Appendix H

Graphviz

Graphviz is a simple, but powerful, package for drawing graphs (both undirected and directed). It was originally developed at AT&T Labs to process diagrams described by the DOT language (`https://graphviz.org/`). The Python version of Graphviz (`https://graphviz.readthedocs.io/en/stable/`) is somewhat different in that it offers an API for creating nodes, edges, and clusters (subgraphs), along with many attributes and layout managers for tweaking the look of a graph. The API can generate DOT files, but all of my examples use the API to create and display graphs directly.

An important issue concerning the distinction between Graphviz and its Python version is that much of the online help about the package uses examples written in the DOT language, which isn't of immediate help when looking for a particular method in the Python API.

The great strength of Graphviz is that the difficult task of laying out the nodes and edges of a graph is left to the software. However, this is also sometimes a weakness, since it's not easy to wrest layout control away from Graphviz; if you want to position lots of graph elements in specific places then Graphviz may not be the right tool for the job.

H.1 A Simple Undirected Graph

The code in Listing H.1 (simpGraph.py) generates an undirected graph using Graphviz's default 'dot' layout engine – the nodes are drawn in a top-to-bottom order based on the ordering of the edges in the code.

```
import graphviz
```

```
TEMP_FNM = 'temp_graph'

g = graphviz.Graph(format='png')  # use Graph or Digraph
# g.attr(rankdir='LR')     # to lay out left-to-right
g.attr(label='A Simple Graph')

g.edge('1', '2');  g.edge('2', '3'); g.edge('3', '1')
g.edge('1', '4');  g.edge('4', '5'); g.edge('4', '6')
g.edge('4', '7');  g.edge('6', '8'); g.edge('8', '9')
g.edge('9', '10'); g.edge('9', '7'); g.edge('10', '7')
g.edge('6', '7')

g.render(filename=TEMP_FNM, view=True)
```

Listing H.1. Generate a Top-to-Bottom Undirected Graph

This program displays the image shown in Fig. H.1, which is also saved to `temp_graph.png`.

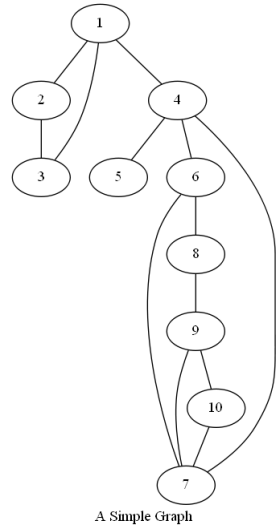

Figure H.1. A Top-to-Bottom Undirected Graph

Note that Graphviz has inferred the graph's nodes from the strings used in the edge() calls.

The `cleanup=True` argument of render() removes any unnecessary files, which in this case will be a DOT text file called `temp_graph`. It contains:

```
graph {
        label="A Simple Graph"
        1 -- 2;  2 -- 3; 3 -- 1
```

```
        1 -- 4;  4 -- 5; 4 -- 6
        4 -- 7;  6 -- 8; 8 -- 9
        9 -- 10; 9 -- 7; 10 -- 7
        6 -- 7
}
```

Actually, I've added line separators (';') to the code to reduce its length.

It's sometimes useful to retain this file (i.e. by removing the `cleanup=True` argument from render()) since it can be employed in other Graphviz tools, such as the 'Make a Graph' website at `https://graphs.grevian.org/graph`.

One easy change to Listing H.1 is to set the direction of the layout to be left-to-right by uncommenting the line:

```
g.attr(rankdir='LR')
```

Rerunning the program produces Fig. H.2.

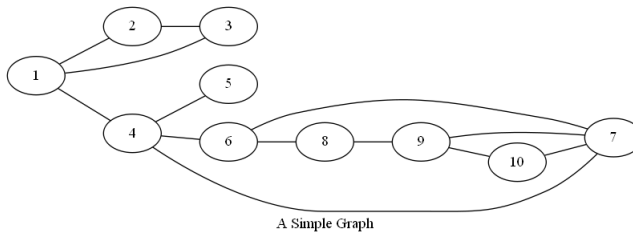

Figure H.2. The Left-to-Right Undirected Graph

'rankdir' is one of *many* graph attributes, which are listed in full at `https://graphviz.org/docs/graph/`.

H.1.1 The Directed Version. Converting the undirected graph to a directed version is another one-line change to Listing H.1 – replacing `graphviz.Graph()` by `graphvis.Digraph()`:

```
g = graphviz.Digraph(format='png')
```

Fig. H.3 shows both the top-down and the left-right versions of this directed graph.

