Drawing graphs with \textit{dot}

\textit{Eleftherios Koutsofios}
\textit{Stephen C. North}

AT&T Bell Laboratories
Murray Hill, NJ

\textit{dot} draws directed graphs as hierarchies. Like its predecessor, \textit{dag}, it is a Unix filter, makes good drawings, and runs quickly. Its important new features are node ports for drawing data structures with pointers; improved placement of nodes, edge splines and labels; cluster layouts; and an underlying file language for graph tools. Here is a reduced module dependency graph of the SML-NJ compiler. The layout took 3.5 seconds of user time on an HP-9000/730 computer.
1 Basic Graph Drawing

dot draws directed graphs. It reads attributed graph text files and writes drawings, either as graph files or in a graphics language such as PostScript.

dot takes four main steps in drawing a graph. Knowing about these helps you to understand what kind of layouts dot makes, and how you can modify its layouts. The first step assigns discrete ranks to nodes. In a top to bottom drawing, ranks determine Y coordinates. Edges that span more than one rank are broken into chains of “virtual” nodes and unit-length edges. The second step orders nodes within ranks to avoid crossings. The third step sets X coordinates of nodes to keep edges short. The last step routes edge splines. This is the same general approach as dag, which in turn builds on the work of Warfield [War77], Carpano [Car80] and Sugiyama [ST81]. We refer the reader to [GKNV93] for explanation of dot’s algorithms.

dot’s graph language has three kinds of items: graphs, nodes, and edges. The main (outermost) graph can be graph (undirected) or a digraph (directed). Because dot makes layouts of directed graphs, all the examples in this user’s guide use digraph. We have written a separate layout utility, neato, to draw undirected graphs [Nor92]. Within a main graph, a subgraph defines a subset of nodes and edges.

Figure 1 is an example graph in dot’s language. Line 1 gives the graph name and type. The following lines create nodes, edges, or subgraphs, and set attributes. Names may be C identifiers, numbers, or quoted C strings. Quotes protect punctuation or white space.

A node is created the first time its name appears in the file. An edge is created when nodes are joined by the edge operator ->. In the example, line 2 makes edges from main to parse and from parse to execute. Running dot on this file (say graph1.dot) yields the drawing of figure 2

```bash
$ dot -Tps graph1.dot -o graph1.ps
```

The command line option -Tps selects PostScript (EPSF) output. graph1.ps may be printed, displayed by a PostScript viewer, or embedded in another document.

It is often useful to adjust the representation or placement of nodes and edges in the layout. This is done by setting attributes of nodes, edges, or subgraphs in the input file. Attributes are name-value pairs of character strings. Figures 3 and 4 illustrate some layout attributes. In the listing of figure 3, line 2 sets the graph’s size to 4,4 (all dimensions are in inches). This attribute controls the bounding box— the drawing is scaled as necessary to fit.

Node or edge attributes are set off in square brackets. In line 3, the node main is assigned shape box. The edge in line 4 is straightened by increasing its weight (the default is 1). The edge in line 6 is drawn as a dotted line. Line 8 makes edges from execute to make_string and printf. In line 10 the default edge color is set to red. This affects any edges created after this point in the file. Line 11 makes a bold edge labeled 100 times. In line 12, node make_string is given a multi-line label. Line 13 changes the default node to be a box filled with a shade of blue. The node compare inherits these values.

---

1Unlike dag, the .GS command is not needed.
Drawing graphs with dot

1: digraph G {
2:     main -> parse -> execute;
3:     main -> init;
4:     main -> cleanup;
5:     execute -> make_string;
6:     execute -> printf
7:     init -> make_string;
8:     main -> printf;
9:     execute -> compare;
10: }

Figure 1: Small graph

Figure 2: Drawing of small graph
1: digraph G {
2:   size ="4,4";
3:   main [shape=box]; /* this is a comment */
4:   main -> parse [weight=8];
5:   parse -> execute;
6:   main -> init [style=dotted];
7:   main -> cleanup;
8:   execute -> { make_string; printf}
9:   init -> make_string;
10:  edge [color=red];
11:  main -> printf [style=bold,label="100 times"];  
12:  make_string [label="make a\nstring"]; 
13:  node [shape=box,style=filled,color=".7 .3 1.0"];
14:  execute -> compare;
15: }

Figure 3: Fancy graph

Figure 4: Drawing of fancy graph
2 Drawing Attributes

The complete list of attributes that affect graph drawing is summarized in table 1.

