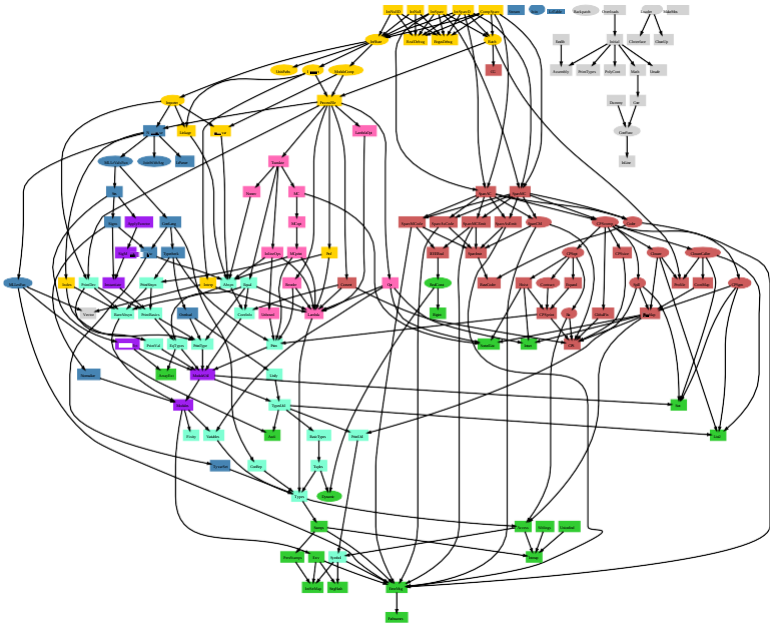
Drawing graphs with *dot*

Emden R. Gansner and Eleftherios Koutsofios and Stephen North

December 22, 2009

dot draws directed graphs as hierarchies. It runs as a command line program, web visualization service, or with a compatible graphical interface. Its features include well-tuned layout algorithms for placing nodes and edge splines, edge labels, "record" shapes with "ports" for drawing data structures; cluster layouts; and an underlying file language for stream-oriented graph tools. Below is a reduced module dependency graph of an SML-NJ compiler that took 0.23 seconds of user time on a 3 GHz Intel Xeon.



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 format such as GIF, PNG, SVG, PDF, or PostScript.

dot draws graphs in four main phases. Knowing this helps you to understand what kind of layouts *dot* makes and how you can control them. The layout procedure used by *dot* relies on the graph being acyclic. Thus, the first step is to break any cycles which occur in the input graph by reversing the internal direction of certain cyclic edges. The next step assigns nodes to discrete ranks or levels. 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 third step orders nodes within ranks to avoid crossings. The fourth step sets X coordinates of nodes to keep edges short, and the final step routes edge splines. This is the same general approach as most hierarchical graph drawing programs, based on the work of Warfield [War77], Carpano [Car80] and Sugiyama [STT81]. We refer the reader to [GKNV93] for a thorough explanation of *dot*'s algorithms.

dot accepts input in the *DOT* language (cf. Appendix D). This language describes three main kinds of objects: graphs, nodes, and edges. The main (outermost) graph can be directed (`digraph`) or undirected `graph`. Because *dot* makes layouts of directed graphs, all the following examples use `digraph`. (A separate layout utility, *neato*, draws undirected graphs [Nor92].) Within a main graph, a `subgraph` defines a subset of nodes and edges.

Figure 1 is an example graph in the *DOT* language. Line 1 gives the graph name and type. The lines that follow create nodes, edges, or subgraphs, and set attributes. Names of all these objects may be C identifiers, numbers, or quoted C strings. Quotes protect punctuation and white space.

A node is created when its name first 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 (call it `graph1.dot`)

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

yields the drawing of Figure 2. 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

```
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

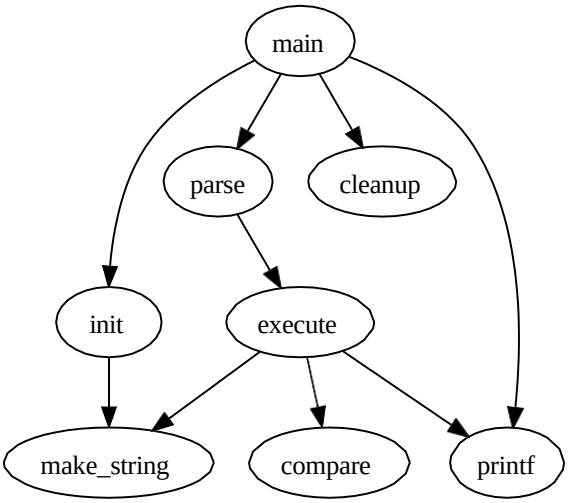


Figure 2: Drawing of small graph

`size` to `4,4` (in inches). This attribute controls the size of the drawing; if the drawing is too large, it is scaled uniformly 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.

2 Drawing Attributes

The main attributes that affect graph drawing are summarized in Appendices A, B and C. For more attributes and a more complete description of the attributes, you should refer to the *Graphviz* web site, specifically

`c/info/attrs.html`

2.1 Node Shapes

Nodes are drawn, by default, with `shape=ellipse`, `width=.75`, `height=.5` and labeled by the node name. Other common shapes include `box`, `circle`, `record` and `plaintext`. A list of the main node shapes is given in Appendix H. 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. In cases where the graph structure is of main concern, and especially when the graph is moderately large, the `point` shape reduces nodes to display minimal content. When drawn, a node's actual size is the greater of the requested size and the area needed for its text label, unless `fixedsize=true`, in which case the `width` and `height` values are enforced.

Node shapes fall into two broad categories: polygon-based and record-based.[1] All node shapes except `record` and `Mrecord` are considered polygonal, and are modeled by the number of sides (ellipses and circles being special cases), and a few other geometric properties. Some of these properties can be specified in a graph. If `regular=true`, the node is forced to be regular. The parameter

[1]There is a way to implement custom node shapes, using `shape=epsf` and the `shapefile` attribute, and relying on PostScript output. The details are beyond the scope of this user's guide. Please contact the authors for further information.