H.2 Using Nodes

It's often best to explicitly define graph nodes by calling `node()` since this gives us access to a node's attributes such as its shape, color, and labeling. Listing H.2 (tennis.py) is an example.

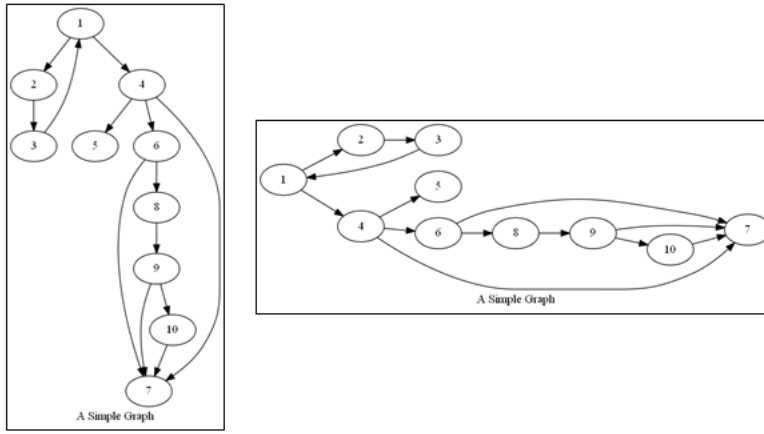

Figure H.3. Top-Bottom and Left-Right Directed Graphs

```
import graphviz as gv

TEMP_FNM = 'temp_graph'

t = gv.Digraph(format='png')
t.attr(label='Tennis Tournament')
t.attr(rankdir='LR')
t.attr('node', shape='box')

# player nodes
t.node('Player 1',shape='none'); t.node('Player 2',shape='none')
t.node('Player 3',shape='none'); t.node('Player 4',shape='none')
t.node('Player 5',shape='none'); t.node('Player 6',shape='none')
t.node('Player 7',shape='none'); t.node('Player 8',shape='none')

# game nodes
t.node('Game 1'); t.node('Game 2'); t.node('Game 3')
t.node('Game 4'); t.node('Game 5'); t.node('Game 6')
t.node('Game 7')

# link players and games
t.edge('Player 1','Game 1'); t.edge('Player 2','Game 1')
t.edge('Player 3','Game 2'); t.edge('Player 4','Game 2')
t.edge('Player 5','Game 3'); t.edge('Player 6','Game 3')
t.edge('Player 7','Game 4'); t.edge('Player 8','Game 4')
t.edge('Game 1', 'Game 5'); t.edge('Game 2', 'Game 5')
t.edge('Game 3', 'Game 6'); t.edge('Game 4', 'Game 6')
t.edge('Game 5', 'Game 7'); t.edge('Game 6', 'Game 7')
```

```
t.render(filename=TEMP_FNM, view=True, cleanup=True)
```

Listing H.2. Drawing a Tennis Tournament

This program displays the details of the players and games in a tennis tournament in Fig. H.4. Note that the list of players is slightly out of order.

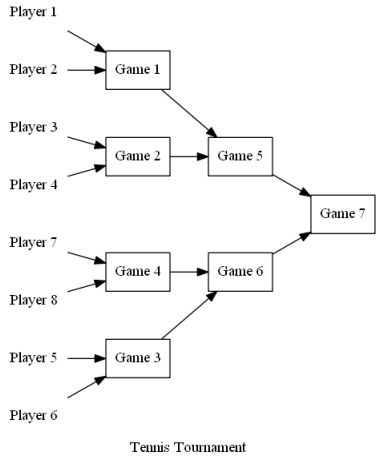

Figure H.4. A Slightly Misordered Tennis Tournament

Node attributes can be set globally by calling the attr() function with a 'node' argument. In `tennis.py`, all the nodes are set to be box-shaped with:

```
t.attr('node', shape='box')
```

A complete list of node attributes can be found at `https://graphviz.org/docs/nodes/`.

The player strings on the left of Fig. H.4 are implemented as nodes with no shape:

```
t.node('Player 1', shape='none')
```

Fixing the ordering of the labels so they are in increasing order going down the page illustrates the trickiness of overriding Graphviz's layout engines.