2.1 Shapes and Labels

By default, nodes are drawn with `shape=ellipse, width=.75, height=.5`, and labeled by the node name. Other common shapes (box, circle, etc.) are listed in table 1. The node shape plaintext is of particularly interest in that it draws a node without any outline, an important convention in some kinds of diagrams. When drawn, a node's actual size is the greater of the requested size and the area needed for its text label. By default, edges are unlabeled. Node and edge labels can be set explicitly as shown in figure 4. Though it is convenient that nodes are labeled with their names by default, sometimes it is essential to set labels explicitly. For example, in drawing a file directory tree, one might have several directories named `src`, but each one must have a unique node identifier. The node number or full path name are suitable unique identifiers. Then the label of each node can be set to the file name within its directory.

In multi-line labels, \n, \l, \r terminate lines that are centered, or left or right justified.\footnote{The escape \N is an internal symbol for node names.} Graphs and cluster subgraphs may also have labels.

The default font is 14-point Times-Roman, in black. Other font families, sizes, and colors may be selected. Font names should be compatible with the target interpreter (usually PostScript). It is best to use only the standard font families Times, Helvetica, Courier, or Symbol as these are guaranteed to work with any target graphics language. For example, Times-Italic, Times-Bold, or Courier are portable, but AvanteGarde-DemiOblique is not.

Nodes with shape `record` or `polygon` have special properties. Section 3 reviews some details of using records. Polygons are useful for many shapes that are not predefined. They are parameterized by number of sides, peripheries, orientation, skew, and distortion, as illustrated in figures 5 and 6. `peripheries` is the number of borders. For example, a doublecircle has 2 peripheries. `orientation` is clockwise rotation from the X axis in degrees. `skew` is a floating point number (usually between \(-1.0\) and \(1.0\)) that distorts the shape by slanting it from top-to-bottom, for example, turning a box into a parallelogram. `distortion` shrinks from top-to-bottom, for example, turning a box into a trapezoid.

Though there is a way to implement custom node shapes, the details are beyond the scope of this user's guide. Please contact the authors for further information.

2.2 Graphics Styles

Nodes and edges have `color` and `style` attributes. A `color` value can be a hue-saturation-brightness triple (three floating point numbers between 0 and 1), or one of the colors names listed in Appendix B (borrowed from some version of the X window system). The numerical form is convenient for scripts or tools that automatically generate colors. Color name lookup case and punctuation and insensitive, so "warmgrey" and Warm_Grey are equivalent.

