How To Plot A Graph Using Rgraphviz

Jeff Gentry, Robert Gentleman, Wolfgang Huber

October 17, 2016

Contents

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)
```

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)
```

The result is shown in Figure **??**. 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 **??**.

2.1 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 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.

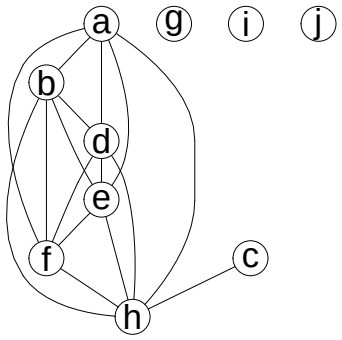

Figure 1: *g1* laid out with *dot*.

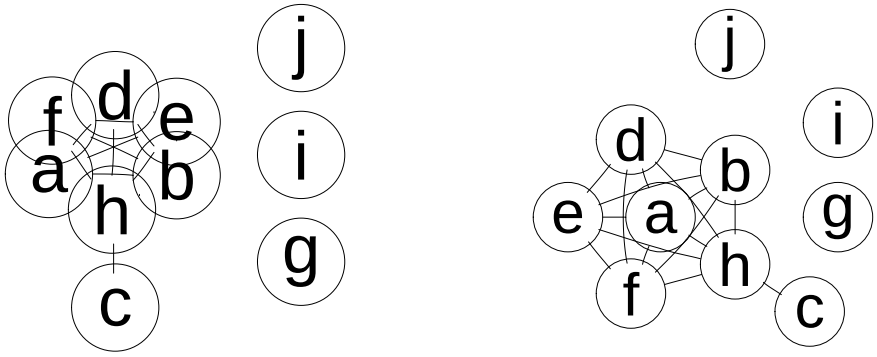

Figure 2: *g1* laid out with *neato* (left) and *twopi* (right).

```
> 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 **??**.

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

A user can request that a subset of the nodes in a graph be kept together. Graphviz then attempts to find a layout where the specified subgraphs are plotted with all nodes relatively close. This is particularly useful when laying out graphs that represent some real physical entity (one biological example is pathways).

In the code below we construct three subgraphs that we will use to group the corresponding nodes when *g1* is rendered.

```
> 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 **??** 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)
```

The result is shown in the left panel of Figure **??**, 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*.

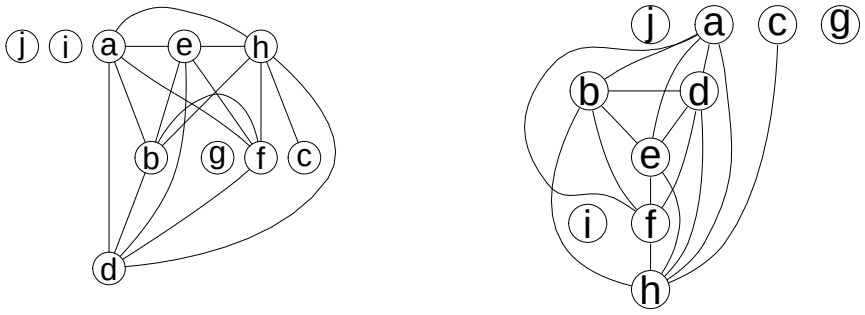

Figure 4: *g1* laid out with two different settings for the parameter *subGList*.

```
> 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] "6.99,6.99"

$graph$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] "\\N"

$node$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] "none"

$edge$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 **??**, 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")))
```

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

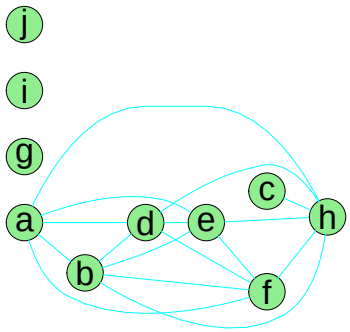

Figure 5: *g1* laid out with user-defined *attrs*.

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)
```

The result is shown in Figure **??**.

