Using *Graphviz* as a Library
(cgraph version)

Emden R. Gansner

August 21, 2014
Contents

1 Introduction 4
  1.1 String-based layouts 4
    1.1.1 dot 5
    1.1.2 xdot 5
    1.1.3 plain 6
    1.1.4 plain-ext 7
    1.1.5 GXL and GML 7
  1.2 Graphviz as a library 7

2 Basic graph drawing 8
  2.1 Creating the graph 8
    2.1.1 Attributes 10
    2.1.2 Attribute and HTML Strings 16
  2.2 Laying out the graph 17
  2.3 Rendering the graph 17
    2.3.1 Drawing nodes and edges 19
  2.4 Cleaning up a graph 20

3 Inside the layouts 21
  3.1 dot 22
  3.2 neato 22
  3.3 fdp 24
  3.4 sfdp 24
  3.5 twopi 24
  3.6 circo 25

4 The Graphviz context 25
  4.1 Version-specific data 26

5 Graphics renderers 26
  5.1 The GVJ_t data structure 30
  5.2 Inside the obj_state_t data structure 30
  5.3 Color information 31

6 Adding Plug-ins 32
  6.1 Writing a renderer plug-in 34
  6.2 Writing a device plug-in 35
  6.3 Writing an image loading plug-in 35

7 Unconnected graphs 37

A Compiling and linking 42

B A sample program: simple.c 43

C A sample program: dot.c 44

D A sample program: demo.c 45
E Some basic types and their string representations
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
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.

```plaintext
graph scale width height
```
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 $2 \times n$ 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.

```c
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 as 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

\[\text{Agraph_t* } G = \text{agmemread}(cp);\]

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

\[\text{Agraph_t* } G = \text{agread}(dp, \&\text{disc});\]

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

The alternative technique is to call agopen.

\[\text{Agraph_t* } G = \text{agopen} \text{name, type, } \&\text{disc};\]

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

\[\text{Agraph_t* } G = \text{agopen} \text{"network", Agdirected, 0};\]

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.

\[\text{Agnode_t* } \text{agnode}(\text{Agraph_t*, char*, int});\]
\[\text{Agedge_t* } \text{agedge}(\text{Agraph_t*, Agnode_t*, char*, int});\]

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

<table>
<thead>
<tr>
<th>Graph Type</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agundirected</td>
<td>Non-strict, undirected graph</td>
</tr>
<tr>
<td>Agstrictundirected</td>
<td>Strict, undirected graph</td>
</tr>
<tr>
<td>Agdirected</td>
<td>Non-strict, directed graph</td>
</tr>
<tr>
<td>Agstrictdirected</td>
<td>Strict, directed graph</td>
</tr>
</tbody>
</table>

Table 3: Graph types
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`:

