Graph Basics in R and Bioconductor

Denise Scholtens

November 8, 2007

1 Introduction

Graphs are data structures containing nodes and edges in which nodes represent objects of interest and edges represent relationships between the nodes. The flexibility of node-and-edge graphs make them ideal modes of representation for high-throughput data common in systems biology research. Nodes are often used to represent genes/proteins and edges are used to represent many different pairwise relationships, for example, coexpression, interactions, etc.

Contributors to the R and Bioconductor projects have recently focused much effort on enabling statistical analyses of graph-like data. Central to appropriate analysis is the creation of convenient R classes for the representation and storage of these data, as well as interfaces with existing analytic software for graph data. In this exercise, we will learn how to construct objects of the classes required for analysis with the available R functions. We will apply common traversal algorithms and compute popular summary statistics for graphs using the R interface to the Boost Graph Library, specifically the R package *RBGL* (Siek et al., 2002). We will also learn how to visualize graphs using the R interface to the AT&T GraphViz plotting software (Gansner and North , 1999), specifically the R package *Rgraphviz*. These basic classes and utilities are central components to methodological development for more sophisticated statistical analyses of graph data.

1.1 Basic definitions

Table 1 contains a set of basic graph definitions. More comprehensive treatments can be found in Gross and Yellen (1999) and Gentleman et al (2005). The definitions presented here are directly tied to some of the exercises we will perform.

2 Creating *graph* objects in R

First load the libraries required for the analyses. *graph* implements basic graph handling capabilities. *RBGL* and *Rgraphviz* interface with the Boost Graph Library and AT&T GraphViz software, respectively.

```
> library("graph")
> library("RBGL")
> library("Rgraphviz")
```

Term	Definition
$\mathbf{G(V, E)}$	a graph G consisting of a set of nodes V and a set of edges E connecting pairs of the members of V
directed graph	a graph in which edges are directed from a tail node to a head node
undirected graph	a graph in which edges are not directed and represent symmetric relationships between a pair of nodes
degree	in an undirected graph, the number of edges incident on a node
indegree	in a directed graph, the number of edges for which the node is the head
outdegree	in a directed graph, the number of edges for which the node is the tail
complement	for a graph $G(V, E)$, the complement G^c contains the same set of nodes V and all edges between pairs of nodes in V not contained in E
path	a sequence of vertices with an edge connecting each vertex to the next in the sequence
adjacent	a pair of nodes directly connected by an edge
accessible	a pair of nodes connected by a path
connected component	in an undirected graph, a maximal subgraph of G with a path between each pair of nodes
strongly connected component	the parallel definition of connected component for a directed graph

Table 1: Basic definitions for graphs

Method	Description
`acc`	find all nodes accessible from the specified node
`adj`	find all nodes adjacent to the specified node
`complement`	return the complement of the graph
`connComp`	return the connected components in a graph
`degree`	return the degree for all nodes in an undirected graph or indegree and outdegree for directed graphs
`edges`	returns edges containing nodes specified in the argument `which`
`nodes`	return a character vector of node names

Table 2: A few of the methods that act on objects of class *graph*.

2.1 *graphNEL* objects

R contains several classes for graph data objects. The class *graph* is a virtual class that all other classes should extend. Several R methods act on all graph classes, some of which are reported in Table 2. First we describe the most general and flexible graph class, *graphNEL*, which is a node and edge list representation of a graph.

2.2 Directed graphs

Let's create a simple directed graph object and see what the object looks like. Suppose we have a graph with four nodes named A, B, C, and D with two directed edges from B to C and D to A as depicted in Figure 1. One simple way to create an object of class `graphNEL` is to create a 'from-to' matrix and use the function `ftM2graphNEL` to put things in the right places.

```
> from <- c("B", "D")
> to <- c("C", "A")
> ft <- cbind(from, to)
> g <- ftM2graphNEL(ft)
> g
```

```
A graphNEL graph with directed edges
Number of Nodes = 4
Number of Edges = 2
```

We see that `g` contains the correct adjacency and in/outdegree data.

```
> adj(g, c("A", "B", "C", "D"))
```

```
$A
character(0)
```

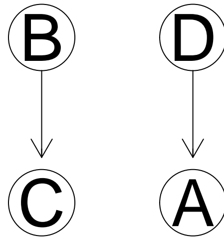

Figure 1: A simple graph with four nodes and two directed edges.

```
$B
[1] "C"

$C
character(0)

$D
[1] "A"

> degree(g)

$inDegree
B D C A
0 0 1 1

$outDegree
B D C A
1 1 0 0
```

