A New Interface to Render Graphs Using Rgraphviz

Florian Hahne

October 24, 2023

Contents

1 Overview

This vignette shows how to use Rgraphviz's updated interface for rendering of graphs. For details on graph layout see the Vignette "How To Plot A Graph Using Rgraphviz". Note that the design of the interface is independent of graphviz, however no bindings to any other graph layout software have been implemented so far.

2 Introduction

There are two distinct processes when plotting graphs: *layout*, which places nodes and edges in a (usually two-dimensional) space, and *rendering*, which is the actual drawing of the graph on a graphics device. The first process

is typically the more computationally expensive one and relies on sophisticated algorithms that arrange the graph's components based on different criteria. The arrangement of the nodes and edges depends on various parameters such as the desired node size, which again may be a function of the size of the node labels. Rendering of a graph is often subject to frequent changes and adaptions, and it makes sense to separate the two processes in the software implementation. It is also important to realize that the process of getting a good layout is iterative, and using default parameter settings seldom yields good plots.

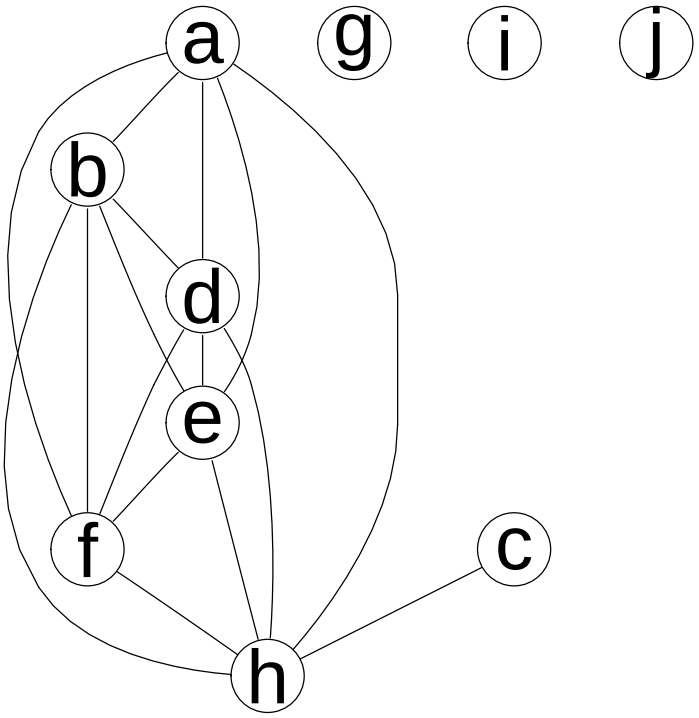
The code available for doing graph layout in Bioconductor is based mainly on the *Graphviz* project and the *Boost graph library*. However, because the rendering of a graph is separated from the layout, one can use other graph layout algorithms, as long as the requirements of the rendering interface are met.

In the process of laying out a graph some amount of information is generated, mostly regarding the locations and dimensions of nodes on a two-dimensional plane and the trajectories of the edges. Bioconductor *graph* objects now contain a slot `renderInfo` to hold this information. The typical workflow of a graph layout is to pass a graph object to the layout function, which returns another graph object containing all the necessary information for subsequent rendering. The process of calling a layout algorithm is encapsulated in the `layoutGraph` function. Calling this function without any further arguments will result in using one of the *Graphviz* layout algorithms via the the *Rgraphviz* package. We assume a knowledge of graph layout and the available *Graphviz* options in the remainder of this Vignette and will mostly deal with the rendering part, here.

The rendering of a graph relies solely on *R*'s internal plotting capabilities. As for all other plotting functions in *R*, many parameters controlling the graphical output can be tuned. However, because there are several parts of a graph one might want to modify (e.g., nodes, edges, captions), setting the graphical parameters is slightly more complex than for other plots. We have established a hierarchy to set global defaults, graph-specific parameters, and settings that apply only to individual rendering operations.

To demonstrate the new rendering interface we first generate a graph using the *graph* package and lay it out using the default *Graphviz* dot layout.

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



3 Default rendering parameters

There is a hierarchy to set rendering parameters for a graph. The levels of this hierarchy are

1. The session: These are the defaults that will be used for a parameter if not set somewhere further down the hierarchy. You can change the session defaults at any time using the function `graph.par`.

2. Rendering operation: Defaults can be set for a single rendering operation, that is, for a single call to `renderGraph` using its `graph.pars` argument.