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

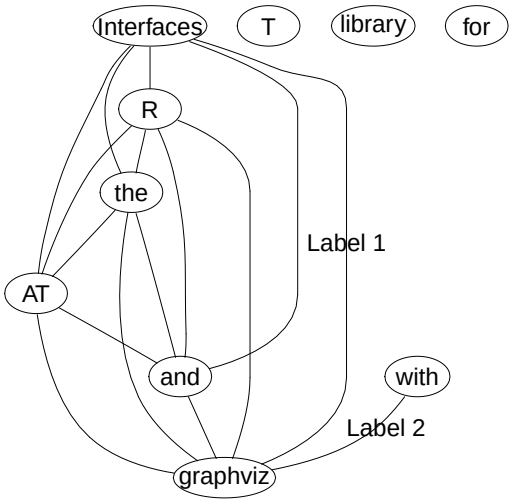

Figure 6: *g1* laid out with user-defined node and edge labels.

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$label <- ew
> attrs$edge$fontsize <- 27

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

The result is shown in Figure **??**.

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.

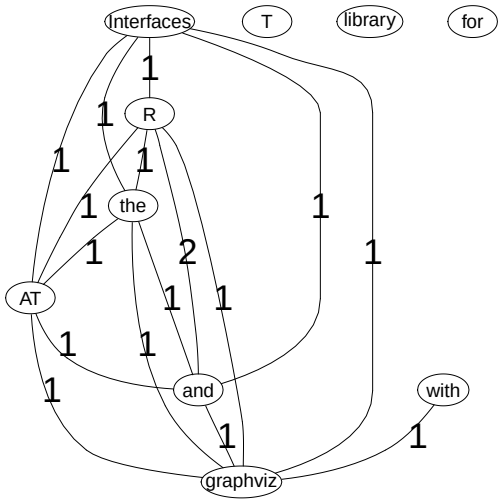

Figure 7: *g1* laid out with edge weights as edge labels.

```
> ## Specify node drawing color
> nAttrs$color <- c(a="red", b="red", g="green", d="blue")
> ## Specify edge drawing color
> eAttrs$color <- c("a~d"="blue", "c~h"="purple")
> ## Specify node fill color
> nAttrs$fillcolor <- c(j="yellow")
> ## label color
> nAttrs$fontcolor <- c(e="green", f="red")
> eAttrs$fontcolor <- c("a~h"="green", "a~b"="red")
> nAttrs

$label
           a            b            c            d            e            f
"Interfaces"          "R"       "with"        "the"         "AT"        "and"
           g            h            i            j
         "T"   "graphviz"    "library"      "for\n"

$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     a~b
"green"   "red"

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

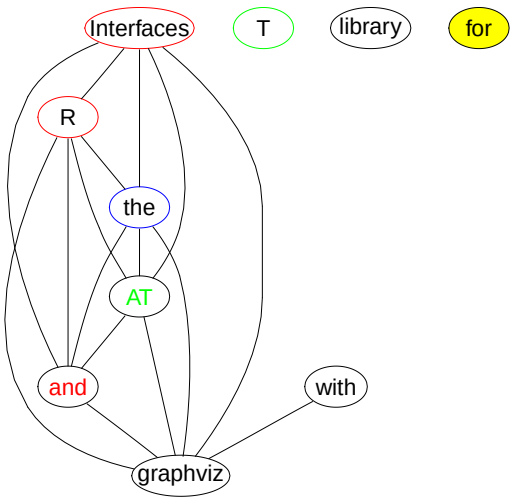

Figure 8: *g1* laid out with colors.

The result is shown in Figure **??**.

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 **??** 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")

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

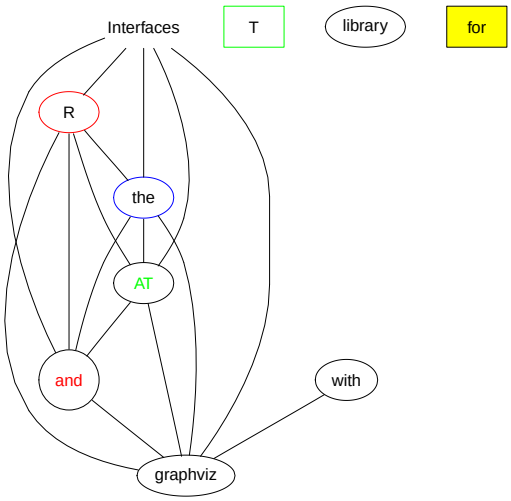

Figure 9: *g1* laid out with user defined node shapes.

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.

