HowTo Render A Graph Using Rgraphviz

Jeff Gentry

April 25, 2006

1 Overview

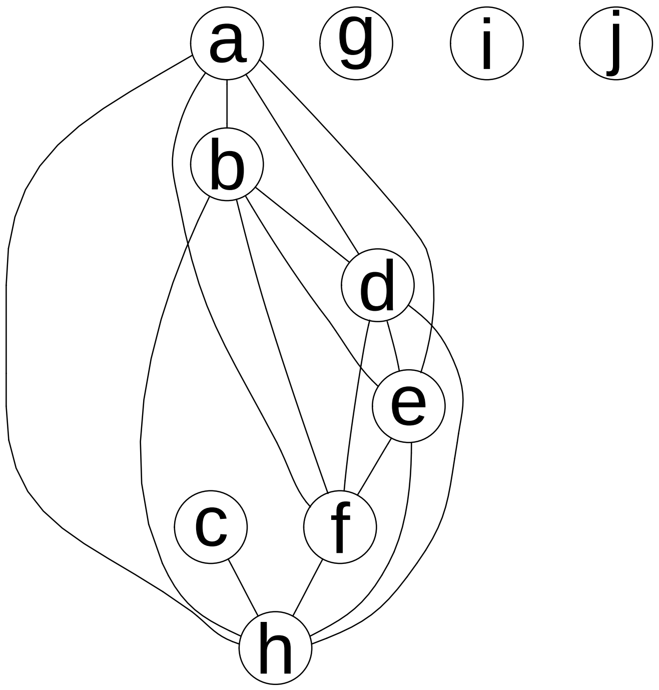
This article will demonstrate how to easily render a graph from R into various formats using the *Rgraphviz*. To do this, first we need to generate a R graph using the *graph* package:

```
> library(Rgraphviz)
> set.seed(123)
> V <- letters[1:10]
> M <- 1:4
> g1 <- randomGraph(V, M, 0.2)
> g1
```

```
A graphNEL graph with undirected edges
Number of Nodes = 10
Number of Edges = 16
```

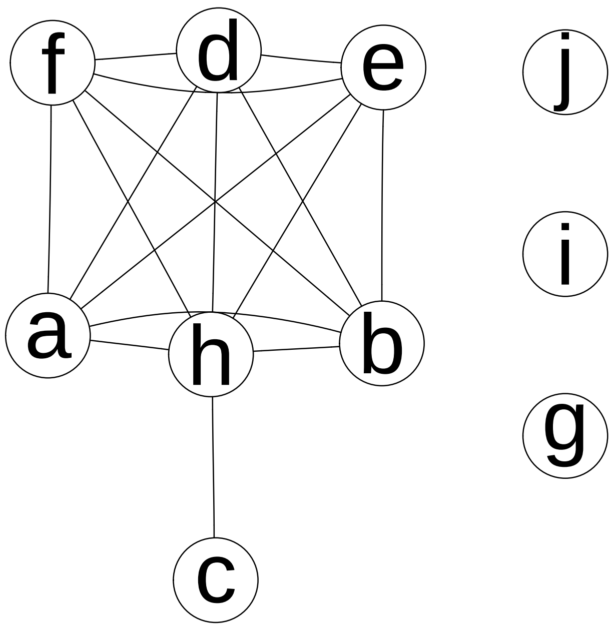
2 Plotting in R Using Different Layout Methods

It is quite simple to generate a R plot window to display your graph. Once you have your graph object, simply use the `plot` method:



The *Rgraphviz* package allows you to specify varying layout engines, such as ”dot” (the default), ”neato”, and ”twopi”. This can be done using the call to `plot`:

```
> z <- plot(g1, "neato")
```



The "twopi" layout method requires a graph to be fully connected. To determine if your graph is fully connected:

```
> isConnected(g1)
```

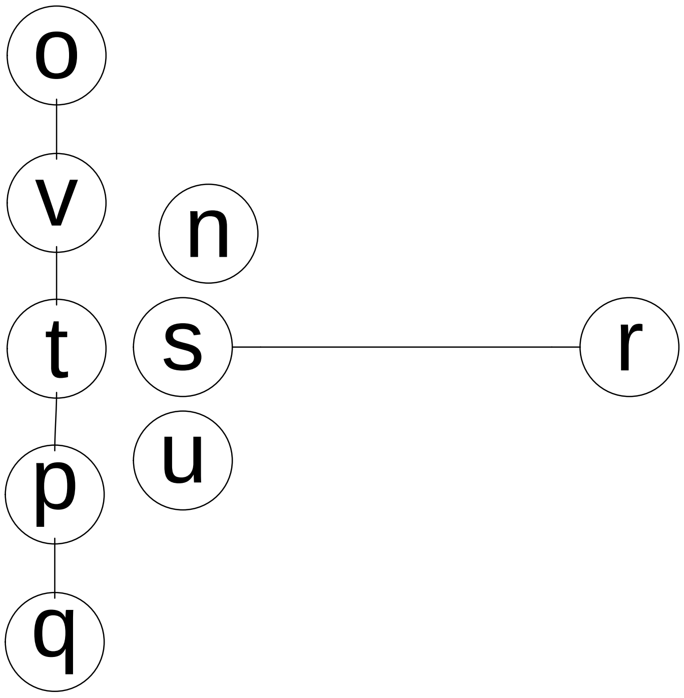
```
[1] FALSE
```

A working "twopi" layout can be seen with this graph:

```
> set.seed(123)
> V <- letters[14:22]
> g2 <- randomEGraph(V, 0.2)
> isConnected(g2)
```