```
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];   // so is this
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

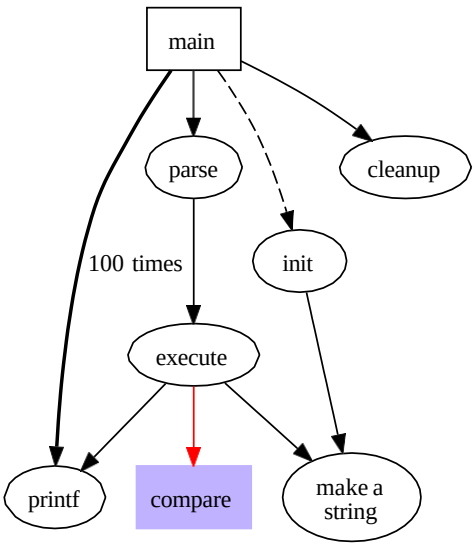


Figure 4: Drawing of fancy graph

`peripheries` sets the number of boundary curves drawn. For example, a doublecircle has `peripheries=2`. The `orientation` attribute specifies a clockwise rotation of the polygon, measured in degrees.

The shape `polygon` exposes all the polygonal parameters, and is useful for creating many shapes that are not predefined. In addition to the parameters `regular`, `peripheries` and `orientation`, mentioned above, polygons are parameterized by number of sides `sides`, `skew` and `distortion`. `skew` is a floating point number (usually between -1.0 and 1.0) that distorts the shape by slanting it from top-to-bottom, with positive values moving the top of the polygon to the right. Thus, `skew` can be used to turn a box into a parallelogram. `distortion` shrinks the polygon from top-to-bottom, with negative values causing the bottom to be larger than the top. `distortion` turns a box into a trapezoid. A variety of these polygonal attributes are illustrated in Figures 6 and 5.

Record-based nodes form the other class of node shapes. These include the shapes `record` and `Mrecord`. The two are identical except that the latter has rounded corners. These nodes represent recursive lists of fields, which are drawn as alternating horizontal and vertical rows of boxes. The recursive structure is determined by the node's `label`, which has the following schema:

rlabel → *field* ('|' *field*)*
field → *boxLabel* | '' *rlabel* ''
boxLabel → ['<' *string* '>'] [*string*]

Literal braces, vertical bars and angle brackets must be escaped. Spaces are interpreted as separators between tokens, so they must be escaped if they are to appear literally in the text. The first string in a *boxLabel* gives a name to the field, and serves as a port name for the box (cf. Section 3.1). The second string is used as a label for the field; it may contain the same escape sequences as multi-line labels (cf. Section 2.2). The example of Figures 7 and 8 illustrates the use and some properties of records.

2.2 Labels

As mentioned above, the default node label is its name. Edges are unlabeled by default. Node and edge labels can be set explicitly using the `label` attribute as shown in Figure 4.

Though it may be convenient to label nodes by name, at other times labels must be set 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.

```
1:  digraph G {
2:      a -> b -> c;
3:      b -> d;
4:      a [shape=polygon,sides=5,peripheries=3,color=lightblue,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 5: Graph with polygonal shapes

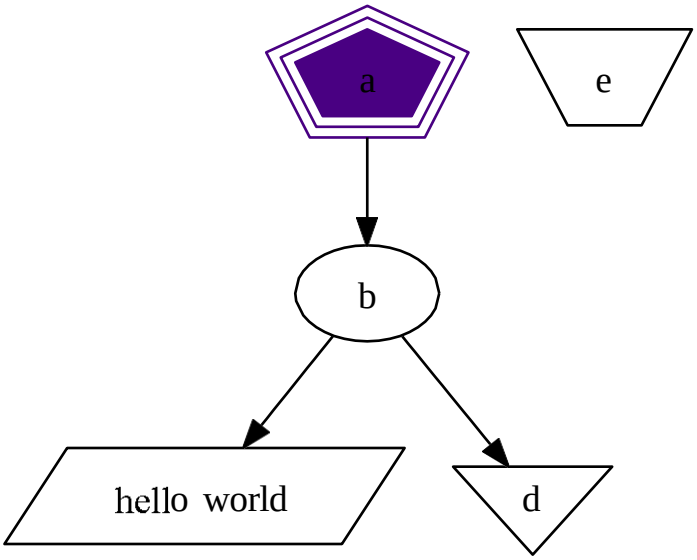


Figure 6: Drawing of polygonal node shapes

