How To Plot A Graph Using Rgraphviz

Jeff Gentry, Robert Gentleman, Wolfgang Huber

December 26, 2006

Contents

1 Overview **1**

2 Different layout methods **2**
2.1 Reciprocated edges 2

3 Subgraphs **4**
3.1 A note about edge names 5

4 Attributes **6**
4.1 Global attributes 6
4.2 Per node attributes 9
4.3 Node labels 9
4.4 Using edge weights for labels 10
4.5 Adding color 11
4.6 Node shapes 12

5 Layout, rendering and the function `agopen` **13**

6 Customized node plots **17**

7 Special types of graphs **20**
7.1 Cluster graphs 20
7.2 Bipartite graphs 20

8 Tooltips and hyperlinks on graphs **22**

1 Overview

This vignette demonstrate how to easily render a graph from R into various formats using the *Rgraphviz* package. To do this, let us generate a graph using the *graph* package:

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

> g1 <- randomGraph(V, M, 0.2)

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

2 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.

```
> plot(g1)
```

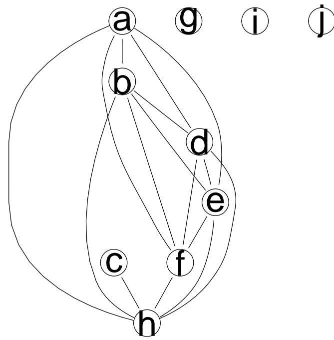

Figure 1: *g1* laid out with *dot*.

The result is shown in Figure 1. The *Rgraphviz* package allows you to specify varying layout engines, such as *dot* (the default), *neato* and *twopi*.

```
> plot(g1, "neato")

> plot(g1, "twopi")
```

The result is shown in Figure 2.

2.1 Reciprocated edges

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

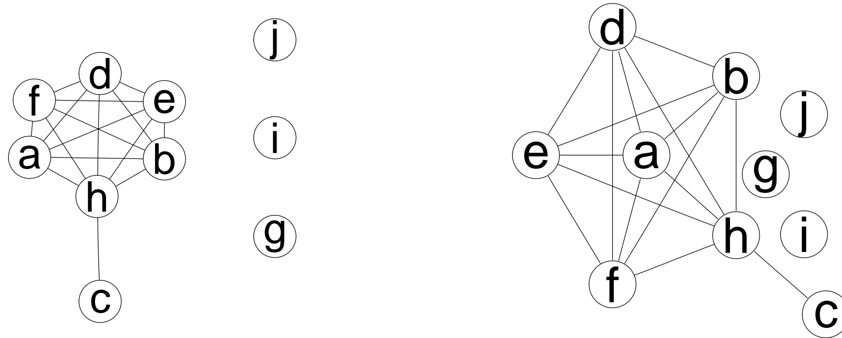

Figure 2: *g1* laid out with *neato* (left) and *twopi* (right).

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)
```

Figure 3: *rEG* laid out with *recipEdges* set to *combined* (left) and *distinct* (right).

```
> plot(rEG)
```

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

The result is shown in Figure 3.

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

```
> removedEdges(g1)
```

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

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 Section 4 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)
```

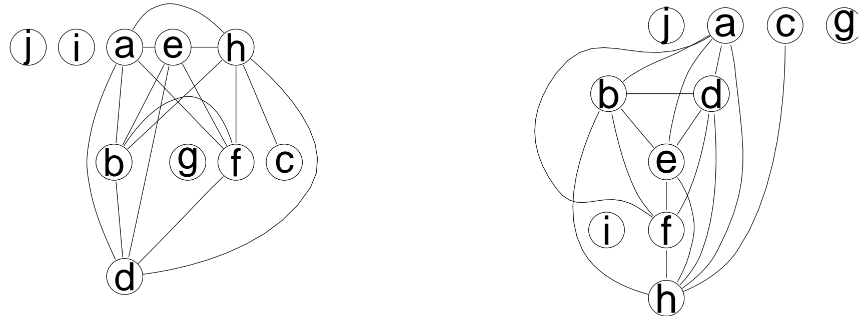

Figure 4: *g1* laid out with two different settings for the parameter *subGList*.

The result is shown in the left panel of Figure 4, and for comparison, another example:

```
> 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. The name of an edge between tail node `x` and head node `y` is `x~y`. The method `edgeNames` can be used to obtain a vector of all edge names, and it 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

4.1 Global attributes

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()

$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 (see Figure 5, we set 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")))
```

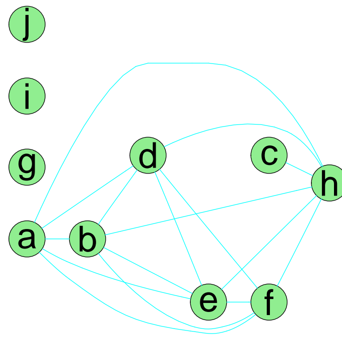

Figure 5: *g1* laid out with user-defined *attrs*.

