Using GraphViz on Clemson Palmetto Cluster

Clemson CCIT Visualization Lab

April 27, 2017

GraphViz is an open source tool which visualizes structural information such as diagrams of abstract graphs and networks. By generating a visual presentation with easily readable layout from simple text descriptions of a graph, this software has the potential to enhance researches and projects in many fields including computer networking, data base design, bioinformatics, and etc.. This document focuses on introducing the usage of GraphViz on Clemson Palmetto Cluster, as well as its basic functionalities. To access more details of GraphViz, we refer readers to its official online tutorials: `http://www.graphviz.org/Documentation.php`.

1 Installation

Before installing GraphViz on Palmetto, we first download its source code from `http://www.graphviz.org/Download_source.php`. A simple way to do that is copying the link of a specific release of source code, e.g. `http://www.graphviz.org/pub/graphviz/stable/SOURCES/graphviz-2.40.1.tar.gz`, and download it to a desired folder on Palmetto by executing the following commands:

```
1 $mkdir your_graphviz_folder
2 $cd your_graphviz_folder
3 $wget http://www.graphviz.org/pub/graphviz/stable/
    ↪ SOURCES/graphviz-2.40.1.tar.gz
4 $tar -zxvf graphviz-2.40.1.tar.gz
```

Second, we need to configure the installing prefix since the default path of the software is under the *usr* folder which we do not have write permission on Palmetto. This can be done by executing the following commands:

```
1 $cd graphviz-2.40.1
2 $./configure --prefix=your_intallation_path
```

Finally, GraphViz can be compiled and installed by simply executing

```
1 $make
2 $make install
```

Once it successfully installed, we can find corresponding executables, e.g. *dot*, in the *bin* directory under *your_installation_path*.

2 Running GraphViz

There are multiple ways to use GraphViz on Palmetto. One simple option is to construct a graph description file, e.g. demo.dot, using text editors, and then generate a corresponding image, e.g. demo.dot.png, by executing a filter command, e.g. *dot*, from terminal:

```
1 $dot -Tpng -O demo.dot
```

GraphViz provides a suite of filter commands to portray different types of graphs:

- dot: filter for drawing direct graphs
- neato: filter for drawing undirected graphs
- twopi: filter for radial layouts of graphs
- circo: filter for circular layout of graphs
- fdp: filter for drawing undirected graphs
- sfdp: filter for drawing large undirected graphs
- patchwork: filter for squarified tree maps
- osage: filter for array-based layouts

The $-T$ option specifies the format of an output image:

- Tdot: Dot format containing layout information
- Txdot: Dot format containing complete layout information
- Tps: PostScript
- Tpdf: PDF
- Tsvg or Tsvgz: Structured Vector Graphics
- Tfig: XFIG graphics
- Tpng: png bitmap graphics
- Tgif: gif bitmap graphics
- Tjpg or Tjpeg: jpeg bitmap graphics
- Tjson: xdot information encoded in JSON
- Timap: imagemap files for httpd servers for each node or edge that has a non-null href attribute
- Tcmapx: client-side imagemap for use in html and xhtml

Beside of running from command line, we can also use GraphViz through tools with graphical user interfaces, e.g. *gveditor*, *vimdot*, *dotty*, *lefty*, etc.. To use these tools, we need to connect to Palmetto with X11 graphical forwarding option, as described in Palmetto User's Guide (`https://www.palmetto.clemson.edu/palmetto/pages/userguide.html#graphical`). For the following of this document, we will use *gveditor* for demonstrations purpose.

3 Drawing Graph Using *dot*

This section demonstrates the process of drawing direct graphs using *dot*, which is one of the most widely adopted filters provided by GraphViz.

dot accepts input in a graph description language which refers to as DOT. Dot defines three primary objects: graph, nodes, and edges, which can be augmented with a suite of parameters specifying the corresponding graphic attributes in the actual visualization, e.g. color, shape, size, etc.. For detailed information fo DOT language, we refer readers to its Wikipedia page `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`, and an introduction supported by GraphViz team `http://www.graphviz.org/pdf/dotguide.pdf`.

```
digraph demo{
 clemson [label="Clemson University",
 style=filled, shape=ellipse, color=orange];

 vizlab [label="CCIT Visualization Lab",
 style=unfilled, shape=box,color=purple];
 clemson->vizlab [label="has"];
}
```

Figure 1: A basic example of DOT language defining a direct graph with two nodes and an edge, where their graphical attributions have been specified.