```
1:  digraph structs {
2:  node [shape=record];
3:      struct1  [shape=record,label="<f0> left|<f1> mid\ dle|<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  -> struct2;
7:      struct1  -> struct3;
8:  }
```

Figure 7: Records with nested fields

The inode 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.

Multi-line labels can be created by using the escape sequences `\n`, `\l`, `\r` to terminate lines that are centered, or left or right justified.[2]

Graphs and cluster subgraphs may also have labels. Graph labels appear, by default, centered below the graph. Setting `labelloc=t` centers the label above the graph. Cluster labels appear within the enclosing rectangle, in the upper left corner. The value `labelloc=b` moves the label to the bottom of the rectangle. The setting `labeljust=r` moves the label to the right.

The default font is 14-point Times-Roman, in black. Other font families, sizes and colors may be selected using the attributes `fontname`, `fontsize` and `fontcolor`. Font names should be compatible with the target interpreter. 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`, and `Courier` are portable; `AvanteGarde-DemiOblique` isn't.

For bitmap output, such as GIF or JPG, *dot* relies on having these fonts available during layout. Most precompiled installations of *Graphviz* use the fontconfig library for matching font names to available fontfiles. fontconfig comes with a set of utilities for showing matches and installing fonts. Please refer to the font- config documentation, or the external *Graphviz* FontFAQ or for further details. If *Graphviz* is built without fontconfig (which usually means you compiled it from source code on your own), the `fontpath` attribute can specify a list of directories[3] which should be searched for the font files. If this is not set, *dot* will use the DOTFONTPATH environment variable or, if this is not set, the GDFONTPATH environment variable. If none of these is set, *dot* uses a built-in list.

Edge labels are positioned near the center of the edge. Usually, care is taken to prevent the edge label from overlapping edges and nodes. It can still be difficult, in a complex graph, to be certain which edge a label belongs to. If the `decorate` attribute is set to true, a line is drawn connecting the label to its edge. Sometimes avoiding collisions among edge labels and edges forces the drawing to be bigger than desired. If `labelfloat=true`, *dot* does not try to prevent such overlaps, allowing a more compact drawing.

An edge can also specify additional labels, using `headlabel` and `taillabel`, which are be placed near the ends of the edge. The characteristics of these labels are specified using the attributes `labelfontname`, `labelfontsize` and

[2] The escape sequence \N is an internal symbol for node names.

[3] For Unix-based systems, this is a concatenated list of pathnames, separated by colons. For Windows-based systems, the pathnames are separated by semi-colons.

`labelfontcolor`. These labels are placed near the intersection of the edge and the node and, as such, may interfere with them. To tune a drawing, the user can set the `labelangle` and `labeldistance` attributes. The former sets the angle, in degrees, which the label is rotated from the angle the edge makes incident with the node. The latter sets a multiplicative scaling factor to adjust the distance that the label is from the node.

2.3 HTML-like Labels

In order to allow a richer collection of attributes at a finer granularity, *dot* accepts HTML-like labels using HTML syntax. These are specified using strings that are delimited by `< . . . >` rather than double-quotes. Within these delimiters, the string must follow the lexical, quoting, and syntactic conventions of HTML.

By using the `<TABLE>` element, these labels can be viewed as an extension of and replacement for `shape=record`. With these, one can alter colors and fonts at the box level, and include images. The `PORT` attribute of a `<TD>` element provides a port name for the cell (cf. Section 3.1).

Although HTML-like labels are just a special type of label attribute, one frequently uses them as though they were a new type of node shape similar to records. Thus, when these are used, one often sees `shape=none` and `margin=0`. Also note that, as a label, these can be used with edges and graphs as well as nodes.

Figures 9 and 10 give an example of the use of HTML-like labels.

2.4 Graphics Styles

Nodes and edges can specify a `color` attribute, with black the default. This is the color used to draw the node's shape or the edge. A `color` value can be a hue-saturation-brightness triple (three floating point numbers between 0 and 1, separated by commas); one of the colors names listed in Appendix J (borrowed from some version of the X window system); or a red-green-blue (RGB) triple[4] (three hexadecimal number between 00 and FF, preceded by the character '#'). Thus, the values `"orchid"`, `"0.8396,0.4862,0.8549"` and `"#DA70D6"` are three ways to specify the same color. The numerical forms are convenient for scripts or tools that automatically generate colors. Color name lookup is case-insensitive and ignores non-alphanumeric characters, 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 is better to

[4] A fourth form, RGBA, is also supported, which has the same format as RGB with an additional fourth hexadecimal number specifying alpha channel or transparency information.

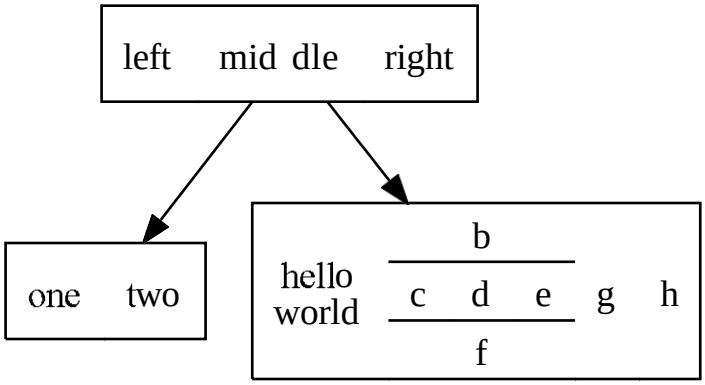


Figure 8: Drawing of records

```
 1:  digraph html {
 2:      abc [shape=none, margin=0, label=<
 3:  <TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0"  CELLPADDING="4">
 4:    <TR><TD ROWSPAN="3"><FONT COLOR="red">hello</FONT><BR/>world</TD>
 5:        <TD COLSPAN="3">b</TD>
 6:        <TD ROWSPAN="3" BGCOLOR="lightgrey">g</TD>
 7:        <TD ROWSPAN="3">h</TD>
 8:    </TR>
 9:    <TR><TD>c</TD>
10:        <TD PORT="here">d</TD>
11:        <TD>e</TD>
12:    </TR>
13:    <TR><TD COLSPAN="3">f</TD>
14:    </TR>
15:  </TABLE>>];
16:  }
```

Figure 9: HTML-like labels

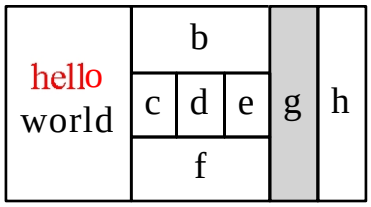


Figure 10: Drawing of HTML-like labels

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, in certain output formats, you can define your own color space. For example, if using PostScript for output, you can redefine `nodecolor`, `edgecolor`, or `graphcolor` in a library file. Thus, to use RGB colors, place the following line in a file `lib.ps`.

