HowTo Render A Graph Using Rgraphviz

Jeff Gentry

May 14, 2004

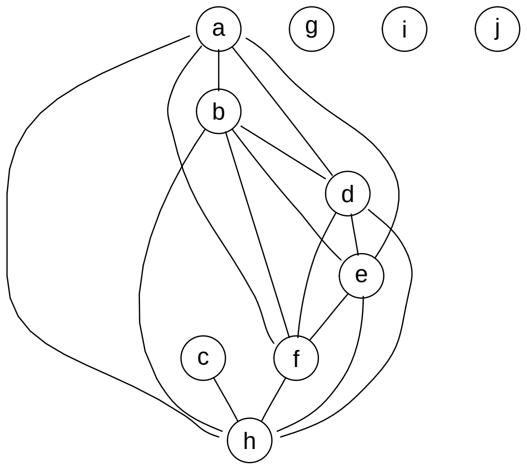
1 Overview

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)

Creating a new generic function for "lines" in ".GlobalEnv"
Creating a new generic function for "plot" in ".GlobalEnv"

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

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

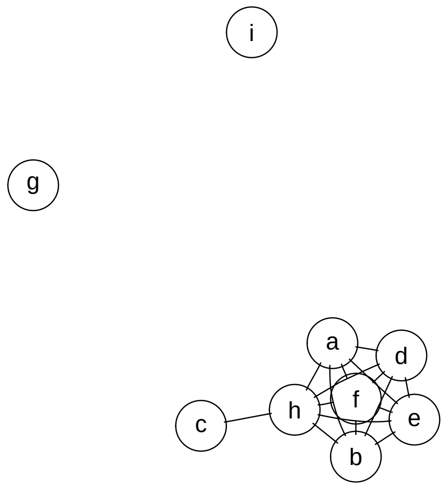
2 Plotting in R Using Different Layout Methods

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)
```

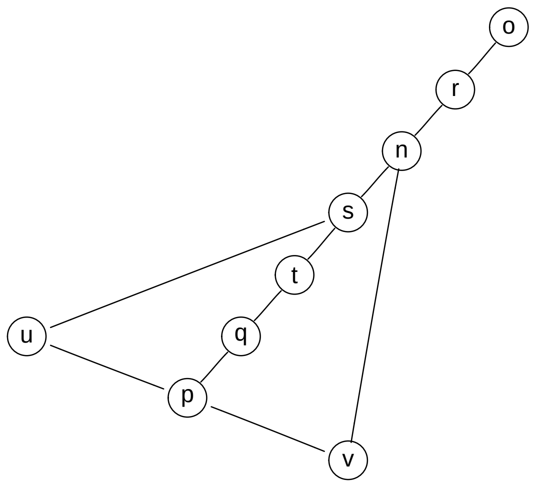
```
[1] FALSE
```

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

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

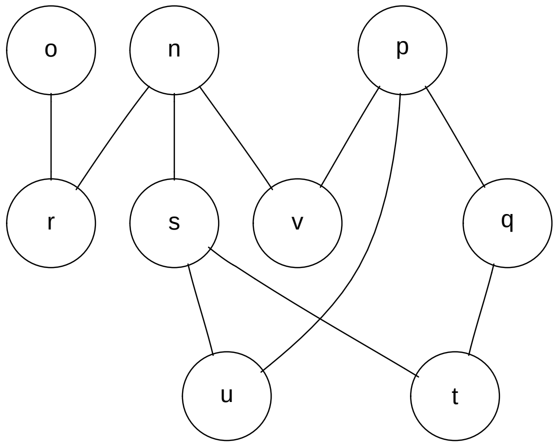
```
[1] TRUE
```

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

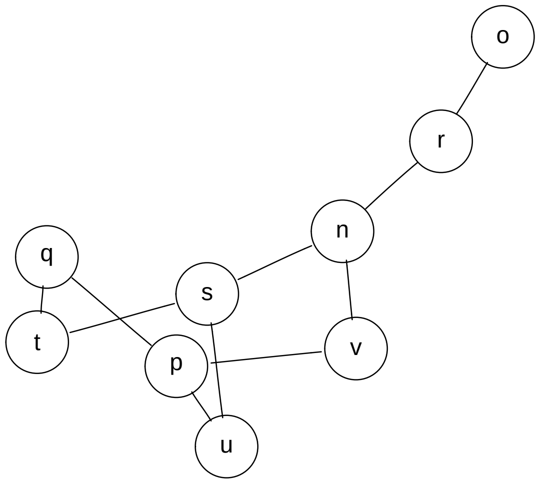


And finally, to demonstrate how the differing layout methods work on this second graph:

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

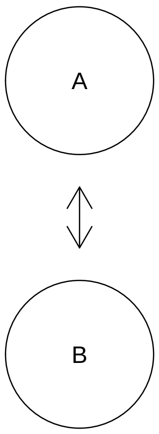


