Package ‘graph’

April 5, 2014

Title graph: A package to handle graph data structures

Version 1.40.1

Author R. Gentleman, Elizabeth Whalen, W. Huber, S. Falcon

Description A package that implements some simple graph handling capabilities.

Maintainer Bioconductor Package Maintainer <`maintainer@bioconductor.org`>

License Artistic-2.0

Depends R (>= 2.10), methods

Imports methods, stats, stats4, tools, utils, BiocGenerics (>= 0.1.11)

Suggests SparseM (>= 0.36), XML, RBGL, RUnit, cluster

Enhances Rgraphviz

biocViews GraphsAndNetworks

Collate AllClasses.R AllGenerics.R bitarray.R buildDepGraph.R
methods-graph.R graphNEL.R clustergraph.R NELhandler.R
edgefunctions.R graphfunctions.R GXLformals.R gxlReader.R
random.R write.tlp.R mat2graph.R settings.R zzz.R
standardLabeling.R TODOT.R methods-graphAM.R attrData.R
reverseEdgeDirections.R nodes-methods.R methods-multiGraph.R
test_graph_package.R MultiGraph.R methods-graphBAM.R graph-constructors.R

LazyLoad yes

R topics documented:

`acc-methods` *Methods for Accessibility Lists*

Description

This generic function takes an object that inherits from the `graph` class and a node in that graph and returns a vector containing information about all other nodes that are accessible from the given node. The methods are vectorized so that `index` can be a vector.

Usage

```
## S4 method for signature graph,character
acc(object, index)
## S4 method for signature clusterGraph,character
acc(object, index)
```

Arguments

`object`	An instance of the appropriate graph class.
`index`	A character vector specifying the nodes for which accessibilty information is wanted.

Value

The methods should return a named list of integer vectors. The `names` of the list correspond to the names of the supplied nodes. For each element of the list the returned vector is named. The names of the vector elements correspond to the nodes that are accessible from the given node. The values in the vector indicate how many edges are between the given node and the node in the return vector.

Methods

object = graph An object of class graph.

object = clusterGraph An instance of the `clusterGraph` class.

index A `character` vector of indices corresponding to nodes in the graph.

Examples

```
set.seed(123)
gR3 <- randomGraph(LETTERS[1:10], M<-1:2, p=.5)
acc(gR3, "A")
acc(gR3, c("B", "D"))
```

---

addEdge *addEdge*

Description

A function to add an edge to a graph.

Usage

```
addEdge(from, to, graph, weights)
```

Arguments

`from`	The node the edge starts at
`to`	The node the edge goes to.
`graph`	The graph that the edge is being added to.
`weights`	A vector of weights, one for each edge.

Details

Both `from` and `to` can be vectors. They need not be the same length (if not the standard rules for replicating the shorter one are used). Edges are added to the graph between the supplied nodes.

The `weights` are given for each edge.

The implementation is a bit too oriented towards the `graphNEL` class and will likely change in the next release to accomodate more general graph classes.

If the graph is undirected then the edge is bidirectional (and only needs to be added once). For directed graphs the edge is directional.

Value

A new instance of a graph object with the same class as `graph` but with the indicated edges added.

Author(s)

R. Gentleman

See Also

`addNode`,`removeEdge`, `removeNode`

Examples

```
V <- LETTERS[1:4]
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")

gX <- addEdge("A", "C", gR2, 1)

gR3 <- randomEGraph(letters[10:14], .4)
gY <- addEdge("n", "l", gR3, 1)
```

`addNode` *addNode*

Description

Add one or more nodes to a graph.

Usage

```
addNode(node, object, edges)
```

Arguments

`node`	A character vector of node names.
`object`	A `graph`
`edges`	A named list of edges.

Details

The supplied `nodes` are added to the set of nodes of the `object`.