A basic example of DOT is shown in Figure 1, defining a direct graph contains two nodes, *clemson* and *vizlab*, where the former has been designed as a filled orange ellipse with label "Clemson University" while the latter is an unfilled purple box. An edge connecting them is also defined to indicate that Clemson University has CCIT Visualization Lab. The corresponding image of this DOT script is shown in Figure 2.

An advanced example is shown by Listing 1, where the abstract structure of Game of Thrones family tree, shown in Figure 3, was reconstructed, shown by Figure 4. In this example, the four families were grouped into clusters using the *subgraph* feature, e.g. line 3 in Listing 1, allowing us to do finer manipulations such as assigning different background colors and adding subtitles to individual

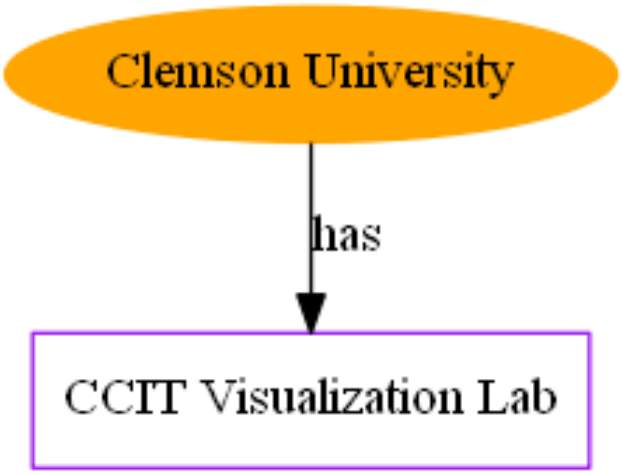

Figure 2: A graph generated by the example described in Figure 1

families, and making the graph clean and structured. For individual families, configurations of different types of nodes were specified, e.g. line 7, before defining actual nodes, e.g. line 8 to line 11. This example also demonstrates utilizations of additional attributes such as shape of a node(line 11), style of an edge(line 84), location, size, color and family of a font(line 94 to line 98), etc..

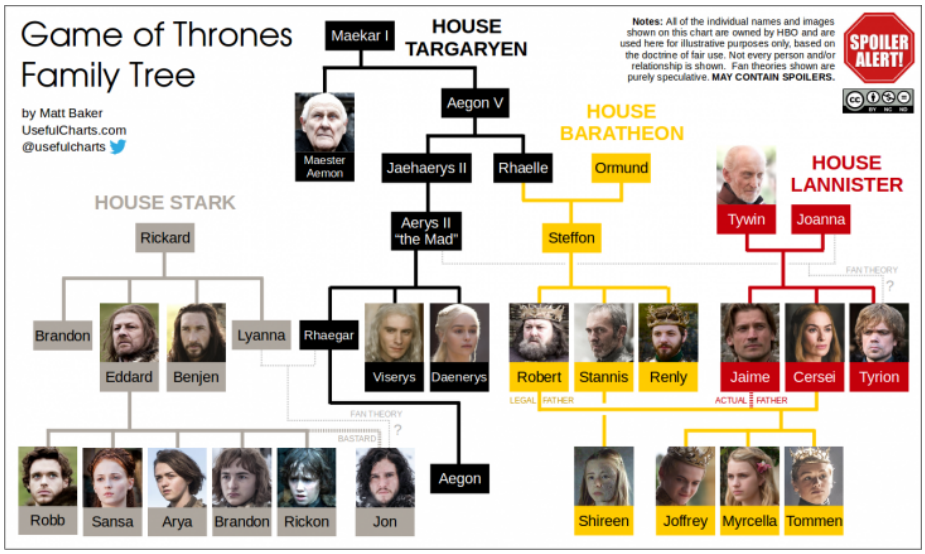

Figure 3: Game of Thrones family tree. *source*:http://www.chartgeek.com/game-of-thrones-family-tree/