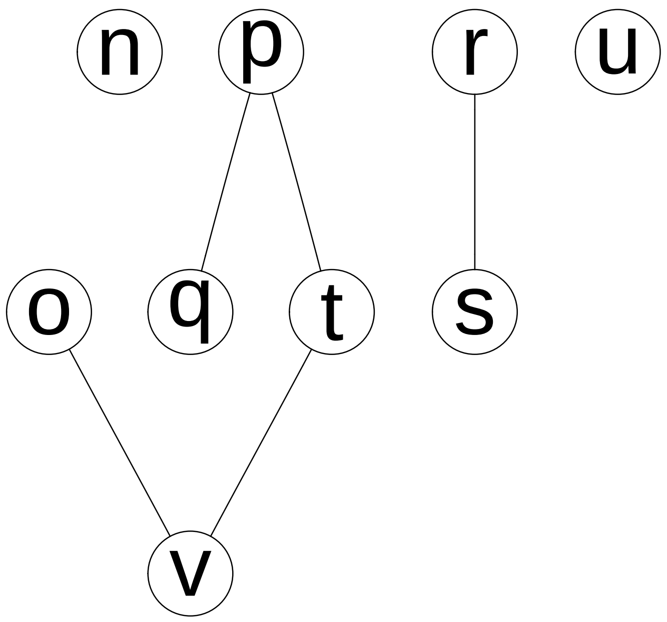
```
[1] FALSE
```

```
> z <- plot(g2, "twopi")
```

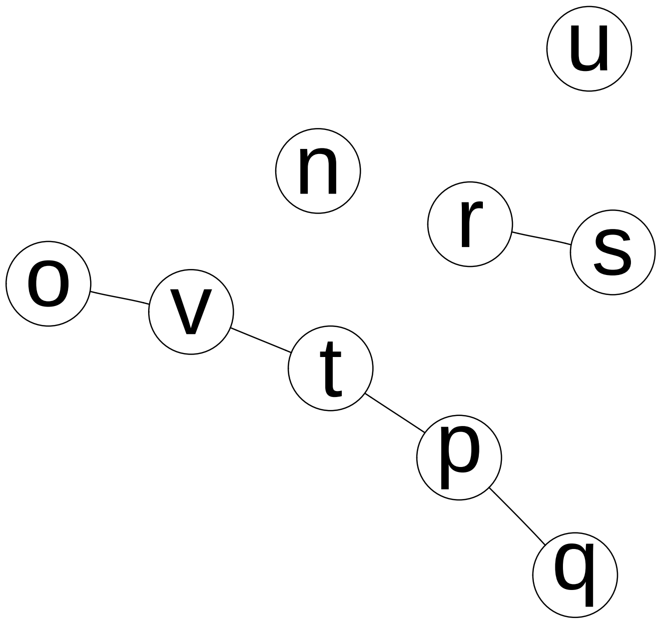


And finally, to demonstrate how the differing layout methods work on this second graph:

```
> z <- plot(g2, "dot")
```

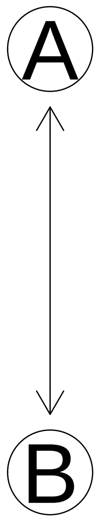


```
> z <- plot(g2, "neato")
```



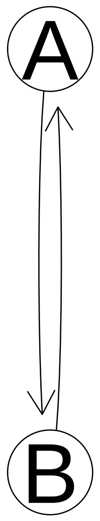
Note that there is an option, *recipEdges* that details how to deal with reciprocated edges in a graph. The two options are *combined* (the default) and *distinct*. This is mostly useful in directed graphs that have reciprocating edges - the *combined* option will display them as a single edge with an arrow on both ends while *distinct* shows them as two separate edges.

```
> rEG <- new("graphNEL", nodes = c("A", "B"), edgemode = "directed")
> rEG <- addEdge("A", "B", rEG, 1)
> rEG <- addEdge("B", "A", rEG, 1)
> plot(rEG)
```



In this first example above, the edges were combined, whereas below they are showed separately.

```
> plot(rEG, recipEdges = "distinct")
```



The function `removedEdges` can be used to return a numerical vector detailing which edges (if any) would be removed by the combining of edges.

```
> a <- removedEdges(g1)
> a
```

```
 [1]  7 12 13 17 18 19 22 23 24 25 27 28 29 30 31 32
```

3 SubGraphs

Rgraphviz supports the ability to define specific clustering of nodes. This will instruct the layout algorithm to attempt to keep the clustered nodes close together. To do this, one must first generate the desired set (one or more) of subgraphs with the *graph* object.

```
> sg1 <- subGraph(c("a", "d", "j", "i"), g1)
> sg1
```

```
A graphNEL graph with undirected edges
Number of Nodes = 4
Number of Edges = 1
```

```
> sg2 <- subGraph(c("b", "e", "h"), g1)
> sg2
```

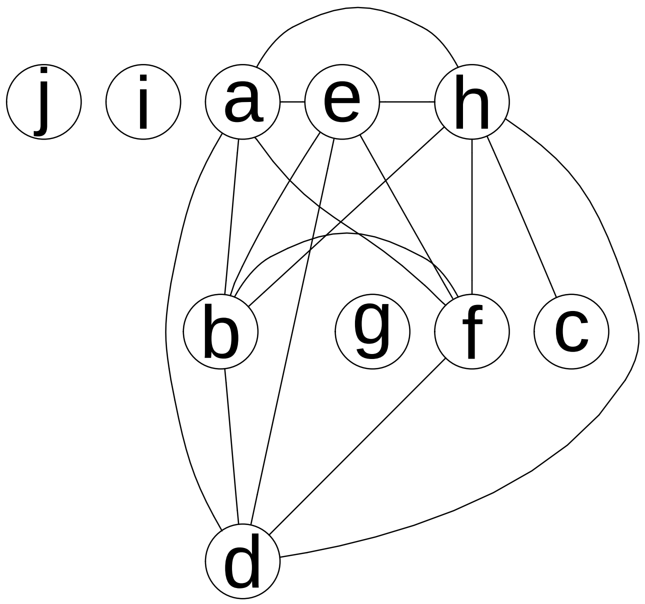
```
A graphNEL graph with undirected edges
Number of Nodes = 3
Number of Edges = 3

> sg3 <- subGraph(c("c", "f", "g"), g1)
> sg3

A graphNEL graph with undirected edges
Number of Nodes = 3
Number of Edges = 0
```

