How To use the graph package

October 26, 2021

1 Introduction

The *graph* package provides an implementation of graphs (the kind with nodes and edges) in R. Software infrastructure is provided by three different, but related packages,

graph Provides the basic class definitions and functionality.

RBGL Provides an interface to graph algorithms (such as shortest path, connectivity etc).

Rgraphviz Provides rendering functionality. Different layout algorithms are provided and node plotting, line type, color etc parameters can be controlled by the user.

A short description of the R classes and methods is given at the end of this document. But here, we begin by creating some graphs and performing different operations on those graphs.

The reader will benefit greatly from also have the *Rgraphviz* package available and from using it to render the different graphs as they proceed through these notes.

2 Getting Started

We will first create a graph and then spend some time examining some of the different functions that can be applied to the graph. We will create a random graph as the basis for our explorations (but will delay explaining the creation of this graph until Section 3).

First we attach the *graph* package and create a random graph (this is based on the Erdos-Renyi model for random graphs).

```
> library(graph)
> set.seed(123)
> g1 = randomEGraph(LETTERS[1:15], edges=100)
> g1

A graphNEL graph with undirected edges
Number of Nodes = 15
Number of Edges = 100
```

We can next list the nodes in our graph, or ask for the degree (since this is an undirected graph we do not distinguish between in-degree and out-degree). For any node in `g1` we can find out which nodes are adjacent to it using the `adj` function. Or we can find out which nodes are accessible from it using the `acc` function. Both functions are *vectorized*, that is, the user can supply a vector of node names, and each returns a named list. The names of the list elements correspond to the names of the nodes that were supplied. For `acc` the elements of the list are named vectors, the names correspond to the nodes that can be reached and the values correspond to their distance from the starting node.

```
> nodes(g1)

 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"

> degree(g1)

 A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
13 13 14 13 14 14 12 14 12 14 12 14 13 14 14

> adj(g1, "A")

$A
 [1] "N" "M" "E" "I" "O" "G" "H" "D" "K" "J" "C" "L" "F"

> acc(g1, c("E", "G"))

$E
A B C D F G H I J K L M N O
1 1 1 1 1 1 1 1 1 1 1 1 1 1

$G
A B C D E F H I J K L M N O
1 1 1 2 1 1 1 2 1 1 1 1 1 1
```

One can obtain subgraphs of a given graph by specifying the set of nodes that they are interested in. A subgraph is actually a copy of the relevant part of the original graph. A subgraph is the set of specified nodes plus any edges between them. We can also compute the boundary of a subgraph. The boundary is the set of all nodes in the original graph that have an edge to the specified subgraph. The `boundary` returns a named list with one component for each node in the subgraph. The elements of this list are vectors which contain all nodes in the original graph that have an edge to that element of the subgraph.

We also demonstrate two edge related functions in the code chunk below. One retrieves all edges from a graph and is called `edges` while the other retrieves the edge weights and is called `edgeWeights`.

```
> sg1 = subGraph(c("A", "E", "F","L"), g1)
> boundary(sg1, g1)

$A
 [1] "C" "D" "G" "H" "I" "J" "K" "M" "N" "O"

$E
 [1] "B" "C" "D" "G" "H" "I" "J" "K" "M" "N" "O"

$F
 [1] "B" "C" "D" "G" "H" "I" "J" "K" "M" "N" "O"

$L
 [1] "B" "C" "D" "G" "H" "I" "J" "K" "M" "N" "O"

> edges(sg1)

$A
[1] "E" "L" "F"
```

```
$E
[1] "A" "F" "L"

$F
[1] "E" "A" "L"

$L
[1] "E" "F" "A"

>  edgeWeights(sg1)

$A
E L F
1 1 1

$E
A F L
1 1 1

$F
E A L
1 1 1

$L
E F A
1 1 1

>
```

2.1 Some Algebraic Manipulations

The examples here originally came from Chris Volinsky at AT&T, but have been modified in places as the *graph* package has evolved. In the code chunk below we demonstrate how to create a graph *from scratch*. In this code chunk two graphs are created, gR and gR2, the first is undirected while the second is a directed graph.

```
> V <- LETTERS[1:4]
> edL1 <- vector("list", length=4)
> names(edL1) <- V
> for(i in 1:4)
+   edL1[[i]] <- list(edges=c(2,1,4,3)[i], weights=sqrt(i))
> gR <- graphNEL(nodes=V, edgeL=edL1)
> edL2 <- vector("list", length=4)
> names(edL2) <- V
> for(i in 1:4)
+   edL2[[i]] <- list(edges=c(2,1,2,1)[i], weights=sqrt(i))
> gR2 <- graphNEL(nodes=V, edgeL=edL2, edgemode="directed")
>
```