We can offer a few hints regarding use of color in graph drawings. First, avoid using too many bright colors. A "rainbow effect" is confusing. It's better to choose a narrower range of colors, or to vary saturation along with hue. Second, when nodes are filled with dark or very saturated colors, labels seem to be more readable with `fontcolor=white` and `fontname=Helvetica`. (We also have PostScript functions for dot that create outline fonts from plain fonts.) Third, you can define your
<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node Attributes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>black</td>
<td>node shape color</td>
</tr>
<tr>
<td>fontcolor</td>
<td>black</td>
<td>type face color</td>
</tr>
<tr>
<td>fontname</td>
<td>Times-Roman</td>
<td>PostScript font family</td>
</tr>
<tr>
<td>fontsize</td>
<td>14</td>
<td>point size of label</td>
</tr>
<tr>
<td>height,width</td>
<td>.5,.75</td>
<td>height and width in inches</td>
</tr>
<tr>
<td>label</td>
<td>node name</td>
<td>any string</td>
</tr>
<tr>
<td>layer</td>
<td>overlay range</td>
<td>all, id or id:id</td>
</tr>
<tr>
<td>shape</td>
<td>ellipse</td>
<td>ellipse, box, circle, doublecircle, diamond, plaintext, record, polygon, epsf</td>
</tr>
<tr>
<td>shapefile</td>
<td></td>
<td>external EPSF file if epsf shape</td>
</tr>
<tr>
<td>style</td>
<td></td>
<td>graphics options, e.g. bold, dotted, filled</td>
</tr>
<tr>
<td><strong>Edge Attributes</strong></td>
<td></td>
<td></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</td>
<td>forward, back, both, or none</td>
</tr>
<tr>
<td>fontcolor</td>
<td>black</td>
<td>type face color</td>
</tr>
<tr>
<td>fontname</td>
<td>Times-Roman</td>
<td>PostScript font family</td>
</tr>
<tr>
<td>fontsize</td>
<td>14</td>
<td>point size of label</td>
</tr>
<tr>
<td>id</td>
<td></td>
<td>optional value to distinguish multiple edges</td>
</tr>
<tr>
<td>label</td>
<td></td>
<td>label, if not empty</td>
</tr>
<tr>
<td>layer</td>
<td>overlay range</td>
<td>all, id or id:id</td>
</tr>
<tr>
<td>minlen</td>
<td>1</td>
<td>minimum rank distance between head and tail</td>
</tr>
<tr>
<td>style</td>
<td></td>
<td>graphics options, e.g. bold, dotted, filled</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>integer reflecting importance of edge.</td>
</tr>
<tr>
<td><strong>Graph Attributes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>center</td>
<td></td>
<td>when true, centers drawing on page</td>
</tr>
<tr>
<td>clusterrank</td>
<td>local</td>
<td>may be global or none</td>
</tr>
<tr>
<td>color</td>
<td>black</td>
<td>background or cluster outline color</td>
</tr>
<tr>
<td>concentrate</td>
<td></td>
<td>enables edge concentrators when TRUE</td>
</tr>
<tr>
<td>fontcolor</td>
<td>black</td>
<td>type face color</td>
</tr>
<tr>
<td>fontname</td>
<td>Times-Roman</td>
<td>PostScript 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>any string</td>
</tr>
<tr>
<td>layerseq</td>
<td></td>
<td>id:id:id:...</td>
</tr>
<tr>
<td>margin</td>
<td>.5,.5</td>
<td>margin included in page</td>
</tr>
<tr>
<td>mclimit</td>
<td>1.0</td>
<td>if set to f, adjusts mincross iterations by (f)</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)/(number of nodes)</td>
</tr>
<tr>
<td>ordering</td>
<td></td>
<td>out (for ordered edges)</td>
</tr>
<tr>
<td>orientation</td>
<td>portrait</td>
<td>may be set to landscape</td>
</tr>
<tr>
<td>page</td>
<td></td>
<td>unit of pagination, e.g. 8.5,11</td>
</tr>
<tr>
<td>rank</td>
<td></td>
<td>same, min, or max</td>
</tr>
<tr>
<td>rankdir</td>
<td>TB</td>
<td>LR (left to right) or TB (top to bottom)</td>
</tr>
<tr>
<td>ranksep</td>
<td>.75</td>
<td>separation between ranks, in inches.</td>
</tr>
<tr>
<td>ratio</td>
<td></td>
<td>approximate aspect ratio desired, or fill</td>
</tr>
<tr>
<td>size</td>
<td></td>
<td>drawing bounding box, in inches</td>
</tr>
</tbody>
</table>

Table 1: Drawing attributes
Figure 5: Example of polygonal shapes for nodes

```
1: digraph G {
2:   a -> b -> c;
3:   b -> d;
4:   a [shape=polygon,sides=5,peripheries=3,color=blue_light,style=filled];
5:   c [shape=polygon,sides=4,skew=.4,label="hello world"]
6:   d [shape=invtriangle];
7:   e [shape=polygon,sides=4,distortion=.7];
8: }
```

Figure 6: Listing of graph with polygonal shapes
Drawing graphs with dot

own color space by redefining nodecolor, edgecolor, or graphcolor in a library file. For example, to use RGB colors, place the following line in a file lib.ps.

```
/nodescolor {setrgbcolor} bind def
```

Use the -l command line option to load this file.

```
dot -Tps -l lib.ps file.dot -o file.ps
```

`style` controls miscellaneous graphics features of nodes, edges, graphs or subgraphs. The style is a list of primitives with optional argument lists. The predefined primitives are `filled solid dashed dotted bold` and `invis`. `filled` when applied to nodes or clusters shades inside the boundary of the object using its color. If the color is not set, light grey is used as the default.

User-defined style primitives can be implemented as custom PostScript procedures. Such primitives are executed inside the `gsave` context of a graph, node, or edge, before any of its marks are drawn. The arg lists are translated to PostScript notation. For example, a node with `style="/setlinewidth(3)"` is drawn with a thick outline. Here, `setlinewidth` is a PostScript built-in, but user-defined PostScript procedures are called the same way. The definition of these procedures can be given in a library file loaded using `-l` as shown above.

Edges have a `dir` attribute to set arrowheads. `dir` may be `forward` (the default), `back`, `both`, or `none`. This refers only to where arrowheads are drawn, and does not change the underlying graph. For example, setting `dir=back` does not exchange the endpoints of a directed edge (unlike the `dag` program).

### 2.3 Drawing Size and Spacing

Often a drawing made with the default node sizes and separations is too big for the target printer or for the space allowed for a figure in a document. There are several ways to try to deal with this problem. First, we will review how `dot` computes the final layout size.