Now let's look a little more deeply at how the `graphNEL` object is constructed by examining the slots.

```
> slotNames(g)

[1] "nodes"     "edgeL"     "edgemode"  "edgeData"  "nodeData"  "graphData"

> nodes(g)

[1] "B" "D" "C" "A"

> edgeL(g)

$B
$B$edges
```

```
[1] 3


$D
$D$edges
[1] 4


$C
$C$edges
numeric(0)


$A
$A$edges
numeric(0)
```

```
> edgemode(g)
```

```
[1] "directed"
```

The slots `edgeData`, `nodeData` and `graphData` are objects of class *attrData* which is a container class to manage generic attributes for objects. These can be very useful for storing information about nodes and edges especially when looking at different node and edge types within the same graph. We will not use these slots here.

We see that a `graphNEL` object contains nodes and a list of edges extending from the nodes. Note that `edgeL` represents edges according to numeric indices rather than node labels supplied by the user. This is done intentionally so that node labels can be permuted to generate randomizations of the graph. This concept will be discussed in more detail in the exercise titled 'Testing Associations between Graphs'. Appropriate adjustments to the indices are made when dealing with a subgraph of the original graph.

```
> sg <- subGraph(c("B", "C"), g)
> sg
```

```
A graphNEL graph with directed edges
Number of Nodes = 2
Number of Edges = 1
```

```
> edgeL(sg)
```

```
$B
$B$edges
[1] 2
```

```
$C
$C$edges
integer(0)
```

Methods such as `acc` return node labels as well as numerical indices for easier interpretation.

```
> acc(g, c("B", "C"))

$B
C
1

$C
named numeric(0)

> acc(sg, c("B", "C"))

$B
C
1

$C
named numeric(0)
```

2.3 Incidence matrices

Graph data are often represented in the form of an incidence matrix. An incidence matrix is a square matrix with rows and columns corresponding to the nodes. If the graph is directed, it is helpful to think of the rows as the tails and the columns as the heads. If an edge extends from the ith node to the jth node in a graph, then there is a 1 in the ith row and the jth column in the incidence matrix and all other entries are zero. An incidence matrix for an undirected graph will always be symmetric, but an incidence matrix for a directed graph may not be.

To create an incidence matrix of our simple graph `g`, we can use the function `ftM2adjM` or we can coerce our *graphNEL* object to an incidence matrix. An incidence matrix can also be coerced back to a *graphNEL* object.

```
> gM <- ftM2adjM(ft)
> gM

  B D C A
B 0 0 1 0
D 0 0 0 1
C 0 0 0 0
A 0 0 0 0

> as(g, "matrix")
```

```
  B D C A
B 0 0 1 0
D 0 0 0 1
C 0 0 0 0
A 0 0 0 0
```

```
> as(gM, "graphNEL")

A graphNEL graph with directed edges
Number of Nodes = 4
Number of Edges = 2
```

Nodes and edges can be added to and removed from graphs using the `addNode`, `removeNode`, `addEdge`, and `removeEdge` commands.

```
> g2 = addNode("E", g)
> g2 = addEdge(from = c("D", "B"), to = c("E", "E"), g2)
> g2

A graphNEL graph with directed edges
Number of Nodes = 5
Number of Edges = 4

> inEdges("E", g2)

$E
[1] "B" "D"
```

2.4 Undirected graphs

When we converted our simple `ft` matrix to a *graphNEL* object using `ftM2graphNEL`, the resultant graph was directed. By default, `edgemode` is set to `directed` unless otherwise specified. We can convert our graph `g` to an undirected graph in several ways. For example, the function `ugraph` removes directionality from all edges and returns the resultant undirected graph.

```
> ug <- ugraph(g)
> ug

A graphNEL graph with undirected edges
Number of Nodes = 4
Number of Edges = 2

> edgeL(ug)

$B
$B$edges
[1] 3
```

```
$D
$D$edges
[1] 4


$C
$C$edges
[1] 1


$A
$A$edges
[1] 2
```

Or, we can read the edges in as undirected in the initial call to `ftM2graphNEL`.

```
> ug <- ftM2graphNEL(ft, edgemode = "undirected")
> ug
```

```
A graphNEL graph with undirected edges
Number of Nodes = 4
Number of Edges = 2
```

```
> edgeL(ug)
```

```
$B
$B$edges
[1] 3


$D
$D$edges
[1] 4


$C
$C$edges
[1] 1


$A
$A$edges
[1] 2
```

Note that `edgeL` now assigns edges to all four nodes in the graph `ug`, but the total edge count remains at two since the edges are interpreted to be completely symmetric.