```
/nodecolor {setrgbcolor} bind def
```

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

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

The `style` attribute controls miscellaneous graphics features of nodes and edges. This attribute is a comma-separated list of primitives with optional argument lists. The predefined primitives include `solid`, `dashed`, `dotted`, `bold` and `invis`. The first four control line drawing in node boundaries and edges and have the obvious meaning. The value `invis` causes the node or edge to be left undrawn. The style for nodes can also include `filled`, `diagonals` and `rounded`. `filled` shades inside the node using the color `fillcolor`. If this is not set, the value of `color` is used. If this also is unset, light grey[5] is used as the default. The `diagonals` style causes short diagonal lines to be drawn between pairs of sides near a vertex. The `rounded` style rounds polygonal corners.

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 argument lists are translated to PostScript notation. For example, a node with `style="setlinewidth(8)"` 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` causes an arrowhead to be drawn at the tail and no arrowhead at the head, but it does not exchange the endpoints of the edge. The attributes `arrowhead` and `arrowtail` specify the style of arrowhead, if any, which is used at the head and tail ends of the edge. Allowed values are `normal`, `inv`, `dot`, `invdot`, `odot`, `invodot`

[5] The default is black if the output format is MIF, or if the shape is `point`.

and `none` (cf. Appendix I). The attribute `arrowsize` specifies a multiplicative factor affecting the size of any arrowhead drawn on the edge. For example, `arrowsize=2.0` makes the arrow twice as long and twice as wide.

In terms of style and color, clusters act somewhat like large box-shaped nodes, in that the cluster boundary is drawn using the cluster's `color` attribute and, in general, the appearance of the cluster is affected the `style`, `color` and `fillcolor` attributes.

If the root graph has a `bgcolor` attribute specified, this color is used as the background for the entire drawing, and also serves as the default fill color.

2.5 Drawing Orientation, Size and Spacing

Two attributes that play an important role in determining the size of a *dot* drawing are `nodesep` and `ranksep`. The first specifies the minimum distance, in inches, between two adjacent nodes on the same rank. The second deals with rank separation, which is the minimum vertical space between the bottoms of nodes in one rank and the tops of nodes in the next. The `ranksep` attribute sets the rank separation, in inches. Alternatively, one can have `ranksep=equally`. This guarantees that all of the ranks are equally spaced, as measured from the centers of nodes on adjacent ranks. In this case, the rank separation between two ranks is at least the default rank separation. As the two uses of `ranksep` are independent, both can be set at the same time. For example, `ranksep="1.0 equally"` causes ranks to be equally spaced, with a minimum rank separation of 1 inch.

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). 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 size 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 scaled up in one dimension to achieve the requested ratio expressed as drawing $height/width$. For example, `ratio=2.0` makes the drawing twice as high as it is wide. Then the layout is scaled using `size` as in Case 1.

Case 2b. If `ratio=fill` and `size=`x, y was set, then the drawing is scaled up in one dimension to achieve the ratio y/x. Then scaling is performed as in Case 1. The effect is that all of the bounding box given by `size` is filled.

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

Case 2d. If `ratio=auto` and the `page` attribute is set and the graph cannot be drawn on a single page, then `size` is ignored and *dot* computes an "ideal" size. In particular, the size in a given dimension will be the smallest integral multiple of the page size in that dimension which is at least half the current size. The two dimensions are then scaled independently to the new size.

If `rotate=90` is set, or `orientation=landscape`, then the drawing is rotated 90° into landscape mode. The X axis of the layout would be along the Y axis of each page. This does not affect *dot*'s interpretation of `size`, `ratio` or `page`.

At this point, if `page` is not set, then the final layout is produced as one page. If `page=`x, y is set, then the layout is printed as a sequence of pages which can be tiled or assembled into a mosaic. Common settings are `page="8.5,11"` or `page="11,17"`. These values refer to the full size of the physical device; the actual area used will be reduced by the margin settings. (For printer output, the default is 0.5 inches; for bitmap-output, the X and Y margins are 10 and 2 points, respectively.) For tiled layouts, it may be helpful to set smaller margins. This can be done by using the `margin` attribute. This can take a single number, used to set both margins, or two numbers separated by a comma to set the x and y margins separately. As usual, units are in inches. Although one can set `margin=0`, unfortunately, many bitmap printers have an internal hardware margin that cannot be overridden.

The order in which pages are printed can be controlled by the `pagedir` attribute. Output is always done using a row-based or column-based ordering, and `pagedir` is set to a two-letter code specifying the major and minor directions. For example, the default is `BL`, specifying a bottom-to-top (`B`) major order and a left-to-right (`L`) minor order. Thus, the bottom row of pages is emitted first, from left to right, then the second row up, from left to right, and finishing with the top row, from left to right. The top-to-bottom order is represented by `T` and the right-to-left

order by `R`.

If `center=true` and the graph can be output on one page, using the default page size of 8.5 by 11 inches if `page` is not set, the graph is repositioned to be centered on that 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 *dot*. We have some tools that help with this.

sccmap decompose the graph into strongly connected components

tred compute transitive reduction (remove edges implied by transitivity)

gvpr graph processor to select nodes or edges, and contract or remove the rest of the graph

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 `fontsize`.
2. Use smaller `ranksep` and `nodesep`.
3. Use `ratio=auto`.
4. Use `ratio=compress` and give a reasonable `size`.
5. A sans serif font (such as Helvetica) may be more readable than Times when reduced.

2.6 Node and Edge Placement

Attributes in *dot* provide many ways to adjust the large-scale layout of nodes and edges, as well as fine-tune the drawing to meet the user's needs and tastes. This section discusses these attributes[6].

Sometimes it is natural to make edges point from left to right instead of from top to bottom. If `rankdir=LR` in the top-level graph, the drawing is rotated in this way. `TB` (top to bottom) is the default. The mode `rankdir=BT` is useful for drawing upward-directed graphs. For completeness, one can also have `rankdir=RL`.

[6]For completeness, we note that *dot* also provides access to various parameters which play technical roles in the layout algorithms. These include `mclimit`, `nslimit`, `nslimit1`, `remincross` and `searchsize`.

In graphs with time-lines, or in drawings that emphasize source and sink nodes, you may need to constrain rank assignments. The `rank` of a subgraph may be set to `same`, `min`, `source`, `max` or `sink`. A value `same` causes all the nodes in the subgraph to occur on the same rank. If set to `min`, all the nodes in the subgraph are guaranteed to be on a rank at least as small as any other node in the layout[7]. This can be made strict by setting `rank=source`, which forces the nodes in the subgraph to be on some rank strictly smaller than the rank of any other nodes (except those also specified by `min` or `source` subgraphs). The values `max` or `sink` play an analogous role for the maximum rank. Note that these constraints induce equivalence classes of nodes. If one subgraph forces nodes `A` and `B` to be on the same rank, and another subgraph forces nodes `C` and `B` to share a rank, then all nodes in both subgraphs must be drawn on the same rank. Figures 11 and 12 illustrate using subgraphs for controlling rank assignment.

In some graphs, the left-to-right ordering of nodes is important. If a subgraph has `ordering=out`, then out-edges within the subgraph that have the same tail node wll fan-out from left to right in their order of creation. (Also note that flat edges involving the head nodes can potentially interfere with their ordering.)

There are many ways to fine-tune the layout of nodes and edges. For example, if the nodes of an edge both have the same `group` attribute, *dot* tries to keep the edge straight and avoid having other edges cross it. The `weight` of an edge provides another way to keep edges straight. An edge's `weight` suggests some measure of an edge's importance; thus, the heavier the weight, the closer together its nodes should be. *dot* causes edges with heavier weights to be drawn shorter and straighter.

Edge weights also play a role when nodes are constrained to the same rank. Edges with non-zero weight between these nodes 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 (`style="invis"`) where needed.

The end points of edges adjacent to the same node can be constrained using the `samehead` and `sametail` attributes. Specifically, all edges with the same head and the same value of `samehead` are constrained to intersect the head node at the same point. The analogous property holds for tail nodes and `sametail`.

During rank assignment, the head node of an edge is constrained to be on a higher rank than the tail node. If the edge has `constraint=false`, however, this requirement is not enforced.

In certain circumstances, the user may desire that the end points of an edge never get too close. This can be obtained by setting the edge's `minlen` attribute.

[7] Recall that the minimum rank occurs at the top of a drawing.