New graphs can be constructed from these graphs in many different ways but in all cases the existing graph itself is not altered, but rather a copy is made and the changes are carried out on that copy. Nodes and or edges can be added to the graphs using the functions addNode, addEdge, removeNode

3

and `removeEdge`. All functions will take a vector of nodes or edges and add or remove all of them at one time. One other function in this family is `combineNodes`, this function takes a vector of nodes and a graph and combines those nodes into a single new node (the name of which must be supplied). The function `clearNode` removes all edges to the specified nodes.

```
> gX = addNode(c("E", "F"), gR)
> gX

A graphNEL graph with undirected edges
Number of Nodes = 6
Number of Edges = 2

> gX2 = addEdge(c("E", "F", "F"), c("A", "D", "E"), gX, c(1,2,3))
> gX2

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

> gR3 = combineNodes(c("A","B"), gR, "W")
> gR3

A graphNEL graph with undirected edges
Number of Nodes = 3
Number of Edges = 1

> clearNode("A", gX)

A graphNEL graph with undirected edges
Number of Nodes = 6
Number of Edges = 1

>
```

When working with directed graphs it is sometimes of interest to find the *underlying* graph. This is the graph with all edge orientation removed. The function `ugraph` provides this functionality.

```
> ##find the underlying graph
> ugraph(gR2)

A graphNEL graph with undirected edges
Number of Nodes = 4
Number of Edges = 3

>
```

Other operations that can be carried out on graphs, that are of some interest, are unions, intersections and complements. We have take a rather specialized definition of these operations and it is not one that is widely used, but it is very useful for the bioinformatics and computational biology projects that we are working on.

For two or more graphs all with **the same nodes** we define:

union to be the graph with the same set of nodes as the inputs and edges between any two nodes that were connected in any one graph.

intersection to be the graph with the same set of nodes as the inputs and with edges between two nodes if there was an edge in all graphs.

complement to be the graph with the same set of nodes as its input and edges in the complement if there were none in the original graph.

In the code chunk below we generate a random graph and then demonstrate the concepts of union, intersection and complement.

```
> set.seed(123)
> gR3 <- randomGraph(LETTERS[1:4], M<-1:2, p=.5)
>  x1 <-  intersection(gR,gR3)
>  x1

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

>  x2 <-  union(gR,gR3)
>  x2

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

>  x3 <- complement(gR)
>  x3

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

>
```

Notice that while the graphs `gR` and `gR2` have different sets of edge weights these are lost when the `union`, `intersection` and `complement` are taken. It is not clear how they should be treated and in the current implementation they are ignored and replaced by weight 1 in the output.

3 Random Graphs

Three basic strategies for finding random graphs have been implemented:

randomEGraph A random edge graph. In this graph edges are randomly generated according to a specified probability, or the number of edges can be specified and they are randomly assigned.

randomGraph For this graph the number of nodes is specified as well as some latent factor. The user provides both the node labels and a factor with some fixed number of levels. Each node is randomly assigned levels of the factor and then edges are created between nodes that share the same levels of the factor.

randomNodeGraph A random graph with a pre-specified node distribution is generated.

The function `randomEGraph` will generate graphs using the random edge model. In the code chunk below we generate a graph, `g1` on 12 nodes (with labels from the first 12 letters of the alphabet) and specify that the probability of each edge existing is 0.1. The graph `g2` is on the same set of nodes but we specify that it will contain 20 edges.

```
> set.seed(333)
> V = letters[1:12]
> g1 = randomEGraph(V, .1)
> g1

A graphNEL graph with undirected edges
Number of Nodes = 12
Number of Edges = 7

> g2 = randomEGraph(V, edges=20)
> g2

A graphNEL graph with undirected edges
Number of Nodes = 12
Number of Edges = 20

>
```

The function `randomGraph` generates graphs according to the latent variable model. In the code chunk bel

```
> set.seed(23)
> V <- LETTERS[1:20]
> M <- 1:4
> g1 <- randomGraph(V, M, .2)
```

Our last example involves the generating random graphs with a prespecified node degree distribution. In the example below we require a node degree distribution of 1, 1, 2 and 4. We note that self-loops are allowed (and if someone wants to provide the code to eliminate them, we would be glad to have it).

```
>       set.seed(123)
>       c1 <- c(1,1,2,4)
>       names(c1) <- letters[1:4]
>       g1 <- randomNodeGraph(c1)
>
>
>
```

4   Some Graph Algorithms

In addition to the simple algebraic operations that we have demonstrated in the preceeding sections of this document we also have available implementations of some more sophisticated graph algorithms. If possible though, one should use the algorithms provided in the *RBGL*.

