIs in more detail.

6.1 Writing a renderer plug-in

A renderer plug-in has two parts. The first consists of a structure of type `gvrender_engine_t` defining the renderer's actions, as described in Section 5. Recall that any field may contain a NULL pointer.

For the second part, the programmer must provide a structure of type `gvrender_features_t`. This record provides *Graphviz* with information about the renderer. Figure 6 list the fields involved. Some of the

```
int flags;
double default_margin;
double default_pad;
pointf default_pagesize;
pointf default_dpi;
char **knowncolors;
int sz_knowncolors;
color_type_t color_type;
char *device;
char *loadimage_target;
```

Figure 6: Features of a renderer

default values may be overridden by the input graph.

We now describe the fields in detail.

`flags` Bit-wise of `or` flags indicating properties of the renderer. These flags are described in Table 16.

`default_margin` Default margin size in points. This is the amount of space left around the drawing.

`default_pad` Default pad size in points. This is the amount by which the graph is inset within the drawing region. Note that the drawing region may be filled with a background color.

`default_pagesize` Default page size size in points. For example, an 8.5 by 11 inch letter-sized page would have a `default_pagesize` of 612 by 792.

[10]Normally, for builds intended for local installation `dot -c` is run during `make install`. It may be necessary to run this manually if cross-compiling or otherwise manually moving binaries to a different system.

`default_dpi` Default resolution, in pixels per inch. Note that the x and y values may be different to support non-square pixels.

`knowncolors` An array of character pointers giving a lexicographically ordered [11] list of the color names supported by the renderer.

`sz_knowncolors` The number of items in the `knowncolors` array.

`color_type` The preferred representation for colors. See Section 5.3.

`device` The name of a device, if any, associated with the renderer. For example, a renderer using GTK for output might specify `"gtk"` as its device. If a name is given, the library will look for a plug-in of type API_device with that name, and use the associated functions to initialize and terminate the device. See Section 6.2.

`loadimage_target` The name of the preferred type of image format for the renderer. When a user-supplied image is given, the library will attempt to find a function that will convert the image from its original format to the renderer's preferred one. A user-defined renderer may need to provide, as additional plug-ins, its own functions for handling the conversion.

Flag	Description
GVRENDER_DOES_ARROWS	Built-in arrowheads on splines
GVRENDER_DOES_LAYERS	Supports graph layers
GVRENDER_DOES_MULTIGRAPH_OUTPUT_FILES	If true, the renderer's output can contain multiple renderings
GVRENDER_DOES_TRUECOLOR	Supports a truecolor color model
GVRENDER_Y_GOES_DOWN	Output coordinate system has the origin in the upper left corner
GVRENDER_X11_EVENTS	For GUI plug-ins, defers actual rendering until the GUI event loop invokes `job->callbacks->refresh()`
GVRENDER_DOES_TRANSFORM	Can handle transformation (scaling, translation, rotation) from universal to device coordinates. If false, the library will do the transformation before passing any coordinates to the renderer
GVRENDER_DOES_LABELS	Wants an object's label, if any, provided as text during rendering
GVRENDER_DOES_MAPS	Supports regions to which URLs can be attached. If true, URLs are provided to the renderer, either as part of the `job->obj` or via the renderer's `begin_anchor` function
GVRENDER_DOES_MAP_RECTANGLE	Rectangular regions can be mapped
GVRENDER_DOES_MAP_CIRCLE	Circular regions can be mapped
GVRENDER_DOES_MAP_POLYGON	Polygons can be mapped
GVRENDER_DOES_MAP_ELLIPSE	Ellipses can be mapped
GVRENDER_DOES_MAP_BSPLINE	B-splines can be mapped
GVRENDER_DOES_TOOLTIPS	If true, tooltips are provided to the renderer, either as part of the `job->obj` or via the renderer's `begin_anchor` function
GVRENDER_DOES_TARGETS	If true, targets are provided to the renderer, either as part of the `job->obj` or via the renderer's `begin_anchor` function
GVRENDER_DOES_Z	Uses a 3D output model

Table 16: Renderer properties

6.2 Writing a device plug-in

A device plug-in provides hooks for *Graphviz* to handle any device-specific operations needed before and after rendering. The related engine of type `gvdevice_engine_t` has 2 entry points:

[11] The ordering must be done byte-wise using the `LANG=C` locale for byte comparison.

```
void (*initialize) (GVJ_t*);
void (*finalize) (GVJ_t*);
```

which are called at the beginning and end of rendering each job. The initialize routine might open a canvas on window system, or set up a new page for printing; the finalize routine might go into an event loop after which it could close the output device.