The most commonly used technique is to link nodes with *invisible* edges to encourage the engine to position the nodes in a particular order. Unfortunately, this isn't sufficient in this case since we're utilizing a left-to-right ordering for the graph, but the players should be positioned top-down. This distinction can be made by placing the invisible edges in a subgraph with their own rank and direction. The new lines in tennis2.py are:

```
with t.subgraph() as ps:
  ps.attr(rankdir='TB', rank='same')
  ps.edge('Player 1', 'Player 2', style='invis')
  ps.edge('Player 2', 'Player 3', style='invis')
  ps.edge('Player 3', 'Player 4', style='invis')
  ps.edge('Player 5', 'Player 6', style='invis')
  ps.edge('Player 6', 'Player 7', style='invis')
  ps.edge('Player 7', 'Player 8', style='invis')
```

The ps subgraph is inside the tournament's graph (t), and the edges all have the same rank in top-to-bottom (rankdir='TB') order. Running tennis2.py produces Fig. H.5.

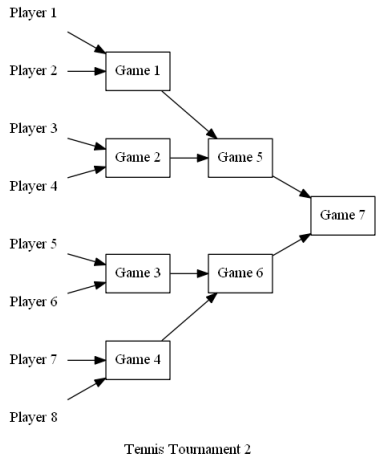

Figure H.5. An Ordered Tennis Tournament

H.2.1 A Probability Tree. The probability tree in Fig. H.6 shows a few other aspects of node rendering.

probs.py generates a top-down directed graph with circular nodes, but the ones at level 2 are colored blue. Less obviously, several nodes use the same labels ('W', 'R', and 'G') which require their node names (which must be unique) to be different from their labels. This is done using code like:

```
tree.node(pNode, label=col, shape='circle', color='blue')
```

pNode is a unique string generated using a counter at run time, while col is one of the three letters.

The probabilities beneath each leaf node in Fig. H.6 are drawn as no-shape nodes connected to the leaves above by invisible edges. This is implemented in probs.py using:

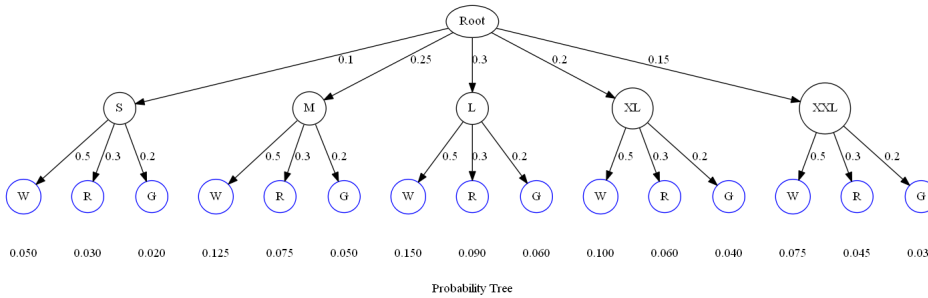

Figure H.6. A Probability Tree

```
# total prob in 'none' shape linked by invisible edge to pNode
tree.node(tNode, label=f"{(sprob*cprob):.3f}",
                    labelloc='t', shape='none')
tree.edge(pNode, tNode, len='0.5', weight='3', style='invis')
                  # shorter and more vertical
```

The invisible edges are shortened using the `len` attribute, and their weight is increased with `weight`. The weight makes it more likely that the probability text will be directly below the letter nodes rather than be spaced out evenly across the row. A complete list of edge attributes can be found at `https://graphviz.org/docs/edges/`.

H.3 More Layouts

Graphviz has a growing list of layout engines (see `https://graphviz.org/docs/layouts/`). `dot` is the default one, best suited to hierarchical drawing such as the trees in the previous section. More general graphs may benefit from being laid out with `neato`, `fdp`, or `sfdp`. Graphs arranged in a circle are best positioned with `circo`.

engines.py uses the same edge data as simpGraph.py, but lets the user decide on the layout engine for drawing it. Fig. H.7 shows the images produced when the user enters `dot`, `neato`, `fdp`, `sfdp`, or `circo`.