2.5 Weighted graphs

In addition to nodes and edges, we can also add weights to the edges. This is done by specifying the argument `W` in `ftM2graphNEL`. If edge weights are not specified, by default they are assigned a value of 1.

```
> W = 1:2
> g3 = ftM2graphNEL(ft, W = W)
> g3

A graphNEL graph with directed edges
Number of Nodes = 4
Number of Edges = 2

> edgeWeights(g3)

$B
C
1

$D
A
2

$C
numeric(0)

$A
numeric(0)
```

2.6 Special graph classes

The *graphNEL* class provides a very general structure for representing graph data and can be especially suitable for a graph with a large number of nodes and relatively few edges. The *graph* package also contains specific classes for cluster graphs and distance graphs: *clusterGraph* and *distGraph*. These graphs put special conditions on the nodes and edges and the classes accommodate these conditions.

A cluster graph contains a set of nodes which are partitioned into distinct non-overlapping sets. All nodes within a cluster are connected to each other and are not connected to any other nodes. Our simple undirected graph `ug` could be viewed as a cluster graph with two small clusters of only two members each.

```
> CG <- new("clusterGraph", clusters = list(c1 = c("A", "D"), c2 = c("B",
+     "C")))
> CG
```

```
A graph with  undirected  edges
Number of Nodes = 4
Number of Edges = 2
```

```
> acc(CG, c("D", "B"))
```

```
$D
[1] "A" "D"

$B
[1] "B" "C"
```

A distance graph is a graph in which all edges between pairs of nodes exist and represent distances between the nodes. These distances are represented as edge weights in the graph.

```
> x <- 1:4
> names(x) <- letters[1:4]
> d1 <- dist(x)
> DG <- new("distGraph", Dist = d1)
> DG
```

```
distGraph with 4 nodes
```

```
> Dist(DG)
```

```
  a b c
b 1
c 2 1
d 3 2 1
```

2.7 Random graphs

In addition to user-specified graphs, the *graph* package also contains functions for randomly generating graphs. The function `randomEGraph` generates undirected graphs according to the classic Erdös-Renyi random graph model. Along with a character vector `V` of node names, the user can supply either the parameter `p` that specifies the probability of any edge existing in the graph, or the parameter `edges` that states the number of edges to be randomly assigned in the graph.

```
> set.seed(123)
> rg <- randomEGraph(LETTERS[1:20], edges = 30)
> rg
```

```
A graphNEL graph with undirected edges
Number of Nodes = 20
Number of Edges = 30
```

```
> summary(degree(rg))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   0.00    1.75    3.50    3.00    4.00    5.00 
```

```
> adj(rg, "B")
```

```
$B
[1] "T" "E" "L" "K"
```

```
> acc(rg, "B")
```

```
$B
C D E F G H J K L M N O P R S T 
2 2 1 3 3 3 2 1 1 2 2 2 2 3 2 1 
```

```
> connComp(rg)
```

```
[[1]]
[1] "A" "Q"

[[2]]
 [1] "B" "C" "D" "E" "F" "G" "H" "J" "K" "L" "M" "N" "O" "P" "R" "S" "T"

[[3]]
[1] "I"
```

This particular instance generated a graph with three connected components: one contains the singleton node "I", one contains only two nodes "A" and "Q", and the other contains all remaining nodes. Note that the mean degree of all nodes is 3. This is expected for an ER graph in which node degree follows a Poisson distribution with mean given by twice the number of edges divided by the number of nodes.

There are other functions to generate random graphs according to different models. For example, `randomNGraph` generates a random graph with a specified node degree distribution and `randomGraph` generates random graphs according to latent variable models.

2.8 Additional exercises

Please consult the *graph* package vignettes and man pages for help with these exercises.