```c
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:

```plaintext
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 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.

---

1 if supported
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 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.

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](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 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

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

Table 4: Geometric node attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint</td>
<td>true</td>
<td>use edge to affect node ranking</td>
</tr>
<tr>
<td>fontname</td>
<td>Times-Roman</td>
<td>font family</td>
</tr>
<tr>
<td>fontsize</td>
<td>14</td>
<td>point size of label</td>
</tr>
<tr>
<td>headclip</td>
<td>true</td>
<td>clip head end to node boundary</td>
</tr>
<tr>
<td>headport</td>
<td>center</td>
<td>position where edge attaches to head node</td>
</tr>
<tr>
<td>label</td>
<td></td>
<td>edge label</td>
</tr>
<tr>
<td>len</td>
<td>1.0/0.3</td>
<td>preferred edge length</td>
</tr>
<tr>
<td>lhead</td>
<td></td>
<td>name of cluster to use as head of edge</td>
</tr>
<tr>
<td>ltail</td>
<td></td>
<td>name of cluster to use as tail of edge</td>
</tr>
<tr>
<td>minlen</td>
<td>1</td>
<td>minimum rank distance between head and tail</td>
</tr>
<tr>
<td>samehead</td>
<td>1</td>
<td>tag for head node; edge heads with the same tag</td>
</tr>
<tr>
<td>sametail</td>
<td></td>
<td>are merged onto the same port</td>
</tr>
<tr>
<td>tailclip</td>
<td>true</td>
<td>clip tail end to node boundary</td>
</tr>
<tr>
<td>tailport</td>
<td>center</td>
<td>position where edge attaches to tail node</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>importance of edge</td>
</tr>
</tbody>
</table>

Table 5: Geometric edge attributes
<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>center</td>
<td>false</td>
<td>† center drawing on page</td>
</tr>
<tr>
<td>clusterrank</td>
<td>local</td>
<td>may be global or none</td>
</tr>
<tr>
<td>compound</td>
<td>false</td>
<td>allow edges between clusters</td>
</tr>
<tr>
<td>concentrate</td>
<td>false</td>
<td>enables edge concentrators</td>
</tr>
<tr>
<td>defaultdist</td>
<td>(1 + (\sum_{e \in E} \text{len})/</td>
<td>E</td>
</tr>
<tr>
<td>dim</td>
<td>2</td>
<td>dimension of layout</td>
</tr>
<tr>
<td>dpi</td>
<td>96/0</td>
<td>dimension of layout</td>
</tr>
<tr>
<td>epsilon</td>
<td>.0001</td>
<td>termination condition</td>
</tr>
<tr>
<td>fontsize</td>
<td>14</td>
<td>point size of label</td>
</tr>
<tr>
<td>margin</td>
<td>† any string</td>
<td>† space placed around drawing</td>
</tr>
<tr>
<td>maxiter</td>
<td>layout dependent</td>
<td>bound on iterations in layout</td>
</tr>
<tr>
<td>mclimit</td>
<td>1.0</td>
<td>scale factor for mincross iterations</td>
</tr>
<tr>
<td>mindist</td>
<td>1.0</td>
<td>minimum distance between nodes</td>
</tr>
<tr>
<td>mode</td>
<td>major</td>
<td>variation of layout</td>
</tr>
<tr>
<td>model</td>
<td>shortpath</td>
<td>model used for distance matrix</td>
</tr>
<tr>
<td>nodesep</td>
<td>.25</td>
<td>separation between nodes, in inches</td>
</tr>
<tr>
<td>nslimit</td>
<td></td>
<td>if set to (f), bounds network simplex iterations by (f(\text{number of nodes})) when setting x-coordinates specify out or in edge ordering</td>
</tr>
<tr>
<td>ordering</td>
<td></td>
<td>† use landscape orientation if rotate is not used and the value is landscape</td>
</tr>
<tr>
<td>orientation</td>
<td>portrait</td>
<td>specify if and how to remove node overlaps</td>
</tr>
<tr>
<td>overlap</td>
<td>true</td>
<td>do components separately, then pack</td>
</tr>
<tr>
<td>pack</td>
<td></td>
<td>† unit of pagination, e.g. &quot;8.5,11&quot;</td>
</tr>
<tr>
<td>packmode</td>
<td>node</td>
<td>granularity of packing</td>
</tr>
<tr>
<td>page</td>
<td></td>
<td>if (\text{quantum} &gt; 0), node label dimensions will be rounded to integral multiples of (\text{quantum})</td>
</tr>
<tr>
<td>quantum</td>
<td></td>
<td>same, min, max, source or sink</td>
</tr>
<tr>
<td>rank</td>
<td></td>
<td>separation between ranks, in inches</td>
</tr>
<tr>
<td>rankdir</td>
<td>TB</td>
<td>approximate aspect ratio desired, fill or auto</td>
</tr>
<tr>
<td>ranksep</td>
<td>.75</td>
<td>If true and there are multiple clusters, re-run crossing minimization</td>
</tr>
<tr>
<td>ratio</td>
<td></td>
<td>synonym for dpi</td>
</tr>
<tr>
<td>remincross</td>