```
> z <- plot(g2, "neato")
```



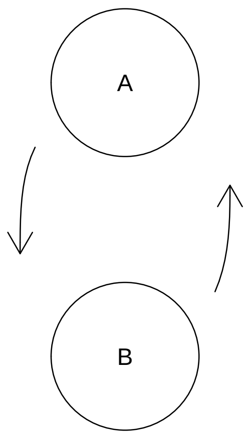
Note that there is an option, *recipEdges* that details how to deal with reciprocated edges in a graph. The two options are *combined* (the default) and *distinct*. This is mostly useful in directed graphs that have reciprocating edges - the *combined* option will display them as a single edge with an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



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

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

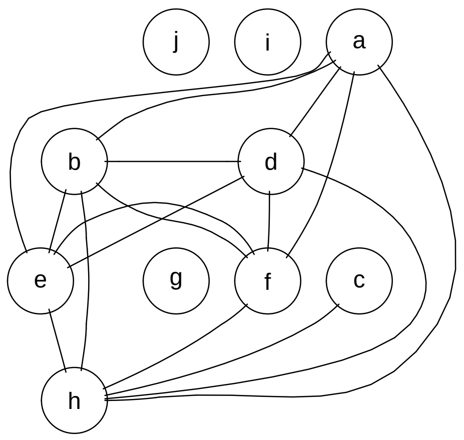
```
A graph with  undirected  edges
Number of Nodes = 3
Number of Edges = 3
```

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

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

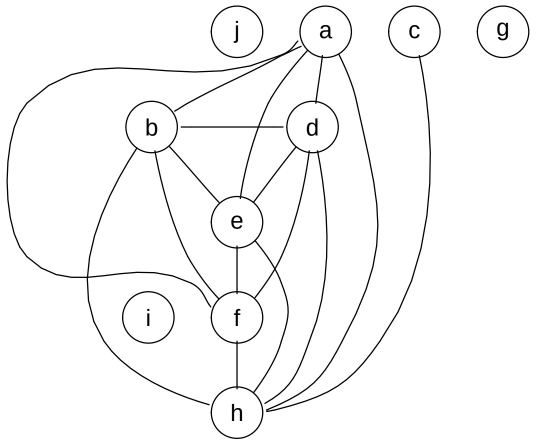
To plot using the subgraphs, one must use the `subGList` argument which accepts a list of every subgraph.

```
> plot(g1, subGList = list(sg1, sg2, sg3))
```



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(sg1, sg2))
```



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.research.att.com/ erg/graphviz/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 if nothing else is provided, using the function `getDefaultAttrs` - users are encouraged to work off of the return of this function instead of creating their own from scratch as then attributes which they don't want to change from the defaults will still be kept intact. The `getDefaultAttrs` function takes as a parameter the layout type to be used (dot, neato or twopi) but defaults to dot. The *attrs* list is a four element list with element names of '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

$graph$rankdir
[1] "TB"


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

$cluster$color
[1] "black"


$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"


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

$edge$dir
[1] "both"