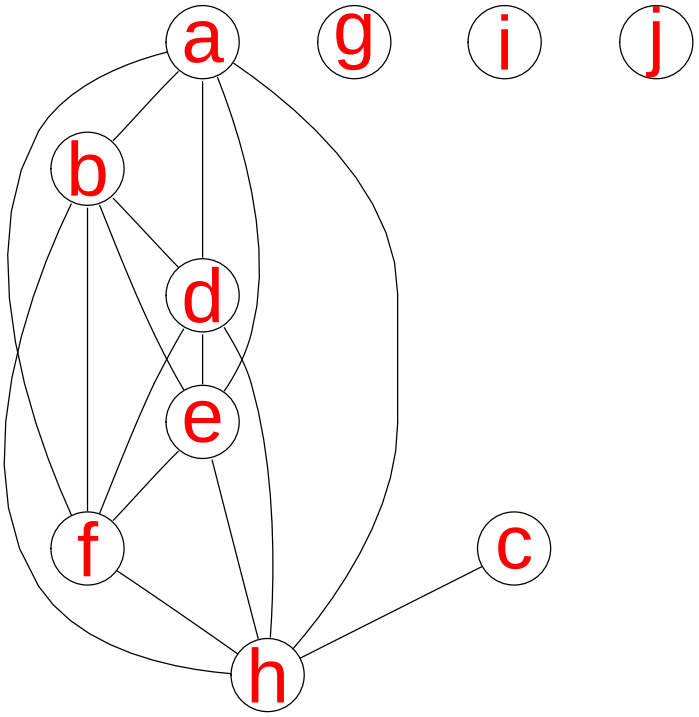
3. Individual nodes or edges: Parameters for individual nodes or edges in a graph object can be set using the `nodeRenderInfo` and `edgeRenderInfo` functions.

Note that all parameters set in `renderGraph's graph.pars` argument are transient, whereas setting session-wide parameters will affect all subsequent rendering operations. Setting parameters via the `nodeRenderInfo` and `edgeRenderInfo` functions affects the individual graph objects, and these changes will obviously be retained in all subsequent layout or rendering operations of that particular graph.

3.1 Default node parameters

We now use our example graph to further explore these options. Let's start with the nodes: We want to fill all our nodes with a gray color and use a red color for the node names. Since this should be applied to all nodes, we set a global rendering parameter using `graph.par`:

```
> graph.par(list(nodes=list(fill="lightgray", textCol="red")))
> renderGraph(g1)
```



Setting a session-wide parameter has side-effects for all subsequent rendering operations, and we will soon see how to archive the same setting using the `nodeRenderInfo` function.

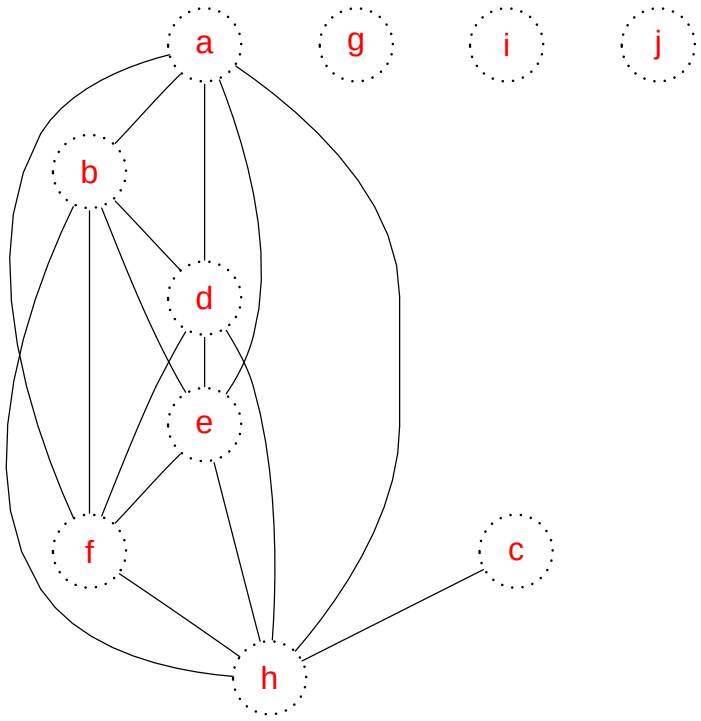
Note that `graph.par` takes as single argument a list of rendering parameters. There are three different types of parameters the user might want to set: nodewide parameters, edgewide parameters and parameters that control features of the whole graph. Accordingly, the parameters list passed to `graph.par` may contain the list items `nodes`, `edges` and `graph`. Each of these list items can again be a list of available plotting parameters. In our example, the parameters are `fill` and `textCol`. All currently available node parameters are:

- col: the color of the line drawn as node border. Defaults to `black`.
- lty: the type of the line drawn as node border. Defaults to `solid`. Valid values are the same as for the R's base graphic parameter `lty`.

- lwd: the width of the line drawn as node border. Defaults to 1.Note that the underlying low level plotting functions do not support vectorized `lwd` values. Instead, only the first item of the vector will be used.
- fill: the color used to fill a node. Defaults to `transparent`.
- textCol:the font color used for the node labels. Defaults to `black`.
- fontsize: the font size for the node labels in points. Defaults to `14`. Note that the fontsize will be automatically adjusted to make sure that all labels fit their respective nodes. You may want to increase the node size by supplying the appropriate layout parameters to *Graphviz* in order to allow for larger fontsizes.
- cex: Expansion factor to further control the fontsize. As default, this parameter is not set, in which case the fontsize will be clipped to the node size. This mainly exists to for consistency with the base graphic parameters and to override the clipping of fontsize to nodesize.
- shape: This is not really a graphical parameter. See Section 5 for details.