If `edges` are provided then their must be the same number as there are `nodes` and the must be in the same order. The elements of the `edges` list are vectors. They can be character vectors of node labels for nodes in `object` and if so then they are added with unit weights. If the vector is numeric then it must be named (with labels corresponding to nodes in the `object`) and the values are taken to be the edge weights.

When the `object` is a `distGraph` then the `edges` must be supplied and they must contain appropriate distances for all nodes both those in `object` and those supplied.

Value

A new graph of the same class as `object` with the supplied node added to the set of nodes.

Author(s)

R. Gentleman

See Also

`removeNode`, `removeEdge`, `addEdge`

Examples

```
   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)
   gX <- addNode("X", gR)

set.seed(123)
g1 <- randomGraph(letters[1:10], 1:4, p=.3)
g2 <- addNode("z", g1, edges=list(c("a", "h", "g")))
```

`adj-methods` *Methods for finding the adjacency list for selected nodes.*

Description

This generic function takes an object that inherits from the `graph` class and a node in that graph and returns a vector containing information about all other nodes that are adjacent to the given node. This means that they are joined to the given node by an edge. The accessibility list, `acc` is the list of all nodes that can be reached from a specified node.

Value

The methods return vector of nodes that are adjacent to the specified node.

Methods

object = graph An object that inherits from glass `graph`

index An index (could be multiple) which can be either the integer offset for the node(s) or their labels.

See Also

`acc-methods`

Examples

```
set.seed(123)
gR3 <- randomGraph(LETTERS[1:4], M<-1:2, p=.5)
adj(gR3, "A")
adj(gR3, c(2,3))
```

`apoptosisGraph` *KEGG apoptosis pathway graph*

Description

A graph representing the apoptosis pathway from KEGG, as well as a data.frame of attributes for use in plotting the graph with `Rgraphviz` and a list to compare the nodes with their respective LocusLink IDs.

Usage

```
data(apopGraph)
```

Details

The `apopGraph` data set contains three objects:

The first is `apopGraph`, which is an object of class `graph-NEL` and represents the hsa04210 graph from `KEGG`.

The second is `apopAttrs`, which is a data.frame with two columns, and a row for every node in `apopGraph`. The first column lists what color the node is represented with on the `KEGG` site. The second column lists the type of the node - either `genesym` or `text`. Most nodes are of type `genesym` as they represent genes, but some of the nodes in the `KEGG` graph were not genes and thus those nodes are of type `text`.

The third, `apopLocusLink` is a named list where the names correspond to the node names in `apopGraph`. The values of the list are the LocusLink IDs that correspond to that node in the KEGG graph.

Source

http://www.genome.ad.jp/kegg/pathway/hsa/hsa04210.html

Examples

```
data(apopGraph)
if (require("Rgraphviz") & interactive())
  plot(apopGraph)
```

`attrData-class` *Class "attrData"*

Description

A container class to manage generic attributes. Supports named attributes with default values with methods for vectorized access.

Objects from the Class

Objects can be created by calls of the form `new("attrData", defaults)`. The `defaults` argument should be a named list containing the initial attribute names and default values.

Slots

`data`**:** Where custom attribute data is stored

`defaults`**:** A named list of known attribute names and defualt values.

Methods

attrDataItem<- `signature(self = "attrData", x = "character", attr = "character")`: ...

attrDataItem `signature(self = "attrData", x = "character", attr = "missing")`: ...

attrDataItem `signature(self = "attrData", x = "character", attr = "character")`: ...

attrDefaults<- `signature(self = "attrData", attr = "character", value = "ANY")`: ...

attrDefaults<- `signature(self = "attrData", attr = "missing", value = "list")`: ...

attrDefaults `signature(self = "attrData", attr = "missing")`: ...

attrDefaults `signature(self = "attrData", attr = "character")`: ...

initialize `signature(.Object = "attrData")`: ...

names return the names of the stored attributes

names<- set the names of the stored attributes

removeAttrDataItem `signature(self="attrData", x="character", value="NULL")`: Remove the data associated with the key specified by `x`.