4.2 Per node attributes

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 use two lists *nAttrs* and *eAttrs*.

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

4.3 Node 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.

```
> z <- strsplit(packageDescription("Rgraphviz")$Description, " ")[[1]]
> z <- z[1:numNodes(g1)]
> names(z) = nodes(g1)
> nAttrs$label <- z

> eAttrs$label <- c("a~h" = "Label 1", "c~h" = "Label 2")

> attrs <- list(node = list(shape = "ellipse", fixedsize = FALSE))

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

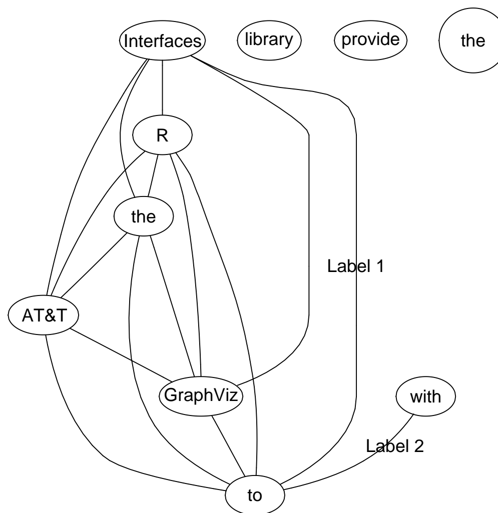

Figure 6: *g1* laid out with user-defined node and edge labels.

The result is shown in Figure 6.

4.4 Using edge weights for labels

A common desire for edge weights is to use the edge weights for the edge labels. 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.

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.

```
> ew <- as.character(unlist(edgeWeights(g1)))
> ew <- ew[setdiff(seq(along = ew), removedEdges(g1))]
> names(ew) <- edgeNames(g1)
> eAttrs$label <- ew
```

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

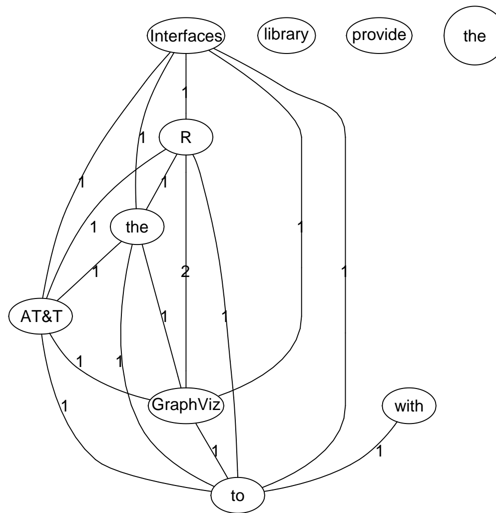

Figure 7: *g1* laid out with edge weights as edge labels.

The result is shown in Figure 7.

4.5 Adding 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            c            d            e
"Interfaces"          "R"       "with"        "the"       "AT&T"
           f            g            h            i            j
  "GraphViz"    "library"         "to"    "provide" "the\nability"

$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, attrs = attrs)
```

The result is shown in Figure 8.

4.6 Node shapes

The *Rgraphviz* package allows you to specify different shapes for your nodes. Currently, the supported shapes are *circle* (the default), *ellipse*, *plaintext* and *box*. *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. Figure 9 shows the graph of 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:

```
> attrs$node$shape <- "ellipse"
> nAttrs$shape <- c(g = "box", f = "circle", j = "box", a = "plaintext")
```

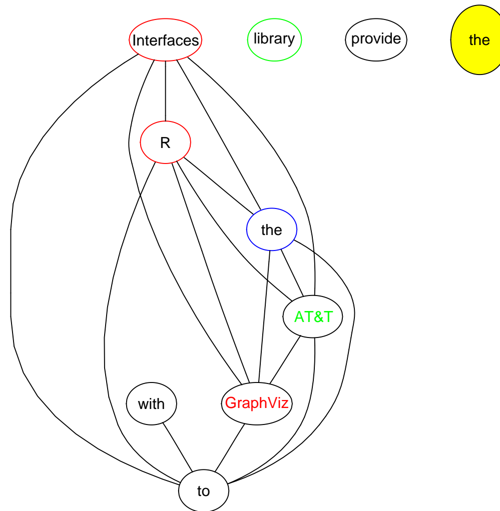

Figure 8: *g1* laid out with colors.

```
> plot(g1, attrs = attrs, nodeAttrs = nAttrs)
```

5 Layout, rendering and the function `agopen`

The calls to the `plot` that we have made above amount to two different processing steps, *layout* and *rendering*. In the layout step, Graphviz lays out the nodes and edges on a (virtual) 2D plotting surface. In the *rendering* step, a plot consisting of lines, shapes, and letters with particular line styles, colors, fonts, font size etc. is created.