<td></td>
<td>specifies node to be used as root of a layout</td>
</tr>
<tr>
<td>resolution</td>
<td></td>
<td>† If 90, set orientation to landscape</td>
</tr>
<tr>
<td>root</td>
<td></td>
<td>maximum edges with negative cut values to check</td>
</tr>
<tr>
<td>rotate</td>
<td></td>
<td>when looking for a minimum one during network simplex</td>
</tr>
<tr>
<td>searchsize</td>
<td>30</td>
<td>manner of initial node placement</td>
</tr>
<tr>
<td>sep</td>
<td>0.1</td>
<td>factor to increase nodes when removing overlap</td>
</tr>
<tr>
<td>size</td>
<td></td>
<td>maximum drawing size, in inches</td>
</tr>
<tr>
<td>splines</td>
<td></td>
<td>render edges using splines</td>
</tr>
<tr>
<td>start</td>
<td>random</td>
<td>manner of initial node placement</td>
</tr>
<tr>
<td>voro_margin</td>
<td>0.05</td>
<td>factor to increase bounding box when more space is necessary during Voronoi adjustment</td>
</tr>
<tr>
<td>viewport</td>
<td></td>
<td>† Clipping window</td>
</tr>
</tbody>
</table>

Table 6: Geometric graph attributes
concrete output format. For example, *Graphviz* only uses the `center` attribute, which specifies that the graph drawing should be centered within its page, when the library generates a concrete representation. The tables distinguish these device-specific attributes by a † symbol at the start of the Use column.

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

### Table 7: Geometric cluster attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>fontname</td>
<td>Times-Roman</td>
<td>font family</td>
</tr>
<tr>
<td>fontsize</td>
<td>14</td>
<td>point size of label</td>
</tr>
<tr>
<td>label</td>
<td></td>
<td>edge label</td>
</tr>
<tr>
<td>peripheries</td>
<td>1</td>
<td>number of cluster boundaries</td>
</tr>
</tbody>
</table>

### Table 8: Decorative node attributes

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

### Table 9: Decorative edge attributes

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

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

Table 10: Decorative graph attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgcolor</td>
<td></td>
<td>background color for cluster</td>
</tr>
<tr>
<td>color</td>
<td>black</td>
<td>cluster boundary color</td>
</tr>
<tr>
<td>fillcolor</td>
<td>black</td>
<td>cluster fill color</td>
</tr>
<tr>
<td>fontcolor</td>
<td>black</td>
<td>text color</td>
</tr>
<tr>
<td>labeljust</td>
<td>centered</td>
<td>left, right or center alignment for cluster labels</td>
</tr>
<tr>
<td>labelloc</td>
<td>top</td>
<td>top or bottom location for cluster labels</td>
</tr>
<tr>
<td>nojustify</td>
<td>false</td>
<td>context for justifying multiple lines of text</td>
</tr>
<tr>
<td>pencolor</td>
<td>black</td>
<td>cluster boundary color</td>
</tr>
<tr>
<td>style</td>
<td></td>
<td>style options, e.g. bold, dotted, filled:</td>
</tr>
</tbody>
</table>

Table 11: Decorative cluster attributes
fillcolor and bgcolor attributes; or with shapes, the most common attributes being shape, height, width, style, fixedsize, peripheries and regular.

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>bb</td>
<td>bounding box of drawing or cluster</td>
</tr>
<tr>
<td>lp</td>
<td>position of graph, cluster or edge label</td>
</tr>
<tr>
<td>pos</td>
<td>position of node or edge control points</td>
</tr>
<tr>
<td>rects</td>
<td>rectangles used in records</td>
</tr>
<tr>
<td>vertices</td>
<td>points defining node’s boundary, if requested</td>
</tr>
</tbody>
</table>

Table 12: Output position 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.

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

Table 13: Miscellaneous attributes

2.1.2 Attribute and HTML Strings

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.
When using strings as labels, one can have some formatting control via the various inline escape sequences such as \"\n\", \"\l\", \"\N\", etc., and attributes such as fontname and fontcolor. To get a great deal more flexibility, one can use HTML-like labels. In the DOT language, these strings are delimited by angle brackets <...> rather than double quotes in order to be work seamlessly with ordinary strings. Even at the library level, these strings are semantically identical to ordinary strings except when used as labels. To create one of these, one uses agstrdup_html() rather than agstrdup(). The agstrfree() is still used to release the string. For example, one might use the following code to attach an HTML string to a node:

```c
Agnode_t* n;
char* l = agstrdup_html(agroot(n), "<B>some bold text</B>"租车)
agset (n,"label",l);
agstrfree (l);
```

In addition, the function aghtmlstr() can be used query if an attribute string is an HTML string.

### 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.

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 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

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

\(^2\) 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.
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

The first and second arguments are a graphviz context handle and a pointer to the graph to be rendered. The final argument gives, respectively, 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 \texttt{-T} flag when the layout program is invoked from a command shell.

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

\begin{verbatim}
  gvLayout (gvc, g, "dot");
  gvRenderFilename (gvc, g, "png", "out.png");
  gvRenderFilename (gvc, g, "cmap", "out.map");
\end{verbatim}

in which the graph is laid out using the \texttt{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

\begin{verbatim}
  gvRenderData (GVC_t *gvc, Agraph_t* g, char *format, char **result,
               unsigned int *length)
\end{verbatim}

which writes the output of the rendering onto an allocated character buffer. A pointer to this buffer is returned in \texttt{*result} and the number of bytes written is stored in \texttt{length}. After using the buffer, the memory should be freed by the application. As the calling program may rely on a different run-time system than that used by Graphviz, the library provides the function

\begin{verbatim}
  gvFreeRenderData (char *data);
\end{verbatim}

which can be used to free the memory pointed to by \texttt{*result}.

Sometimes, an application will decide to do its own rendering. An application-supplied drawing routine, such as \texttt{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 \texttt{dot} or \texttt{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`, `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.

Edges, even if a line segment, are represented as cubic B-splines or piecewise Bezier curves. The `ED_bezier` 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.

\(^4\)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.
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 \( \text{ND\_shape(n)->name} \) is "record" or "Mrecord", the node consists of a packed collection of rectangles. In this case, \( \text{ND\_shape\_info(n)} \) can be cast to \( \text{field\_t*} \), which describes the recursive partition of the node into rectangles. The value \( b \) of \( \text{field\_t} \) 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 \( \text{ND\_shape(n)->usershape} \) 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, which includes the limiting cases of circles, ellipses, and none. In this case, \( \text{ND\_shape\_info(n)} \) can be cast to \( \text{polygon\_t*} \), 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 \( \text{ND\_shape(n)->name} \) is "box", and use the \( \text{ND\_coord, ND\_width and ND\_height} \) 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 \( \text{shape\_info} \) field of the node. For example, given a polygonal node, its \( \text{ND\_shape\_info(n)} \) field contains a \( \text{vertices} \) 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 \text{xdot} information or to supply its own renderer plugin.

For edges, each \text{bezier} structure has a \text{list} field pointing to an array containing the control points and a \text{size} field giving the number of points in \text{list}, which will always have the form \((3 * n + 1)\). In addition, there are fields for specifying arrowheads. If \( bp \) points to a \text{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 \text{arrowtail} and \text{arrowsize} attributes. Analogously, an arrowhead at the head node is specified by \( bp->eflag \) and the vector from \( bp->list[bp->size-1] \) to \( bp->ep \).

The label field \((\text{ND\_label(n)}, \text{ED\_label(e)}, \text{GD\_label(g)})\) 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{text} field, while the \text{fontname, fontcolor} and \text{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 \text{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.

---

\*This is not quite true but close enough for now.
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.

Note that if you render a graph into the `dot` or `xdot` format, this attaches attributes onto the graph. Some of these attributes are used during layout. For example, the `neato` layout will use the `pos` attribute of the nodes for an initial layout, while the `twopi` layout may set the `root` attribute, which will lock in this attribute for any future layouts using `twopi`. To avoid having these attributes affecting another layout of the graph, the user should set these attributes to the empty string before calling `gvLayout` again.

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 given 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
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\cite{GKN04} mode, (`mode="major"`), or the gradient descent technique proposed by Kamada and Kawai\cite{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 \( 1.0 + L_{\text{avg}} \cdot \sqrt{|V|} \), where \( L_{\text{avg}} \) is the average edge length and \( |V| \) is the number of nodes in the graph. This supplies sufficient separation between components so that they do not overlap. Typically, the larger components will be centrally located, while smaller components will form a ring around the outside.

In some cases, an application may decide to use the circuit model (`model="circuit"`), a model based on electrical circuits as first proposed by Cohen\cite{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\cite{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  \textit{fdp}

The \textit{fdp} layout is similar in appearance to \textit{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 \textit{neato}, \textit{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.

\begin{verbatim}
initialize
position
splines
\end{verbatim}

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

3.4  \textit{sfdp}

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

Unlike \textit{fdp}, \textit{sfdp} does not support cluster subgraphs. It also does not model edge lengths or weights.

\begin{verbatim}
initialize
position
adjust
splines
\end{verbatim}

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

3.5  \textit{twopi}

The radial layout algorithm represented by \textit{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.

\begin{verbatim}
initialize
position
adjust
splines
\end{verbatim}

As usual, the layout commences by initializing the graph. This is followed by the \texttt{position} step, which is parameterized by the central node, specified by the graph’s \texttt{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 \textit{neato}, the layout allows an \texttt{adjust} step to eliminate node-node overlaps. Again as with \textit{neato}, the call to \texttt{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
circle
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 the base part of argv[0] (argv[0] with the directory portion removed) is the name of one of the layout algorithms, this will be bound to the GVC_t value and used at layout time. (This can always be overridden by supplying a "-K" flag, or by supplying a "layout" attribute in the graph.) 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. All of the functions take a GVJ_t* value, which contains various

---

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