In the next code chunk we set the defaults for all remaining node parameters:

```
> graph.par(list(nodes=list(col="darkgreen", lty="dotted", lwd=2, fontsize=6)))
> renderGraph(g1)
```



Similar to *R*'s base `par` function, the original values of a modified paramter are returned by `graph.par` and you may want to assign them to an object in order to later revert your changes.

A useful feature when plotting graphs is to control the shape of the nodes. However, shapes have an impact on the ideal location of nodes and, even more so, the edges between them. Hence, the layout algorithm needs to be re-run whenever there are changes in node shapes. In Section 5 we will learn how to control node shapes and also some features of the edges.

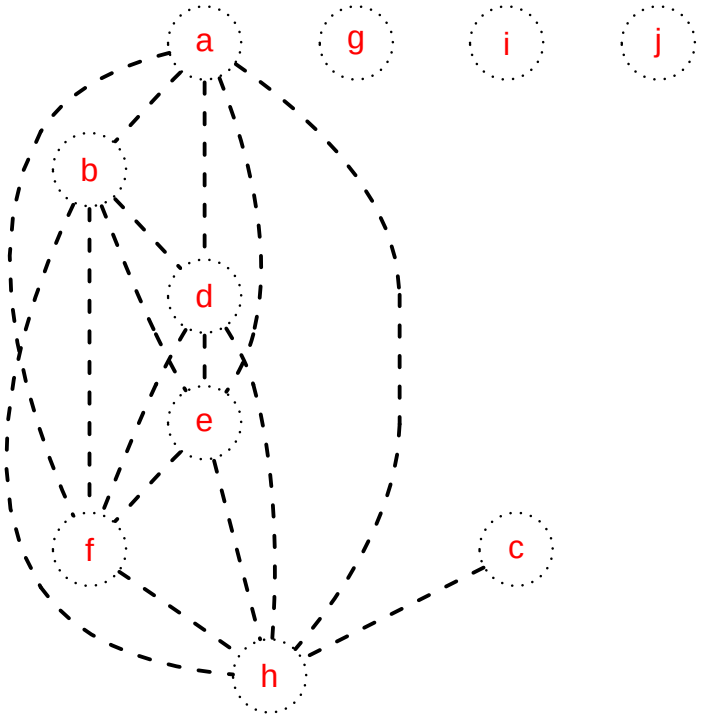
3.2 Default edge parameters

Now, let's take a look at the parameters that control the appearence of edges. They are:

- col: the color of the edge line. Defaults to `black`.
- lty: the type of the edge line. Defaults to `solid`. Valid values are the same as for the R's base graphic parameter `lty`.
- lwd: the width of the edge line. Defaults to `1`.
- textCol:the font color used for the edge labels. Defaults to `black`.
- fontsize: the font size for the edge labels in points. Defaults to `14`.
- cex: Expansion factor to further control the fontsize. This mainly exists to be consistent with the base graphic parameters.
- arrowhead, arrowtail Again, not really a plotting parameter. Section 5 provides details.

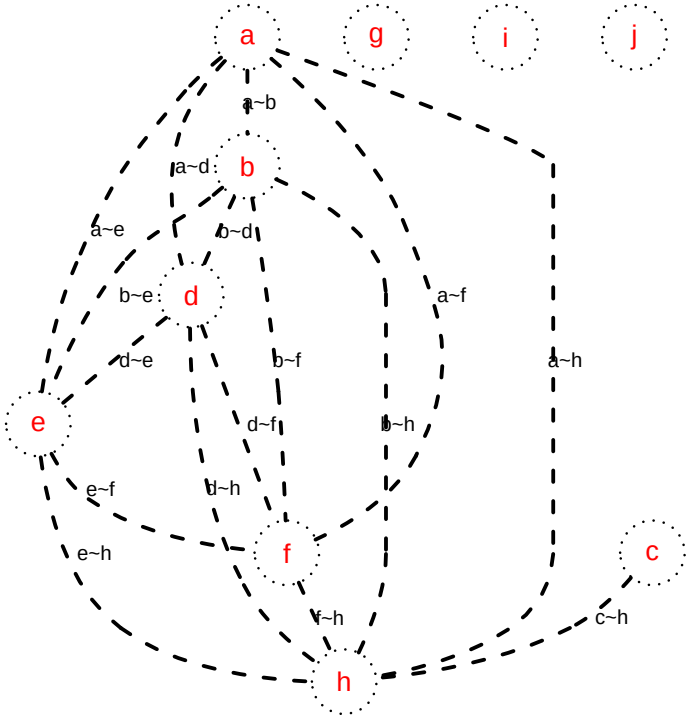
First, we set some attributes that control the edge lines.

```
> graph.par(list(edges=list(col="lightblue", lty="dashed", lwd=3)))
> renderGraph(g1)
```