The layout managers can be adjusted using various graph attributes (see `https://graphviz.org/docs/graph/`). engines.py uses:

```
g.attr(rankdir='LR')      # for dot
g.attr(overlap='false')   # avoids node overlapping
g.attr(splines='true')    # helps edges avoid crossing nodes
g.attr(oneblock='true')   # for circo
```

The `splines` attribute lets the layout manager employ splines to curve edges around nodes; this avoids overlapping but can slow down rendering when the

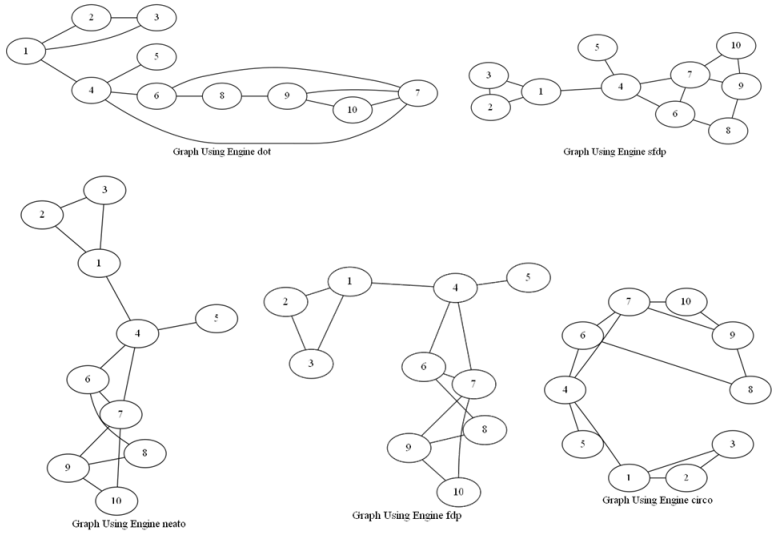

Figure H.7. The Same Graph; Five Different Layouts.
Top row: `dot`, `sfdp`; Bottom row: `neato`, `fdp`, `circo`

graph is large. `oneblock` tells `circo` that all the nodes are to be grouped into a single circle.

H.4 More Attributes

Listing H.3 (arcs.py) draws the directed graph in Fig. H.8, which illustrates a few more node and edge attributes.

```
import graphviz

def ns(name, val):
  return '<<I>'+name+'</I><SUB>'+str(val)+'</SUB>>'

TEMP_FNM = 'temp_graph'
ALPHA_BLUE = '#0096FF80'  # 50% alpha

dg = graphviz.Digraph(format='png', engine='neato',
      node_attr={'shape':'point'},
      edge_attr={'fontsize':'12',
                 'arrowhead':'open',
                 # 'color':'cadetblue3'
                 'color':ALPHA_BLUE
                })
dg.attr(label='Directed Graph')
```

```
for v in ['v1','v2','v3','v4','v5','v6','v7']:
  dg.node(v, '', xlabel=v)

dg.edge('v1','v2',label=ns('e',1))
dg.edge('v2','v3',label=ns('e',2))
dg.edge('v3','v4',label=ns('e',3))
dg.edge('v4','v6',label=ns('e',4))
dg.edge('v6','v5',label=ns('e',6))
dg.edge('v6','v7',label=ns('e',7))
dg.edge('v6','v1',label=ns('e',8))
dg.edge('v2','v7',label=ns('e',9))
dg.edge('v3','v5',label=ns('e',5))

dg.render(filename=TEMP_FNM, view=True, cleanup=True)
```

Listing H.3. Generate a Fancy Directed Graph

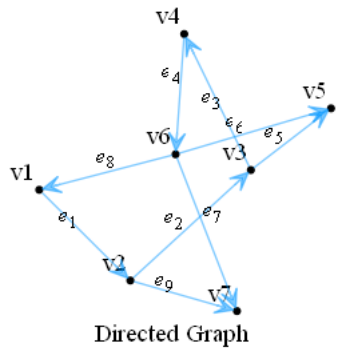

Figure H.8. Directed Graph Using Attributes

The node and edge attributes are set with the `node_attr` and `edge_attr` parameters of Digraph(), which allow multiple attributes to be set at once via a key:value map; this requires less code than calling `dg.attr('node', ...)` and `dg.attr('edge', ...)` multiple times. There's also a `graph_attr` parameter which isn't used in this example.

The API structure for Digraph() (and Graph()) are described in the Graphviz Python documentation at `https://graphviz.readthedocs.io/en/stable/api.html`. However, the meaning of the attributes, such as the possible node shapes, is explained in the Graphviz package documentation (e.g. at `https://graphviz.org/doc/info/shapes.html`).