By dissecting these steps and manually interfering, we can achieve finer control over the appearance of the rendered graph.

The functions `buildNodeList` and `buildEdgeList` generate a list of *pNode* and *pEdge* objects respectively. These are used to provide the information for the Graphviz layout, and by default they are generated automatically during the call to the *plot* function. 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)
```

You can now see the contents of the first *pNode* and first *pEdge* objects in their respective lists.

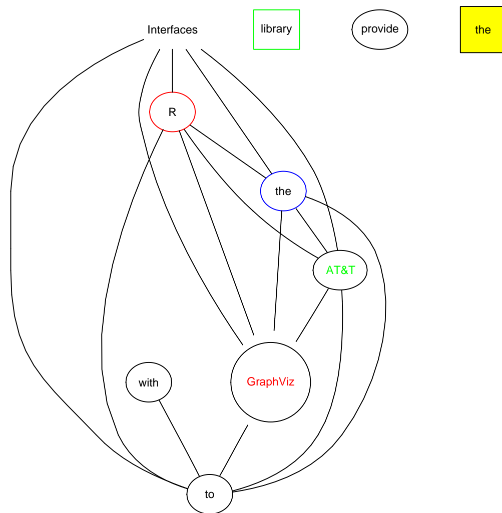

Figure 9: *g1* laid out with user defined node shapes.

```
> 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
```

The functions buildNodeList and buildEdgeList can also use the attribute lists constructed above.

```
> 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] "Interfaces"

$color
[1] "red"

$fillcolor
[1] "transparent"

$fontcolor
[1] "black"

$shape
[1] "plaintext"


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
```

Note the difference between the objects in the second example as compared with the first.

We can add arrowheads to the a e and a h edges

```
> for (j in c("a~e", "a~h")) edges[[j]]@attrs$arrowhead <- "open"
```

While visually indicating direction, these will have no bearing on the layout itself as Graphviz views these edges as undirected.

Now we can plot this graph (see Figure 10):

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

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

```
> data(graphExamples)
> z <- graphExamples[[8]]
> nNodes <- length(nodes(z))
> nA <- list()
> nA$fixedSize <- rep(FALSE, nNodes)
> nA$height <- nA$width <- rep("1", nNodes)
> nA$label <- rep("z", nNodes)
> nA$color <- rep("green", nNodes)
```

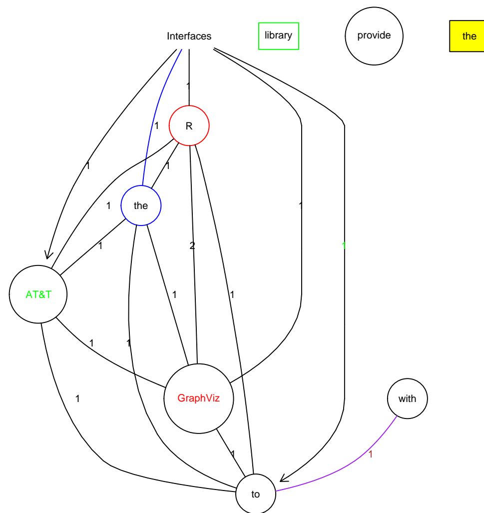

Figure 10: *g1* laid out via nodes and edge lists.