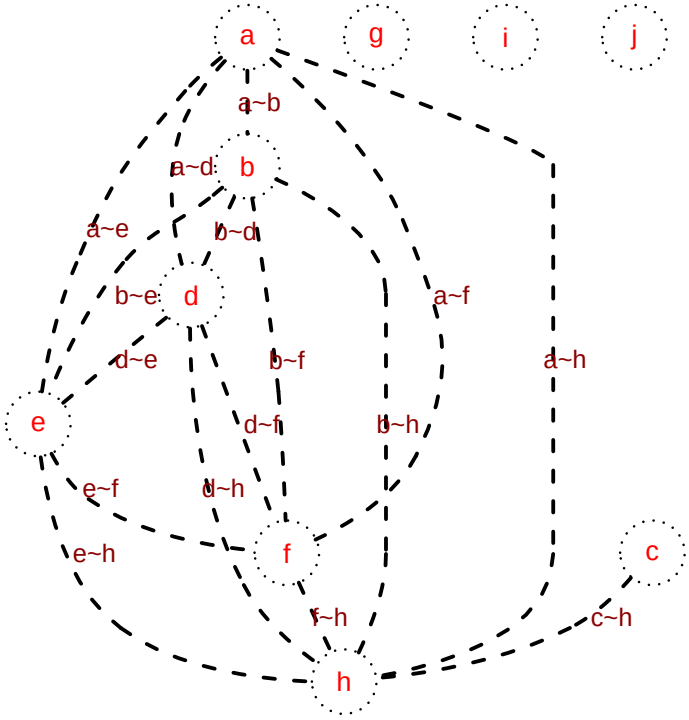
In order to show the effects of the edge label parameters, we first have to add such labels. `layoutGraph` will pass them on to *Graphviz* when they are specified as `edgeAttrs`:

```
> labels <- edgeNames(g1)
> names(labels) <- labels
> g1 <- layoutGraph(g1, edgeAttrs=list(label=labels))
> renderGraph(g1)
```



Now we can start tweaking them:

```
> graph.par(list(edges=list(fontsize=18, textCol="darkred")))
> renderGraph(g1)
```



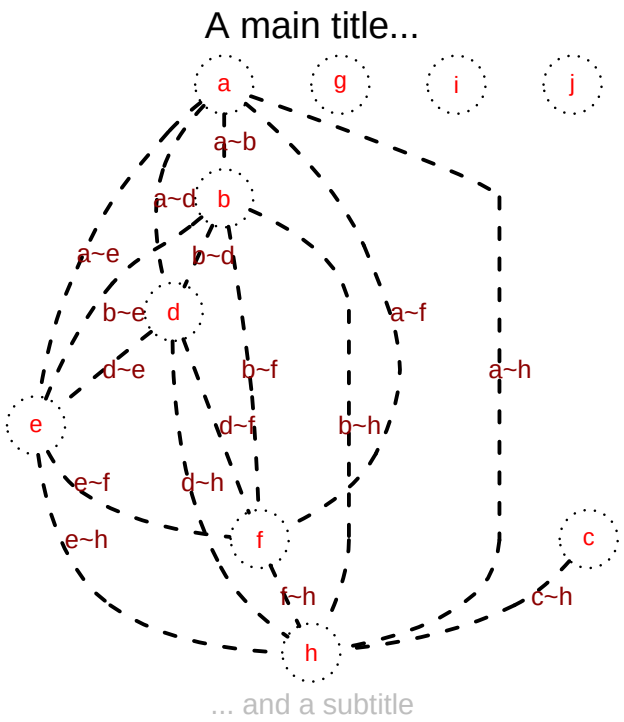
3.3 Default graphwide parameters

Some features of a graph are not really attributes of either nodes or edges. They can be controlled through the graphwide rendering parameters:

- main: text that is plotted as the main title. Unless set explicitely, no title will be plotted.
- sub: text that is plotted as subtitle at the bottom of the graph. Unless set explicitely, no subtitle will be plotted.
- col.main: the font color used for the title. Defaults to `black`.
- cex.main: Expansion factor for the fontsize used for the title. Defaults to `1.2`
- col.sub: the font color used for the subtitle. Defaults to `black`.
- cex.sub: Expansion factor for the fontsize used for the subtitle. Defaults to `1`

Here, we add both a title and a subtitle to the plot.

```
> graph.par(list(graph=list(main="A main title...",
+                 sub="... and a subtitle", cex.main=1.8,
+                 cex.sub=1.4, col.sub="gray")))
> renderGraph(g1)
```



Of course we could set all graph-, node-, and edgewide parameters in one single call to `graph.par`. Instead of defining global settings with `graph.par` we could also provide a list with the same structure to `renderGraph` through its `graph.pars` argument. Those will only be applied in the respective rendering operation, whereas options set using the function `graph.par` are retained throughout the whole *R* session.