```c
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*, int);
void (*end_cluster) (GVJ_t*);
void (*begin_edges) (GVJ_t*);
void (*end_edges) (GVJ_t*);
void (*begin_node) (GVJ_t*, char*, int);
void (*end_node) (GVJ_t*);
void (*begin_edge) (GVJ_t*, char*, bool, char*, int);
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.

**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`. 

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

delayer(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.

del_page(job) Called when the drawing of the 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.

del_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.

del_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.

del_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.

del_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.

del_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.

del_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.

<table>
<thead>
<tr>
<th>txt-&gt;just</th>
<th>p.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>'n'</td>
<td>Center of text</td>
</tr>
<tr>
<td>'l'</td>
<td>Left edge of text</td>
</tr>
<tr>
<td>'r'</td>
<td>Right edge of text</td>
</tr>
</tbody>
</table>

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 place-
ment 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 representa-
tion of some particular type. The renderer can use this information to resolve the color to a represen-
tation 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.

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.

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

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

cenwidth 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.
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.

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.
### Table 14: Color type representations

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

### 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 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 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.

### Table 15: Plug-in API types

<table>
<thead>
<tr>
<th>Kind</th>
<th>Functions</th>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API.render</td>
<td>gvrender_engine_t</td>
<td>gvrender_features_t</td>
<td>Functions for rendering a graph</td>
</tr>
<tr>
<td>API_device</td>
<td>gvdevice_engine_t</td>
<td>-</td>
<td>Functions for initializing and terminating a device</td>
</tr>
<tr>
<td>API_loadimage</td>
<td>gvlloadimage_engine_t</td>
<td>-</td>
<td>Functions for converting from one image format to another</td>
</tr>
<tr>
<td>API_layout</td>
<td>gvlayout_engine_t</td>
<td>gvlayout_features_t</td>
<td>Functions for laying out a graph</td>
</tr>
<tr>
<td>API_textlayout</td>
<td>gvtextlayout_engine_t</td>
<td>-</td>
<td>Functions for resolving font names and text size</td>
</tr>
</tbody>
</table>

![Figure 5: Plug-in fields](image-url)
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.

```c
const 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

```c
const 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:

```c
const 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.\footnote{Normally, for builds intended for local installation dot -c is run during \texttt{make install}. It may be necessary to run this manually if cross-compiling or otherwise manually moving binaries to a different system.}

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 \texttt{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 \texttt{gvrender\_features\_t}. This record provides Graphviz with information about the renderer. Figure 6 list the fields involved. Some of the 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.
- **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 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.

\footnote{The ordering must be done byte-wise using the \texttt{LANG=C} locale for byte comparison.}

```
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
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.