```
digraph asde91 {
    ranksep=.75; size = "7.5,7.5";

    {
        node [shape=plaintext, fontsize=16];
        /* the time-line graph */
        past -> 1978 -> 1980 -> 1982 -> 1983 -> 1985 -> 1986 ->
                1987 -> 1988 -> 1989 -> 1990 -> "future";
        /* ancestor programs */
        "Bourne sh"; "make"; "SCCS"; "yacc"; "cron"; "Reiser cpp";
        "Cshell"; "emacs"; "build"; "vi"; "<curses>"; "RCS"; "C*";
    }

    { rank = same;
        "Software IS"; "Configuration Mgt"; "Architecture & Libraries";
        "Process";
    };

    node [shape=box];
    { rank = same; "past"; "SCCS"; "make"; "Bourne sh"; "yacc"; "cron"; }
    { rank = same; 1978; "Reiser cpp"; "Cshell"; }
    { rank = same; 1980; "build"; "emacs"; "vi"; }
    { rank = same; 1982; "RCS"; "<curses>"; "IMX"; "SYNED"; }
    { rank = same; 1983; "ksh"; "IFS"; "TTU"; }
    { rank = same; 1985; "nmake"; "Peggy"; }
    { rank = same; 1986; "C*"; "ncpp"; "ksh-i"; "<curses-i>"; "PG2"; }
    { rank = same; 1987; "Ansi cpp"; "nmake 2.0"; "3D File System"; "fdelta";
        "DAG"; "CSAS";}
    { rank = same; 1988;  "CIA"; "SBCS"; "ksh-88"; "PEGASUS/PML"; "PAX";
        "backtalk"; }
    { rank = same; 1989;  "CIA++"; "APP"; "SHIP"; "DataShare"; "ryacc";
        "Mosaic"; }
    { rank = same; 1990;  "libft"; "CoShell"; "DIA"; "IFS-i"; "kyacc"; "sfio";
        "yeast"; "ML-X";  "DOT";  }
    { rank = same; "future"; "Adv. Software Technology"; }

    "PEGASUS/PML" -> "ML-X";
    "SCCS" -> "nmake";
    "SCCS" -> "3D File System";
    "SCCS" -> "RCS";
    "make" -> "nmake";
    "make" -> "build";
    .
    .
    .
}
```

Figure 11: Graph with constrained ranks

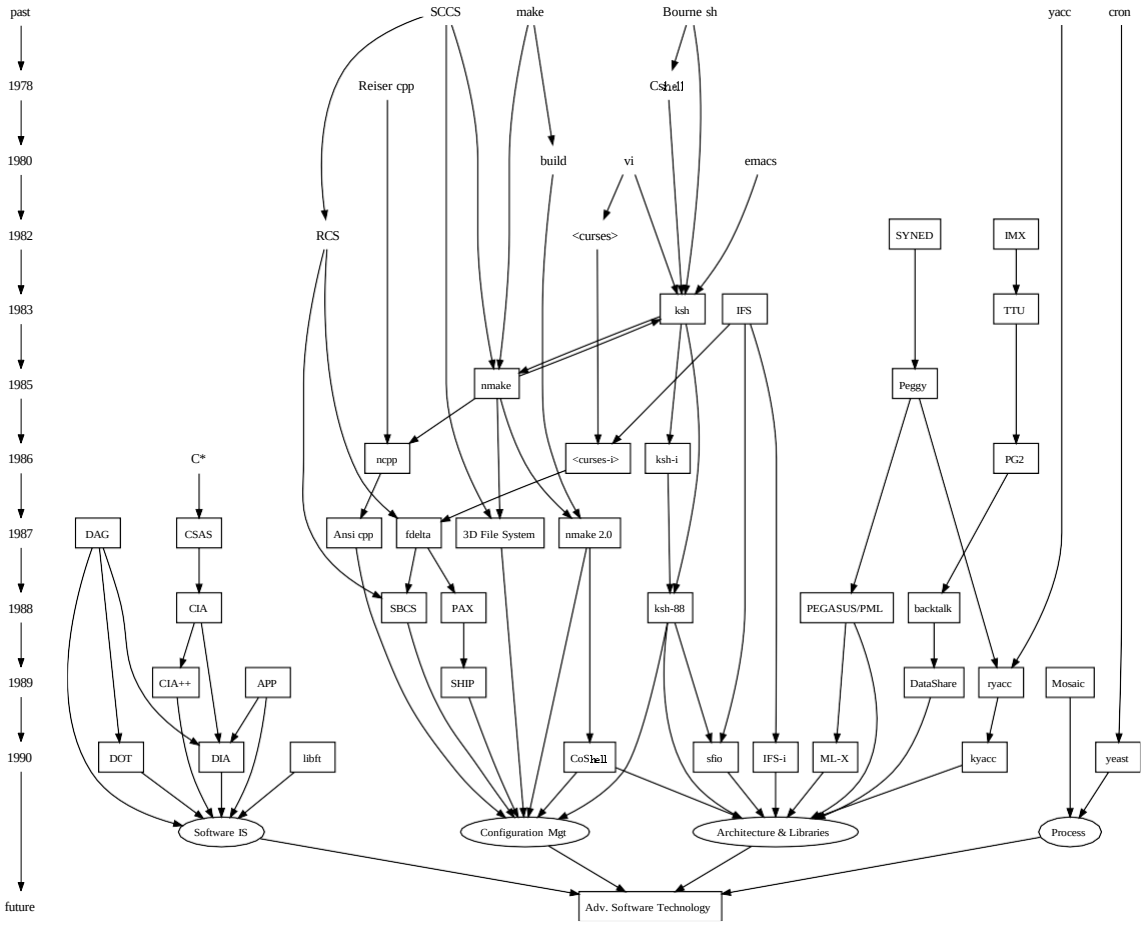


Figure 12: Drawing with constrained ranks

This defines the minimum difference between the ranks of the head and tail. For example, if `minlen=2`, there will always be at least one intervening rank between the head and tail. Note that this is not concerned with the geometric distance between the two nodes.

Fine-tuning should be approached cautiously. *dot* works best when it can makes 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 `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 *dot* with an interactive front-end that allows user-defined hints and constraints.

3 Advanced Features

3.1 Node Ports

A node port is a point where edges can attach to a node. (When an edge is not attached to a port, it is aimed at the node's center and the edge is clipped at the node's boundary.)

There are two types of ports. Ports based on the 8 compass points `"n"`, `"ne"`, `"e"`, `"se"`, `"s"`, `"sw"`, `"w"` or `"nw"` can be specified for any node. The end of the node will then be aimed at that position on the node. Thus, if `se` port is specified, the edge will connect to the node at its southeast "corner".

In addition, nodes with a `record` shape can use the record structure to define ports, while HTML-like labels with tables can make any cell a port using the `PORT` attribute of a `<TD>` element. If a record box or table cell defines a port name, an edge can use that port name to indicate that it should be aimed at the center of the box. (By default, the edge is clipped to the box's boundary.)

There are also two ways to specify ports. One way is to use an edge's `headport` and `tailport` attributes, e.g.

