RCytoscape

Paul Shannon

October 11, 2010

Cytoscape is a well-known bioinformatics tool for displaying and exploring biological networks. The virtues of ***R*** will be known to the reader. The *RCytoscape* package uses XMLRPC to communicate between ***R*** and Cytoscape, allowing Bioconductor graphs to be viewed, explored and manipulated using 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 1.1, 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 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 differenet 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 <- CytoscapeWindow("vignette", graph = g)
> ping(cw)

[1] "It works!"

> 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. From the Cytoscape 'Layout' menu, a reasonable choice is to choose 'JGraph Layouts' -> 'GEM Layout'. Other possibilities include '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

```
> redraw(cw)

[1] TRUE
```

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

We often know quite a lot about the objectes 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 begin with the small 3-node graph created in the previous code example. You will ecounter one obscurity in the code chunk just below: the assignment of a 'class' attribute to each nodeData category in the graph. 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 mandatory: if you do not stipulate the class of each node and edge attribute, as you see below, than RCytoscape will balk. 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. The first method called on the CytoscapeWindow object, below, serves to remind you of the legal names you can use in assigning the class of an attribute.

```
> print(getAttributeClassNames(cw))

[1] "floating|numeric|double" "integer|int"
[3] "string|char|character"

> g <- cw@graph
> nodeDataDefaults(g, attr = "moleculeType") <- "undefined"
> attr(nodeDataDefaults(g, attr = "moleculeType"), "class") <- "string"
> nodeDataDefaults(g, attr = "lfc") <- 0
> attr(nodeDataDefaults(g, attr = "lfc"), "class") <- "numeric"
> 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@graph <- g
> displayGraph(cw)

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

> redraw(cw)

[1] TRUE
```

You can now explore these simple node attribute values in Cytoscape, selecting nodes, and navigating around inside the Cytoscape 'Data Panel'.

Now we will add some visual mapping (vizmap) rules, whose goal is to feed the eye more information about the network. First, the node shape. We begin by inquiring of Cytoscape what the permitted node shapes currently are:

```
> getNodeShapes(cw)

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

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

[1] TRUE

> redraw(cw)

[1] TRUE
```

The node shape rule, above, is an example of a 'lookup' rule, which is sometimes referred to as 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 values, 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, by default, interpolation rules. 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"))
```

Note that there *five* colors, but only three control.points. The extra two colors instruct the 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"))
```

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")
```

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

g <- cw@graph edgeDataDefaults (g, attr='edgeType') <- 'unspecified' attr (edgeDataDefaults (g, attr='edgeType'), 'class') <- 'string'

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)

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)

6 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")

[1] TRUE

> floatPanel(cw, "D")

[1] TRUE

> dockPanel(cw, "d")

[1] TRUE

> hidePanel(cw, "Control Panel")

[1] TRUE

> floatPanel(cw, "control")

[1] TRUE

> dockPanel(cw, "c")

[1] TRUE
```

7 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:

```
> sfn(cw)
```

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

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

8 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