Author(s)

Seth Falcon

Examples

```
defaultProps <- list(weight=1, color="blue", friends=c("Bob", "Alice"))
adat <- new("attrData", defaults=defaultProps)

## Get all defaults
attrDefaults(adat)

## Or get only a specific attribute
attrDefaults(adat, attr="color")

## Update default weight
attrDefaults(adat, attr="weight") <- 500

## Add new attribute
attrDefaults(adat, attr="length") <- 0

## Asking for the attributes of an element you havent customized
## returns the defaults
attrDataItem(adat, x=c("n1", "n2"), attr="length")

## You can customize values
attrDataItem(adat, x=c("n1", "n2"), attr="length") <- 5

## What keys have been customized?
names(adat)
```

`attrDataItem-methods` *Get and set attributes values for items in an attrData object*

Description

The `attrDataItem` method provides get/set access to items stored in a `attrData-class` object.

Usage

```
attrDataItem(self, x, attr)
attrDataItem(self, x, attr) <- value
```

Arguments

`self`	A `attrData-class` instance
`x`	A `character` vector of item names
`attr`	A `character` vector of length 1 giving the attribute name to get/set. Note that the attribute name must have already been defined for the `attrData` object via `attrDefaults`. If missing, return a list of all attributes for the specified nodes.

`value`	An R object to set as the attribute value for the specified items. If the object has length one or does not have a length method defined, it will be assigned to all items in `x`. If the length of `value` is the same as `x`, the corresponding elements will be assigned. We will add an argument to indicate that the `value` is to be taken as-is for those cases where the lengths are the same coincidentally.

`attrDefaults-methods` *Get and set the default attributes of an attrData object*

Description

The `attrDefaults` method provides access to a `attrData-class` object's default attribute list. The default attribute list of a `attrData-class` object defines what attributes can be customized for individual data elements by defining attribute names and default values.

Usage

```
attrDefaults(self, attr)
attrDefaults(self, attr) <- value
```

Arguments

`self`	A `attrData-class` instance
`attr`	A `character` vector of length 1 giving the name of an attribute. Can be missing.
`value`	An R object that will be used as the default value of the specified attribute, or a named list of attribute name/default value pairs if `attr` is missing.

`aveNumEdges` *Calculate the average number of edges in a graph*

Description

aveNumEdges divides the number of edges in the graph by the number of nodes to give the average number of edges.

Usage

```
aveNumEdges(objgraph)
```

Arguments

`objgraph`	the graph object

Value

A double representing the average number of edges will be returned.

Author(s)

Elizabeth Whalen

See Also

`numEdges`, `mostEdges`, `numNoEdges`

Examples

```
set.seed(124)
g1 <- randomGraph(1:10, letters[7:12], p=.6)
aveNumEdges(g1)
```

`biocRepos` *A graph representing the Bioconductor package repository*

Description

This graph is a rendition of the Bioconductor package repository and represents the dependency graph of that repository. An edge between two package denotes a dependency on the 'to' package by the 'from' package.

Usage

```
data(biocRepos)
```

Examples

```
data(biocRepos)
## An example of usage will be here soon
```

`boundary` *Returns the Boundary between a Graph and a SubGraph*

Description

The boundary of a subgraph is the set of nodes in the original graph that have edges to nodes in the subgraph. The function `boundary` computes the boundary and returns it as a list whose length is the same length as the number of nodes in the subgraph.

Usage

```
boundary(subgraph, graph)
```

Arguments

`graph`	the original graph from which the boundary will be created
`subgraph`	can either be the vector of the node labels or the subgraph itself.

Details

The *boundary* of a *subgraph* is the set of nodes in the graph which have an edge that connects them to the specified subgraph but which are themselves not elements of the subgraph.

For convenience users can specify the subgraph as either a graph or a vector of node labels.

Value

This function returns a named list of length equal to the number of nodes in `subgraph`. The elements of the list correspond to the nodes in the `subgraph`. The elements are lists of the nodes in `graph` which share an edge with the respective node in `subgraph`.