---

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GVRENDER_DOES_ARRONS</td>
<td>Built-in arrowheads on splines</td>
</tr>
<tr>
<td>GVRENDER_DOES LAYERS</td>
<td>Supports graph layers</td>
</tr>
<tr>
<td>GVRENDER_DOES_MULTIGRAPH_OUTPUT FILES</td>
<td>If true, the renderer’s output can contain multiple renderings</td>
</tr>
<tr>
<td>GVRENDER_DOES_TRUECOLOR</td>
<td>Supports a truecolor color model</td>
</tr>
<tr>
<td>GVRENDER_Y GOES DOWN</td>
<td>Output coordinate system has the origin in the upper left corner</td>
</tr>
<tr>
<td>GVRENDER_X11_EVENTS</td>
<td>For GUI plug-ins, defers actual rendering until the GUI event loop invokes job-&gt;callbacks-&gt;refresh()</td>
</tr>
<tr>
<td>GVRENDER_DOES_TRANSFORM</td>
<td>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</td>
</tr>
<tr>
<td>GVRENDER_DOES_LABELS</td>
<td>Wants an object’s label, if any, provided as text during rendering</td>
</tr>
<tr>
<td>GVRENDER_DOES_MAPS</td>
<td>Supports regions to which URLs can be attached. If true, URLs are provided to the renderer, either as part of the job-&gt;obj or via the renderer’s begin_anchor function</td>
</tr>
<tr>
<td>GVRENDER_DOES_MAP_RECTANGLE</td>
<td>Rectangular regions can be mapped</td>
</tr>
<tr>
<td>GVRENDER_DOES_MAP_CIRCLE</td>
<td>Circular regions can be mapped</td>
</tr>
<tr>
<td>GVRENDER_DOES_MAP_POLYGON</td>
<td>Polygons can be mapped</td>
</tr>
<tr>
<td>GVRENDER_DOES_MAP_ELLIPSE</td>
<td>Ellipses can be mapped</td>
</tr>
<tr>
<td>GVRENDER_DOES_MAP_BSPLINE</td>
<td>B-splines can be mapped</td>
</tr>
<tr>
<td>GVRENDER_DOES_TOOLTIP</td>
<td>If true, tooltips are provided to the renderer, either as part of the job-&gt;obj or via the renderer’s begin_anchor function</td>
</tr>
<tr>
<td>GVRENDER_DOES_TARGETS</td>
<td>If true, targets are provided to the renderer, either as part of the job-&gt;obj or via the renderer’s begin_anchor function</td>
</tr>
<tr>
<td>GVRENDER_DOES_Z</td>
<td>Uses a 3D output model</td>
</tr>
</tbody>
</table>

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:

```c
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.

```c
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 the following meanings:

```c
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`

- **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 then 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");
}
The call to \texttt{ccomps} splits the graph \texttt{g} into its connected components. \texttt{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 \texttt{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 \texttt{neato}, allow the input graph to fix the position of certain nodes, indicated by \texttt{NDpinned(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 \texttt{pack} library provides \texttt{pccomps}, an analogue to \texttt{ccomps} for this situation. It has almost the same interface as \texttt{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 \texttt{nodeInduce} to create the corresponding node-induced subgraph, and then lay out the component with \texttt{gvLayout}. Here, we use \texttt{neato} for each layout, but it is possible to use a different layout for each component.\textsuperscript{12}

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

The \texttt{pack_graph} function uses the graph's \texttt{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 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 \texttt{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, \texttt{pack_graph} finishes by recomputing and setting the bounding box attribute \texttt{GD_bb} of the graph.

The final step is to free the component subgraphs.

Although \texttt{dot} and \texttt{neato} have their specialized approaches to unconnected graphs, it should be noted that these are not without their deficiencies. The approach used by \texttt{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

\footnote{At present, the \texttt{dot} layout has a limitation that it only works on a root graph. Thus, to use \texttt{dot} for a component, one needs to create a new copy of the subgraph, apply \texttt{dot} and then copy the position attributes back to the component.}
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


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, lib and bin 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
- cgraph
- 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

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 could have the form:

```bash
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
```

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.

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 writes 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).

```c
#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.

```c
#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.

Note that either the graph or the argv[] values have to specify which layout algorithm is used, as explained in Section 4. Specifically, the input graph must have the layout attribute set, or the command line arguments must contain a valid "-K" flag. If not, gvParseArgs will look at the base name part of argv[0] and use that as the name of desired layout program. For this to work, the executable program needs to be renamed as one of the Graphviz layout programs (cf. Section 1).

```c
#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 */
```
E Some basic types and their string representations

A point type is the structure

```c
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

```c
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 \( p_N \).