RCytoscape

Paul Shannon

June 12, 2011

Cytoscape is a well-known bioinformatics tool for displaying and exploring biological networks. The virtues of ***R*** are very likely already known to the reader. *RCytoscape* uses XMLRPC to communicate between ***R*** and Cytoscape, allowing Bioconductor graphs to be viewed, explored and manipulated with the Cytoscape point-and-click visual interface. Thus these two quite different, quite useful bioinformatics software environments are connected, mutually enhancing each other, providing new possibilities for exploring biological data.

1 Prerequisites

In addition to this package (RCytoscape), you will need:

- XMLRPC, an ***R*** package which provides the communication layer for your ***R*** session. It can be downloaded from `http://www.omegahat.org/XMLRPC/`
- Cytoscape version 2.7, which can be downloaded from `http://cytoscape.org`.
- CytoscapeRPC, version 1.2 or later, a Cytoscape plugin which provides the Cytoscape end of the communication layer, can be downloaded from the Cytoscape plugins website: `http://cytoscape.org/plugins.html`.

2 Getting Started

Install the CytoscapeRPC plugin. This process is explained at

`http://cytoscape.org/manual/Cytoscape2_7Manual.html#PluginsandthePluginManager`

but briefly (and more simply), installation consists of copying the plugin's jar file to the 'Plugins' directory of your installed Cytoscape application, and then restarting Cytoscape. You should see **CytoscapeRPC** in Cytoscape's **Plugins** menu; click to

activate the plugin, which starts the XMLRPC server, and which listens for commands from ***R*** (or elsewhere). The default and usual choice is to communicate over port 9000 on localhost. You can choose a different port from the Plugins->CytoscapeRPC activate menu item. If you choose a different port, be sure to use that port number when you call the RCytoscape constructor – the default value on in the constructor argument list is also 9000.

3 A minimal example

Here we create a 3-node graphNEL in R, send it to Cytoscape for display and layout. For the sake of simplicity, no node attributes, no edges, and no vizmapping is included; those topics are covered in subsequent examples.

```
> library(RCytoscape)
> g <- new("graphNEL", edgemode = "directed")
> g <- graph::addNode("A", g)
> g <- graph::addNode("B", g)
> g <- graph::addNode("C", g)
> cw <- new.CytoscapeWindow("vignette", graph = g)
> displayGraph(cw)

[1] "label"
list()
```

You should see a single red dot in the middle of a small Cytoscape window titled 'simple', contained along with possibly other windows within the Cytoscape Desktop. This graph needs layout. You can accomplish this in two ways: by function call (see below), or interactively from the Cytoscape 'Layout' menu, where a reasonable choice is 'JGraph Layouts' -> 'GEM Layout', or 'yFiles Organic'.

After layout, you will see the structure of this graph: simply 3 unconnected nodes. These nodes will be unlabeled, small and colored red: not very informative. Fortunately, Cytoscape has some built-in rendering rules in which (and unless instructed otherwise) nodes are rendered round in shape, pale red in color, and labeled with the names supplied when they were added to the graph; edges (if any) are simple blue lines. To get this default rendering, simply type

```
> layout(cw, layout.name = "grid")
> redraw(cw)
```

4 Somewhat more complicated: some node attributes and an introduction to 'vizmap' rules

We often know quite a lot about the nodes and edges in our graphs. By conveying this information visually, the graph will be easier to explore. For instance, we may know that gene A phosphorylates gene B, that A is a kinase and B a transcription factor, and that their mRNA expression (compared to a control) is a log fold change, base 2, of 1.8 and 3.2 respectively. One of the core features of Cytoscape, the 'vizmapper', allows you to specify how data values (e.g., 'kinase', 'transcription factor'; expression ratios) should control the visual attributes of the graph (for instance, node shape. Here is a simple example. (Note: edge attributes work just like node attributes, both in assigning them, and in using them to control visual attributes of the graph. In this exercise, however, and to keep things simple, only node attributes are discussed. Edges and edge attributes are discussed below.)