Author(s)

Elizabeth Whalen and R. Gentleman

See Also

`subGraph`, `graph-class`

Examples

```
set.seed(123)
g1 <- randomGraph(letters[1:10], 1:4, p=.3)
##both should be "a"
boundary(c("g", "i"), g1)
```

`calcProb` *Calculate the hypergeometric probability of the subgraph's number of edges.*

Description

`calcProb` calculates the probability of having the number of edges found in the subgraph given that it was made from `origgraph`. The hypergeometric distribution is used to calculate the probability (using the pdf).

Usage

```
calcProb(subgraph, origgraph)
```

Arguments

`subgraph`	subgraph made from the original graph
`origgraph`	original graph object from which the subgraph was made

Value

The probability of the subgraph's number of edges is returned.

Author(s)

Elizabeth Whalen

See Also

`calcSumProb`

Examples

```
#none right now
```

`calcSumProb` *Calculate the probability that a subgraph has an unusual number of edges.*

Description

For any graph a set of nodes can be used to obtain an induced subgraph (see `subGraph`). An interesting question is whether that subgraph has an unusually large number of edges. This function computes the probability that a *random* subgraph with the same number of nodes has more edges than the number observed in the presented subgraph. The appropriate probability distribution is the hypergeometric.

Usage

```
calcSumProb(sg, g)
```

Arguments

`sg`	subgraph made from the original graph
`g`	original graph object from which the subgraph was made

Details

The computation is based on the following argument. In the original graph there are n nodes and hence $N = n * (n - 1)/2$ edges in the complete graph. If we consider these N nodes to be of two types, corresponding to those that are either in our graph, g, or not in it. Then we think of the subgraph which has say m nodes and $M = m * (m - 1)/2$ possible edges as representing M draws from an urn containing N balls of which some are white (those in g) and some are black. We count the number of edges in the subgraph and use a Hypergeomtric distribution to ask whether our subgraph is particularly dense.

Value

The probability of having greater than or equal to the subgraph's number of edges is returned.

Author(s)

Elizabeth Whalen

See Also

`calcProb`

Examples

```
set.seed(123)
V <- letters[14:22]
g1 <- randomEGraph(V, .2)

sg1 <- subGraph(letters[c(15,17,20,21,22)], g1)
calcSumProb(sg1, g1)
```

`clearNode` *clearNode*

Description

This function removes all edges to or from the specified node in the graph.

Usage

```
clearNode(node, object)
```

Arguments

`node`	a node
`object`	a graph

Details

All edges to and from `node` are removed. `node` can be a vector.

Value

A new instance of the graph with all edges to and from the specified node(s) removed.

Author(s)

R. Gentleman

See Also

`removeNode`, `removeEdge`

Examples

```
V <- LETTERS[1:4]
edL3 <- vector("list", length=4)
for(i in 1:4)
  edL3[[i]] <- list(edges=(i%%4)+1, weights=i)
names(edL3) <- V
gR3 <- graphNEL(nodes=V, edgeL=edL3, "directed")
g4 <- clearNode("A", gR3)
```

`clusterGraph-class` *Class "clusterGraph"*

Description

A cluster graph is a special sort of graph for clustered data. Each cluster forms a completely connected subgraph. Three are no edges between clusters.

Objects from the Class

Objects can be created by calls of the form `new("clusterGraph", ...)`.

Slots

`clusters`**:** Object of class `"list"` a list of the labels of the elements, one element of the list for each cluster.

Extends

Class `"graph"`, directly.