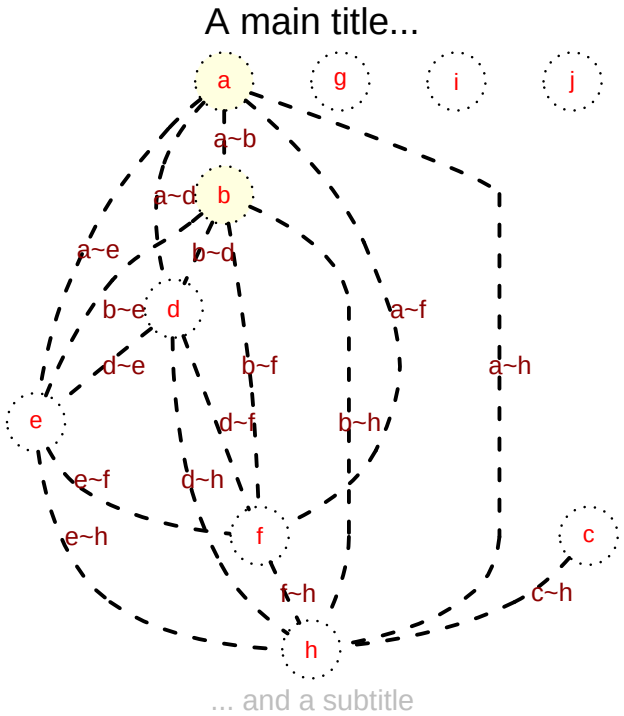
4 Parameters for individual nodes/edges

In many cases we don't want to globally change certain parameters for all nodes or edges, but rather do this selectively to highlight individual nodes/edges or subsets thereof. To this end, parameters for individual nodes and edges can be set using the `nodeRenderInfo` and `edgeRenderInfo` functions. Both `nodeRenderInfo` and `edgeRenderInfo` are replacement functions that operate directly on the *graph* object. For completeness, `graphRenderInfo` is the function that can be used to control the graph-wide attributes (like captions and subtitles). When you change a parameter in the *graph* object this will be car-

ried on across all further rendering and layout operations. The settings made by `edgeRenderInfo` and `nodeRenderInfo` take precedence over all other default settings.

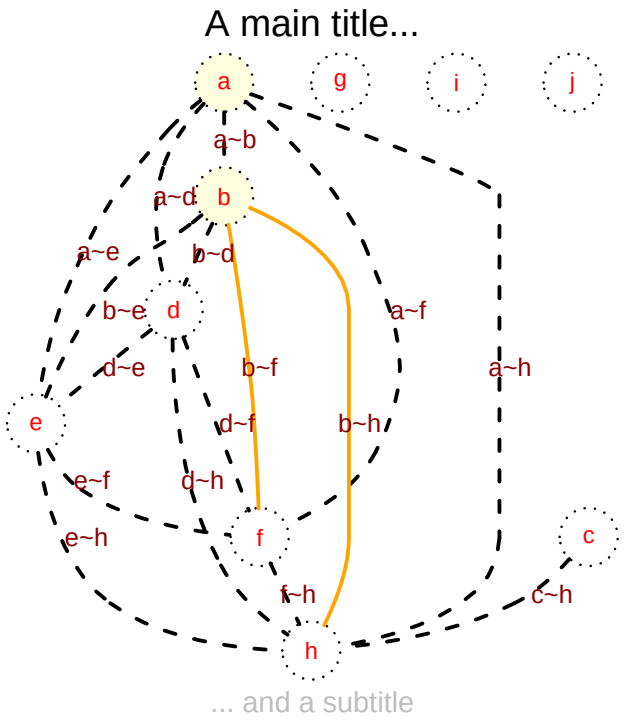
The parameters to be set have to be given as named lists, where each list item can contain named vectors for certain options. For example, the following code sets the fill color of nodes `a` and `b` to `yellow`.

```
> nodeRenderInfo(g1) <- list(fill=c(a="lightyellow", b="lightyellow"))
> renderGraph(g1)
```



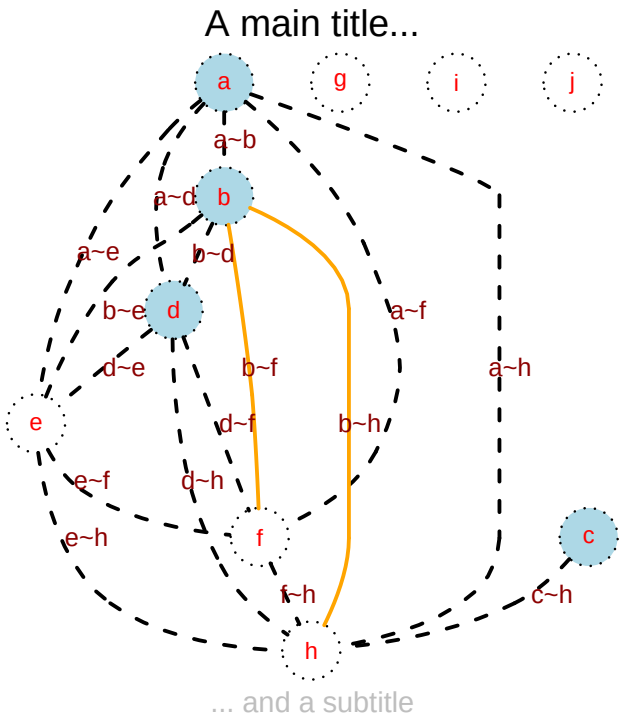
The names of the vectors have to match the node or edge names of the graph. Node names are straightforward (the result of calling the function `nodes` on a *graph* object), however edge names are made up of the names of the connected nodes separated by `~`, the tilde symbol. An edge between nodes `a` and `b` would be named `a~b`. For a directed graph `a~b` is the edge fom `a` to `b`, and `b~a` is the edge from `b` to `a`. For undirected graphs the two are equivalent. `edgeNames` returns the names of all edges in a graph. The following code changes the line type of the edges between nodes `b` and `f` and nodes `b` and `h` to `solid` and their line color to `orange`.

```
> edgeRenderInfo(g1) <- list(lty=c("b~f"="solid", "b~h"="solid"),
+                            col=c("b~f"="orange", "b~h"="orange"))
> renderGraph(g1)
```