A layout is initially made internally at its "natural" size, using default settings (unless `ratio=compress` was set, as described below). By default, nodes are at least .75 (inches) wide by .5 tall; fonts are 14 points high; nodes are separated by at least .25 and ranks by .5. There is no bound on the size or aspect ratio of the drawing, so if the graph is large, the layout is also large. If you don't specify `size` or `ratio`, then the natural size layout is printed.

The easiest way to control the output size of the drawing is to set `size=x,y` in the graph file (or on the command line using `-G`). This determines the bounding box of the final layout. For example, `size="7.5,10"` fits on an 8.5x11 page (assuming the default page orientation) no matter how big the initial layout. `ratio` also affects layout size. There are a number of cases, depending on the settings of `size` and `ratio`.

**Case 1.** `ratio` was not set. If the drawing already fits within the given `size` then nothing happens. Otherwise, the drawing is reduced uniformly enough to make the critical dimension fit.

If `ratio` was set, there are four subcases.

**Case 2a.** If `ratio=x` where `x` is a floating point number, then the drawing is stretched (adding whitespace) to achieve the requested ratio expressed as drawing `width/height`. For example, `ratio=2.0` makes the drawing twice as wide as it is high. Then the layout is scaled using `size` as in Case 1.
Case 2b. If \texttt{ratio=fill} and \texttt{size=x,y} was set, then the drawing is stretched (adding whitespace) to achieve the ratio $x/y$. The effect is that all of the bounding box given by \texttt{size} is filled. Then scaling is performed as in Case 1.

Case 2c. If \texttt{ratio=compress} and \texttt{size=x,y} was set, then the initial layout is compressed to attempt to fit it on the given bounding box. This trades off layout quality, balance, and symmetry, to pack the layout more tightly. Then scaling is performed as in Case 1.

Case 2d. If \texttt{ratio=auto} then \texttt{size} is ignored and \texttt{dot} computes an “ideal” size using the following heuristic: it first attempts to fit the drawing on one page by reducing to not less than 50\% of its original size. Otherwise, the drawing is printed on multiple pages, using the full area of each page and not reducing under 50\%.

At this point, if \texttt{page} is not set, then the final layout is printed as one page.

If \texttt{page=x,y} is set, then the layout is printed as a sequence of pages that can tiled or assembled into a mosaic. Common settings are \texttt{page="8.5,11"} or \texttt{page="11,17"}. These values refers to the size of the physical device, and are independent of landscape mode. For tiled layouts, you may find it helpful to set smaller margins (the default is \texttt{margin=.5}). Although you can set \texttt{margin=0}, unfortunately, many bitmap printers have an internal hardware margin that cannot be overridden.

If \texttt{rotate=90} is set, then the layout is printed in landscape mode. The $X$ axis of the layout would be along the $Y$ axis of each page. This does not affect the \texttt{dot}'s interpretation of \texttt{size,ratio}, or \texttt{"page}.

A common problem is that a large graph drawn at a small size yields unreadable node labels. To make larger labels, something has to give. There is a limit to the amount of readable text that can fit on one page. Often you can draw a smaller graph by extracting an interesting piece of the original graph before running \texttt{dot}. We have some tools that help with this.

- \texttt{sccmap} - decompose into strongly connected components
- \texttt{tred} - compute transitive reduction (remove edges implied by transitivity)
- \texttt{gpr} - graph processor to select nodes or edges, and contract or remove the rest of the graph
- \texttt{unflatten} - improve aspect ratio of trees by staggering the lengths of leaf edges

With this in mind, here are some thing to try on a given graph:
1. Increase the node \texttt{fontsize}.
2. Use smaller \texttt{ranksep} and \texttt{nodesep}.
3. Use \texttt{ratio=auto}.
4. Use \texttt{ratio=compress} and give a reasonable \texttt{size}.
5. A sans serif font (such as Helvetica) may be more readable than Times when reduced.

2.4 Node and Edge Placement

Sometimes it is natural to make edges point from left to right instead of from top to bottom. If \texttt{rankdir=LR} in the top-level graph, the drawing is rotated in this way. \texttt{TB} (top to bottom) is the default. (\texttt{TB} seems potentially useful for drawing upward-directed graphs, but hasn’t been implemented. In some graphs you could achieve the same effect by reversing the endpoints of edges and setting their \texttt{dir=back}.)