Methods

- **connComp** `signature(object = "clusterGraph")`: find the connected components; simply the clusters in this case.
- **acc** `signature(object = "clusterGraph")`: find the accessible nodes from the supplied node.
- **adj** `signature(object = "clusterGraph")`: find the adjacent nodes to the supplied node.
- **nodes** `signature(object = "clusterGraph")`: return the nodes.
- **nodes<-** `signature(object="clusterGraph", value="character")`: replace the node names with the new labels given in `value`.
- **numNodes** `signature(object = "clusterGraph")`: return the number of nodes.
- **edgeWeights** Return a list of edge weights in a list format similar to the `edges` method.
- **edgeL** `signature(graph = "clusterGraph")`: A method for obtaining the edge list.
- **coerce** `signature(from = "clusterGraph", to =     "matrix")`: Convert the `clusterGraph` to an adjacency matrix. Currently, weights are ignored. The conversion assumes no self-loops.

Author(s)

R. Gentleman

See Also

`graph-class`, `distGraph-class`

Examples

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

`clusteringCoefficient-methods`

Clustering coefficient of a graph

Description

This generic function takes an object that inherits from the `graph` class. The graph needs to have `edgemode=="undirected"`. If it has `edgemode=="directed"`, the function will return NULL.

Usage

```
## S4 method for signature graph
clusteringCoefficient(object, selfLoops=FALSE)
```

Arguments

`object`	An instance of the appropriate graph class.
`selfLoops`	Logical. If true, the calculation takes self loops into account.

Details

For a node with n adjacent nodes, if `selfLoops` is `FALSE`, the clustering coefficent is N/(n*(n-1)), where N is the number of edges between these nodes. The graph may not have self loops. If `selfLoops` is `TRUE`, the clustering coefficent is N/(n*n), where N is the number of edges between these nodes, including self loops.

Value

A named numeric vector with the clustering coefficients for each node. For nodes with 2 or more edges, the values are between 0 and 1. For nodes that have no edges, the function returns the value NA. For nodes that have exactly one edge, the function returns NaN.

Author(s)

Wolfgang Huber `http://www.dkfz.de/mga/whuber`

Examples

```
set.seed(123)
g1 <- randomGraph(letters[1:10], 1:4, p=.3)
clusteringCoefficient(g1)
clusteringCoefficient(g1, selfLoops=TRUE)
```

`Coercions between matrix and graph representations` *Coercions between matrix and graph representations*

Description

A collection of functions and methods to convert various forms of matrices into graph objects.

Usage

```
aM2bpG(aM)
ftM2adjM(ft, W=NULL, V=NULL, edgemode="directed")
ftM2graphNEL(ft, W=NULL, V=NULL, edgemode="directed")
## S4 method for signature graphNEL,matrix
coerce(from,to="matrix",strict=TRUE)
## S4 method for signature matrix,graphNEL
coerce(from,to="graphNEL",strict=TRUE)
```

Arguments

`ft`	An nx2 matrix containing the `from/to` representation of graph edges.
`W`	An optional vector of edge weights.
`V`	An optional vector of node names.
`aM`	An affiliation matrix for a bipartite graph.

`edgemode`	Character. Specifies if the resulting graph is to be directed or undirected.
`from`	Object to coerce from, either of type `matrix` or `grpahNEL`
`to`	Character giving class to coerce to. Either "matrix" or "graphNEL".
`strict`	Strict object checking.

Details

In the functions `ftM2adjM` and `ftM2graphNEL`, a `from/to` matrix `ft` is converted into an `adjacency` matrix or a `graphNEL` object respectively. In `ft`, the first column represents the `from` nodes and the second column the `to` nodes.

To have unconnected nodes, use the `V` argument (see below). The `edgemode` parameter can be used to specify if the desired output is a directed or undirected graph.

The same edge must not occur twice in the `from/to` matrix. If `edgemode` is `undirected`, the edge `(u,v)` and `(v,u)` must only be specified once.

`W` is an optional vector of edge weights. The order of the edge weights in the vector should correspond to the order of the edges recorded in `ft`. If it is not specified, edge weights of 1 are assigned by default.