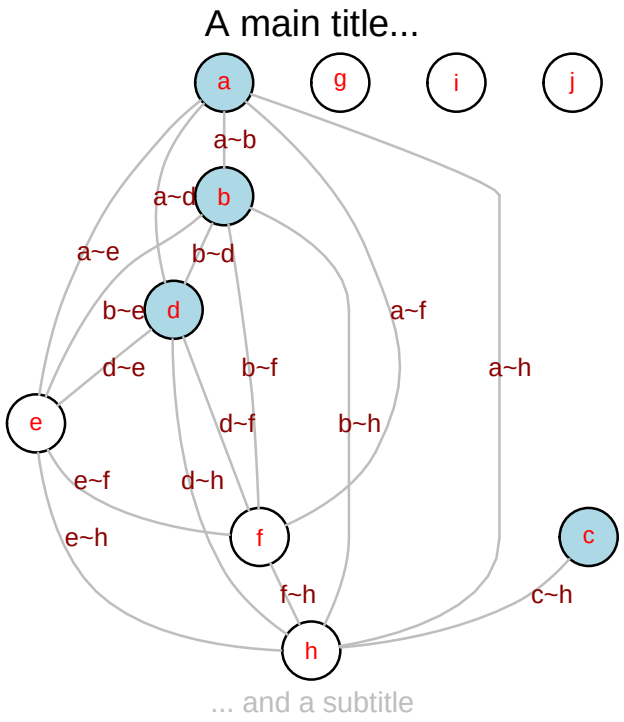
Changes in the rendering of specific nodes or edges is often motivated by certain classes or features they represent and we don't want to set this manually but rather use a programmatic approach:

```
> baseNodes <- letters[1:4]
> fill <- rep("lightblue", length(baseNodes))
> names(fill) <- baseNodes
> nodeRenderInfo(g1) <- list(fill=fill)
> renderGraph(g1)
```



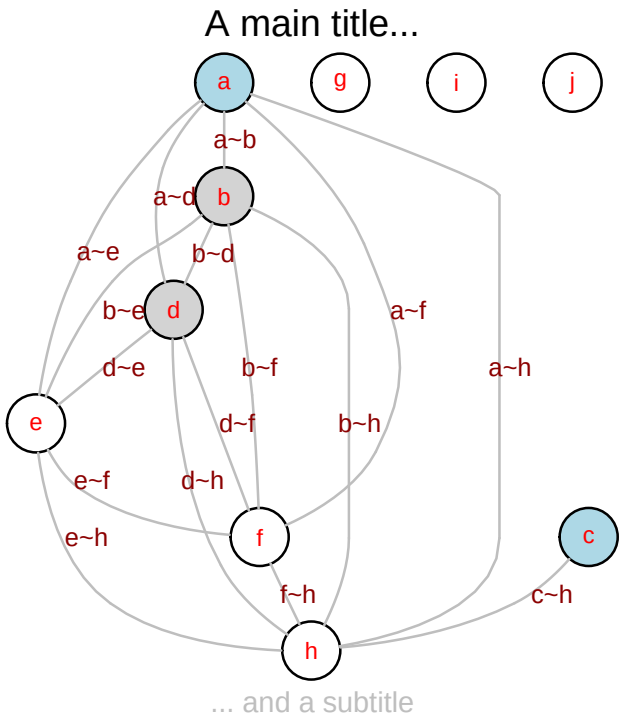
Both `nodeRenderInfo` and `edgeRenderInfo` also allow to set parameters for all edges or nodes of the graph at once. The syntax is simple: instead of a named vector one can pass a scalar for a given parameter. This value will then be assigned to all available nodes or edges.

```
> nodeRenderInfo(g1) <- list(lty=1)
> edgeRenderInfo(g1) <- list(lty=1, lwd=2, col="gray")
> renderGraph(g1)
```



The easiest way to set an attribute back to its default (the session default if it has been set) is to pass in a `NULL` list item via the respective setter function.

```
> nodeRenderInfo(g1) <- list(fill=list(b=NULL, d=NULL))
> renderGraph(g1)
```



5 Graphical parameters that affect the layout

As mentioned before, some graphical parameters are somewhat on the border between graph layout and graph rendering. The shape of the node for example does not drastically affect it's location, however the layout of the edges pointing to and from this node might slightly change. Since manipulating these attributes is a frequent operation in graph plotting, we decided to make them available through the rendering interface.