- *Generate a graph with 50 nodes and 50 randomly selected directed edges. Name it 'myDirectedGraph'.*
- *Form the undirected graph counterpart of myDirectedGraph. Name it 'myUndirectedGraph'.*
- *Compare the output of the 'degree' method for the two graphs. What is returned in each instance?*
- *Add randomly generated weights to the edges in myUndirectedGraph. Call this new graph 'myWeightedGraph'.*
- *Create a subgraph of myUndirectedGraph containing the largest connected component. Call this new graph 'myLCCGraph'.*

3 Probing graphs using *RBGL*

3.1 Paths between nodes

RBGL provides an R interface to the Boost Graph Library. This library contains algorithms for probing and analyzing mathematical graph objects. A complete list of the available algorithms is available with the package vignette. Here we demonstrate a few of the functions for our ER random graph.

Suppose we want to know the shortest distance between all pairs of nodes in the graph where distance is defined as the number of edges connecting any pair of nodes. *RBGL* provides a variety of algorithms to answer this question, including the R functions `bellman.ford.sp`, `dag.sp`, `dijkstra.sp`, `johnson.all.pairs.sp`, and `sp.between`. Different algorithms are appropriate for graphs with different structures. In our case we use `johnson.all.pairs.sp`. Note that node A has path length reported as 'Inf' for all nodes except Q. Recall that A and Q formed their own connected component and were disconnected from the remainder of the nodes in the graph. The function `johnson.all.pairs.sp` reports path length between nodes that are not accessible to each other as 'Inf'. Analyses of graph data often report the average path length for all pairs of nodes in a graph. This is not particularly interpretable if the graph contains multiple connected components.

```
> l <- johnson.all.pairs.sp(rg)
> l[1:5, 1:5]
```

```
    A   B   C   D   E
A   0 Inf Inf Inf Inf
B Inf   0   2   2   1
C Inf   2   0   2   1
D Inf   2   2   0   2
E Inf   1   1   2   0
```

```
> l[1, ]
```

```
  A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P   Q   R   S   T
  0 Inf Inf Inf Inf Inf Inf Inf Inf Inf Inf Inf Inf Inf Inf Inf   1 Inf Inf Inf
```

3.2 Sets of highly connected nodes

Sometimes we are also interested in finding which nodes in a graph are highly connected to similar nodes. The function `highlyConnSG` partitions a graph into a set of highly connected subgraphs in which each node is connected to at least half of the other nodes in the subgraph. For `rg`, we see that we have a set of five nodes that are all connected to at least two of the other four. There are several functions available for detecting what might be called 'clusters' of nodes, for example `maxClique`, `kcores` and `kCliques`. See Figure 2 for a plot of `rg` and its largest highly connected subgraph.

```
> hcs <- highlyConnSG(rg)
> hcsN = unlist(lapply(hcs$clusters, FUN = length))
> hcsMax = hcs$clusters[[which.max(hcsN)]]
```

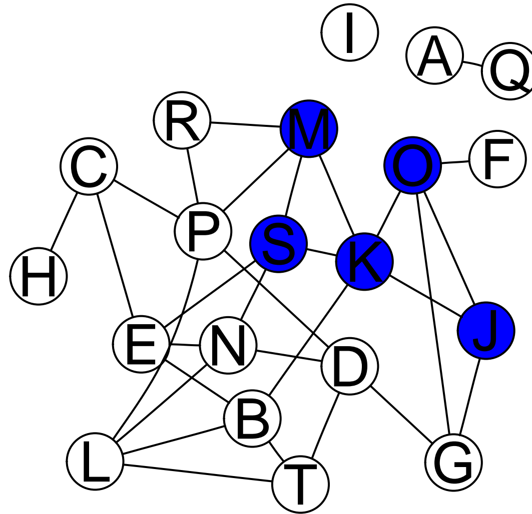

Figure 2: A plot of `rg` and with nodes in its largest highly connected subgraph colored blue.

```
> hcsSG = subGraph(hcsMax, rg)
> hcsSG

A graphNEL graph with undirected edges
Number of Nodes = 5
Number of Edges = 6
```

3.3 Additional Exercises

Please consult the *RBGL* package vignettes and man pages for help with these exercises.

- *Using myDirectedGraph, myUndirectedGraph, myWeightedGraph, and myLCCgraph, explore the functions* `maxClique`*,* `mstree.kruskal`*,*`connComp`*,*`strongConnComp`*, and* `kcores`*. Determine the types of graph for which these functions are appropriate. Determine what it is that the functions are returning.*

4 Visualizing graphs with *Rgraphviz*

While the *graph* classes provide helpful storage, representation, and manipulation of graph data, they do not necessarily lend to easy comprehension of the topology of connectedness between nodes. Visualization can help clarify the nature of relationships in a graph. The R package *Rgraphviz* interfaces with AT&T GraphViz software to accomplish visualization.