`V` is an optional vector of node names. All elements of `ft` must be contained in `V`, but not all names in `V` need to be contained in `ft`. If `V` is not specified, it is set to all nodes represented in `ft`. Specifying `V` is most useful for creating a graph that includes nodes with degree 0.

`aM` is an affiliation matrix as frequently used in social networks analysis. The rows of `aM` represent actors, and the columns represent events. An entry of "1" in the ith row and jth column represents affiliation of the ith actor with the jth event. Weighted entries may also be used. `aM2bpG` returns a `graphNEL` object with nodes consisting of the set of actors and events, and directed (possibly weighted) edges from the actors to their corresponding events. If plotted using `Rgraphviz` and the `dot` layout, the bipartite structure of the graph returned by `aM2bpG` should be evident.

An `adjacency` matrix can be coerced into a `graphNEL` using the `as` method. If the matrix is a symmetric matrix, then the resulting graph will be `undirected`, otherwise it will be `directed`.

Value

For `ftM2graphNEL` and `aM2bpG`, an object of class `graphNEL`. For `ftM2adjM`, a matrix (the adjacency matrix representation).

Author(s)

Denise Scholtens, Wolfgang Huber

Examples

```
## From-To matrix

From <- c("A","A","C","C")
To   <- c("B","C","B","D")
L <- cbind(From,To)
```

```
W  <- 1:4
M1 <- ftM2adjM(L, W, edgemode="directed")
M2 <- ftM2adjM(L, W, edgemode="undirected")
stopifnot(all(M1+t(M1)==M2))

G1 <- ftM2graphNEL(L, W, edgemode="directed")
G2 <- ftM2graphNEL(L, W, edgemode="undirected")

## Adjacency matrix

From <- matrix(runif(100), nrow=10, ncol=10)
From <- (From+t(From)) > pi/4
rownames(From) <- colnames(From) <- LETTERS[1:10]

To <- as(From,"graphNEL")
Back <- as(To,"matrix")

stopifnot(all(From == Back))
```

`combineNodes` *combineNodes*

Description

A function to combine, or collapse, a specified set of nodes in a graph.

Usage

```
combineNodes(nodes, graph, newName, ...)
## S4 method for signature character,graphNEL,character
combineNodes(nodes, graph, newName, collapseFunction=sum)
```

Arguments

`nodes`	A set of nodes that are to be collapsed.
`graph`	The graph containing the nodes
`newName`	The name for the new, collapsed node.
`collapseFunction`	Function or character giving the name of a function used to collapse the edge weights after combining nodes. The default is to sum up the weights, but mean would be a useful alternative.
`...`	Additional arguments for the generic

Details

The nodes specified are reduced to a single new node with label given by `newName`. The in and out edges of the set of nodes are all made into in and out edges for the new node.

Value

An new instance of a graph of the same class as `graph` is returned. This new graph has the specified nodes reduced to a single node.

Author(s)

R. Gentleman

See Also

`inEdges`, `addNode`

Examples

```
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, edgemode="directed")
gR <- addNode("M", gR)
gR <- addEdge("M", "A", gR, 1)
gR <- addEdge("B", "D", gR, 1)
gX <- combineNodes(c("B","D"), gR, "X")

gR <- addNode("K", gR)
gR <- addEdge(c("K","K"), c("D", "B"), gR, c(5,3))
edgeWeights(combineNodes(c("B","D"), gR, "X"))$K
edgeWeights(combineNodes(c("B","D"), gR, "X", mean))$K
```

`DFS` *Depth First Search*

Description

This function implements algorithm 4.2.1 of Gross and Yellen. The input is a `graph` and a `node` to start from. It returns a standard vertex labeling of `graph`. This is a vector with elements corresponding to the nodes of `graph` and with values that correspond to point in the depth first search the node is visited.

Usage

```
DFS(object, node, checkConn=TRUE)
```

Arguments

`object`	An instance of the `graph` class.
`node`	A `character` indicating the starting node.
`checkConn`	A `logical` indicating whether the connectivity of the graph should be checked.