6.3 Writing an image loading plug-in

A image loading plug-in has engine type `gvimageload_engine_t` and provides a single entry point which can be used to read in an image, convert the image from one format to another, and write the result. Since the function actually does rendering, it is usually closely tied to a specific renderer plug-in.

```
void (*loadimage) (GVJ_t *job, usershape_t *us, boxf b, bool filled);
```

When called, `loadimage` is given the current job, a pointer to the input image `us`, and the bounding box `b` in device coordinates where the image should be written. The boolean `filled` value indicates whether the bounding box should first be filled.

The `type` value for an image loading plug-in's `gvplugin_installed_t` entry should specify the input and output formats it handles. Thus, a plug-in converting JPEG to GIF would be called "`jpeg2gif`". Since an image loader may well want to read in an image in some format, and then render the image using the same format, it is quite reasonable for the input and output formats to be identical, e.g. "`gif2gif`".

Concerning the type `usershape_t`, its most important fields are shown in Figure 7. These fields have

```
char *name;
FILE *f;
imagetype_t type;
unsigned int x, y;
unsigned int w, h;
unsigned int dpi;
void *data;
size_t datasize;
void (*datafree)(usershape_t *us);
```

Figure 7: Fields in `usershape_t`

the following meanings:

`name` The name of the image.

`f` An open input stream to the image's data. Since the image might be processed multiple times, the application should use a function such as `fseek` to make sure the file pointer points to the beginning of the file.

`type` The format of the image. The formats supported in *Graphviz* are `FT_BMP`, `FT_GIF`, `FT_PNG`, `FT_JPEG`, `FT_PDF`, `FT_PS` and `FT_EPS`. The value `FT_NULL` indicates an unknown image type.

`x` **and** `y` The coordinates of the lower-left corner of image in image units. This is usually the origin but some images such as those in PostScript format may be translated away from the origin.

`w` **and** `h` The width and height of image in image units

`dpi` The number of image units per inch

`data`, `datasize`, `datafree` These fields can be used to cache the converted image data so that the file I/O and conversion need only be done once. The data can be stored via `data`, with `datasize` giving the number of bytes used. In this case, the image loading code should store a clean-up handler in `datafree`, which can be called to release any memory allocated.

If `loadimage` does caching, it can check if `us->data` is NULL. If so, it can read and cache the image. If not, it should check that the `us->datafree` value points to its own `datafree` routing. If not, then some other image loader has cached data there. The `loadimage` function must them call the current `us->datafree` function before caching its own version of the image.

The code template in Figure 8 indicates how caching should be handled.

```
if (us->data) {
    if (us->datafree != my_datafree) {
        us->datafree(us);  /* free incompatible cache data */
        us->data = NULL;
        us->datafree = NULL;
        us->datasize = 0;
    }
}

if (!us->data) {
   /* read image data from us->f and convert it;
    * store the image data into memory pointed to by us->data;
    * set us->datasize and us->datafree to the appropriate values.
    */
}

if (us->data) {
   /* emit the image data in us->data */
}
```

Figure 8: Caching converted images

7 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 `"KK"` and `pack="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 `pack` 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 layout algorithms given an input graph `g` and a `GVC_t` value `gvc`.

```
Agraph_t *sg;
FILE *fp;
Agraph_t** cc;
int       i, ncc;

cc = ccomps(g, &ncc, (char*)0);

for (i = 0; i < ncc; i++) {
sg = cc[i];
    nodeInduce (sg);
    gvLayout(gvc, sg, "neato");
}
pack_graph (ncc, cc, g, 0);

gvRender(gvc, g, "ps", stdout);

for (i = 0; i < ncc; i++) {
    sg = cc[i];
    gvFreeLayout(gvc, sg);
    agdelete(g, sg);
}
```

The call to `ccomps` splits the graph `g` 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 `pack` 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, we take one component at a time, use `nodeInduce` to create the corresponding node-induced subgraph, and then lay out the component with `gvLayout`. Here, we use *neato* for each layout, but it is possible to use a different layout for each component.[12]

Next, we use the `pack` function `pack_graph` to reassemble the graph into a single drawing. To position the components, `pack` uses the polyomino-based approach described by Freivalds et al[FDK02]. The first three arguments to the function are clear. The fourth argument indicates whether or not there are fixed components.

The `pack_graph` function uses the graph's `packmode` attribute to determine how the packing should be done. At present, packing uses the single algorithm mentioned above, but allows three varying granularities, represented by the values `"node"`, `"clust"` and `"graph"`. In the first case, packing is done at the

[12] At present, the *dot* layout has a limitation that it only works on a root graph. Thus, to use *dot* for a component, one needs to create a new copy of the subgraph, apply *dot* and then copy the position attributes back to the component.

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, `"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.

Note that the library automatically computes the bounding box of each of the components. Also, as a side-effect, `pack_graph` 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 `pack` library by default.

References

[BHvW00] M. Bruls, K. Huizing, and J. van Wijk. Squarified Treemaps. In W. de Leeuw and R. van Liere, editors, *Proceedings of Eurographics and IEEE TVCG Symposium on Visualization*, pages 33–42, 2000.