\textit{dot User's Manual, November 20, 1996}
In graphs with time-lines, or in drawings that emphasize source and sink nodes, you may need to constrain rank assignments. The set of a subgraph may be set to \texttt{minrank}, \texttt{maxrank}, or \texttt{samerank}. This constrains the nodes in the subgraph. Figures 7 and 8 illustrate using subgraphs for controlling rank assignment.

In some graphs, the left-to-right ordering of nodes is important. If a subgraph has \texttt{ordering=\texttt{out}} then out-edges within the subgraph having the same tail node fan out from left to right in their order of creation.

Also, when nodes are constrained to the same rank, edges with non-zero weight between them are aimed across the rank in the same direction (left-to-right, or top-to-bottom in a rotated drawing) as far as possible. This fact may be exploited to adjust node ordering by placing invisible edges (\texttt{style="invis"}) where needed.

Fine tuning should be approached cautiously. \texttt{dot} works best when it can make a layout without much "help" or interference in its placement of individual nodes and edges. Layouts can be adjusted somewhat by increasing the weight of certain edges, or by creating invisible edges or nodes using \texttt{style=invis}, and sometimes even by rearranging the order of nodes and edges in the file. But this can backfire because the layouts are not necessarily stable with respect to changes in the input graph. One last adjustment can invalidate all previous changes and make a very bad drawing. A future project we have in mind is to combine the mathematical layout techniques of \texttt{dot} with an interactive front end that allows user-defined hints and constraints.

3 \hspace{1em} Node Ports

A node port is a point where edges may attach to a node. (When an edge is not attached to a port, it is aimed at the node's center.) A node with a port specifier has the syntax \texttt{name:port}. The names and bindings of ports can differ from one node to another, depending on shapes and other attributes. Presently only the \texttt{record} shape has ports. This shape represents a record as recursive lists of labeled boxes. A port refers to the center of one of the boxes. Ports are created by inserting the construct \texttt{<portid>} in a box label, as shown in figures 9 and 10.

Figures 11 and 12 shows how recursive records are drawn. Vertical bars separate fields at the same level, while curly braces enclose subfield lists. Port identifiers are enclosed in angle brackets. (Literal braces, vertical bars, and angle brackets must be escaped.) Spaces are interpreted as separators between tokens (similar to the way most typesetting programs work) so they must be escaped if you want fixed or "hard" spaces. Also, note that records sometimes look better if their input height is set to a small value so the text labels dominate the actual size, as illustrated in figure 9. Otherwise the default node size (:75 by .5) is assumed, as in figure 11.

The example of figures 13 and 14 uses left-to-right drawing in a layout of a hash table.

3.1 \hspace{1em} Clusters

A cluster is a subgraph placed in its own distinct rectangle of the layout. A subgraph is recognized as a cluster when its name has the prefix \texttt{cluster} (unless the graph's \texttt{clusterrank=none}). Cluster labels, fonts, colors, and styles can be set in the usual way. Clusters are drawn by a recursive technique that computes a rank assignment and internal ordering of nodes within clusters. Figure 15 through 17 are cluster layouts and the corresponding graph files.

Figure 7: Graph with constrained ranks
Figure 8: Drawing with constrained ranks
Drawing graphs with dot

```plaintext
digraph g {
    node [shape = record, height = .1];
    node0[label = "<f0> \f1 G <f2> "];
    node1[label = "<f0> \f1 E <f2> "];
    node2[label = "<f0> \f1 B <f2> "];
    node3[label = "<f0> \f1 F <f2> "];
    node4[label = "<f0> \f1 R <f2> "];
    node5[label = "<f0> \f1 H <f2> "];
    node6[label = "<f0> \f1 Y <f2> "];
    node7[label = "<f0> \f1 A <f2> "];
    node8[label = "<f0> \f1 C <f2> "];
    "node0":f2 -> "node4":f1;
    "node0":f0 -> "node1":f1;
    "node1":f0 -> "node2":f1;
    "node1":f2 -> "node3":f1;
    "node2":f2 -> "node8":f1;
    "node2":f0 -> "node7":f1;
    "node4":f2 -> "node6":f1;
    "node4":f0 -> "node5":f1;
    }
```

Figure 9: Binary search tree using records

![Binary search tree using records](image)

Figure 10: Drawing of binary search tree
Drawing graphs with dot

```plaintext
1: digraph structs {
2:   node [shape=record];
3:   struct1 [shape=record, label="<f0> left|<f1> middle|<f2> right"];
4:   struct2 [shape=record, label="<f0> one|<f1> two"];
5:   struct3 [shape=record, label="hello\nworld \{ b |c|here| d|e| f\} g | h"];
6:   struct1:f1 -> struct2:f0;
7:   struct1:f2 -> struct3:here;
8: }
```