5.1 Node shapes

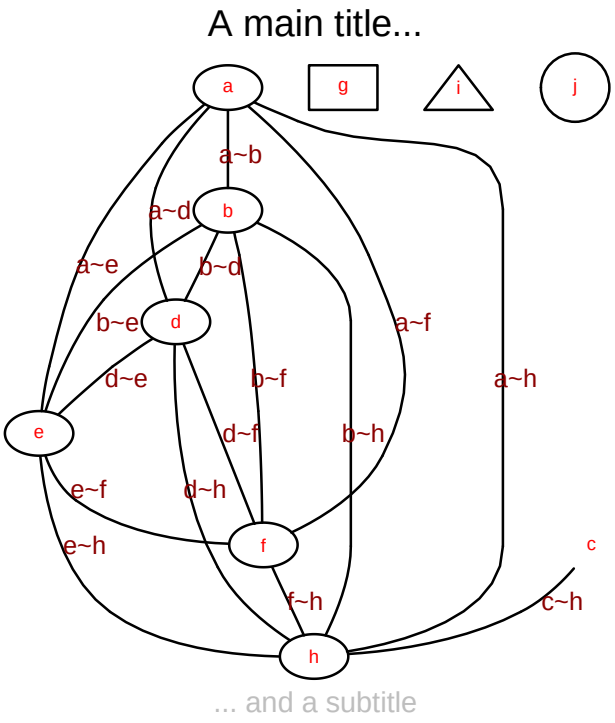
Node shapes can be set using the `shape` parameter. Currently, the rendering function supports the following shape types:

- circle this is the default value. Circular nodes are not affected by changes in width or height, the algorithm will always use a quadratic bounding box
- ellipse this shape allows for differences in width and height. Elliptical nodes often fit node labels better without wasting too much real estate.
- box, rectangle A rectangular node.

- triangle this is currently only partly supported due to restrictions in the Graphviz interface. Edges locations might not be optimal when triangular nodes are used.
- plaintext no node shape at all, only the node labels are plotted.

Lets change all nodes shapes to ellipses first and then try out some of the remaining shapes on single nodes. Because of their impact on the overall layout, we have to run `layoutGraph` again in order for these modifications to work.

```
> nodeRenderInfo(g1) <- list(shape="ellipse")
> nodeRenderInfo(g1) <- list(shape=c(g="box", i="triangle",
+                                    j="circle", c="plaintext"))
> g1 <- layoutGraph(g1)
> renderGraph(g1)
```

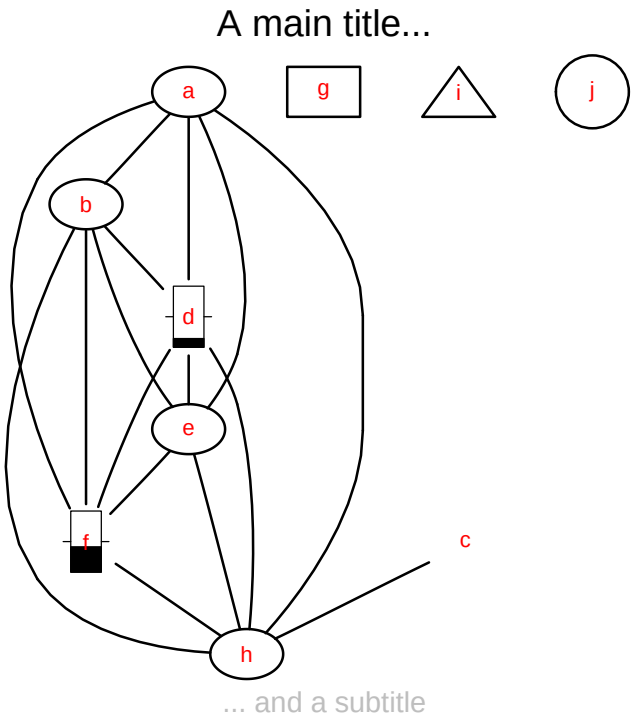


To provide even more flexibility, the values of the `shape` attributes can also be user-defined functions. `renderGraph` will call these functions internally, passing on a number of parameters. The most important parameter is a two-by-two matrix giving the bounding box of the respective node. This information should be used in the self-defined node plotting function to control the node location and also its size. No clipping ever occurs to this bounding box, and if the user decides to extend the node size beyond these limits, overplotting with other nodes or edges is very likely. The additional parameters that are passsed on to

the function are: `labelX`, `labelY`, `fill`, `col`, `lwd`, `lty`, `textCol`, `style`, `label` and `fontsize`. Consult the help pages of the `layoutGraph` and `renderGraph` functions for details.

As an example we will use a function that creates random thermometer plots as node glyphs in the following code chunk. Note that we make use of the `...` argument to catch all of the passed parameters that we don't really need. We also remove the edge labels from the graph to tidy it up a little.

```
> edgeRenderInfo(g1) <- list(label=NULL)
> myNode <- function(x, col, fill, ...)
+ symbols(x=mean(x[,1]), y=mean(x[,2]), thermometers=cbind(.5, 1,
+ runif(1)), inches=0.5,
+ fg=col, bg=fill, add=TRUE)
> nodeRenderInfo(g1) <- list(shape=list(d=myNode, f=myNode),
+                            fill=c(d="white", f="white"),
+                            col=c(d="black", f="black"))
> g1 <- layoutGraph(g1)
> renderGraph(g1)
```