To plot using the subgraphs, one must use the `subGList` argument which is a list of lists, with each sublist having three elements:

- *graph* : The actual *graph* object for this subgraph.
- *cluster* : A logical value noting if this is a `cluster` or a `subgraph`. A value of *TRUE* (the default, if this element is not used) indicates a `cluster`. In Graphviz, `subgraphs` are used as an organizational mechanism but are not necessarily laid out in such a way that they are visually together. Clusters are laid out as a separate graph, and thus Graphviz will tend to keep nodes of a cluster together. Typically for *Rgraphviz* users, a `cluster` is what one wants to use.
- *attrs* : A named vector of attributes, where the names are the attribute and the elements are the value. For more information about attributes, see the section `Attributes` below.

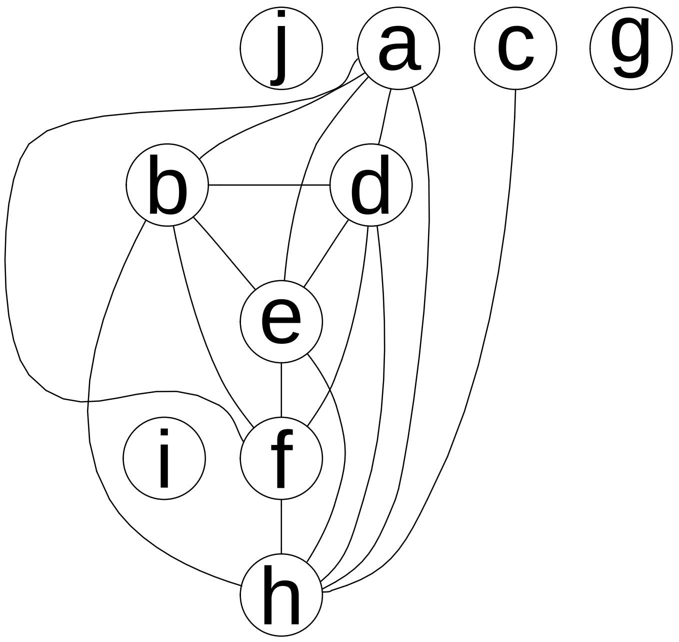
Please note that only the *graph* element is required. If the *cluster* element is not specified, the subgraph is assumed to be a `cluster` and if there are no attributes to specify for this subgraph then *attrs* is unnecessary.

```
> subGList <- vector(mode = "list", length = 3)
> subGList[[1]] <- list(graph = sg1)
> subGList[[2]] <- list(graph = sg2, cluster = FALSE)
> subGList[[3]] <- list(graph = sg3)
> plot(g1, subGList = subGList)
```



To demonstrate the differences that will appear with different subgraph patterns, another example is provided:

```
> sg1 <- subGraph(c("a", "c", "d", "e", "j"), g1)
> sg2 <- subGraph(c("f", "h", "i"), g1)
> plot(g1, subGList = list(list(graph = sg1), list(graph = sg2)))
```



3.1 A Note About Edge Names

While internal node naming is quite straight forward (it is simply taken from the *graph* object), *Rgraphviz* needs to be able to uniquely identify edges by name. End users as well will need to be able to do this to correctly assign attributes (see `Attributes`). The name of an edge is `x~y` where `x` is the tail node and `y` is the head node. The method `edgeNames` can be used to obtain a vector of all edge names, and takes the argument `recipEdges` so that the output correctly matches which edges will be used by *Rgraphviz*.

```
> edgeNames(g1)

 [1] "a~b" "a~d" "a~e" "a~f" "a~h" "b~f" "b~d" "b~e" "b~h" "c~h" "d~e" "d~f"
[13] "d~h" "e~f" "e~h" "f~h"

> edgeNames(g1, recipEdges = "distinct")

 [1] "a~b" "a~d" "a~e" "a~f" "a~h" "b~f" "b~a" "b~d" "b~e" "b~h" "c~h" "d~a"
[13] "d~b" "d~e" "d~f" "d~h" "e~a" "e~b" "e~d" "e~f" "e~h" "f~b" "f~a" "f~d"
[25] "f~e" "f~h" "h~c" "h~a" "h~b" "h~d" "h~e" "h~f"
```

4 Attributes

5 The Attributes List

There are many visualization options in Graphviz that can be set beyond those which are given explicit options using Rgraphviz - such as colors of nodes and edges, which node to center on for twopi plots, node labels, edge labels, edge weights, arrow heads and tails, etc. A list of all available attributes is accessible online at: `http://www.graphviz.org/pub/scm/graphviz2/doc/info/attrs.html`. Note that there are some differences between default values and also some attributes will not have an effect in Rgraphviz. Please see the man page for `graphvizAttributes` for more details.

Attributes can be set both globally (for the entire graph, for all edges, all nodes, etc) as well as on a per-node and per-edge basis. Global attributes are set via a list and passed in as the *attrs* argument to `plot`. A default set of global attributes are used for global values which are not specified (by using the `getDefaultAttrs` function). The `getDefaultAttrs` function will take a partial global attribute list (see below for a description) and/or the layout type to be used (dot, neato, or twopi) and will generate an attribute list to be used with defaults for values that the user did not specify. The list has four elements: 'graph', 'cluster', 'edge' and 'node'. Within each element is another list, where the names correspond to attributes and the values correspond to the value to use globally on that attribute. An example of this structure can be seen with the default list provided by `getDefaultAttrs`:

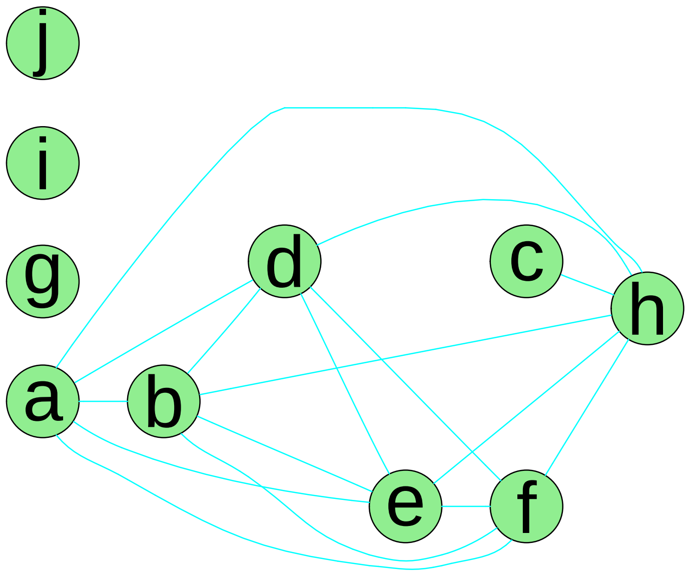
```
> defAttrs <- getDefaultAttrs()
> defAttrs

$graph
$graph$bgcolor
[1] "transparent"

$graph$fontcolor
[1] "black"

$graph$ratio
[1] "fill"

$graph$overlap
[1] ""

$graph$splines
[1] TRUE

$graph$rank
[1] "same"
```

```
$graph$size
[1] "11.1929133858268,7.76771653543307"

$graph$rankdir
[1] "TB"


$cluster
$cluster$bgcolor
[1] "transparent"

$cluster$color
[1] "black"

$cluster$rank
[1] "same"


$node
$node$shape
[1] "circle"

$node$fixedsize
[1] TRUE

$node$fillcolor
[1] "transparent"

$node$label
[1] ""

$node$color
[1] "black"

$node$fontcolor
[1] "black"

$node$fontsize
[1] "14"

$node$height
[1] "0.5"

$node$width
[1] "0.75"
```

```
$edge
$edge$color
[1] "black"

$edge$dir
[1] "both"

$edge$weight
[1] "1.0"

$edge$label
[1] ""

$edge$fontcolor
[1] "black"

$edge$arrowhead
[1] "none"

$edge$arrowtail
[1] "none"

$edge$fontsize
[1] "14"

$edge$labelfontsize
[1] "11"

$edge$arrowsize
[1] "1"

$edge$headport
[1] "center"

$edge$layer
[1] ""

$edge$style
[1] "solid"

$edge$minlen
[1] "1"
```

To manually set some attributes, but not others, pass in a list with the specific attributes that you desire. In the following example, we will be setting two attributes (*label* and *fillcolor* for nodes, one for edges (*color*) and one for the graph itself (*rankdir*). We could also have called `getDefaultAttrs` with the same list that we are passing as the *attrs* argument, but there is no need here.

```
> plot(g1, attrs = list(node = list(label = "foo", fillcolor = "lightgreen"),
+     edge = list(color = "cyan"), graph = list(rankdir = "LR")))
```



Users can also set attributes per-node and per-edge. In this case, if an attribute is defined for a particular node then that node uses the specified attribute and the rest of the nodes use the global default. Note that any attribute that is set on a per-node or per-edge basis `must` have a default set globally, due to the way that Graphviz sets attributes. Both the per-node and per-edge attributes are set in the same basic manner - the attributes are set using a list where the names of the elements are the attributes, and each element contains a named vector. The names of this vector correspond to either node names or edge names, and the values of the vector are the values to set the attribute to for that node or edge. The following sections will demonstrate how to set per-node and per-edge attributes for commonly desired tasks. For these we will construct two lists, *nAttrs* and *eAttrs* to pass in to `plot`.

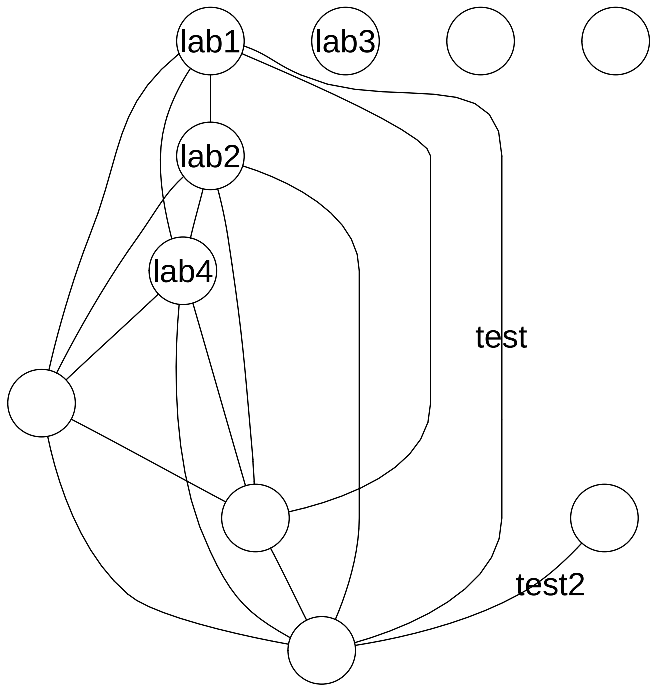
```
> nAttrs <- list()
> eAttrs <- list()
```

Please note to take care with edge names. If *recipEdges* is set to *combined*, then only one of any pair of reciprocal edges will actually be used. Users should utilize the `edgeNames` method to be sure that they are setting attributes for the right edge names.

6 Labels

By default, nodes use the node name as their label and edges do not have a label. However, both can have custom labels supplied via attributes.