We continue with the small 3-node graph created in the previous code example. You will encounter one obscurity in the code chunk just below: the explicit initialization of node (and edge) attributes for a graph. This is similar to, but goes a bit further than, calls to *nodeDataDefaults* required for Bioconductor graph objects. In fact, such calls are executed within the RCytocape initNodeAttribute and initEdgeAttribute methods. An additional bit of information is required here: the 'class' or data type of the attribute you are initializing. It can be 'char', 'integer' or 'numeric'.

For instance, the node attribute called 'moleculeType' has values which are character strings. The node attribute called 'lfc' (log fold change) is declared to be numeric. This extra fuss is unavoidable: if you do not stipulate the class of each node and edge attribute, as is demonstrated below, then RCytoscape will report an error. We require this because it is only by way of such explicit assignment that RCytoscape can resolve the difference between integer and floating point values – necessary in the statically typed Java language (in which Cytoscape is written) though not in R.

```
> g <- cw@graph
> g <- initNodeAttribute(graph = g, attribute.name = "moleculeType",
+     attribute.type = "char", default.value = "undefined")
> g <- initNodeAttribute(graph = g, "lfc", "numeric", 0)
> nodeData(g, "A", "moleculeType") <- "kinase"
> nodeData(g, "B", "moleculeType") <- "TF"
> nodeData(g, "C", "moleculeType") <- "cytokine"
> nodeData(g, "A", "lfc") <- -1.2
> nodeData(g, "B", "lfc") <- 1.8
> nodeData(g, "C", "lfc") <- 3.2
> cw = setGraph(cw, g)
> displayGraph(cw)
```

```
[1] "label"
```

```
[1] "moleculeType"
[1] "lfc"
list()
```

```
> redraw(cw)
```

You can now explore these simple node attribute values in Cytoscape, selecting nodes, and navigating around inside the Cytoscape 'Data Panel' – which is designed for exactly that purpose.

5 Modifying the display: default values and mapping rules

By default, Cytoscape displays nodes as pale red circles circles, and edges as blue undecorated lines. RCytoscape provides an easy way to change these defaults. More interestingly, RCytoscape provides easy programmatic access to the ***vizmapper***, by means of which the size, shape and color of nodes and edges is determined by the data attributes you have attached to those nodes and edges.

First, let's change the the defaults. Note that 'redraw' must be called on the CytoscapeWindow to force an update of the display. Cytoscape is a little inconsistent about this: setting defauls, like you see below, requires the redraw, but setting rules (about which more below) does not. We hope to get more consistency of Cytoscape soon. In the meantime, if you are rendering a larg graph, you may want to call this only when you are sure you need to.

```
> setDefaultNodeShape(cw, "octagon")
```

```
[1] TRUE
```

```
> setDefaultNodeColor(cw, "#AAFF88")
```

```
[1] TRUE
```

```
> setDefaultNodeSize(cw, 80)
```

```
[1] TRUE
```

```
> setDefaultNodeFontSize(cw, 40)
```

```
[1] TRUE
```

```
> redraw(cw)
```

Now we will add some visual mapping – *(vizmap)* – rules. These rules connect data attributes to visual attributes, thereby giving the eye with additional information about the network. In our first mapping, we use map from the data attribute 'moleculeType' to node shapes to. To begin, ask Cytoscape for the node shape names it recognizes. Then query the graph for a list of the data attributes, establishing that 'moleculeType' is indeed a current data attribute of all nodes. Then find out what distinct values are defined for moleculeType across the nodes of the graph. Finally, define and set the rule.

```
> getNodeShapes(cw)

 [1] "trapezoid"     "round_rect"    "ellipse"       "triangle"
 [5] "rect_3d"       "diamond"       "parallelogram" "octagon"
 [9] "trapezoid_2"   "rect"          "vee"           "hexagon"

> print(noa.names(getGraph(cw)))

[1] "label"        "moleculeType" "lfc"

> print(noa(getGraph(cw), "moleculeType"))

         A          B          C
  "kinase"       "TF" "cytokine"

> attribute.values <- c("kinase", "TF", "cytokine")
> node.shapes <- c("diamond", "triangle", "rect")
> setNodeShapeRule(cw, node.attribute.name = "moleculeType", attribute.values,
+     node.shapes)
> redraw(cw)
```