```
a -> b [tailport=se]
```

Alternatively, the portname can be used to modify the node name as part of the edge declaration using the syntax *node_name*`:`*port_name*. Thus, another way to handle the example given above would be

```
a -> b:se
```

Since a record box has its own corners, one can add a compass point port to record name port. Thus, the edge

```
a -> b:f0:se
```

will attach to the southeast corner of the box in record node `b` whose port name is `f0`.

Figure 13 illustrates the declaration and use of port names in record nodes, with the resulting drawing shown in Figure 14.

Figures 15 and 16 give another example of the use of record nodes and ports. This repeats the example of Figures 7 and 8 but now using ports as connectors for edges. 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 13. Otherwise the default node size (.75 by .5) is assumed, as in Figure 16. The example of Figures 17 and 18 uses left-to-right drawing in a layout of a hash table.

3.2 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 `cluster`. (If the top-level graph has `clusterrank=none`, this special processing is turned off). Labels, font characteristics and the `labelloc` attribute can be set as they would be for the top-level graph, though cluster labels appear above the graph by default. For clusters, the label is left-justified by default; if `labeljust="r"`, the label is right-justified. The `color` attribute specifies the color of the enclosing rectangle. In addition, clusters may have `style="filled"`, in which case the rectangle is filled with the color specified by `fillcolor` before the cluster is drawn. (If `fillcolor` is not specified, the cluster's `color` attribute is used.)

Clusters are drawn by a recursive technique that computes a rank assignment and internal ordering of nodes within clusters. Figure 19 through 21 are cluster layouts and the corresponding graph files.

```
 1:  digraph   g
 2:  {       node = record,height=.1];
 3:  [shape      = "<f0>  |<f1> G|<f2>  "];
 4:  node0[label  = "<f0>  |<f1> E|<f2>  "];
 5:  node1[label  = "<f0>  |<f1> B|<f2>  "];
 6:  node2[label  = "<f0>  |<f1> F|<f2>  "];
 7:  node3[label  = "<f0>  |<f1> R|<f2>  "];
 8:  node4[label  = "<f0>  |<f1> H|<f2>  "];
 9:  node5[label  = "<f0>  |<f1> Y|<f2>  "];
10:  node6[label  = "<f0>  |<f1> A|<f2>  "];
11:  node7[label  = "<f0>  |<f1> C|<f2>  "];
12:  "node0":f2  -> "node4":f1;
13:  "node0":f0  -> "node1":f1;
14:  "node1":f0  -> "node2":f1;
15:  "node1":f2  -> "node3":f1;
16:  "node2":f2  -> "node8":f1;
17:  "node2":f0  -> "node7":f1;
18:  "node4":f2  -> "node6":f1;
19:  "node4":f0  -> "node5":f1;
20:  }
```

Figure 13: Binary search tree using records

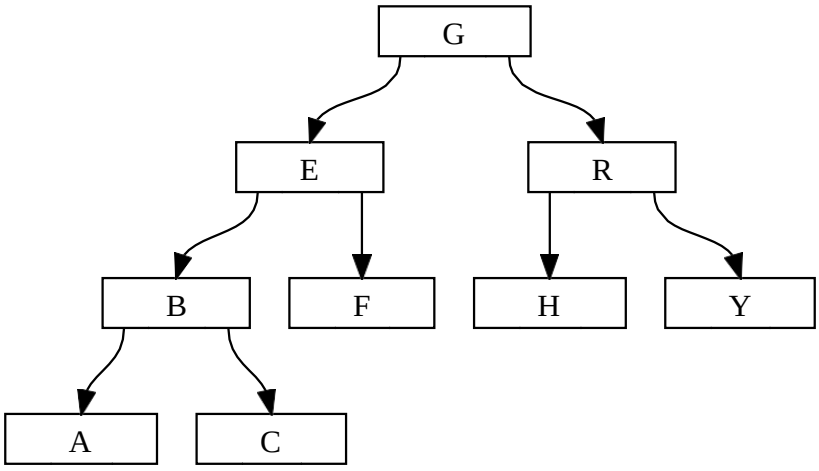


Figure 14: Drawing of binary search tree

```
1:  digraph structs
2:  { node
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 15: Records with nested fields (revisited)

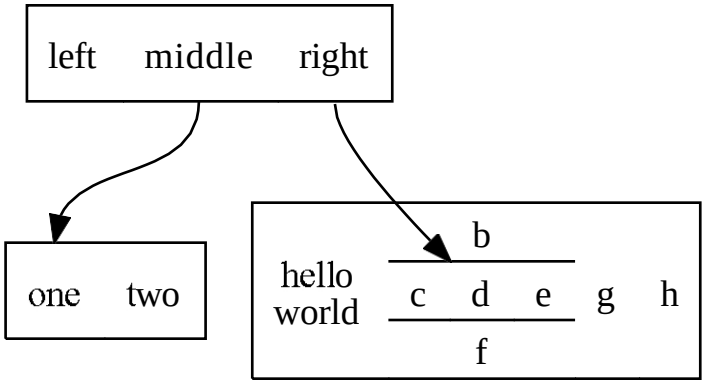


Figure 16: Drawing of records (revisited)

```
 1:  digraph G
 2:    { nodesep=.05;
 3:    rankdir=LR;
 4:    node [shape=record,width=.1,height=.1];
 5:
 6:    node0 [label = "<f0> |<f1> |<f2> |<f3> |<f4> |<f5> |<f6> | ",height=2.5];
 7:    node [width = 1.5];
 8:    node1 [label = "{<n> n14 | 719 |<p> }"];
 9:    node2 [label = "{<n> a1  | 805 |<p> }"];
10:    node3 [label = "{<n> i9  | 718 |<p> }"];
11:    node4 [label = "{<n> e5  | 989 |<p> }"];
12:    node5 [label = "{<n> t20 | 959 |<p> }"] ;
13:    node6 [label = "{<n> o15 | 794 |<p> }"] ;
14:    node7 [label = "{<n> s19 | 659 |<p> }"] ;
15:
16:    node0:f0 -> node1:n;
17:    node0:f1 -> node2:n;
18:    node0:f2 -> node3:n;
19:    node0:f5 -> node4:n;
20:    node0:f6 -> node5:n;
21:    node2:p -> node6:n;
22:    node4:p -> node7:n;
23:  }
```