```
> 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)
```

6 Customized node plots

The *Rgraphviz* package provides for customized drawing of nodes. Customized nodes must have one of the standard node shapes, but are able to provide for richer information inside.

To do this, lay out the graph using the shape desired, then, when plotting the laid out graph, 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 that same function) 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:

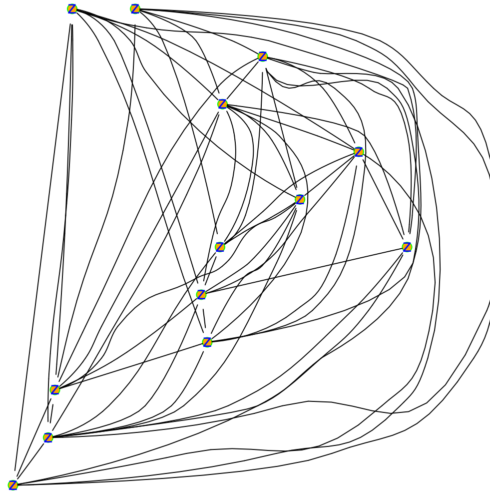

Figure 11: Customized layout of graph z.

node is an object of class *AgNode* describing the node's location and other information

ur is of class $XYPoint$ and describes the upper right hand point of the bounding box (the lower left is 0,0)

attrs is a node attribute list as discussed in Section 4. It can be used for post-layout attribute changes to override values that were used for the layout.

radConv is used by *Rgraphviz* to convert Graphviz units to R plotting units. This argument will probably not need to be used a custom drawing function, but does need to exist.

A custom drawing function is free to ignore these arguments, but the argument must exist in the function signature.

The default function for node drawing on all nodes is `drawAgNode`, and users who want to supply their own node drawing function are encouraged to inspect this function as a template.

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.

```
> set.seed(123)
> counts = matrix(rexp(numNodes(g1) * 4), ncol = 4)
> g1layout <- agopen(g1, name = "foo")
> makeNodeDrawFunction <- function(x) {
+     force(x)
+     function(node, ur, attrs, radConv) {
+         nc <- getNodeCenter(node)
+         pieGlyph(x, xpos = getX(nc), ypos = getY(nc), radius = getNodeRW(node),
+             col = rainbow(4))
+         text(getX(nc), getY(nc), paste(signif(sum(x), 2)), cex = 0.5,
+             col = "white", font = 2)
+     }
+ }
> drawFuns <- apply(counts, 1, makeNodeDrawFunction)

> plot(g1layout, drawNode = drawFuns, main = "Example Pie Chart Plot")
```

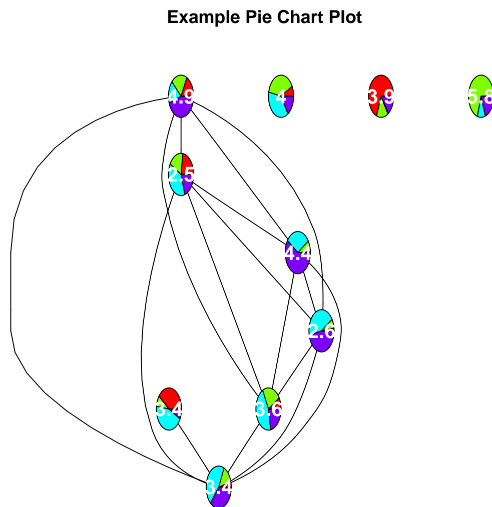

Figure 12: *g1* with pie charts as nodes.

The result is shown in Figure 12.

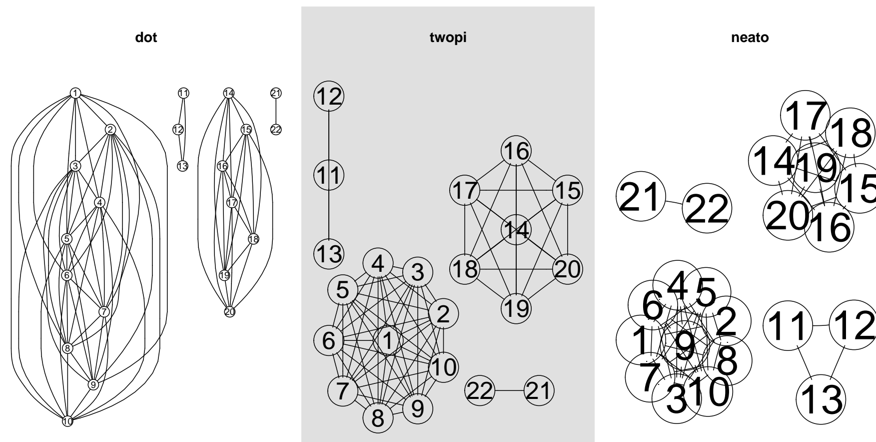

Figure 13: A cluster graph laid out using the three layout algorithms.

7 Special types of graphs

Up to this point, we have only been working with objects of class *graphNEL*, but the other subclasses of the virtual class *graph* (such as *distGraph* and *clusterGraph*) will work as well, provided that they support the `nodes` and `edgeL` methods.

In this section, we demonstrate a few examples. Users should not notice a difference in the interface, but this will provide some visual examples as to how these types of graphs will appear.

7.1 Cluster graphs

For our first set of examples, we 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)))
```

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

In Figure 13 we show *cG* laid out using three different algorithms.