Figure 11: Records with nested fields

```plaintext
left middle right
```

```plaintext
one two
hello world
```

Figure 12: Drawing of records

### 3.2 Concentrators

Setting `concentrate=true` on the top level graph enables an edge merging technique to reduce clutter in dense layouts. Edges are merged when they run parallel and have a common endpoint. A beneficial side-effect in fixed-sized layouts is that removal of these edges often permits larger, more readable labels. While `dot`'s concentrators look somewhat like Newbery's [New89], they are found by searching the edges in the layout, not by detecting complete bipartite graphs in the underlying graph. Thus the `dot` approach runs much faster but doesn’t collapse as many edges as Newbery’s algorithm.

### 4 Command Line Options

By default, `dot` operates in filter mode, writing graphs in the input format with layout attributes appended. `-Tps` sets PostScript output. `-Tpcl` emits HPGL/2 with PCL-5 wrappers, for HP Laserwriters. `-Thpgl` emits pure HPGL for wide bed pen plotters. `-Tmif` emits FrameMaker MIF files. In this mode, graph layouts can be loaded into FrameMaker and edited manually. FrameMaker is limited to 8 basic colors.

`-Gname=value` sets a graph attribute default value. Often it is convenient to set size, pagination, and related values on the command line rather than in the graph file. Note that file contents override command line arguments. `-N` or `-E` instead of `-G` set default node or edge attributes.

`-l` loads graphics library files.

`-o` sets the output file.
Drawing graphs with `dot`

```
digraph G {
  nodesep=.05;
  rankdir=LR;
  node [shape=record,width=.1,height=.1];
  node0 [label = "|f0> |<f1> |<f2> |<f3> |<f4> |<f5> |<f6> |",height=2.5];
  node [width = 1.5];
  node1 [label = "{<n> n14 | 719 |<p> }"];
  node2 [label = "{<n> a1 | 805 |<p> }"];
  node3 [label = "{<n> t20 | 959 |<p> }"]; 
  node4 [label = "{<n> o15 | 794 |<p> }"];
  node5 [label = "{<n> i9 | 718 |<p> }"]; 
  node6 [label = "{<n> s19 | 659 |<p> }"]; 
  node7 [label = "{<n> e5 | 989 |<p> }";//
  node0:f0 -> node1:n;
  node0:f1 -> node2:n;
  node0:f2 -> node3:n;
  node0:f5 -> node4:n;
  node0:f6 -> node5:n;
  node2:p -> node6:n;
  node4:p -> node7:n;
}
```

Figure 13: Hash table graph file

![Hash table graph file](image1)

Figure 14: Drawing of hash table

digraph G {
    subgraph cluster0 {
        node [style=filled, color=white];
        style=filled;
        color=lightgrey;
        a0 -> a1 -> a2 -> a3;
        label = "process #1";
    }
    subgraph cluster1 {
        node [style=filled];
        b0 -> b1 -> b2 -> b3;
        label = "process #2";
        color=blue
    }
    start -> a0;
    start -> b0;
    a1 -> b3;
    b2 -> a3;
    a3 -> a0;
    a3 -> end;
    b3 -> end;
    start [shape=Mdiamond];
    end [shape=Msquare];
}

Figure 15: Process diagram with clusters
Figure 16: Call graph with labeled clusters
Drawing graphs with dot

```plaintext
digraph G {
  size="8,6"; ratio=fill; node[fontsize=24];
  ciafan->computeфан; fan->increment; computeфан->fan; stringdup->fatal;
  main->exit; main->interp_err; main->ciafan; main->fatal; main->malloc;
  main->strcpy; main->getopt; main->init_index; main->strlen; fan->fatal;
  fan->ref; fan->interp_err; ciafan->def; fan->free; computeфан->strftime;
  computeфан->get_sym_fields; fan->exit; fan->malloc; increment->strcmp;
  computeфан->malloc; fan->strftime; fan->strlen; computeфан->strcpy;
  computeфан->realloc; computeфан->strlen; debug->strftime; debug->strcat;
  stringdup->malloc; fatal->strftime; stringdup->strcpy; stringdup->strlen;
  fatal->exit;
  subgraph "cluster_error.h" { label="error.h"; interp_err; }
  subgraph "cluster_nfio.h" { label="nfio.h"; strftime; }
  subgraph "cluster_ciafan.c" { label="ciafan.c"; ciafan; computeфан;
    increment; }
  subgraph "cluster_util.c" { label="util.c"; stringdup; fatal; debug; }
  subgraph "cluster_query.h" { label="query.h"; ref; def; }
  subgraph "cluster_field.h" { get_sym_fields; }
  subgraph "cluster_stdio.h" { label="stdio.h"; strftime; strftime; }
  subgraph "cluster_libc.a" { getopt; }
  subgraph "cluster_stdlib.h" { label="stdlib.h"; exit; malloc; free; realloc; }
  subgraph "cluster_index.h" { init_index; }
  subgraph "cluster_string.h" { label="string.h"; strcpy;_strlen; strcmp; strcat;
}
```