A `plot` method does exist for *graph* objects, so simple plotting of graphs is straightforward. Recall our simple graph `g`. The code below produced the plot in Figure 1.

```
> plot(g)
```

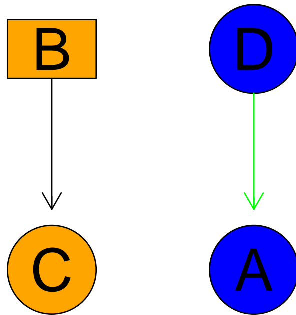

Figure 3: A simple graph with specified node and edge attributes.

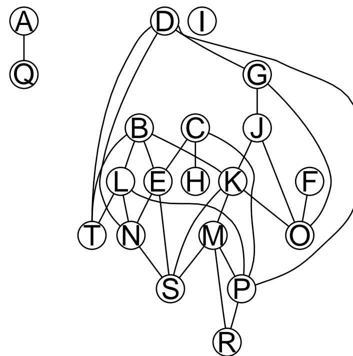

Figure 4: `rg` plotted using the `dot` layout.

In many cases, we want to add interest to graphs using color or different edge types. This can be done by specifying `nodeAttrs` and `edgeAttrs` in the call to `plot`. Here are a few simple examples. The documentation for the *Rgraphviz* package contains extensive instructions on how to construct graphs with particular attributes.

```
> nAttrs <- list()
> eAttrs <- list()
> nAttrs$fillcolor <- c(A = "blue", B = "orange", C = "orange",
+     D = "blue")
> nAttrs$shape <- c(B = "box")
> eAttrs$color <- c("D~A" = "green")
> plot(g, nodeAttrs = nAttrs, edgeAttrs = eAttrs)
```

In addition to the ability to change node and edge attributes, *Rgraphviz* also provides several different layout features. Figure 2 used the 'neato' layout to plot `rg`. Figures 4-5 plot our the same graph `rg` using the `dot` and `twopi` layouts, respectively. Layouts are specified using the `y` argument in the `plot` method for *graph* objects.

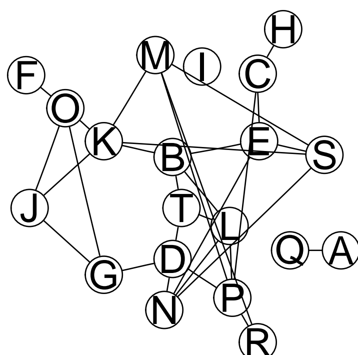

Figure 5: `rg` plotted using the `twopi` layout.

4.1 Additional Exercises

Please consult the `Rgraphviz` package vignettes and man pages for assistance with these exercises.

- *For myDirectedGraph, specify edgeAttrs such that reciprocated edges are plotted distinctly from one another. Then change the attrs so that they are plotted as one edge with an arrow on either end.*
- *Investigate global attributes specified through 'attrs'. Apply some chagnes to any of the graphs you generated.*
- *For myWeightedGraph, plot the edges with text labels corresponding to the weights.*

5 Summary

The *graph* package provides a set of classes for storing, manipulating and analyzing graph data. In addition, it contains many methods for accessing basic, but very important characteristics of graph nodes and edges. *RBGL* provides an interface to classic graph theoretic analysis algorithms contained in the Boost Graph Library. *Rgraphviz* provides an interface to the AT&T GraphViz software for visualizing and plotting graphs.

These three packages lay the foundation for the use and development of more complicated statistical analyses of graph data. Much work remains to be done to develop useful analytical tools for making statistical inference on graphs generated by high-throughput biological data.

SessionInfo

```
R version 2.6.0 (2007-10-03)
x86_64-unknown-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.
```

```
attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] Rgraphviz_1.16.0 RBGL_1.14.0      graph_1.16.1

loaded via a namespace (and not attached):
[1] cluster_1.11.9  rcompgen_0.1-15 tools_2.6.0
```

References

R.C. Gentleman et al. *Bioinformatics and Computational Biology Solutions Using R and Bioconductor.* Springer, 2005.

E. R. Gansner and S. C. North An open graph visualization system and its applications to software engineering *Software Practice and Experience*, 30: 1203–1233, 1999.

J. Gross and J. Yellen *Graph Theory and its Applications.* CRC Press, 1999.

J. G. Siek, L.-Q. Lee and A. Lumsdaine *The Boost Graph Library.* Addison Wesley, Boston, 2002.