The node shape rule, above, is an example of a 'lookup' rule, which we sometimes call a 'discrete' rule. The network has three nodes, each of them a gene, and each of them has a 'moleculeType' attribute. In this case, the values are 'kinase', 'TF' (for transcription factor) and 'cytokine'. These are the discrete values taken on by the moleculeType node attribute. For each, we specify the shape we wish to use in rendering that type of molecule. Thus, there is a discrete set of values, and we map from those values to shapes by a simple 'lookup' process.

(Note (7 October 2010): Cytoscape 2.7 apparently has some bugs in handling default values in vizmap rules. For the time being, in creating a lookup rule, it is best to specify a complete mapping between data values and, i.e., node shape. That is, specify all possible discrete data values, and provide an explicit mapping for all of them.)

A second class of rules uses interpolation. In the classic example, every node (every gene) has a mRNA expression value, expressed as a log fold change ('lfc') ratio, experiment vs. control. Nodes with negative lfc are rendered in shades of green; nodes with positive lfc are rendered in shades of red. Nodes with lfc == 0 are rendered in white.

All intermediate values are rendered in appropriately interpolated shades of either green or red.

setNodeColorRule and setNodeSizeRule are commonly interpolation rules: use mode='interpolate'. But they may also be called as lookup rules: use parameter mode='lookup' to accomplish this, and provide as many data points and sizes (or colors) needed to cover all possible values of the attribute you are mapping.

```
> setNodeColorRule(cw, "lfc", c(-3, 0, 3), c("#00AA00", "#00FF00",
+     "#FFFFFF", "#FF0000", "#AA0000"), mode = "interpolate")
```

Note that there *five* colors, but only three control.points. The two additional colors tell the interpolated mapper which colors to use if the stated data attribute (lfc) has a value less than the smallest control point (paint it a darkish green, #00AA00) or larger than the largets control point (paint it a darkish red, #AA0000). These extreme (or out-of-bounds) colors may be omitted:

```
> setNodeColorRule(cw, "lfc", c(-3, 0, 3), c("#00FF00", "#FFFFFF",
+     "#FF0000"), mode = "interpolate")
```

in which case RCytoscape will resuse the first and last values (green and red) for out-of-bounds values, and issue a warning to the console.

Now, add a node size rule, using 'lfc' again as the controlling node attribute.

```
> control.points = c(-1.2, 2, 4)
> node.sizes = c(10, 20, 50, 200, 205)
> setNodeSizeRule(cw, "lfc", control.points, node.sizes, mode = "interpolate")
```

6 Add some edges, edge attributes, and some rules for their rendering.

```
> g <- cw@graph
> g <- initEdgeAttribute(graph = g, attribute.name = "edgeType",
+     attribute.type = "char", default.value = "unspecified")
> g <- graph::addEdge("A", "B", g)
> g <- graph::addEdge("B", "C", g)
> g <- graph::addEdge("C", "A", g)
> edgeData(g, "A", "B", "edgeType") <- "phosphorylates"
> edgeData(g, "B", "C", "edgeType") <- "promotes"
> edgeData(g, "C", "A", "edgeType") <- "indirectly activates"
> cw@graph <- g
> displayGraph(cw)
```

```
[1] "label"
[1] "moleculeType"
[1] "lfc"
[1] "edgeType"
edgeType
    TRUE
```

```
> line.styles = c("DOT", "SOLID", "SINEWAVE")
> edgeType.values = c("phosphorylates", "promotes", "indirectly activates")
> setEdgeLineStyleRule(cw, "edgeType", edgeType.values, line.styles)
> redraw(cw)
> arrow.styles = c("Arrow", "Delta", "Circle")
> setEdgeTargetArrowRule(cw, "edgeType", edgeType.values, arrow.styles)
```