Figure 17: Call graph file
-v requests verbose output. In processing large layouts, the verbose messages may give some estimate of dot’s progress.

-\v prints the version number.

5 Miscellaneous

In the top-level graph heading, a graph may be declared a strict digraph. This forbids the creation of self-arcs and multi-edges; they are ignored in the input file.

If a subgraph appears with a body more than once in a graph file, its contents are the union of all the nodes and edges. An edge id is an optional string for referencing an edge that was previously created. When set, the triple (tail node, head node, key) form a unique edge key. Otherwise, a new internal id is generated for each distinct edge between the same pair of nodes. An id may be any string.

6 Conclusions

dot produces nicer drawings than dag and has some features to help make more readable drawings. It is not as fast as dag. Since it still takes only a second or two on reasonable inputs, the new features more than compensate.

In writing graph drawing programs, we have found that it does not take long to get the first drawings, but it takes a great deal of work to get truly good drawings. While there is still plenty of room for improvement in dot, we have accomplished our principal goals concerning aesthetics, performance and new features. Since the basic algorithms of dot work well, we have a good basis for further research into problems such as methods for drawing large graphs and on-line (animated) graph drawing.

7 Acknowledgements

We thank Emden Gansner and Phong Vo for their advice about graph drawing algorithms and programming. The graph library uses Phong’s splay tree dictionary library. Also, the users of dag gave us many good suggestions: Emden Gansner, Guy Jacobson, and Randy Hackbarth reviewed earlier drafts of this manual. John Ellson wrote the generalized polygon shape and spent considerable effort to make it robust and efficient. He also wrote the GIF and ISMAP generators.

References


A  Graph File Grammar

The following is an abstract grammar of graph files. Terminals are shown in typewriter font and nonterminals in italics. Angle brackets ⟨ and ⟩ indicate grouping when needed. Double-line brackets [ and ] enclose optional items. Vertical bars | separate alternatives.

```
grap[h] → [strict] ⟨ digraph | graph ⟩ id { stmt-list }
stmt-list → [ stmt ; ] [ stmt-list ]
stmt → attr-stmt | node-stmt | edge-stmt | subgraph | id = id
attr-stmt → ⟨ graph | node | edge ⟩ [ attr-list ]
attr-list → id = id [ attr-list ]
node-stmt → node-id [ opt-attrs ]
node-id → id [: id ]
opt-attrs → [ attr-list ]
edge-stmt → ⟨ node-id | subgraph ⟩ edgeRHS [ opt-attrs ]
edgeRHS → edgeop ⟨ node-id | subgraph ⟩ [ edgeRHS ]
subgraph → [ subgraph id ] { stmt-list } | subgraph id
```

An `id` is any alphanumeric string not beginning with a digit, but possibly including underscores; or a number; or any quoted string possibly containing escaped quotes.

An `edgeop` is `->` in directed graphs and `--` in undirected graphs.

Semicolons aid readability but are not required except in the rare case that a named subgraph with no body immediately precedes an anonymous subgraph, because under precedence rules this sequence is parsed as a subgraph with a heading and a body.

B  Plain Output File Format (-Tplain)

The "plain" output format of `dot` lists node and edge coordinates that are usually needed by front end programs, in a line-oriented style.

The first line is:

```
grap[h] scalefactor bounding-box x bounding-box y
```

All coordinates are in default units (1/72 of an inch), unscaled.

The next group of lines list the nodes in the format:

```
nod[e] name x y xsize ysize label text
```

The name is a unique identifier. If it contains whitespace or punctuation, it is quoted.

The next group of lines list edges:

```
edge tailname headname n x₁ y₁ x₂ y₂ ... xₙ yₙ opt-text opt-x opt-y
```

`n` is the number of coordinate pairs that follow as Bezier spline control points. If the edge is labeled, then the label text and coordinates are listed as the rightmost three items on the line.