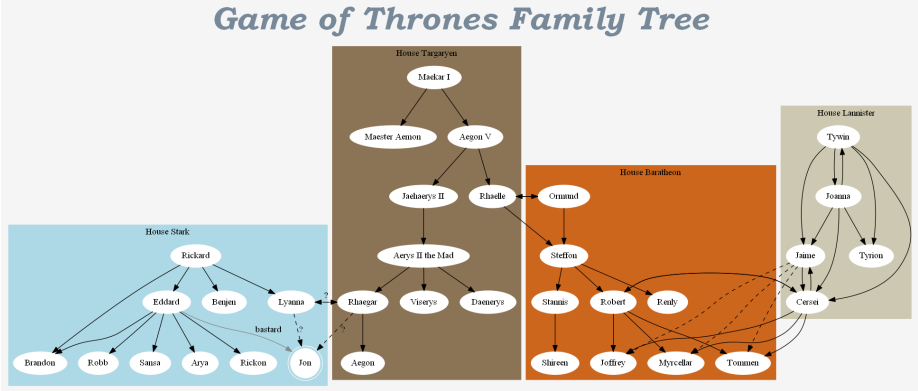

Figure 4: A graph represents Game of Thrones family tree portrayed by Figure 3.

Listing 1: DOT script defines Game of Thrones family tree described in Figure 3

```
digraph GOT{
graph[bgcolor=whitesmoke]
subgraph cluster_stark {
style=filled;
color=lightblue;
label="House Stark";
node [style=filled,color=white];
Rickard;
Brandon; Eddard; Benjen; Lyanna;
Robb; Sansa; Arya; Brandon; Rickon;
node [shape=doublecircle, style=filled, color=white];
Jon;
Rickard->Brandon;
Rickard->Eddard;
Rickard->Benjen;
Rickard->Lyanna;
Eddard->Robb;
Eddard->Sansa;
Eddard->Arya;
Eddard->Brandon;
Eddard->Rickon;
Eddard->Jon [label="bastard",color=azure4];
}
subgraph cluster_targaryen {
style=filled;
color=burlywood4;
label="House Targaryen";
node [style=filled,color=white];
Rhaelle; Rhaegar; Viserys; Daenerys; Aegon;
node [style=filled,color=white,label="Maekar I"];
    ↪ Maekar_I;
node [style=filled,color=white,label="Maester Aemon"];
    ↪  Maester_Aemon;
node [style=filled,color=white,label="Aegon V"];
    ↪ Aegon_V;
node [style=filled,color=white,label="Jaehaerys II"];
    ↪ Jaehaerys_II;
node [style=filled,color=white,label="Aerys II the Mad
    ↪ "]; Aerys_II;
Maekar_I->Maester_Aemon;
Maekar_I->Aegon_V;
Aegon_V->Jaehaerys_II;
Aegon_V->Rhaelle;
Jaehaerys_II->Aerys_II;
```

```
40 Aerys_II->Rhaegar;
41 Aerys_II->Viserys;
42 Aerys_II->Daenerys;
43 Rhaegar->Aegon;
44 }
45 subgraph cluster_baratheon{
46 style=filled;
47 color=chocolate3;
48 label="House Baratheon";
49 node [style=filled,color=white];
50 Ormund; Steffon; Robert; Stannis; Renly; Shireen;
      ↪ Joffrey; Myrcellar; Tommen;
51 Ormund->Steffon;
52 Rhaelle->Steffon;
53 Ormund->Rhaelle;
54 Rhaelle->Ormund;
55 Steffon->Robert;
56 Steffon->Stannis;
57 Steffon->Renly;
58 Stannis->Shireen;
59 Robert->Joffrey;
60 Robert->Myrcellar;
61 Robert->Tommen;
62 }
63 subgraph cluster_lannister{
64 style=filled;
65 color=cornsilk3;
66 label="House Lannister";
67 node [style=filled,color=white];
68 Tywin;Joanna;Jaime;Cersei; Tyrion;
69 Tywin->Joanna;
70 Joanna->Tywin;
71 Joanna->Jaime;
72 Joanna->Cersei;
73 Joanna->Tyrion;
74 Tywin->Jaime;
75 Tywin->Cersei;
76 Tywin->Tyrion;
77 Jaime->Cersei;
78 Cersei->Jaime;
79 Robert->Cersei;
80 Cersei->Robert;
81 Cersei->Joffrey;
82 Cersei->Myrcellar;
83 Cersei->Tommen;
84 Jaime->Joffrey [style=dashed];
```

```
85  Jaime->Myrcellar [style=dashed];
86  Jaime->Tommen [style=dashed];
87  }
88
89  Lyanna->Rhaegar [style=dashed, label="?"];
90  Rhaegar->Lyanna [style=dashed, label="?"];
91  Lyanna->Jon  [style=dashed, label="?"];
92  Rhaegar->Jon [style=dashed, label="?"];
93
94  labelloc="t";
95  fontsize=50;
96  fontcolor=lightslategrey;
97  fontname="Bookman Old Style Bold Italic" ;
98  label="Game of Thrones Family Tree"
99  }
```