```
> 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"

$dir
[1] "none"


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"
```

```
$dir
[1] "none"

$label
[1] "1"

$color
[1] "black"

$fontcolor
[1] "red"


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 **??**):

```
> 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$color <- rep("green", nNodes)
> nA$fillcolor <- rep("orange", nNodes)
> nA$shape <- rep("circle", nNodes)
> nA$fontcolor <- rep("blue", nNodes)
> nA$fontsize <- rep(10, nNodes)
> nA <- lapply(nA, function(x) { names(x) <- nodes(z); x})
```

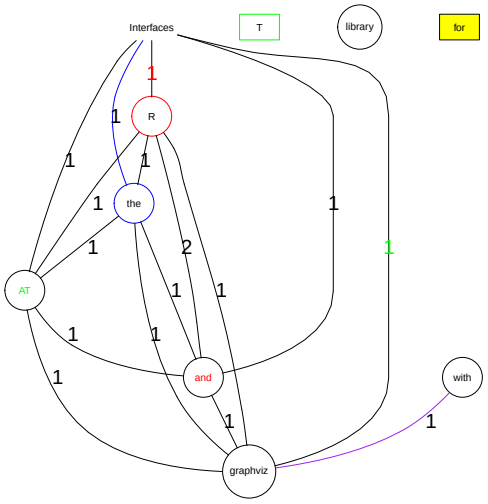

Figure 10: *g1* laid out via nodes and edge lists.

```
> 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:

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 **??**. It can be used for post-layout attribute changes to override values that were used for the layout.

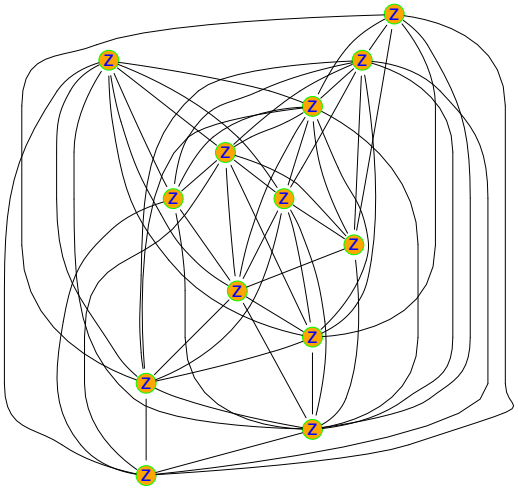

Figure 11: Customized layout of graph z.

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")
```

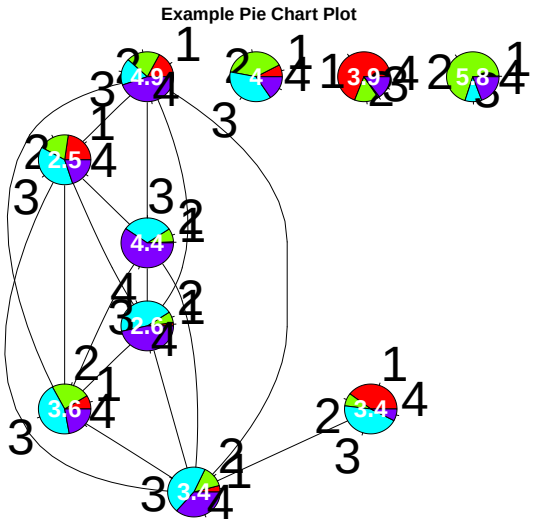

Figure 12: *g1* with pie charts as nodes.

The result is shown in Figure **??**.

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.