[Coh87] J. Cohen. Drawing graphs to convey proximity: an incremental arrangement meth od. *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.

[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.

[Him] Michael Himsolt. GML: A portable Graph File Format. Technical report, Universitat Passau.

[Hu05] Y. F. Hu. Efficient and high quality force-directed graph drawing. *Mathematica Journal*, 10:37–71, 2005.

[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.

[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.

[Wil97] 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

This appendix provides a brief description of how to build your program using *Graphviz* as a library. It also notes the various libraries involved. As compilation systems vary greatly, we make no attempt to provide low-level build instructions. We assume that the user is capable of tailoring the build environment to use the necessary include files and libraries.

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 `gvc.h`, which provides (indirectly) all of the *Graphviz* types and functions, compile the code, and link the program with the necessary libraries.

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

- `gvc`
- `graph`
- `cdt`

If the system is configured to use plug-ins, these libraries are all that are necessary. At run time, the program will load the dynamic libraries it needs.

If the program does not use plug-ins, then these libraries need to be incorporated at link time. These libraries may include

- `gvplugin_core`
- `gvplugin_dot_layout`
- `gvplugin_neato_layout`
- `gvplugin_gd`
- `gvplugin_pango`[13]

plus any other plug-ins the program requires.

If *Graphviz* is built and installed with the GNU build tools, there are package configure files created in the `lib/pkgconfig` directory which can be used with the `pkg-config` program to obtain the include file and library information for a given installation. Assuming a Unix-like environment, a sample `Makefile` for building the programs listed in Appendices B, C and D[14] could have the form:

```
CFLAGS=`pkg-config libgvc --cflags` -Wall -g -O2
LDFLAGS=`pkg-config libgvc --libs`

all: simple dot demo

simple: simple.o
dot: dot.o
demo: demo.o

clean:
    rm -rf simple dot demo *.o
```

[13] For completeness, we note that it may be necessary to explicitly link in the following additional libraries, depending on the options set when *Graphviz* was built: `expat`, `fontconfig`, `freetype2`, `pangocairo`, `cairo`, `pango`, `gd`, `jpeg`, `png`, `z`, `ltdl`, and other libraries required by Cairo and Pango. Typically, though, most builds handle these implicitly.

[14] They can also be found, along with the `Makefile`, in the `dot.demo` directory of the *Graphviz* source.

B A sample program: `simple.c`

This following code illustrates an application which uses *Graphviz* to position a graph using the *dot* layout and then write the output using the `plain` format. An application can replace the call to `gvRender` with its own function for rendering the graph, using the layout information encoded in the graph structure (cf. Section 2.3).

```
#include <gvc.h>

int main(int argc, char **argv)
{
    GVC_t *gvc;
    Agraph_t *g;
    FILE *fp;

    gvc = gvContext();

    if (argc > 1)
        fp = fopen(argv[1], "r");
    else
        fp = stdin;
    g = agread(fp, 0);

    gvLayout(gvc, g, "dot");

    gvRender(gvc, g, "plain", stdout);

    gvFreeLayout(gvc, g);

    agclose(g);

    return (gvFreeContext(gvc));
}
```

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 4.1), and a few other minor details.

```
#include <gvc.h>

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

    gvc = gvContext();
    gvParseArgs(gvc, argc, argv);

    while ((g = gvNextInputGraph(gvc))) {
        if (prev) {
            gvFreeLayout(gvc, prev);
            agclose(prev);
        }
        gvLayoutJobs(gvc, g);
        gvRenderJobs(gvc, g);
        prev = g;
    }
    return (gvFreeContext(gvc));
}
```

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 <gvc.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 a graphviz context */
    gvc = gvContext();

    /* parse command line args - minimally argv[0] sets layout engine */
    gvParseArgs(gvc, argc, argv);

    /* Create a simple digraph */
    g = agopen("g", Agdirected);
    n = agnode(g, "n", 1);
    m = agnode(g, "m", 1);
    e = agedge(g, n, m, 0, 1);

    /* Set an attribute - in this case one that affects the visible rendering */
    agsafeset(n, "color", "red", "");

    /* Compute a layout using layout engine from command line args */
    gvLayoutJobs(gvc, g);

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

    /* Free layout data */
    gvFreeLayout(gvc, g);

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

    /* close output file, free context, and return number of errors */
    return (gvFreeContext(gvc));
}
```

E Some basic types and their string representations

A `point` type is the structure

```
struct {
  int x, y;
}
```

The fields can either give an absolute position or represent a vector displacement. A `pointf` type is the same, with `int` replaced with `double`. A `box` type is the structure

```
struct {
  point LL, UR;
}
```

representing a rectangle. The `LL` gives the coordinates of the lower-left corner, while the `UR` is the upper-right corner. A `boxf` type is the same, with `point` replaced with `pointf`.

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 $p_1, p_2, \ldots, p_N$ 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.