4 Drawing Graph Using *lefty* and *dotty*

lefty Besides generating graphs from DOT scripts, GraphViz also implements a general purpose programmable editor for technical pictures, with its own interactive programming language, allowing users to specify a graph by constructing a *lefty* script and augmenting the display with user interactions, i.e. keyboard pressing, mouse clicking and dragging.

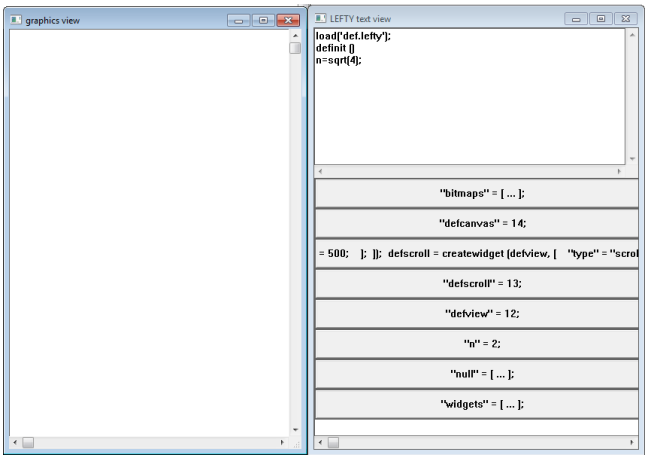

Figure 5: A screen-shot of *lefty* editor. The left is the graphics view showing the actual visualization of a graph while the right is the text view allowing user to manipulate the program used to generate the graph.

lefty implements two views to portray a graph: graphics view and text view. The former explicitly draws an image of the graph (the left in Figure 5), while

the latter shows the actual program specifying the graph (the right in Figure 5). Users can modify the picture by changing the program in the text view, or performing interactive manipulations in the graphics view if appropriate event functions have been created in the program. Possible event functions include *leftdown*, *leftmove*, *leftup*, *middledown*, *middlemove*, *middlepup*, *rightdown*, *rightmove*, *rightup*, *keydown*, *keyup*.

lefty language implements *scalar* and *table*, where the former is a number of character string of arbitrary length while the latter is a 1D array indexed by numbers or strings. The smallest program unit is the expression which is parsed and executed in a sequential fashion, meaning an expression is evaluated instantly unless a function containing it has to be defined and called later. Listing 2 exemplifies a simple lefty function creating an empty graphics view. To execute this function, user can simply type *definit()*.

Listing 2: A *lefty* program creating an empty graphics view. *source:*`http://www.graphviz.org/pdf/leftyguide.pdf`

```
1  definit = function () {
2          defview = createwidget (-1, [
3                  'type' = 'view';
4                  'name' = 'graphics view';
5                  'origin' = ['x' = 1; 'y' = 1;];
6                  'size' = ['x' = 400; 'y' = 500;];
7          ]);
8          defscroll = createwidget (defview, ['type' = '
               ↪ scroll';]);
9          defcanvas = createwidget (defscroll, [
10                 'type' = 'canvas';
11                 'origin' = ['x' = 1; 'y' = 1;];
12                 'size' = ['x' = 400; 'y' = 500;];
13                 'borderwidth' = 1;
14         ]);
15 };
```

A fancier example is shown in Listing 3, allowing user to interactively draw, move, and delete boxes. Note that some *lefty* scripts only implements generic functions without performing any actions, e.g. the previous example shown in Listing 2, thus, being able to reused without modifications. This can be conveniently done by saving the code to a file with extension ".lefty", and importing it by calling the *load()* function, as shown in line 1 in Listing 3. For detailed specifications of *lefty* language, we refer readers to GraphViz official tutorials.