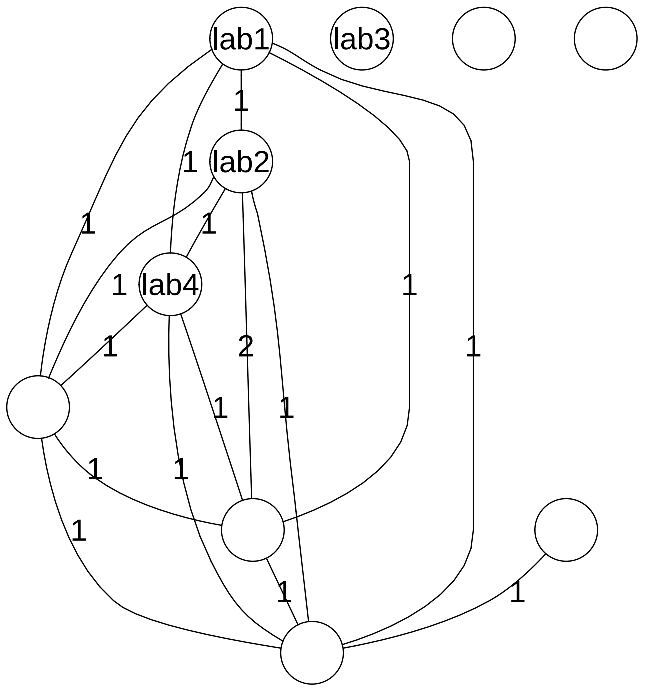
```
> nAttrs$label <- c(a = "lab1", b = "lab2", g = "lab3", d = "lab4")
> nAttrs

$label
     a      b      g      d 
"lab1" "lab2" "lab3" "lab4" 

> eAttrs$label <- c("a~h" = "test", "c~h" = "test2")
> eAttrs

$label
    a~h     c~h 
 "test" "test2" 

> plot(g1, nodeAttrs = nAttrs, edgeAttrs = eAttrs)
```



6.1 Using Edge Weights For Labels

A common desire for edge weights is to use the edge weights of the edges for the labels on a plotted graph. This can be done with just a couple of extra steps. First we will get the edge weights, and unlist them, to provide them in vector format. Then, first we will determine which of those to remove (this step is only necessary if *recipEdges* is set to *TRUE*, which is default behavior for both undirected and directed graphs) and remove those positions from our vector. Finally, we will get the set of edge names which will be used for plotting and bundle that into the appropriate structure for plotting.

```
> ew <- edgeWeights(g1)
> lw <- unlist(unlist(ew))
> toRemove <- removedEdges(g1)
> if (length(toRemove) > 0) lw <- lw[-toRemove]
> names(lw) <- edgeNames(g1)
> eAttrs$label <- lw
> plot(g1, nodeAttrs = nAttrs, edgeAttrs = eAttrs)
```



7 Adding Some Color

There are many areas where color can be specified to the plotted graph. Edges can be drawn in a non-default color, as can nodes. Nodes can also have a specific *fillcolor* defined, detailing what color the interior of the node should be. The color used for the labels can also be specified with the *fontcolor* attribute.

```
> nAttrs$color <- c(a = "red", b = "red", g = "green", d = "blue")
> eAttrs$color <- c("a~d" = "blue", "c~h" = "purple")
> nAttrs$fillcolor <- c(j = "yellow")
> nAttrs$fontcolor <- c(e = "green", f = "red")
> eAttrs$fontcolor <- c("a~h" = "green", "c~h" = "brown")
> nAttrs
```

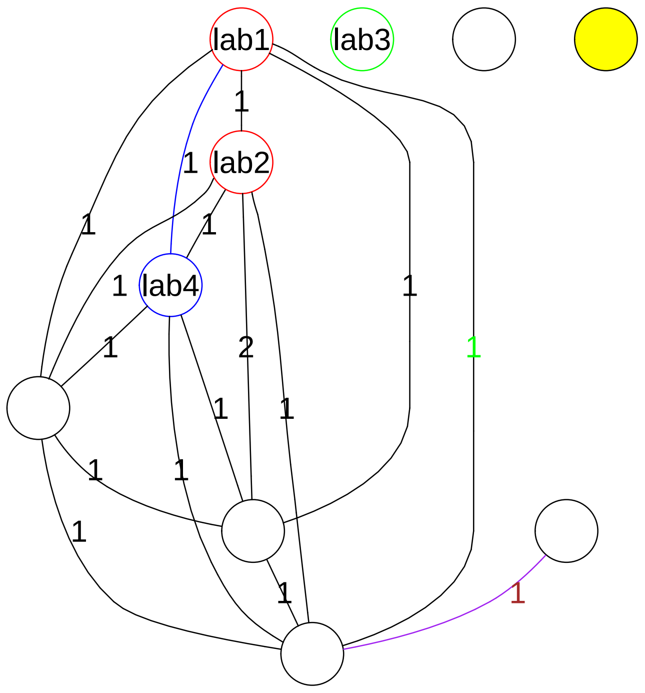
```
$label
      a       b       g       d 
"lab1" "lab2" "lab3" "lab4" 

$color
       a        b        g        d 
   "red"    "red" "green"   "blue" 
```

```
$fillcolor
       j
"yellow"

$fontcolor
      e       f
"green"   "red"
```

```
> eAttrs
```

```
$label
a~b a~d a~e a~f a~h b~f b~d b~e b~h c~h d~e d~f d~h e~f e~h f~h
  1   1   1   1   1   2   1   1   1   1   1   1   1   1   1   1

$color
     a~d      c~h
  "blue" "purple"

$fontcolor
    a~h     c~h
"green" "brown"
```