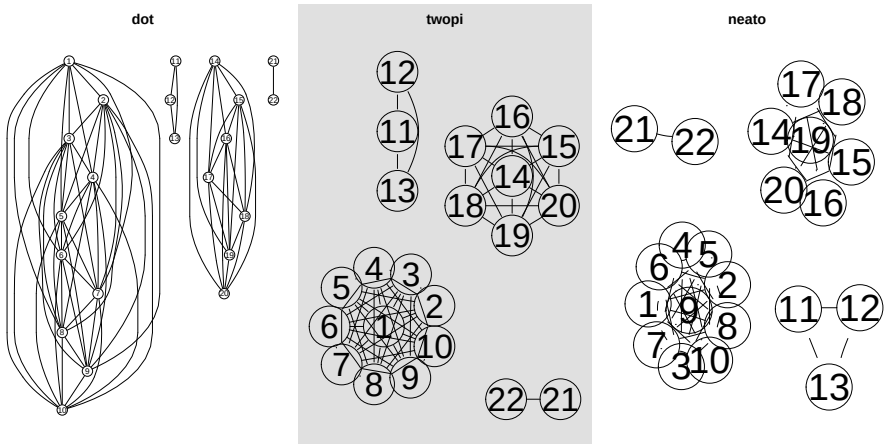

Figure 13: A cluster graph laid out using the three layout algorithms.

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 **??** 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 **??** we plot the bipartite graph.

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

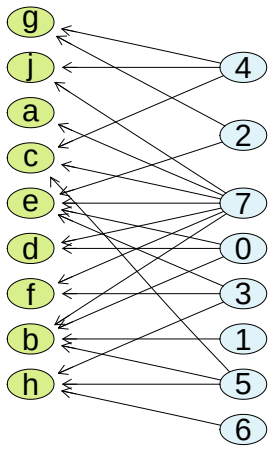

Figure 14: A bipartite graph.

8 NEW Another workflow to plot a graph

Another workflow in doing layout, rendering is as following: (1) convert a graph in *graph* class to *Ragraph* class, (2) set its default attributes for graph, cluster(s), nodes and edges, (3) set attributes for individual cluster, node(s) and/or edge(s), (4) layout it with desired algorithm, (5) render it to desired media.

Repeat steps (3), (4) and/or (5) as needed.

8.1 Convert a graph to *Ragraph* class

We prepare a graph with two subgraphs: graphExample-01.gxl.gz and graphExample-11.gxl.gz from package *graph*. We specify the 1st one is NOT a cluster, while the 2nd one is.

Just as in graphviz, *Rgraphviz* treats a cluster as a special subgraph, its nodes are layed out and drawn together and within a bounding rectangel.

```
> library(graph)
> library(XML)
> g1_gz <- gzfile(system.file("GXL/graphExample-01.gxl.gz",package="graph"))
> g11_gz <- gzfile(system.file("GXL/graphExample-11.gxl.gz",package="graph"))
> g1 <- fromGXL(g1_gz)
> g11 <- fromGXL(g11_gz)
> g1_11 <- join(g1, g11)
> sgl <- vector(mode="list", length=2)
> sgl[[1]] <- list(graph=g1, cluster=FALSE)
> sgl[[2]] <- list(graph=g11, cluster=TRUE)
> ng <- agopenSimple(g1_11, "tmpsg", subGList=sgl)
> close(g1_gz)
> close(g11_gz)
```

Note: this instance of *Ragraph* class from `agopenSimple` contains less content than that you obtain from calling *agopen*: it maintains a pointer for access to graphviz; after setting various attributes and doing layout, other entries are filled in for drawing only.

The following workflow works well for an instance of *Ragraph* class obtained from `agopen` as well.

8.2 Get default attributes

There are default attributes associated with a graph, a cluster (a subgraph), nodes and edges. You can find out what they are as follows:

After you explicitly set one or more default attributes, you'll see the default attributes.