Figure 17: Hash table graph file

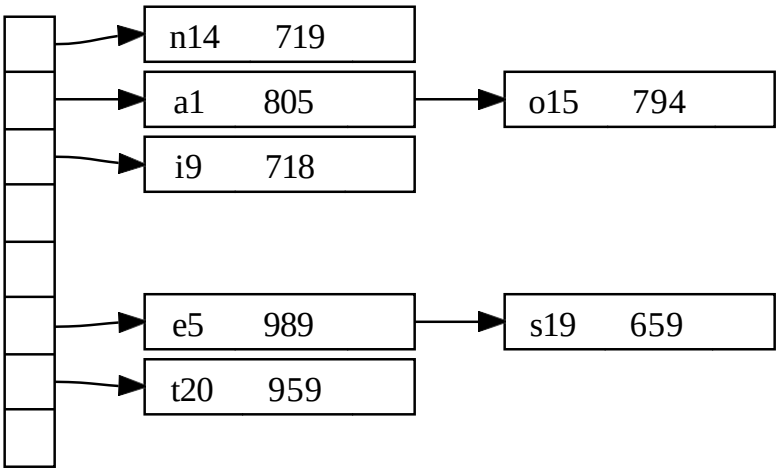


Figure 18: Drawing of hash table

```
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];
}
```

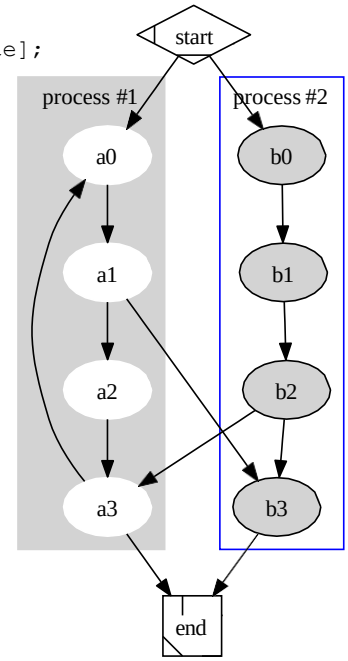


Figure 19: Process diagram with clusters

If the top-level graph has the `compound` attribute set to true, *dot* will allow edges connecting nodes and clusters. This is accomplished by an edge defining an `lhead` or `ltail` attribute. The value of these attributes must be the name of a cluster containing the head or tail node, respectively. In this case, the edge is clipped at the cluster boundary. All other edge attributes, such as `arrowhead` or `dir`, are applied to the truncated edge. For example, Figure 22 shows a graph using the `compound` attribute and the resulting diagram.

3.3 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, have a common endpoint and have length greater than 1. A beneficial side-effect in fixed-sized layouts is that removal of these edges often permits larger, more readable labels. While concentrators in *dot* 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, reading a graph from `stdin`, and writing the graph on `stdout` in the *DOT* format with layout attributes appended. *dot* supports a variety of command-line options:

`-T`*format* sets the format of the output. Allowed values for *format* are:

`bmp` Windoes bitMap format.

`canon` Prettyprint input; no layout is done.

`dot` Attributed *DOT*. Prints input with layout information attached as attributes, cf. Appendix F.

`fig` FIG output.

`gd` GD format. This is the internal format used by the GD Graphics Library. An alternate format is `gd2`.

`gif` GIF output.

`imap` Produces map files for server-side image maps. This can be combined with a graphical form of the output, e.g., using `-Tgif` or `-Tjpg`, in web pages to attach links to nodes and edges.

```
1:digraph G {
2:  size="8,6"; ratio=fill; node[fontsize=24];
3:
4:  ciafan->computefan; fan->increment; computefan->fan; stringdup->fatal;
5:  main->exit; main->interp_err; main->ciafan; main->fatal; main->malloc;
6:  main->strcpy; main->getopt; main->init_index; main->strlen; fan->fatal;
7:  fan->ref; fan->interp_err; ciafan->def; fan->free; computefan->stdprintf;
8:  computefan->get_sym_fields; fan->exit; fan->malloc; increment->strcmp;
9:  computefan->malloc; fan->stdsprintf; fan->strlen; computefan->strcmp;
10:  computefan->realloc; computefan->strlen; debug->sfprintf; debug->strcat;
11:  stringdup->malloc; fatal->sfprintf; stringdup->strcpy; stringdup->strlen;
12:  fatal->exit;
13:
14:  subgraph "cluster_error.h" { label="error.h"; interp_err; }
15:
16:  subgraph "cluster_sfio.h" { label="sfio.h"; sfprintf; }
17:
18:  subgraph "cluster_ciafan.c" { label="ciafan.c"; ciafan; computefan;
19:      increment; }
20:
21:  subgraph "cluster_util.c" { label="util.c"; stringdup; fatal; debug; }
22:
23:  subgraph "cluster_query.h" { label="query.h"; ref; def; }
24:
25:  subgraph "cluster_field.h" { get_sym_fields; }
26:
27:  subgraph "cluster_stdio.h" { label="stdio.h"; stdprintf; stdsprintf; }
28:
29:  subgraph "cluster_<libc.a>" { getopt; }
30:
31:  subgraph "cluster_stdlib.h" { label="stdlib.h"; exit; malloc; free; realloc;  }
32:
33:  subgraph "cluster_main.c" { main; }
34:
35:  subgraph "cluster_index.h" { init_index; }
36:
37:  subgraph "cluster_string.h" { label="string.h"; strcpy; strlen; strcmp; strcat; }
38:}
```