The function `connComp` returns a list of the connected components of the given graph. For a *directed graph* or *digraph* the underlying graph is the graph that results from removing all direction from the edges. This can be achieved using the function `ugraph`. A weakly connected component of a digraph is one that is a connected component of the underlying graph and this is the default behavior of `connComp`.

```
>  g1

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

```
> g1cc <- connComp(g1)
> g1cc

[[1]]
[1] "a" "b" "c" "d"

> g1.sub <- subGraph(g1cc[[1]], g1)
> g1.sub

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

>
```

Another useful set of graph algorithms are the so-called searching algorithm. For the *graph* package we have implemented the depth first searching algorithm as described in Algorithm 4.2.1 of [?]. More efficient and comprehensive algorithms are available through the *RBGL* package. The returned value is a named vector. The names correspond to the nodes of the graph and the values correspond to the distance (often the number of steps) or sum of the edgeweights along the path to that node.

```
> DFS(gX2, "E")

A B C D E F
1 2 5 4 0 3

>
```

5 Special Types of Graphs

We have found it useful to define a few special types or classes of graphs for some bioinformatic problems but they likely have broader applicability. All of the functions described above should have methods for these special types of graphs (although we may not yet have implemented all of them, please let the maintainer know if you detect any omissions).

First is the `clusterGraph`. A cluster graph is a graph where the nodes are separated into groups or clusters. Within a cluster all nodes are connected (a complete graph) but between clusters there are no edges. Such graphs are useful representations of the output of clustering algorithms.

```
> cG1 <- new("clusterGraph", clusters=list(a=c(1,2,3), b=c(4,5,6)))
> cG1

A graph with  undirected  edges
Number of Nodes = 6
Number of Edges = 6

> acc(cG1, c("1", "2"))

$`1`
[1] 1 2 3

$`2`
[1] 1 2 3

>
```

The other special type of graph that we have implemented is based on distances. This graph is completely connected but the edge weights come from inter-node distances (perhaps computed from an expression experiment).

```
> set.seed(123)
> x <- rnorm(26)
> names(x) <- letters
> library(stats)
> d1 <- dist(x)
> g1 <- new("distGraph", Dist=d1)
> g1

distGraph with 26 nodes

>
```

6 Coercion

There are very many different ways to represent graphs. The one chosen for our basic implementation is a node and edge-list representation. However, many others use an adjacency matrix representation. We provide a number of different tools that should help users coerce graphs between the different representations.

Coercion from an adjacency matrix to a *graphNEL* object requires a numeric matrix with both row and column names. These are taken to define the nodes of the graph and the edge weights in the resultant graph are determined by the values in the array (weights zero are taken to indicate the absence of an edge).

The function ftM2adjM converts a *from-to* matrix into an adjacency matrix. Conversion to a *graphNEL* graph can be carried out using the as method for that class.

An aM is an affiliation matrix which is frequently used in social networks analysis. The rows of aM represent actors, and the columns represent events. A one, 1, in the ith row and jth column represents the affiliation of the ith actor with the jth event. The function aM2bpG coerces a aM into an instance of the *graphNEL* where the nodes are both the actors and the events (there is currently no bipartite graph representation, although one could be added).

The two functions sparseM2Graph and graph2SparseM provide coercion between *graphNEL* instances and sparse matrix representations. Currently we rely on the *SparseM* of Koncker and Ng for the sparse matrix implementation.

6.1 Classes

We briefly review some of the class structure here and refer the reader to the technical documentation for this package for more details.

The basic class, *graph*, is a virtual class and all other classes will extend this class. There are three main implementations available. Which is best will depend on the particular data set and what the user wants to do with it. The only slot defined in the virtual class is edgemode which can be either *directed* or *undirected* indicating whether the edges are directed or not.

The class *graphNEL* is a node and edge-list representation of a graph. That is the graph is comprised of two components a list of nodes and a list of the out edges for each node.

The class *graphAM* is an adjacency matrix implementation. It will be developed next and will use the *SparseM* package if it is available.

The class *clusterGraph* is a special form of graph for clustering. In this graph each cluster is a completely connected component (a clique) and there are no between cluster edges.