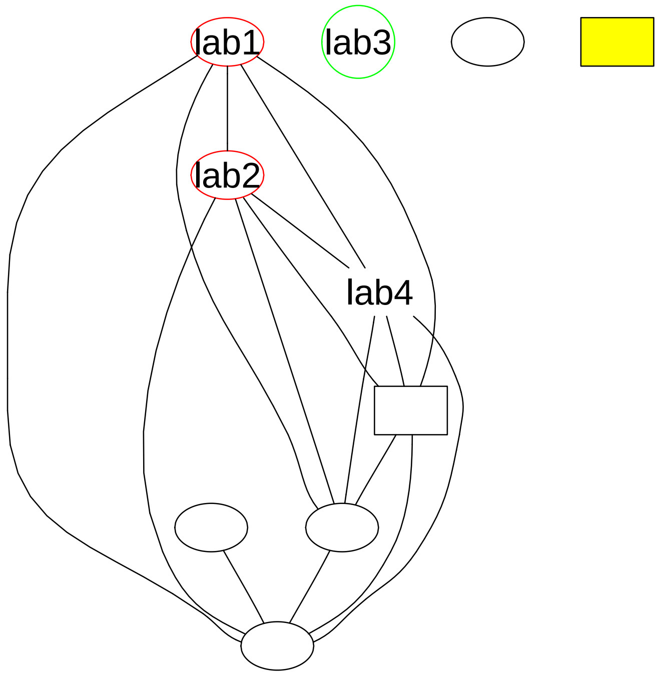
```
> plot(g1, nodeAttrs = nAttrs, edgeAttrs = eAttrs)
```



8 Node Shapes

The *Rgraphviz* package allows you to specify different shapes for your nodes. Currently, the only shapes allowed are *circle* (the default), *ellipse*, *plaintext* and *box* (Note that *plaintext* is simply a *box* that is not displayed for purposes of layout). As with previous attributes, the shape can be set globally or for specific nodes. Here is the same graph from the previous example, with the default shape as *ellipse* and with two nodes specified as being *box*, one as a *circle* and one as a *plaintext* node:

```
> defAttrs$node$shape <- "ellipse"
> nAttrs$shape <- c(e = "box", g = "circle", j = "box", d = "plaintext")
> plot(g1, attrs = defAttrs, nodeAttrs = nAttrs)
```



9 Setting attributes via node and edge lists

The user can take a different direction in setting up attributes and laying out the graph then the one presented above. The following method can be used to replicate exactly the same sorts of behaviour described above, but can be more flexible in some other cases. The functions `buildNodeList` and `buildEdgeList` will generate a list of *pNode* and *pEdge* objects respectively. These are used to provide the information for the actual Graphviz layout (and by default are generated automatically). By generating these manually before the layout, one can edit these objects and perform the layout with these edited lists.

For example:

```
> nodes <- buildNodeList(g1)
> edges <- buildEdgeList(g1)
> nodes[[1]]
```

```
An object of class "pNode"
Slot "name":
[1] "a"

Slot "attrs":
$label
[1] "a"


Slot "subG":
[1] 0
```

```
> edges[[1]]
```

```
An object of class "pEdge"
Slot "from":
[1] "a"

Slot "to":
[1] "b"

Slot "attrs":
$arrowhead
[1] "none"

$weight
[1] "1"


Slot "subG":
[1] 0
```

You can now see the contents of the first *pNode* and first *pEdge* objects in their respective lists. These two functions can also utilize the attribute lists that were passed into `agopen`. Note that if we are using default attributes, that for the `buildNodeList` and `buildEdgeList` functions we only want to pass in defaults for node and edges, respectively.

```
> nodes <- buildNodeList(g1, nodeAttrs = nAttrs, defAttrs = defAttrs$node)
> edges <- buildEdgeList(g1, edgeAttrs = eAttrs, defAttrs = defAttrs$edge)
> nodes[[1]]
```

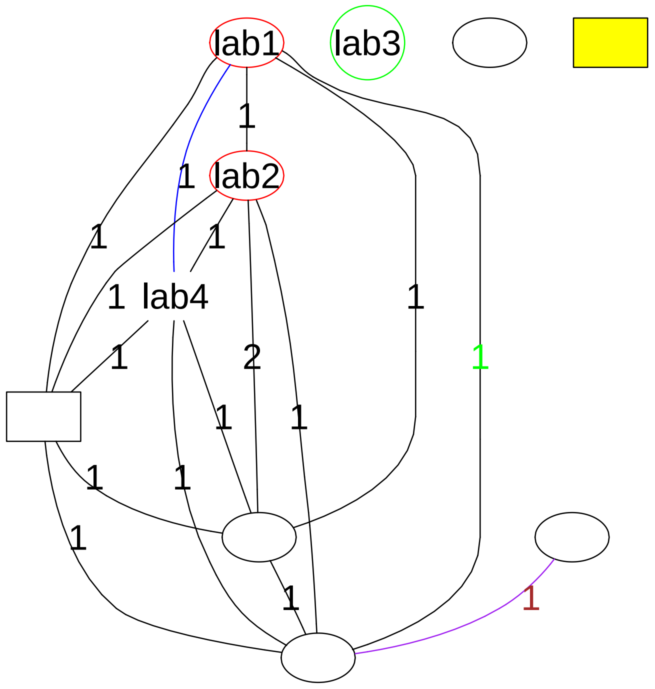
```
An object of class "pNode"
Slot "name":
[1] "a"

Slot "attrs":
$label
[1] "lab1"

$color
[1] "red"

$fillcolor
[1] "transparent"

$fontcolor
[1] "black"

$shape
[1] "ellipse"


Slot "subG":
[1] 0
```

```
> edges[[1]]

An object of class "pEdge"
Slot "from":
[1] "a"

Slot "to":
[1] "b"

Slot "attrs":
$arrowhead
[1] "none"

$weight
[1] "1"

$label
[1] "1"

$color
[1] "black"
```

```
$fontcolor
[1] "black"


Slot "subG":
[1] 0
```

Notice the difference between the objects in the second example as compared with the first, containing the specified attributes. Now we can plot this graph, which should look identical to the previous plot:

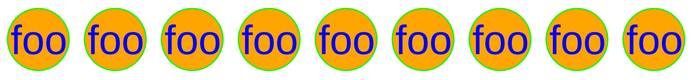
```
> vv <- agopen(name = "foo", nodes = nodes, edges = edges, edgeMode = "undirected")
> plot(vv)
```



Here we've added our own arrowheads to the a e and a h edges as well as added an arrowtail to the graph - while visually indicating direction, these will have no bearing on the layout itself as Graphviz will view these edges as undirected. This same technique can be used in the case where a directed graph has reciprocated edges and one wants to combine those edges into single edges with arrows in both directions.