The last line is always:

```
stop
```
C Layout Attributes

Layout coordinates are in the default PostScript coordinate system. Node coordinates refer to their center points. The edge spline is a list of $3n + 1$ points, plus optional $p_s$ and optional $p_e$ points.

The $3n + 1$ points are the Bezier control points. Points $p_0, p_1, p_2, p_3$ are the first bezier spline, $p_3, p_4, p_5, p_6$ are the second, etc.

The $p_s$ point is present if there’s an arrow at $p_0$. In this case the arrow is from $p_0$ to point $p_s$, where $p_s$ is actually on the node’s boundary and $p_0$ is further away. If there is no arrow, $p_0$ is on the node’s boundary. Similarly, $p_e$ is for an arrow on the other endpoint of the edge.

Currently, edge points are listed top-to-bottom (or left-to-right) regardless of the orientation of the edge. This may change.

D Layers

dot has a feature for drawing a parts of a single diagram on a sequence of overlapping “layers.” Typically the layers are overhead transparencies. To activate this feature, one must set the graph’s global layerseq to a list of identifiers. A node and edge can be assigned to a a layer or range of layers. all is a reserved name for all layers (and can be used at either end of a range, e.g. design:all or all:code). For example:

```plaintext
layerseq = "spec:design:code:debug:ship";
node90 [layer = "code"];
node91 [layer = "design:debug"];  
nodes0 -> node91 [layer = "all"];  
nodes2 [layer = "all:code"];
```

In a layered graph, if a node or edge has no layer assignment, but incident edges or nodes do, then its layer specification is inferred from these. To change the default so that nodes and edges with no layer appear on all layers, insert near the beginning of the graph file:

```plaintext
node [layer=all];
edge [layer=all];
```

There is presently no way to specify a set of layers that are not a continuous range.

When PostScript output is selected, the color sequence for layers is set in the array layercolorseq. This array is indexed starting from 1, and every element must be a 3-array to be interpreted as a color coordinate. The adventurous may learn further from reading dot’s PostScript output.
## E Color Names

<table>
<thead>
<tr>
<th>Whites</th>
<th>Reds</th>
<th>Yellows</th>
<th>Blues</th>
</tr>
</thead>
<tbody>
<tr>
<td>floralwhite</td>
<td>hotpink[1-4]</td>
<td>lightgoldenrodyellow</td>
<td>darkslateblue</td>
</tr>
<tr>
<td>lavender</td>
<td>mediumvioletred</td>
<td></td>
<td>lightskyblue[1-4]</td>
</tr>
<tr>
<td>linen</td>
<td>pink[1-4]</td>
<td></td>
<td>mediumslateblue</td>
</tr>
<tr>
<td>mintcream</td>
<td>red[1-4]</td>
<td></td>
<td>midnightblue</td>
</tr>
<tr>
<td>mocassin</td>
<td>tomato[1-4]</td>
<td></td>
<td>navyblue</td>
</tr>
<tr>
<td>oldlace</td>
<td></td>
<td></td>
<td>royalblue[1-4]</td>
</tr>
<tr>
<td>papayawhip</td>
<td></td>
<td></td>
<td>skyblue[1-4]</td>
</tr>
<tr>
<td>thistle[1-4]</td>
<td></td>
<td></td>
<td>Magentas</td>
</tr>
<tr>
<td>wheat[1-4]</td>
<td></td>
<td></td>
<td>blueviolet</td>
</tr>
<tr>
<td>white</td>
<td></td>
<td></td>
<td>darkorchid[1-4]</td>
</tr>
<tr>
<td>whitesmoke</td>
<td></td>
<td></td>
<td>darkviolet</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>magenta[1-4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mediumorchid[1-4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mediumpurple[1-4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mediumvioletred</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>orchid[1-4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>palevioletred</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>plum[1-4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>purple[1-4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>violet</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>violetred[1-4]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Greens</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>chartreuse[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>darkgreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>darkslategreen[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>darseagreen[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>forestgreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>green[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>greennyellow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lightgreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lightseagreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>limegreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mediumseagreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mediumslategreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mediumspringgreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mintcream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>olive[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>paledgreen[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>seagreen[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>springgreen[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yellowgreen</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cyans</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aquamarine[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cyan[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>darkturquoise</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lightcyan[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mediumaquamarine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mediumturquoise</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>paleturquoise[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Oranges</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>darkorange[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>orange[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>orangered[1-4]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Blacks</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>black</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The color names are listed in the order of their appearance in the documentation. The numbers in square brackets indicate the range of the color index values for each color.