7.2 Bipartite graphs

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.

```
> set.seed(123)
> nodes1 <- paste(0:7)
> nodes2 <- letters[1:10]
> ft <- cbind(sample(nodes1, 24, replace = TRUE), sample(nodes2,
+     24, replace = TRUE))
> ft <- ft[!duplicated(apply(ft, 1, paste, collapse = "")), ]
> g <- ftM2graphNEL(ft, edgemode = "directed")
> g
```

```
A graphNEL graph with directed edges
Number of Nodes = 17
Number of Edges = 23
```

Next we set up the node attributes and create subgraphs so that we can better control the layout. We want to have color for the nodes, and we want to lay the graph out from left to right, rather than vertically.

```
> twocolors <- c("#D9EF8B", "#E0F3F8")
> nodeType <- 1 + (nodes(g) %in% nodes1)
> nA = makeNodeAttrs(g, fillcolor = twocolors[nodeType])
> sg1 = subGraph(nodes1, g)
> sgL = list(list(graph = sg1, cluster = FALSE, attrs = c(rank = "sink")))
> att = list(graph = list(rankdir = "LR", rank = ""))
```

Finally, in Figure 14 we plot the bipartite graph.

```
> plot(g, attrs = att, nodeAttrs = nA, subGList = sgL)
```

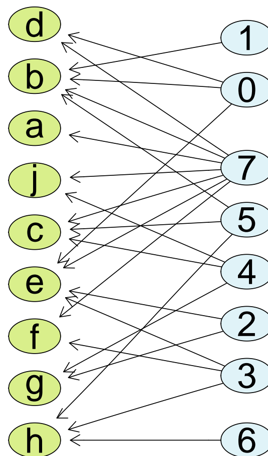

Figure 14: A bipartite graph.

8 Tooltips and hyperlinks on graphs

This section is for advanced users. It gives an example for how the `imageMap` function of the *Rgraphviz* package can be used to create clickable graph renderings with drill-down capability.

First, we load an example graph

```
> data("integrinMediatedCellAdhesion")
```

In the next code chunk, we create a set of plots, one for each node. Here they are just meaningless "dummy" plots, but of course in your application these could be showing real data, corresponding to the nodes.

```
> outdir = tempdir()
> nd = nodes(IMCAGraph)
> plotFiles = paste(seq(along = nd), "png", sep = ".")
> for (i in seq(along = nd)) {
+     png(file.path(outdir, plotFiles[i]), width = 400, height = 600)
+     plot(cumsum(rnorm(100)), type = "l", col = "blue", lwd = 2,
+         main = nd[i])
+     dev.off()
+ }
```

Now we create a HTML page that consists of two parts, so-called frames, one to hold the graph plot, one for the per-node data plots that we generated above.

```
> fhtml = file.path(outdir, "index.html")
> con = file(fhtml, open = "wt")
> cat("<HTML><HEAD><TITLE>", "Integrin Mediated Cell Adhesion graph with tooltips and hyperl
+     "</TITLE></HEAD>", "<FRAMESET COLS=\"3*,2*\" BORDER=0>",
+     "  <FRAME SRC=\"graph.html\">", "  <FRAME NAME=\"nodedata\">",
+     "</FRAMESET></HTML>", sep = "\n", file = con)
> close(con)
```

Finally, we create the graph plot, and the associated HTML image map.

```
> width = 600
> height = 512
> imgname = "graph.png"
> png(file.path(outdir, imgname), width = width, height = height)
> par(mai = rep(0, 4))
> lg = agopen(IMCAGraph, name = "foo", attrs = list(graph = list(rankdir = "LR",
+     rank = ""), node = list(fixedsize = FALSE)), nodeAttrs = makeNodeAttrs(IMCAGraph),
+     subGList = IMCAAttrs$subGList)
> plot(lg)
> con = file(file.path(outdir, "graph.html"), open = "wt")
> writeLines("<html><body>\n", con)
```

```
> imageMap(lg, con = con, tags = list(HREF = plotFiles, TITLE = nd,
+     TARGET = rep("nodedata", length(nd))), imgname = imgname,
+     width = width, height = height)
> writeLines("</body></html>", con)
> close(con)
> dev.off()
```

```
postscript
         2
```

We can have a look at the result by pointing a web browser to it.

```
> fhtml
```

```
[1] "/tmp/RtmpWFxYWw/index.html"
```

```
> if (interactive()) browseURL(fhtml)
```