Next we will use a completely different graph, one of the graphs as part of the *graphExamples* dataset in the *graph* package and provide another demonstration of working with attributes to customize your plot.

```
> data(graphExamples)
> z <- graphExamples[[17]]
> nNodes <- length(nodes(z))
> nA <- list()
> nA$fixedSize <- rep(FALSE, nNodes)
> nA$height <- nA$width <- rep("1", nNodes)
> nA$label <- rep("foo", nNodes)
> nA$color <- rep("green", nNodes)
> nA$fillcolor <- rep("orange", nNodes)
> nA$shape <- rep("circle", nNodes)
> nA$fontcolor <- rep("blue", nNodes)
> nA$fontsize <- rep(14, nNodes)
> nA <- lapply(nA, function(x) {
+     names(x) <- nodes(z)
+     x
+ })
> plot(z, nodeAttrs = nA)
```



10 Plotting with non-standard nodes

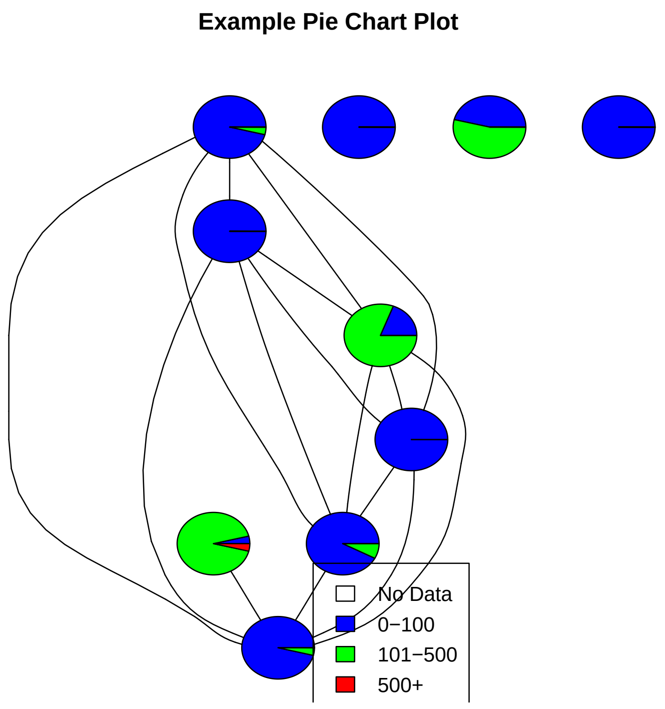
The *Rgraphviz* package provides for non-standard node drawing. Note that these nodes are shaped the same as standard nodes, but are able to provide for richer information in the actual display.

To do this, lay out the graph using the shape desired - then, when plotting the laid out graph, one can use the *drawNode* argument to `plot` to define how the nodes are drawn. This argument can be either of length one (in which case all nodes are drawn with it) or a list of length equal to the number of nodes in the graph (in which case the first element of the list is used to draw the first node, etc). To work correctly, the function will take four arguments - the first *node* is an object of class *AgNode*, which describes the node's location and other information and the second parameter, *ur* is of class *XYPoint* and describes the upper right hand point of the bounding box (where the lower left is 0,0). The third parameter, *attrs*, is a node attribute list as discussed in the "Attributes" section and represents post-layout attribute changes where the user wants to override values present in the layout. The fourth argument, *radConv* is used by *Rgraphviz* to convert Graphviz units to R plotting units. This argument will probably not need to be specified by any custom drawing function, but does need to exist. A custom drawing function is free to ignore these values, but the argument must exist in the function declaration to at least accept the value being passed in. The default function for node drawing on all nodes is `drawAgNode`, so if one wants to use a custom function for some nodes but the standard function for others, the list passed in to *drawNode* can have the custom functions in the elements corresponding to those nodes desired to have special display and `drawAgNode` in the elements corresponding to the nodes where standard display is desired.

One function included with the *Rgraphviz* package that can be used for such alternate node drawing is `pieGlyph`. This allows users to put arbitrary pie charts in as circular nodes. As an example, we will take the *eset* dataset from the *Biobase* package and will create a graph where each node corresponds to one of a set of Affymetrix probes represented in that exprSet and draw each node with a pie chart representing the expression levels of the samples in the exprSet for that probe.

```
> if (require("Biobase")) {
+     data(eset)
+     exprs <- exprs(eset)[100:109, ]
+     probes <- rownames(exprs)
+     set.seed(123)
+     pieGraph <- randomGraph(probes, 1:4, 0.2)
+     pgLayout <- agopen(pieGraph, "foo")
+     counts <- apply(exprs, 1, function(x) {
+         table(cut(x, breaks = c(-Inf, 100, 500, Inf)))
+     })
+     plotPieChart <- function(curPlot, counts) {
```

```
+         buildDrawing <- function(x) {
+             force(x)
+             y <- x * 100 + 1
+             function(node, ur, attrs = list(), radConv = 1) {
+                 nodeCenter <- getNodeCenter(node)
+                 pieGlyph(y, xpos = getX(nodeCenter), ypos = getY(nodeCenter),
+                   radius = getNodeRW(node), col = c("blue", "green",
+                     "red"))
+             }
+         }
+         drawing <- vector(mode = "list", length = length(probes))
+         for (i in 1:length(drawing)) {
+             drawing[[i]] <- buildDrawing(counts[, i])
+         }
+         plot(curPlot, drawNode = drawing, main = "Example Pie Chart Plot")
+         legend(240, 100, legend = c("No Data", "0-100", "101-500",
+             "500+"), fill = c("white", "blue", "green", "red"))
+     }
+     plotPieChart(pgLayout, counts)
+ } else {
+     cat("This example is missing since you do not have Biobase")
+ }
```



To construct this plot, we constructed a complete function, although this is not necessary - one can take any path they desire to build the list of drawing functions. Also note that in this plot the nodes do not have labels as it would look confusing, but those could be easily added with a line such as *drawTxtLabel(txtLabel(node), getX(nodeCenter), getY(nodeCenter))* in the `buildDrawing` sub-function above. The `drawAgNode` should be used as a guide for basic activities such as this.