$edge$weight
[1] 1

$edge$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"
```

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 one place to take care about is with the edge names in that the name of an edge is x y where x is the 'from' node and y is the 'to' node. Note that even with an undirected graph that x y is not the same thing as y x as it depends on how the edge was explicitly defined. These lists are then passed in to `plot` as the arguments *nodeAttrs* and *edgeAttrs*. 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`.

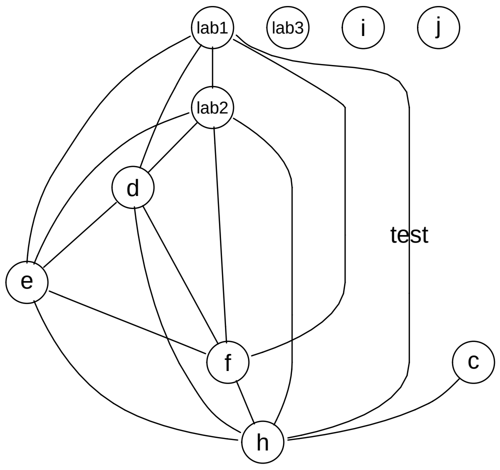
```
> nAttrs <- list()
> eAttrs <- list()
```

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")
> nAttrs

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

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

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

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

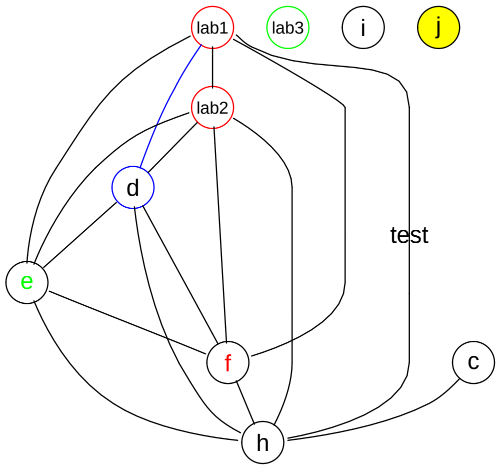


7 Adding Some Color

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

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

$color
      a       b       g       d
```

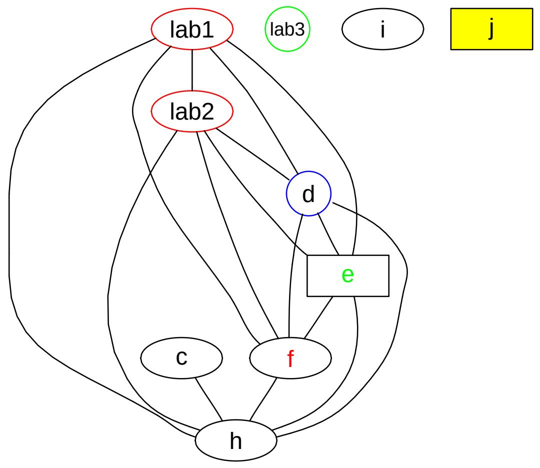
```
  "red"   "red" "green"  "blue"

$fillcolor
       j
"yellow"

$fontcolor
      e       f
"green"   "red"

> eAttrs

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

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

$fontcolor
    a~h     h~c
"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* and *box*. 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* and two as *circle*:

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


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

You can now see the contents of the first *pNode* and first *pEdge* objects in their respective lists. Now, to demonstrate some simple attribute examples such as node and edge color, perhaps we'd like to have the "c" node be blue and the between "a" and "b" to be colored green, we can set it up as such:

```
> nodes$c@attrs$fillcolor <- "blue"
> nodes$c
```

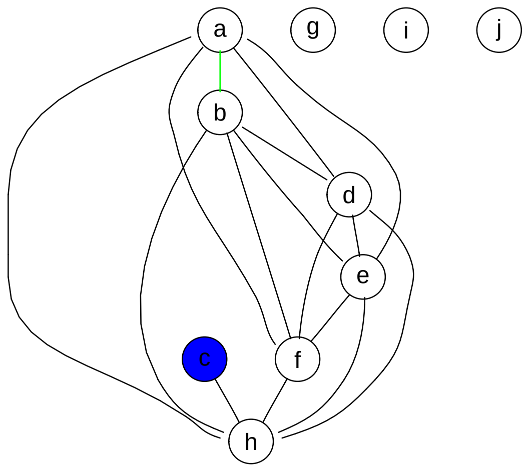
```
An object of class "pNode"
Slot "name":
[1] "c"

