1   Introduction

The purpose of this document is to describe the implementation the classes used to represent graphs in the *graph* package and to discuss design issues for future development.

There are many different ways to represent a graph and to deal with the edges and nodes within that graph. Below we discuss the graph representations implemented in the *graph* package and define the set of methods that form the *graph interface* as determined empiracally by the methods used by packages like *RBGL* when interacting with `graph` objects.

A graph is a pair of sets, $G = (V, E)$ where V is the set of nodes and E is the set of edges, which are determined by relationships that exist between the nodes. If we let $n = |V|$, be the number of nodes then, excluding self-loops there are at most n choose 2 edges in G. A *simple graph* is a graph with at most one edge between any pair of nodes and no self-loops.

2   The *graph* class

The *graph* class and its subclasses support simple graphs as well as graphs with at most one self-loop on any given node. Not all graph representations can easily support more general graphs. Limiting to simple graphs with self-loops allows for reversible conversions between different graph representations. Furthermore, this limitation simplifies the interface of edge related methods which would otherwise have to support ways of identifying one of many edges between the same pair of nodes.

Arbitrary attributes can be associated with a graph, with a node, or with an edge. For both nodes and edges if one edge or node has a particular attribute then all nodes and edges must have that attribute. Nodes and edges can have more than one attribute associated with them.

This raises the question of whether we should use the AnnotatedDataFrame class from Biobase here as a way to implement general node and edge attributes.

However, currently AnnotatedDataFrame is based on a data.frame and cannot easily support arbitrary attributes. Even having a vector of length greater than one as the value of an attribute could cause problems.

The *graph* class itself is VIRTUAL and has the following definition:

```
> library("graph")
> getClass("graph")

Virtual Class "graph" [package "graph"]

Slots:

Name:    edgeData    nodeData renderInfo   graphData
Class:    attrData    attrData renderInfo        list

Known Subclasses: "graphNEL", "graphAM", "distGraph", "clusterGraph"
```

The `edgemode` slot indicates whether the graph is *directed* or *undirected.* Since some graph algorithms only make sense in a directed graph, the edgemode is a property of the entire graph, rather than a property of an edge.

The `graphData` slot was recently added to hold arbitrary attributes for the graph. Although edgemode is such an attribute, it isn't clear whether it should move inside the generic container since edgemode is of such high semantic importance. It probably doesn't matter as long as methods such as `isDirected` do the right thing.

The `edgeData` and `nodeData` slots store the attributes for the edges and nodes of the graph, respectively.

There are currently implementations for the *graphNEL* class, where nodes are a vector and edges are a list, each element of the list correspondes to one node and the values are nodes corresponding to the out-edges from that node. If the graph is directed then all edges essentially appear twice.

The *graphAM* class, which stores the edge information in an adjacency matrix. The matrix must be square and the row names must match the column names. If the graph is undirected then the matrix must also be symmetric.

There are two specialized classes, *distGraph* which takes a distance matrix directly and has special thresholding capabilities. It is not clear whether this should be a specialization of the *graphAM* class or not.

The second specialized class is a *clusterGraph* which can be used to represent the output of a clustering algorithm as a graph. Samples represent nodes and all samples in the same cluster have edges, while samples in distinct clusters do not. Instances of this class must have their edgemode as `undirected`, if the edgemode is reset then coercion to some other mode of graph is needed.

2.1   Methods of graphs

Here are some of the methods that all graph-like objects should support:

nodes(object) Return a character vector of the node labels. The order is not defined.

nodes<-(object) Return a new graph object with the node labels set as specified by a character vector. This is slightly fragile since here order does matter, but the order can only really be determined by first calling `nodes`. by providing a character vector of the appropriate length.

addNode(node, object, edges) Return a new graph object with additional nodes and (optionally) edges. The methods that have been implemented expect `node` to be the node labels of the new nodes specified as a character vector. Optional edges can be specified.

removeNode(node, object) Return a new graph object with nodes (and their incident edges) removed. Current methods are implemented for `node` being a character vector of node labels to remove.

edges(object, which) Return a list with an element for each node in the graph. The names of the list are the node labels. Each element is a character vector giving the node labels of the nodes which the given element shares an edge with. For undirected graphs, reciprocal edges should be included. This representation is very similar to the NEL edgeL structure.

edgeWeights(object, index)

addEdge(from, to, graph, weights) Return a new graph object with additional edges.

removeEdge(from, to, graph) Return a new graph object with the specified edges removed.

numNodes(object) Return a count of the nodes in the graph.

numEdges(object) Return a count of the edges in the graph.

isDirected(object) Return TRUE if the graph is directed and false otherwise.

acc(object, index) See man page.

adj(object, index) See man page.

nodeData Access to node attributes. See man page.

edgeData Access to edge attributes. See man page.