Listing 3: A *lefty* program allowing to draw boxes via mouse clicks. *source:*`http://www.graphviz.org/pdf/leftyguide.pdf`

```
1  load ('def.lefty');
2  definit ();
3  #
```

```
4  # initialize window data
5  #
6  canvas = defcanvas;
7  wrect = [0 = ['x' = 0; 'y' = 0;]; 1 = ['x' = 800; 'y'
       ↪ = 500;];];
8  setwidgetattr (canvas, ['window' = wrect;]);
9  #
10 # data structures
11 #
12 objarray = [];
13 objnum = 0;
14 #
15 # misc functions
16 #
17 min = function (a, b) {
18 if (a <= b)
19 return a;
20 return b;
21 };
22 max = function (a, b) {
23 if (b <= a)
24 return a;
25 return b;
26 };
27 rectof = function (p1, p2) {
28 return [
29 0 = ['x' = min (p1.x, p2.x); 'y' = min (p1.y, p2.y);];
30 1 = ['x' = max (p1.x, p2.x); 'y' = max (p1.y, p2.y);];
31 ];
32 };
33 pointadd = function (p1, p2) {
34 return ['x' = p2.x + p1.x; 'y' = p2.y + p1.y;];
35 };
36 pointsub = function (p1, p2) {
37 return ['x' = p2.x - p1.x; 'y' = p2.y - p1.y;];
38 };
39 #
40 # rendering functions
41 #
42 drawbox = function (obj, color) {
43 box (canvas, obj, obj.rect, ['color' = 'red';]);
44 };
45 redrawboxes = function () {
46 local i;
47 clear (canvas);
48 for (i = 0; i < objnum; i = i + 1)
```

```
49  drawbox (objarray[i], 1);
50  };
51  redraw = function (canvas) {
52  redrawboxes ();
53  };
54  #
55  # editing functions
56  #
57  new = function (rect) {
58  objarray[objnum] = [
59  'rect' = rect;
60  'id' = objnum;
61  ];
62  objnum = objnum + 1;
63  return objarray[objnum - 1];
64  };
65  reshape = function (obj, rect) {
66  obj.rect = rect;
67  return obj;
68  };
69  move = function (obj, p) {
70  obj.rect[0] = pointadd (obj.rect[0], p);
71  obj.rect[1] = pointadd (obj.rect[1], p);
72  return obj;
73  };
74  delete = function (obj) {
75  if (obj.id ~= objnum - 1) {
76  objarray[obj.id] = objarray[objnum - 1];
77  objarray[obj.id].id = obj.id;
78  }
79  remove (objnum - 1, objarray);
80  objnum = objnum - 1;
81  };
82  #
83  # user interface functions
84  #
85  # left mouse button creates new box
86  # middle button moves a box
87  # right button deletes a box
88  #
89  leftdown = function (data) {
90  if (data.obj ~= null)
91  return;
92  leftbox = new (rectof (data.pos, data.pos));
93  drawbox (leftbox, 1);
94  setgfxattr (canvas, ['mode' = 'xor';]);
```

```
95  };
96  leftmove = function (data) {
97  if (~leftbox)
98  return;
99  drawbox (leftbox, 1);
100 clearpick (canvas, leftbox);
101 reshape (leftbox, rectof (data.ppos, data.pos));
102 drawbox (leftbox, 1);
103 };
104 leftup = function (data) {
105 if (~leftbox)
106 return;
107 drawbox (leftbox, 1);
108 clearpick (canvas, leftbox);
109 reshape (leftbox, rectof (data.ppos, data.pos));
110 setgfxattr (canvas, ['mode' = 'src';]);
111 drawbox (leftbox, 1);
112 remove ('leftbox');
113 };
114 middledown = function (data) {
115 if (data.obj == null)
116 return;
117 middlebox = data.obj;
118 middlepos = data.pos;
119 setgfxattr (canvas, ['mode' = 'xor';]);
120 };
121 middlemove = function (data) {
122 if (~middlebox)
123 return;
124 drawbox (middlebox, 1);
125 clearpick (canvas, middlebox);
126 move (middlebox, pointsub (middlepos, data.pos));
127 middlepos = data.pos;
128 drawbox (middlebox, 1);
129 };
130 middleup = function (data) {
131 if (~middlebox)
132 return;
133 drawbox (middlebox, 1);
134 clearpick (canvas, middlebox);
135 move (middlebox, pointsub (middlepos, data.pos));
136 setgfxattr (canvas, ['mode' = 'src';]);
137 drawbox (middlebox, 1);
138 remove ('middlepos');
139 remove ('middlebox');
140 };
```

```
141 rightup = function (data) {
142 if (data.pobj == null)
143 return;
144 drawbox (data.obj, 0);
145 clearpick (canvas, data.obj);
146 delete (data.obj);
147 };
148 dops = function () {
149 local s;
150
151 s = ['x' = 8 * 300; 'y' = 10.5 * 300;];
152 canvas = createwidget (-1, ['type' = 'ps'; 'size' = s
      ↪ ;]);
153 setwidgetattr (canvas, ['window' = wrect;]);
154 redraw (canvas);
155 destroywidget (canvas);
156 canvas=defcanvas;
157 };
```