If you like the notation from package *graph* better, the following codes accomplish the same:

```
> graphDataDefaults(ng)

     [,1] [,2]

> nodeDataDefaults(ng)

            attr name attr value
node attr 1 "label"   "\\N"

> edgeDataDefaults(ng)
```

```
            attr name  attr value
edge attr 1 "key"      ""
edge attr 2 "tailport" ""
edge attr 3 "headport" ""
```

8.3 Set default attributes

If you want to set default attributes yourself, you could call functions like following:

```
> graphDataDefaults(ng, c("size", "bgcolor")) <- c("1", "yellow")
> nodeDataDefaults(ng, c("fontcolor", "width")) <- c("blue", 0.5)
> edgeDataDefaults(ng, c("color", "style")) <- c("green", "dotted")
```

As shown in the examples, you can specify multiple attributes and their corresponding values in one call. R's circular rule applies.

Currently, only the 1st call to set default attributes has effects. Subsequent calls yield no effect.

8.4 Get attributes

You could set attributes for individual elements: graph, cluster(s), node(s) and edge(s). As R's circular rule applies, you could get multiple attributes to multiple elements in one call.

The package *graph* like notations are:

```
> graphData(ng, "bgcolor")

 bgcolor
"yellow"

> nodeData(ng, "a", c("fontcolor", "width"))

     a      a
"blue"  "0.5"

> edgeData(ng, "f", "h", c("color", "arrowhead"))

    f~h     f~h
"green"      NA

>
```

The return value, "default graph, node, cluster, node attr val 1", indicates that the attribute is NOT defined.

The return value, "default graph, node, cluster, node attr val 2", indicates that the attribute is NOT set for this object, the software will use default instead.

8.5 Set attributes

Likewise, you can set attributes for each element: graph, cluster(s), node(s) and edge(s). R's circular rule applies to element(s), attribute name(s) and attribute value(s).

A note on default value(s): default value could be set once and only once at this time. Only the first call has effect.

```
> graphData(ng, "bgcolor") <- "orange"
> clusterData(ng, 2, "bgcolor") <- "red"
> nodeData(ng, "a", c("fontcolor", "width")) <- c("red", "0.8")
> edgeData(ng, "f", "h", c("color", "style")) <- c("blue", "solid")
```

You can set as many attributes as you like.

Not every rendering software honors all the attributes, even for those provided by graphviz. Different attributes could have different impact in layout: some affect layout and drawing, such as node shapes, font size; others affect only drawing, such as fill color, font color.

Some attribute seem only take effect for certain layout, for instance, *bgcolor* for cluster shows the effect when layout is "dot", but not for "circo".

8.6 Layout and render the graph in various formats

To do the layout and render the results, there are two main output channels: (1) interactive output, and (2) file output which you could use various viewers to view the results.

To plot a graph interactively, simply do:

```
> plot(ng, "neato")
> plot(ng, "circo")
```

Currently, interactive output only honors a small number of attributes, mainly those from *buildNodelist* and *buildEdgeList*.

To do layout and then output to a file, you can do:

```
> # toFile(ng, layoutType="dot", filename="test_dot.svg", fileType="svg")
> # toFile(ng, layoutType="circo", filename="test_circo.ps", fileType="ps")
> toFile(ng, layoutType="dot", filename="test_twopi.dot", fileType="dot")

NULL
```

These examples use various renderer provided by graphviz, which honor a lot more attributes. You can look into graphviz doc to find out more, for instance, node shapes, color schemes, line types/sizes, etc.

You need corresponding viewers installed to be able to view the output files.

9 Tooltips and hyperlinks on graphs

Users that want to create a clickable graph renderings with drill-down capability should see the `imageMap` function in the *biocGraph* package.

10 Sessioninfo

- R version 3.3.1 (2016-06-21), `x86_64-apple-darwin13.4.0`
- Locale: `C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, grDevices, graphics, grid, methods, parallel, stats, utils
- Other packages: BiocGenerics 0.20.0, Rgraphviz 2.18.0, XML 3.98-1.4, graph 1.52.0
- Loaded via a namespace (and not attached): stats4 3.3.1, tools 3.3.1