7 Hide, Show, and Float Cytoscape Panels

If you want to have more of the Cytoscape Desktop devoted to displaying your graph, you can hide the panels which normally occupy the left and bottom proting of that Desktop. There are related panels for 'docking' and 'floating' those panels. To save on typing, your arguments to these functions (see below) can be very terse, and are case-independent.

```
> hidePanel(cw, "Data Panel")
> floatPanel(cw, "D")
> dockPanel(cw, "d")
> hidePanel(cw, "Control Panel")
> floatPanel(cw, "control")
> dockPanel(cw, "c")
```

8 Selecting Nodes

Let us now try some simple back-and-forth between Cytoscape and R. In Cytoscape, click the 'B' node. In R:

```
> getSelectedNodes(cw)
```

```
[1] NA
```

Now we wish to extend the selected nodes to include the first neighbors of the already-selected node 'B'. This is a common operation: for instance, after selecting one or more nodes based on experimental data or annotation, you may want to explore these in the context of interaction partners (in a protein-protein network) or in relation to upstream and downstream partners in a signaling or metabolic network. Type:

```
> selectFirstNeighborsOfSelectedNodes(cw)
```

You will see that all three nodes are now selected. Get their identifers back to R:

```
> nodes <- getSelectedNodes(cw)
```

9 Positioning nodes (and possibilities for data-driven graph layout)

Despite its simplicity, RCytoscape's ***setPosition*** method provides the means to create custom layout algorithms and animations. Here is a rather nonsensical example, in which the 'A' node orbits around a fixed center, with the 'B' and 'C' nodes left unchanged.

```
> cwe <- CytoscapeWindow("test.setPosition", graph = RCytoscape::makeSimpleGraph())
> displayGraph(cwe)

[1] "type"
[1] "lfc"
[1] "label"
[1] "count"
[1] "edgeType"
[1] "score"
[1] "misc"
edgeType    score     misc
    TRUE     TRUE     TRUE

> layout(cwe, "jgraph-spring")
> redraw(cwe)
> center.x <- 200
> center.y <- 200
> radius <- 200
> angles <- rep(seq(0, 360, 5), 3)
> for (angle in angles) {
+     angle.in.radians <- angle * pi/180
+     x <- center.x + (radius * cos(angle.in.radians))
+     y <- center.y + (radius * sin(angle.in.radians))
+     setPosition(cwe, "A", x, y)
+ }
```

10 Movies: explore time course data by animating node color and size

Timecourse microarray expression experiments are commonplace in molecular biology research. Once reduced, the expression data is often summarized as a lfc (log fold change, experiment vs control) and a p-value for each gene at each time point. Many Bioconductor packages can be used to find biologically interesting patterns in such data, but visualization – as one aspect of exploratory data analysis – should not be overlooked. Here is a toy problem, a small example illustrating how this can be accomplished with RCytoscape.

We begin with the small sample graph just like that created in the 'setPosition' example above. We add some made-up statistical signficicance data into a new node attribute, 'pval'.

```
> g <- RCytoscape::makeSimpleGraph()
> g <- initNodeAttribute(g, "pval", "numeric", 1)
> cwm <- CytoscapeWindow("movie", graph = g)
> displayGraph(cwm)

[1] "type"
[1] "lfc"
[1] "label"
[1] "count"
[1] "pval"
[1] "edgeType"
[1] "score"
[1] "misc"
edgeType    score     misc
    TRUE     TRUE     TRUE

> layout(cwm, "jgraph-spring")
> redraw(cwm)
```