dotty Based on *lefty*, GraphViz also builds a tool allowing user to interactively create a new graph or edit an existing graph specified by DOT script.

The default view of *dotty* is an empty white canvas, allowing user to draw nodes and edges using mouse operations. Specifically, a node, which is initially represented as a black unfilled ellipse, can be added by left clicking at an arbitrary location inside the canvas. An edge between two nodes can be drawn by pressing down the middle button at the first one, moving cursor to the second, then releasing the button. Figure 6 shows a screen-shot of two nodes and an edge.

To manipulate graphical attributes of a node or an edge, we need to right click on the object to open the operation menu, and select "set attr", as shown in Figure 7. This will open a dialog allowing use to modify the value of an attribute, e.g. *color=blue*, as shown in Figure 8. We can repeat this process to specify multiple attributes, e.g. *style=filled*, *label="source"*, *shape=box*, etc..

Note that *dotty* will not arrange an appropriate layout until we explicitly call the function, i.e. we need to right click at an empty location inside the canvas and select *do layout* option, as shown in Figure 9, and the result is shown in Figure 10.

To open an existing graph, we need to go to the operation menu again as shown in Figure 9 and click "load graph" to select a DOT script, e.g. the family tree example we created, in the file selection dialog. By doing this, we can interactively modify its graphical attributes and manually rearrange locations of nodes and edges to create a layout based on our own specification.