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

$fillcolor
[1] "blue"


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

```
> edges$"a~b"@attrs$color <- "green"
> edges$"a~b"
```

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

Slot "to":
[1] "b"

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

$color
[1] "green"


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

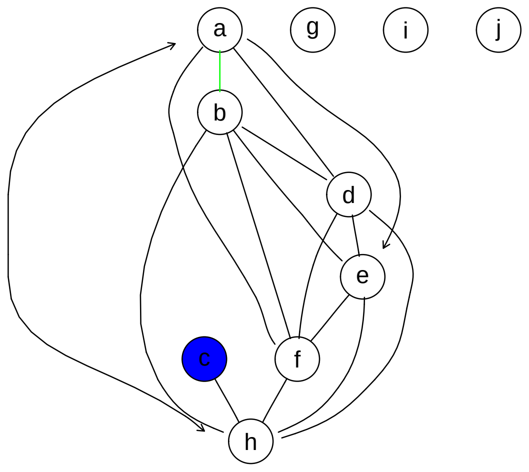
You can see that the *attrs* slot now has a 'color' element to it.

```
> vv <- agopen(name = "test1", nodes = nodes, edges = edges, edgeMode = "undirected")
> plot(vv)
```



Note that the name argument is required by Graphviz but is not currently used for anything in *Rgraphviz*. The edgeMode argument is required in this case because Graphviz needs to know if this graph is directed or not and *Rgraphviz* can't necessarily determine the status of the graph from the node and edge lists. One could suggest to look at the arrowhead/tails (or lack thereof) and derive it that way, but the arrowhead/tails are completely independent from the edgemode of the graph:

```
> edges$"a~e"@attrs$arrowhead <- "open"
> edges$"a~h"@attrs$arrowhead <- "open"
> edges$"a~h"@attrs$arrowtail <- "open"
> vv <- agopen(name = "test1", 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.

10 Plotting with non-standard nodes

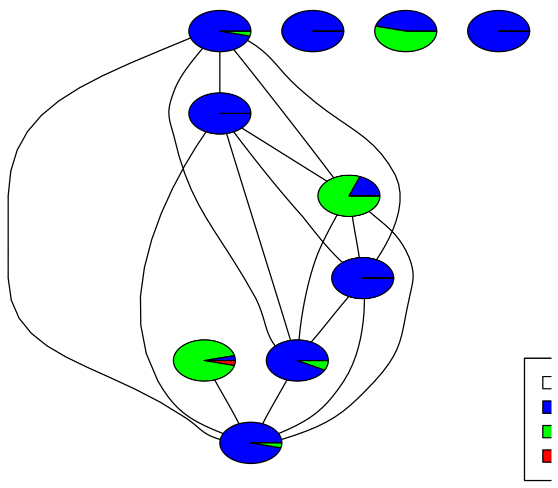
The *Rgraphviz* package provides for non-standard node drawing. Note that these nodes are shaped the same as standard nodes, but are able to provide for richer information in the actual display.

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 three 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. 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.

```
> require("Biobase") || stop("Biobase needed for this example")

Loading required package: Biobase
Welcome to Bioconductor
         Vignettes contain introductory material.  To view,
         simply type: openVignette()
         For details on reading vignettes, see
         the openVignette help page.
[1] TRUE

> 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()) {
+             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(310, 100, legend = c("No Data", "0-100", "101-500",
+         "500+"), fill = c("white", "blue", "green", "red"))
+ }
> plotPieChart(pgLayout, counts)
```

Example Pie Chart Plot



To perform 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.