The nodes are drawn by arcs.py as small black points, so their labels must be placed along side with the `xlabel` attribute (`https://graphviz.org/docs/nodes/`).

The edge labels use a smaller font size and subscripts. The latter is best archived by writing the label using a snippet of HTML, which is done by ns() in arcs.py. In fact, the HTML is more accurately 'HTML-like' since a lot of Graphviz-specific elements have been added; the details can be found at `https://graphviz.org/doc/info/shapes.html#html`. Another major use of this feature is to subdivide box node shapes with horizontal and vertical lines to organize record or tabular data (see the 'More Shapes' section below for an example).

The edge colors are fixed in two ways – using a preexisting color (`cadetblue3`) which is commented out, or with a hexadecimal consisting of values for the RGB and alpha components. Color names are listed at `https://graphviz.org/doc/info/colors.html`, while various color schemes are detailed at `https://graphviz.org/docs/attr-types/color/`. My `ALPHA_BLUE` sets the alpha part of the edges to 0x80, which is roughly 50% transparent, making it a little easier to read the node labels.

H.5 Clusters: An Automaton

A cluster is a subgraph with a visual appearance (by default, a rectangle), and support for many drawing attributes. A cluster is created in the same way as a subgraph (i.e. with subgraph()) but must have a name that begins with the string 'cluster'.

Listing H.4 (automaton.py) draws the automaton in Fig. H.9 which includes a cluster holding the 's1' and 's2' nodes.

```
import graphviz

TEMP_FNM = 'temp_graph'

dg = graphviz.Digraph(format='png')
dg.attr(rankdir='LR')
dg.attr(label='An Automaton')
dg.attr('edge', fontcolor='blue')

# start arrow with no node
dg.node('', shape='none')
dg.edge('', 's0', label='Start')

# nodes
dg.node('s0', shape='doublecircle')
dg.node('s3', shape='doublecircle')

with dg.subgraph(name='cluster1') as c1:
  c1.attr(label='Internal', style='rounded,dashed')
  c1.node('s1')   # nodes inside a cluster
  c1.node('s2')
```

```
# transitions
dg.edge('s0','s0',label='0'); dg.edge('s0','s1',label='1')
dg.edge('s1','s2',label='1'); dg.edge('s2','s3',label='0')
dg.edge('s3','s3',label='0, 1')
dg.edge('s2','s2',label='1'); dg.edge('s1','s3',label='0')

dg.render(filename=TEMP_FNM, view=True, cleanup=True)
```

Listing H.4. Drawing an Automaton

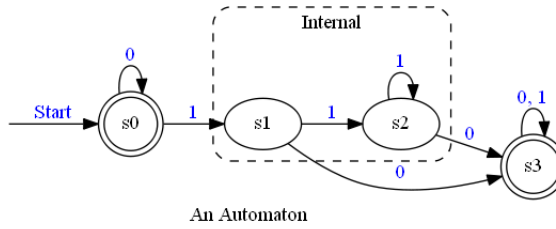

Figure H.9. An Automaton with a Cluster

The cluster code only utilizes the `label` and `style` attributes, but many other elements can be adjusted, as listed at `https://graphviz.org/docs/clusters/`.

Note that although the 's1' and 's2' nodes are defined inside the cluster subgraph `c1`, the edge code only employs their string names.

H.5.1 More Clusters: a Statechart.

states.py draws the nested statechart shown in Fig. H.10.

The outer cluster `c` is created with:

```
with dg.subgraph(name='cluster_op') as c:
  c.attr(label='Operational', style='rounded')
    : # create nodes, an inner cluster, and edges
```

The inner cluster `p` is created with:

```
  with c.subgraph(name='cluster_pul') as p:
    p.attr(label='Pulsing', style='rounded')
      :  # define three nodes and edges
```

Note that `p` is defined as a subgraph of the outer cluster `c`.

The only completely new feature here are the two dashed arrows that link the inner cluster to the 'Standby' node; they are implemented using:

```
  # edges to/from cluster p
  c.edge('standby', 'pulsingStart',
```

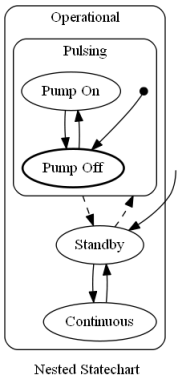

Figure H.10. A Nested Statechart

```
                lhead='cluster_pul', style='dashed')
  c.edge('pOff', 'standby',
                ltail='cluster_pul', style='dashed')
```

The calls to `edge()` link the arrows to the inner cluster using its string name 'cluster_pul' not the cluster variable `p`.