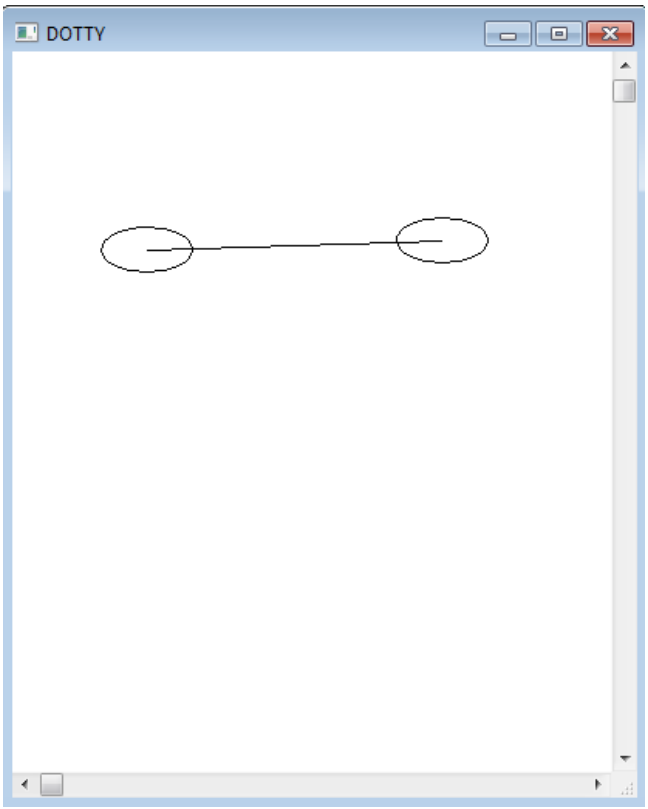

Figure 6: Drawing two nodes with an edge conneting them in *dotty*, where the nodes are represented by black unfilled ellipses.

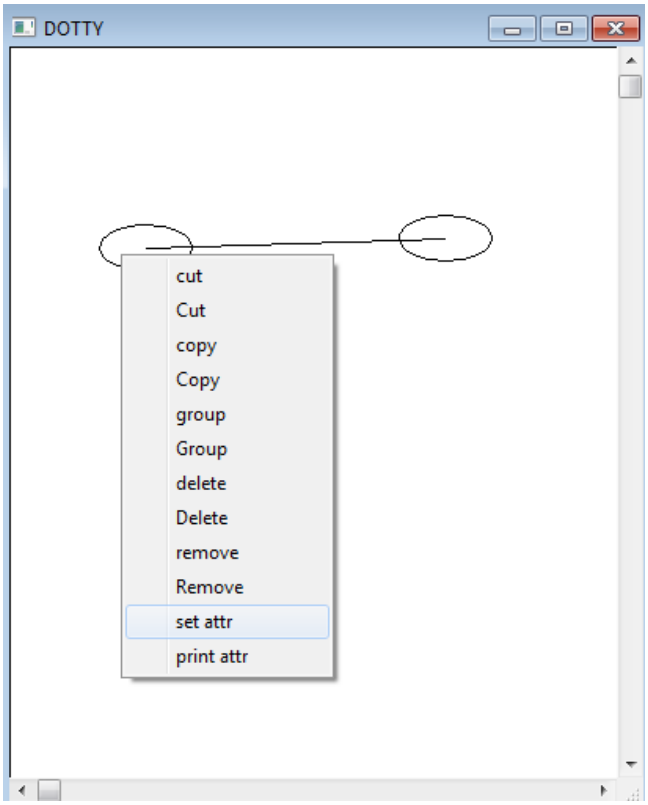

Figure 7: A example of specifying graphcial atrributes of a node.

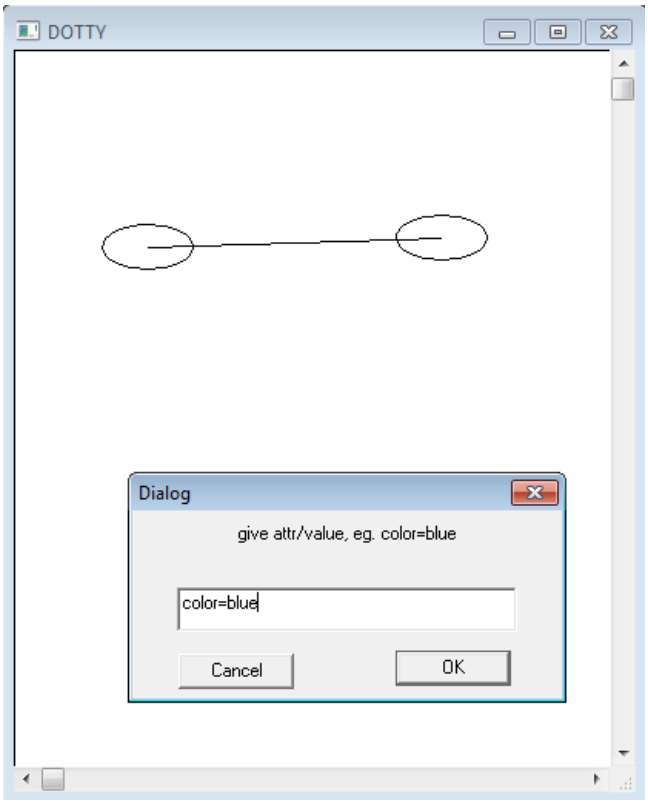

Figure 8: A screen-shot of changing color of a node to blue.

