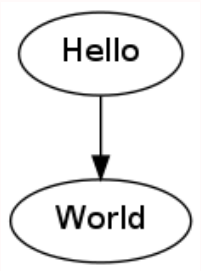
# Building Directed Graphs with GraphViz and Dot

Brian A. Malloy



# 1. Resource

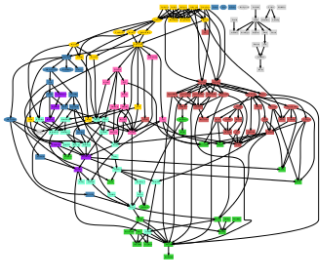
Drawing graphs with *dot*

Emden R. Gansner and Eleftherios Koutsofios and Stephen North

November 2, 2010

**Abstract**

*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.



# 2. Overview

- dot reads a text file and draws a digraph in GIF, PNG, SVG, PDF, ...
- E.g., assume graph.gv is a dot text file:

```
dot -Tpng graph.gv -o graph.png
```

will produce a png file of graph.gv




# 3. dot Layout Algorithm

1. Break cycles by reversing certain cyclic edges (dot graphs are acyclic.
2. Assign nodes to discrete ranks or levels; levels determine Y coordinates
3. Order nodes within levels to avoid crossings
4. Set X coordinates to keep edges short







# 4. dot Format

- dot accepts input in dot language.
- The language describes 3 kinds of objects: graphs, nodes, and edges.
- The outermost/main graph can be directed or undirected
- Undirected graphs are formatted with neato
- Subgraphs, within the main graph, define a set of nodes and edges.




## 4.1. Nodes and Edges

- A node is created when its name first appears in the file.
- An edge is created when nodes are joined with -> operator.
- An example spec is listed in Figure 1, where line 2 creates an edge from *main* to *parse* and from *parse* to *execute*.
- And produces the graph in Figure 2




```
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: Spec for basic digraph.



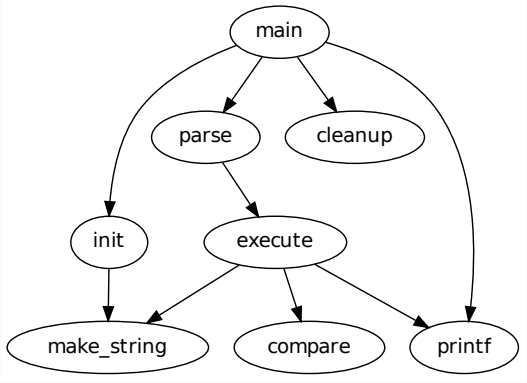



Figure 2: Resulting png for basic digraph drawn from spec in Figure 1.

# 5. dot Drawing Attributes

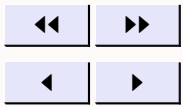
- You can adjust the representation or placement of nodes and edges with attributes.
- Attributes are (name, value) pairs
- Most attributes are in Appendices A, B, and C or the graphviz web site.
- Attributes are set off in square brackets
- An edge can be straightened by increasing it's *weight*; default is 1.
- Edges can be dotted, colored, or labeled




# 6. Node Shapes

- Default: *shape=ellipse*, *width=.75*, *height=.5* and labeled by the name of the node.
- Others: *box*, *circle*, *record*, or *plaintext*.
- A list of common shapes is in Appendix H.
- *plaintext* draws a node without an outline
- The *point* shape reduces node to display minimal content.
- A node's actual size is the greater of: (requested size, area needed for its label) unless *fixedsize=true* ⇒ requested size
- Two kinds of shapes: Polygon-based and Record-based



# 7. Polygon-based Node Shapes

- All shapes, except *record* and *Mrecord* are polygonal
- Modeled by number of sides (except ellipse & circle) and a few other properties
- Some properties are graph: e.g. *regular=true*
- Figures 3 and 4 illustrate a Polygon based spec with some specific attributes, and the resulting dot graph.




```
digraph G {
  size ="4,4";
  main [shape=box]; // this is a comment
  main -> parse [weight=8];
  parse -> execute;
  main -> init [style=dotted];
  main -> cleanup;
  execute -> { make_string; printf}
  init -> make_string;
  edge [color=red]; // so is this
  main->printf [style=bold,label="100 times"];
  make_string [label="make a\nstring"];
  node [shape=box,style=filled,color=".7 .3 1.0"];
  execute -> compare;
}
```

Figure 3: Spec for polygonal digraph.

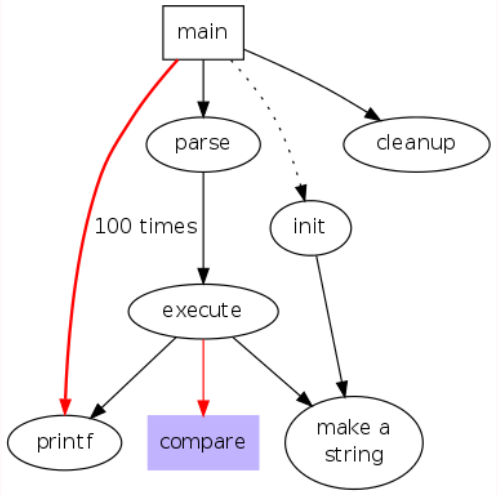



Figure 4: Resulting png for Polygonal digraph.



# 8. Record-based Node Shapes

- Two types *record* and *Mrecord*; identical except *Mrecord* corners are rounded
- Nodes represent recursive lists of fields that are drawn as alternating horizontal and vertical rows of boxes.
- Recursive structure is determined by node's label:

| | |
|---|---|
| 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
- The second string is used as a label for the field; it may contain the same escape sequences as multi-line labels
- See Figures 5 and 6.




```
1 digraph structs {
2   node [shape=record];
3   struct1 [shape=Mrecord,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 5: Spec for digraph with record nodes. For readable e.g., see code/records.gv

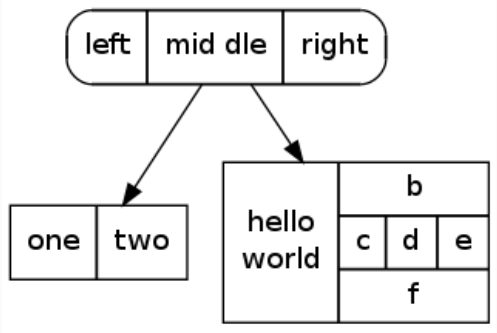



Figure 6: Resulting digraph for record spec drawn using spec in Figure 5.

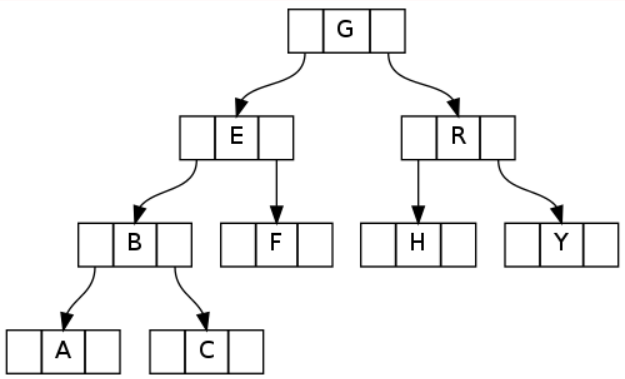



Figure 7: Digraph of a Binary Search Tree.