2.2 Some Details

Once both nodes and edges are instances of classes they will be quite large. In order to reduce the storage requirements (especially for large graphs) using integer indices may be beneficial.

The minimum amount of storage required is $|V| + |E|$. If we use an incidence matrix representation then the storage is $|V|^2$. If we use a node and edge list representation then the storage requirements are $|V| + 2|E|$. When either $|V|$ or $|E|$ are large these mechanisms will not be especially efficient. In some cases it may be better to keep the actual node and edge data stored in hash tables and keep other integer vectors available for accessing the necessary components.

2.2.1 Representation of Edges

We have taken the approach of allowing the representation of the edge sets to not contain every node. When the graphs are sparse this can be a fairly large savings in space, but it means that one cannot determine the nodes in a graph from the edges in the graph.

Also, for the *graphNEL* class we do not store the names of the nodes in the NEL, but rather indexes into a the node vector. This is important for allowing us to perform permutations on the nodes of a graph, but causes a number of problems when subsetting graphs, and means that knowledge of the edges does not provide knowledge of the nodes.

3 Multi-graphs

There are no clear and widely used definitions for multi-graphs, so here we will make clear a definition that we believe will be useful for biological computations. We define a multi-graph to consist of two components, one a set of nodes and the second a list of edge sets. Each edge set corresponds to a potentially different set of relationships between the nodes (which are common to all edge sets). We denote this by $G = (V, E_L)$, where V is the set of nodes and $E_L = (E_1, \ldots, E_L)$ is a collection of L edge sets. Each with a potentially distinct set of relationships. The edge sets are essentially identical to the edge sets for a graph, and hence can have arbitrary attributes associated with them, the edges can be either *directed* or *undirected* and self-loops are allowed.

It is not clear whether there should be distinct types of multigraphs as there are graphs. It will surely be more flexible to support a list of edge sets, and to allow these to have different structures.

Current definition does not extend the *graph* class. The definition is:

```
> getClass("multiGraph")

Class "multiGraph" [package "graph"]

Slots:

Name:      nodes     edgeL  nodeData graphData
Class:     vector     list  attrData     list
```

nodes A vector of node identifiers.

edgeL A possibly named list of instances of the *edgeSet* class.

The *edgeSet* class is a virtual class with several different extensions. These include a *edgeSetNEL* and an *edgeSetAM*, others will be added once the interface stabilizes.

Edge attributes are in the edgeData slot in the edgeSet class. This implies that edgeSets in a multiGraph can have completely unrelated edge attributes. Another approach would be to maintain a list conforming to the edgeSet list containing edge attributes that would enforce the same attributes to be defined for all edges in the multiGraph.

3.1 Methods

In some ways it would be most natural to have `edges` methods for the *edgeSet* class the issues raised in Section 2.2.1 seem to preclude this and it only seems to make sense to have `node` and `edges` methods for the *multiGraph* class.

It will probably make sense to be able to name the edgeSets within a multiGraph and to be able to extract graph objects from the multiGraph representing any of the edgeSets.

There should be methods to produce graph objects based on intersection, union, and more complex combination algorithms. The edgeSets may represent interaction data with reliability estimates as edge weights. The user may want to produce a graph object combining the available data to obtain most reliable edges.

We may want to consider apply type operations to apply an operation across all edgeSets in a multiGraph.

3.2 Use Cases

An important motivator for the *multiGraph* class is the representation of data from protein interaction experiments. Our goal is to represent these data in terms of what interactions were tested, and of those which ones are either positive or negative.

4 Bipartite Graphs

A bipartite graph graph is a graph where the nodes can be divided into two sets, say V_1 and V_2, such that all edges are between members of V_1 and members of V_2 and there are no edges between any two elements of V_1, nor of V_2.