5.2 Edge arrowheads and arrowtails

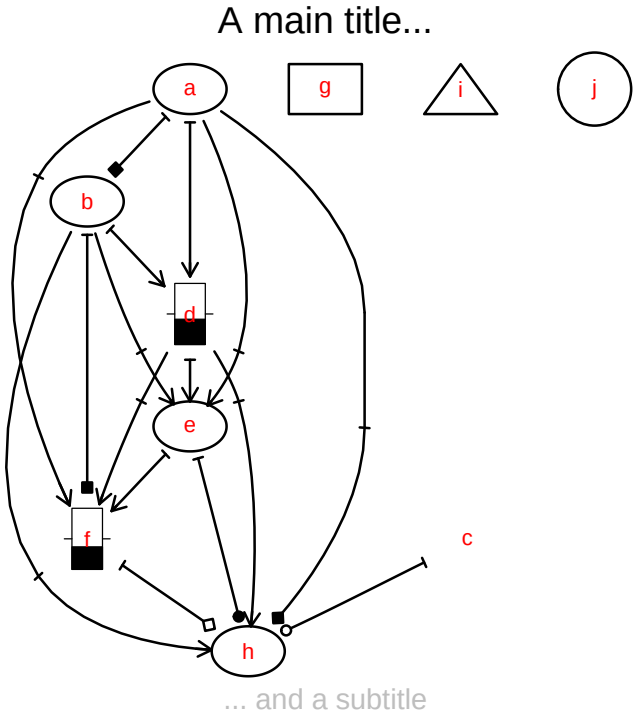
Similar to the control of node shapes, `renderGraph` supports different types of shapes for the tips of the edges. In undirected graphs this feature is not really

supported because edges are merely connecting nodes, they don't convey any information about direction. We can change the mode of our sample graph to `directed` to further explore this feature.

```
> edgemode(g1) <- "directed"
```

The valid values for both arrowheads and arrowtails of the edges are `open`, `normal`, `dot`, `odot`, `box`, `obox`, `tee`, `diamond`, `odiamond` and `none`. The little helper function `edgeNames` can be used to list all available edge names.

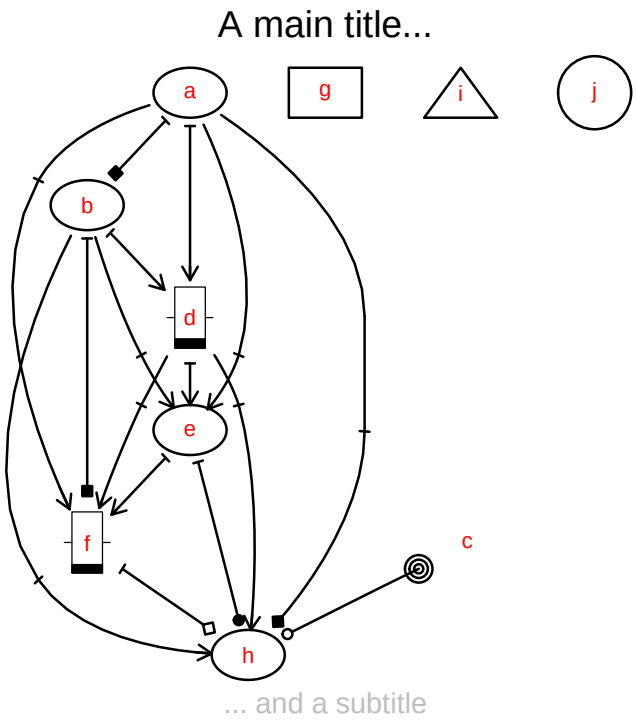
```
> edgeRenderInfo(g1) <- list(arrowhead=c("e~h"="dot", "c~h"="odot",
+                                        "a~h"="diamond", "b~f"="box",
+                                        "a~b"="box", "f~h"="odiamond"),
+                            arrowtail="tee")
> g1 <- layoutGraph(g1)
> renderGraph(g1)
```



There is also the option to pass a user-defined function as `arrowhead` or `arrowtail` parameter to gain even more control. Similar to node shapes, the function has to be able to deal with several parameters: The first parameter gives the center of the location of the arrowhead or tail. The additional parameters `col`, `lwd` and `lty` are color and line styles defined for the edges.

```
> myArrows <- function(x, ...)
+ {
```

```
+ for(i in 1:3)
+ points(x,cex=i, ...)
+ }
> edgeRenderInfo(g1) <- list(arrowtail=c("c~h"=myArrows))
> g1 <- layoutGraph(g1)
> renderGraph(g1)
```



6 Sessioninfo

- R Under development (unstable) (2023-10-22 r85388), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=en_US.UTF-8`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Time zone: `America/New_York`
- TZcode source: `system (glibc)`
- Running under: `Ubuntu 22.04.3 LTS`

- Matrix products: default
- BLAS: `/home/biocbuild/bbs-3.19-bioc/R/lib/libRblas.so`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0`
- Base packages: base, datasets, graphics, grDevices, grid, methods, stats, utils
- Other packages: BiocGenerics 0.49.0, graph 1.81.0, Rgraphviz 2.47.0
- Loaded via a namespace (and not attached): compiler 4.4.0, stats4 4.4.0, tools 4.4.0