Now define the node size and node color rules we will use in the animation:

```
> lfc.control.points <- c(-3, 0, 3)
> lfc.colors <- c("#00AA00", "#00FF00", "#FFFFFF", "#FF0000", "#AA0000")
> setNodeColorRule(cwm, "lfc", lfc.control.points, lfc.colors,
+     mode = "interpolate")
> pval.control.points <- c(0.1, 0.05, 0.01, 1e-04)
> pval.sizes <- c(30, 50, 70, 100)
> setNodeSizeRule(cwm, "pval", pval.control.points, pval.sizes,
+     mode = "interpolate")
```

Once the rules are in effect, node color and size will always be a function of each nodes' lfc and pval numeric attributes. Animation is accomplished easily: all you have to do is to update these two attributes for each node at each time point, and asking Cytoscape to render the nodes following the rules it already has been instructed to follow. Note that a message is written to the Cytoscape status message bar – found at the lower left corner of the Cytoscape desktop – to help keep track of what the current timepoint is.

```
> pval.timepoint.1 <- c(0.01, 0.3, 0.05)
> pval.timepoint.2 <- c(0.05, 0.01, 0.01)
> pval.timepoint.3 <- c(1e-04, 0.005, 0.1)
> lfc.timepoint.1 <- c(-1, 1, 0)
> lfc.timepoint.2 <- c(2, 3, -2)
> lfc.timepoint.3 <- c(2.5, 2, 0)
> for (i in 1:5) {
+     setNodeAttributesDirect(cwm, "lfc", "numeric", c("A", "B",
+         "C"), lfc.timepoint.1)
+     setNodeAttributesDirect(cwm, "pval", "numeric", c("A", "B",
+         "C"), pval.timepoint.1)
+     redraw(cwm)
+     msg(cwm, "timepoint 1")
+     system("sleep 1")
+     setNodeAttributesDirect(cwm, "lfc", "numeric", c("A", "B",
+         "C"), lfc.timepoint.2)
+     setNodeAttributesDirect(cwm, "pval", "numeric", c("A", "B",
+         "C"), pval.timepoint.2)
+     redraw(cwm)
+     msg(cwm, "timepoint 2")
+     system("sleep 1")
+     setNodeAttributesDirect(cwm, "lfc", "numeric", c("A", "B",
+         "C"), lfc.timepoint.3)
+     setNodeAttributesDirect(cwm, "pval", "numeric", c("A", "B",
+         "C"), pval.timepoint.3)
+     redraw(cwm)
+     msg(cwm, "timepoint 3")
+     system("sleep 1")
+ }
```

Though this example is very simple and quite without biologcal value, we hope that adequately illustrate the technique. We commonly use just this approach to explore large scale timecourse expression data experiments in the context of curated, biologically relevant pathways, such as those provided by KEGG, and selected by packages such as SPIA or Category.

11 Future Work

- All vizmap rules are currently added to the 'default' vizmap. It is not yet possible to delete rules from this map, either selectively or as a clean sweep. More flexible use of vizmaps is high on the to-do list.

12 Credits

- RCytoscape would not be possible without the generosity and skill of Jan Bot, whose Cytoscape plugin 'CytoscapeRPC' is the indispensable link in commuication between R and Cytoscape.

- Duncan Temple Lang's XMLRPC R package is also indispensable, providing the R end of the communication link. Like Jan, Duncan has cheerfully provided excellent support during the development of RCytoscape.

- Nishant Gopalakrishnan kindly saw me through the Bioconductor package approval process.

- Dan Tenenbaum and Herve Pages, also of Bioconductor, have offered steadfast assistance and skill in creating the testing framework for RCytoscape.

  I am very grateful to all of these computer scientists, to the Cytoscape project, and to my colleagues at the Institute for Systems Biology.

13 References

- Shannon P, Markiel A, Ozier O, Baliga NS, Wang JT, Ramage D, Amin N, Schwikowski B, Ideker T. 2003. Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome Res. Nov;13(11):2498-504

- Huber W, Carey VJ, Long L, Falcon S, Gentleman R. 2007. Graphs in molecular biology. BMC Bioinformatics. 2007 Sep 27;8.