Figure 20: Call graph file

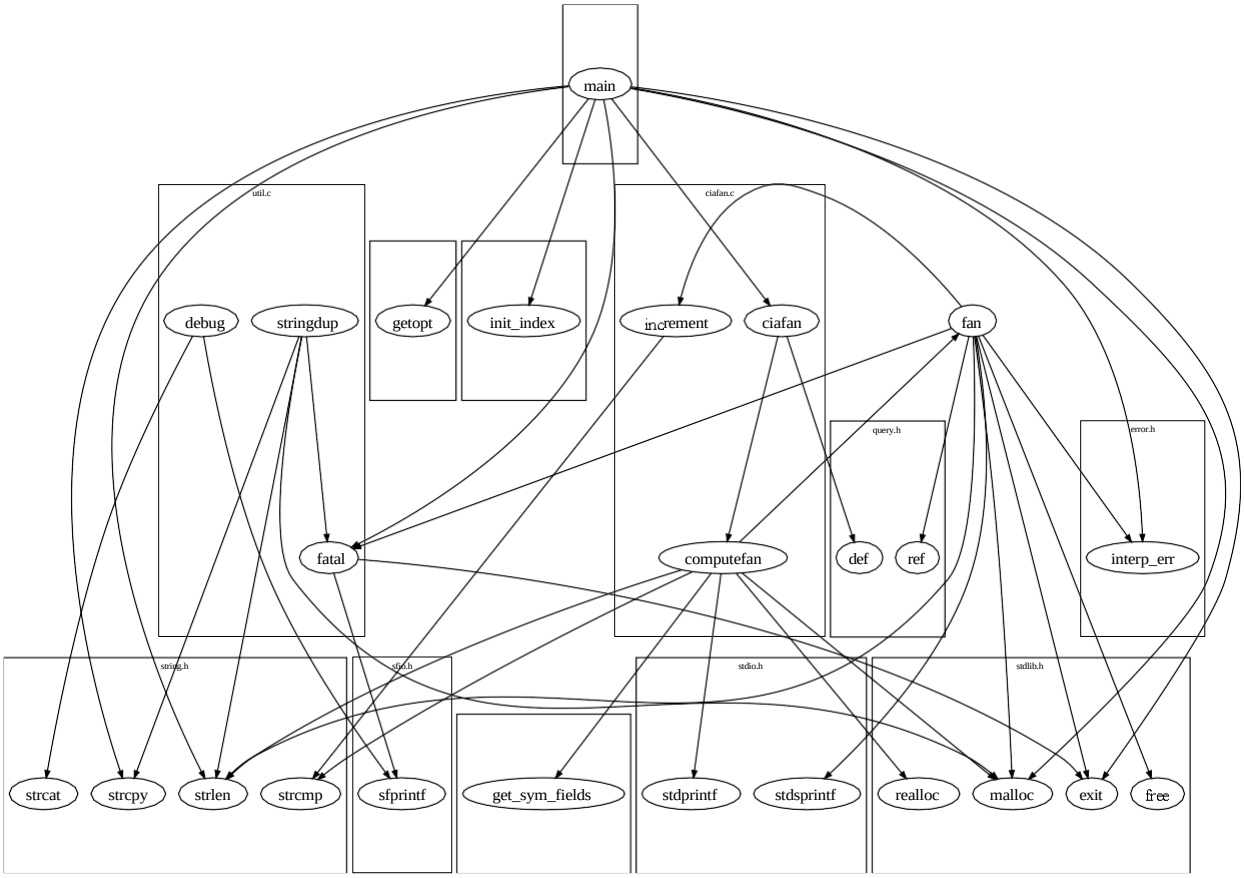


Figure 21: Call graph with labeled clusters

```
digraph G {
    compound=true;
    subgraph cluster0 {
      a -> b;
      a -> c;
      b -> d;
      c -> d;
    }
    subgraph cluster1 {
      e -> g;
      e -> f;
    }
    b -> f [lhead=cluster1];
    d -> e;
    c -> g [ltail=cluster0,
              lhead=cluster1];
    c -> e [ltail=cluster0];
    d -> h;
}
```

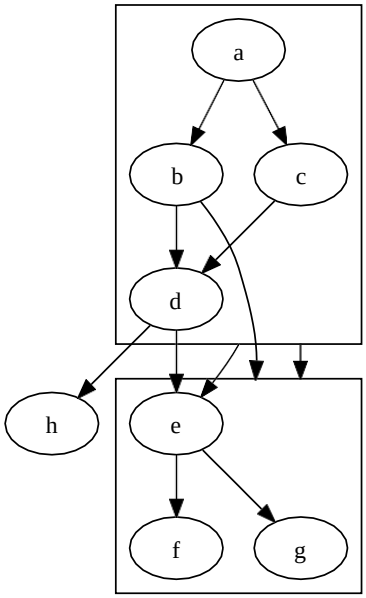


Figure 22: Graph with edges on clusters

`cmapx` Produces HTML map files for client-side image maps.

`pdf` Adobe PDF via the Cairo library. We have seen problems when embedding into other documents. Instead, use -Tps2 as described below.

`plain` Simple, line-based ASCII format. Appendix E describes this output. An alternate format is `plain-ext`, which provides port names on the head and tail nodes of edges.

`png` PNG (Portable Network Graphics) output.

`ps` PostScript (EPSF) output.

`ps2` PostScript (EPSF) output with PDF annotations. This output should be distilled into PDF, such as for pdflatex, before being included in a document. (Use ps2pdf; epstopdf doesn't handle `%%BoundingBox: (atend)`.)

`svg` SVG output. The alternate form `svgz` produces compressed SVG.

`vrml` VRML output.

`wbmp` Wireless BitMap (WBMP) format.

`-G`*name*=*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. The analogous flags `-N` or `-E` set default node or edge attributes. Note that file contents override command line arguments.

`-l`*libfile* specifies a device-dependent graphics library file. Multiple libraries may be given. These names are passed to the code generator at the beginning of output.

`-o`*outfile* writes output into file *outfile*.

`-v` requests verbose output. In processing large layouts, the verbose messages may give some estimate of *dot*'s progress.

`-V` prints the version number and exits.

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.

Nodes, edges and graphs may have a `URL` attribute. In certain output formats (`ps2`, `imap`, `cmapx`, or `svg`), this information is integrated in the output so that

nodes, edges and clusters become active links when displayed with the appropriate tools. Typically, URLs attached to top-level graphs serve as base URLs, supporting relative URLs on components. When the output format is `imap`, or `cmapx`, a similar processing takes place with the `headURL` and `tailURL` attributes.

For certain formats (`ps`, `fig` or `svg`), `comment` attributes can be used to embed human-readable notations in the output.

6 Conclusions

dot produces pleasing hierarchical drawings and can be applied in many settings.

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 Acknowledgments

We thank Phong Vo for his advice about graph drawing algorithms and programming. The graph library uses Phong's splay tree dictionary library. Also, the users of *dag*, the predecessor of *dot*, gave us many good suggestions. Guy Jacobson and Randy Hackbarth reviewed earlier drafts of this manual, and Emden contributed substantially to the current revision. 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 and other tools to bring *Graphviz* to the web.

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/365023133133011124