11 Other types of graphs

Up to this point, we have only been working with objects of class *graphNEL*, but the other subclasses of graph (such as *distGraph* and *clusterGraph*) will work as well (provided that they support the `nodes` method as well as have an `edgeL` method defined to generate an edge list like the one for *graphNEL*).

In this section, we'll demonstrate a few examples of using graphs of classes other then *graphNEL*. Users should not notice a difference in the actual interface, but this will also provide some visual examples as to how these types of graphs will appear.

For our first set of examples, we will create an object of class *clusterGraph* and then plot it using all three layout methods:

```
> cG <- new("clusterGraph", clusters = list(a = c(1:10), b = c(11:13),
+     c = c(14:20), d = c(21, 22)))
> cG

A graph with  undirected  edges
Number of Nodes = 22
Number of Edges = 70
```

In Figure 1 we show the same graph, *cG* laid out using three different algorithms.

12 Laying out a bipartite graph

Here we provide a simple example of laying out a bipartite graph. There are two types of nodes, and edges go only from one type to the other. We first construct the bipartite graph and set various node attributes. We want to have color for the nodes, and we want to lay the graph out from left to right, rather than vertically.

```
> library("RColorBrewer")
> twocolors <- c(brewer.pal(11, "RdYlGn")[7], brewer.pal(11, "RdYlBu")[7])
> myNodes <- c("m1", "m2", "m3", "m4", "r1", "r2")
> myEdges <- list(m1 = list(edges = c("r1")), m2 = list(edges = c("r1")),
+     m3 = list(edges = c("r2")), m4 = list(edges = c("r2")), r1 = list(edges = c("m3")),
```

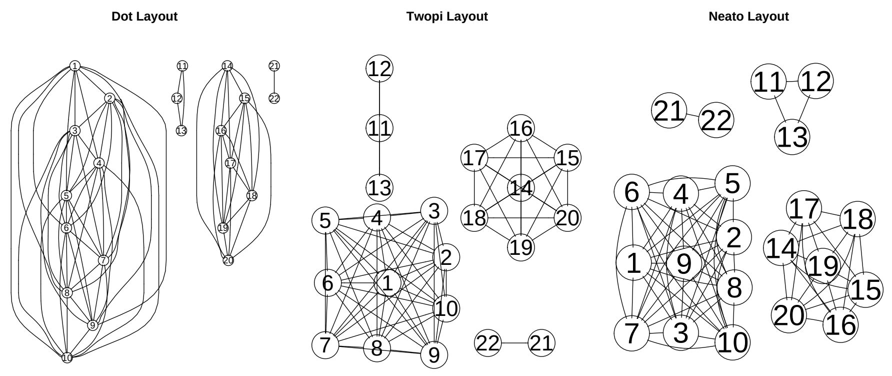


Figure 1: A cluster graph laid out using the three layout algorithm.

```
+     r2 = list())
> g <- new("graphNEL", nodes = myNodes, edgeL = myEdges, edgemode = "directed")
> g
```

```
A graphNEL graph with directed edges
Number of Nodes = 6
Number of Edges = 5
```

Next we set up the node attributes and create subgraphs so that we can better control the layout.

```
> nA = makeNodeAttrs(g, fillcolor = twocolors[1])
> sg1 = subGraph(c("r1", "r2"), g)
> v1 = "sink"
> names(v1) = "rank"
> sgL = list(list(graph = sg1, cluster = FALSE, attrs = v1))
> defA = list(graph = list(rankdir = "LR", rank = ""))
```

Finally, in Figure 2 we plot the bipartite graph.

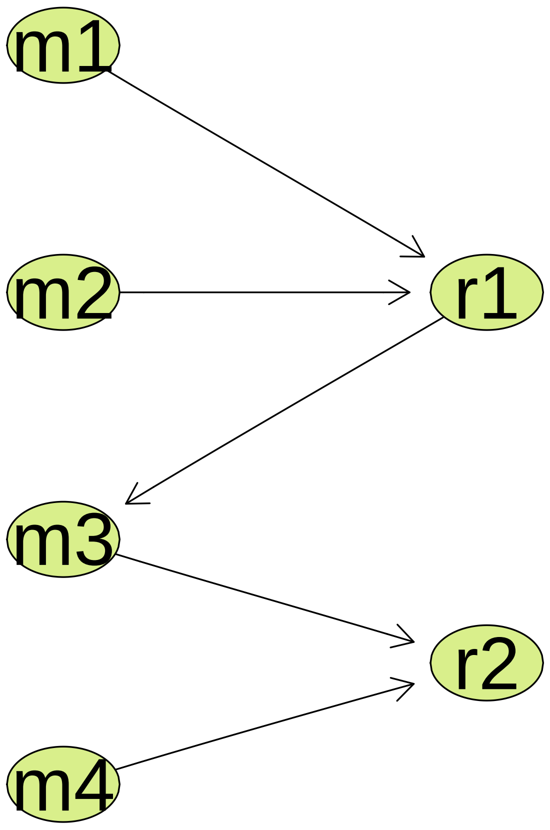


Figure 2: A bipartite graph.