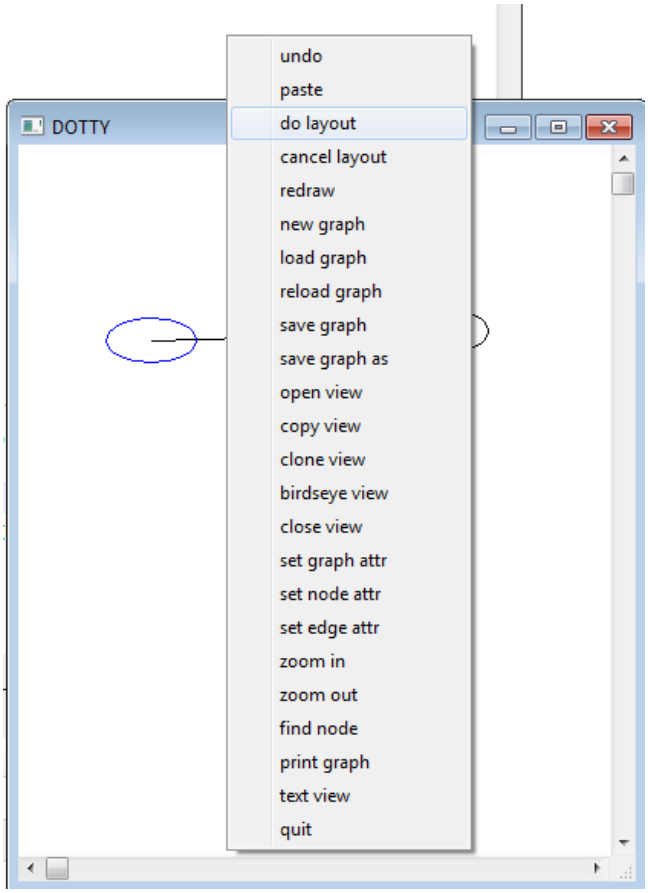

Figure 9: A screen-shot of opening the operation menu through right clicking.

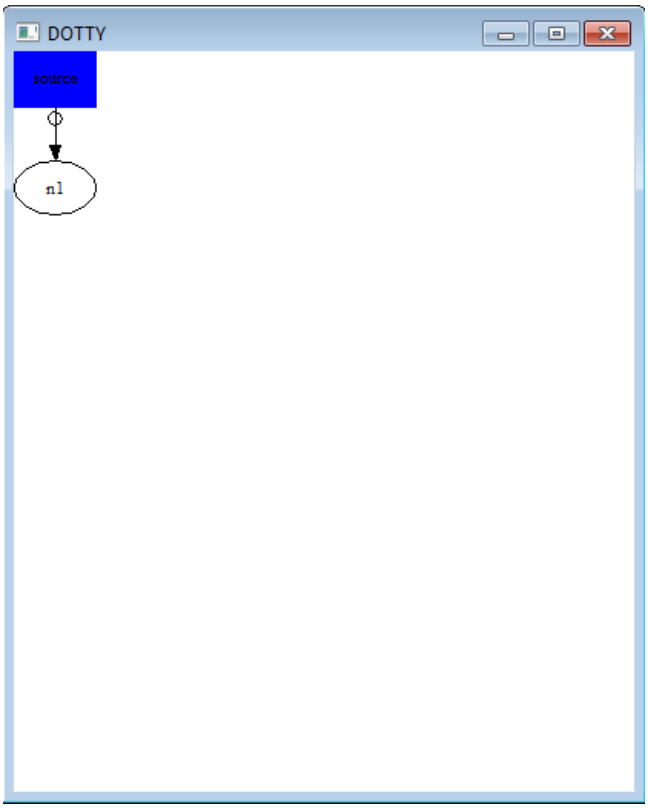

Figure 10: A screen-shot of graph after doing layout.