H.6 More Shapes

Listing H.5 (`someShapes.py`) draws a variety of node shapes in Fig. H.11.

```
import graphviz

TEMP_FNM = 'temp_graph'

dg = graphviz.Digraph(format='png')
dg.attr(rankdir='LR')   # LR or TB
dg.attr(label='Some Shapes')

dg.node('n1', '{record|{a|b|<f0>c}}', shape='record')
   # uses graph's rankdir to interpret '|'

dg.node('n2', label=(
  '''<<table border="0" cellborder="1" cellspacing="0">
      <tr>
        <td colspan="3">table</td>
      </tr>
      <tr>
        <td port="f0">d</td>
        <td port="f1">e</td>
        <td port="f2">f</td>
      </tr>
```

```
    </table>
  >''' ), shape='plaintext')

dg.edge('n1:f0', 'n2:f2')

# pentagon
dg.node('P', '5-sided', shape='polygon', sides='5')
# hexagon
dg.node('H', '6-sided', shape='polygon', sides='6')
# distorted hexagon
dg.node('E','Distorted',shape='polygon',sides='6',distortion='0.7')
# skewed hexagon
dg.node('F', 'Skewed', shape='polygon', sides='6', skew='0.7')
# rotated hexagon
dg.node('G', 'Rot 30',shape='polygon',sides='6',orientation='30')
dg.edges(['PH', 'HE', 'EF', 'FG'])
   # e.g. PH means link nodes P --> H

dg.render(filename=TEMP_FNM, view=True, cleanup=True)
```

Listing H.5. Generating Different Node Shapes

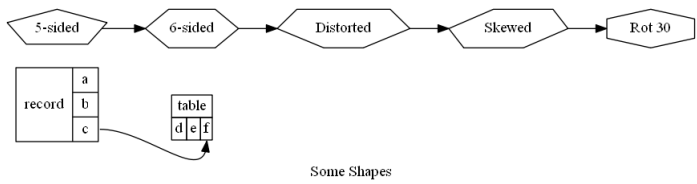

Figure H.11. Different Node Shapes

The bottom row of Fig. H.11 is generated first by the code, using a node record and HTML. (More complex examples of these two approaches can be found at `https://graphviz.readthedocs.io/en/stable/examples.html`). Note that the edge between the nodes employs `f` fields in the labels.

Although the record coding is considerably shorter, the meaning of its '|' separator is tied to the layout direction (in this case left-to-right, `LR`). If the layout is changed to top-to-bottom (`TB`), the HTML node isn't affected, but the record's organization changes as shown in Fig. H.12.

The top row in Fig. H.11 utilizes a 'polygon' shape combined with various attributes. The first shape is 5-sided, but the others are hexagons. Details on the many shape types and polygon attributes are given at `https://graphviz.org/doc/info/shapes.html`.

H.6.1 Images: A Circuit Diagram. Probably the easiest way to utilize an arbitrary shape as a node is to save its outline as an image. `circuits.py` loads the three PNG images shown in Fig. H.13.

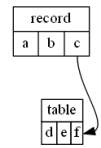

Figure H.12. A top-bottom Record and HTML Node

Figure H.13. AND, OR, and NOT Gate Images

The relevant snippets of code are:

```
# gates using images for AND, OR, and NOT
g.node('AND', image='andGate.png',imagescale='true',shape='none')
g.node('OR', image='orGate.png',imagescale='true',shape='none')
g.node('NOT\l', image='notGate.png',imagescale='true',shape='none')
```

The default shape (an ellipse) is switched off, and the loaded image is scaled to fit its space.

A drawback is that edges only extend to the default shape's boundary, so are unlikely to touch the image, as shown in Fig. H.14.

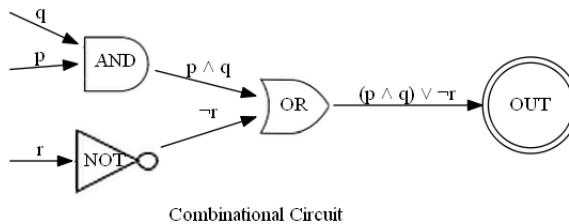

Figure H.14. A Combinatorial Circuit

Another occasional issue is positioning a node's label on top of an image. For instance, I had to left-justify the 'NOT' string using `\l`.