Details

This function implements algorithm 4.2.1 of Gross and Yellen. Specific details are given there.

It requires that the graph be connected. By default, this is checked, but since the checking can be expensive it is optional.

A faster and mostly likely better implementation of depth first searching is given by `dfs` in the **RBGL** package.

Value

A vector with names given by the nodes of `graph` whose values are `0` to one less than the number of nodes. These indices indicate the point at which the node will be visited.

Author(s)

R. Gentleman

References

Graph Theory and its Applications, J. Gross and J. Yellen.

See Also

`boundary`

Examples

```
RNGkind("Mersenne-Twister")
set.seed(123)
g1 <- randomGraph(letters[1:10], 1:4, p=.3)
RNGkind()
DFS(g1, "a")
```

`distGraph-class` *Class "distGraph"*

Description

A class definition for graphs that are based on distances.

Objects from the Class

Objects can be created by calls of the form `new("distGraph", ...)`.

Slots

`Dist`: Object of class `"dist"` that forms the basis for the edge weights used in the `distGraph`.

Extends

Class `"graph"`, directly.

Methods

show `signature(object = "distGraph")`: a print method

Dist `signature(object = "distGraph")`: return the dist object.

adj `signature(object = "distGraph")`: find the nodes adjacent to the supplied node.

nodes `signature(object = "distGraph")`: return the nodes in the graph.

numNodes `signature(object = "distGraph")`: return the number of nodes.

threshold `signature(object = "distGraph", k, value)`: set all distances that are larger than the supplied threshold, `k`, to the supplied value. The default is value is zero (and so is appropriate for similarities, rather than distances).

initialize `signature(object = "distGraph")`: initialize a `distGraph` instance.

edgeWeights Return a list of edge weights in a list format similar to the `edges` method.

edgeL `signature(graph = "distGraph")`: A method for obtaining the edge list.

Author(s)

R. Gentleman

References

Shamir's paper and Butte et al

See Also

`graph-class`, `clusterGraph-class`

Examples

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

`duplicatedEdges` *duplicatedEdges*

Description

A multigraph is a graph where edges between nodes can be represented several times. For some algorithms this causes problems. `duplicatedEdges` tests an instance of the `graphNEL` class to see if it has duplicated edges and returns `TRUE` if it does and `FALSE` otherwise.

Usage

```
duplicatedEdges(graph)
```

Arguments

`graph`	An instance of the class `graphNEL`

Details

It would be nice to handle other types of graphs.

Value

A logical, either `TRUE` if the graph has duplicated edges or `FALSE` it not.

Author(s)

R. Gentleman

See Also

`connComp`, `ugraph`

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
```

`edgeData-methods` *Get and set attributes for the edges of a graph object*

Description

Attributes of the edges of a graph can be accessed using `edgeData`. The attributes must be defined using `edgeDataDefaults`. You can ommit the `from` or `to` argument to retrieve attribute values for all edges to (respectively, from) a given node.

Usage

```
edgeData(self, from, to, attr)
edgeData(self, from, to, attr) <- value
```

Arguments

`self`	A `graph-class` instance
`from`	A `character` vector of node names
`to`	A `character` vector of node names
`attr`	A `character` vector of length one specifying the name of a node attribute
`value`	An R object to store as the attribute value

`edgeDataDefaults-methods` *Get and set default attributes for the edges of a graph*

Description

Set default values for attributes associated with the edges of a graph.

Usage

```
edgeDataDefaults(self, attr)
edgeDataDefaults(self, attr) <- value
```

Arguments

`self`	A `graph-class` instance
`attr`	A `character` vector of length one giving the name of the attribute
`value`	An R class to use as the default value for the specified attribute

---

`edgeMatrix` *Compute an Edge Matrix or weight vector for a Graph*

Description

For our purposes an *edge matrix* is a matrix with two rows and as many columns as there are edges. The entries in the first row are the index of the node the edge is *from*, those in the second row indicate the node the edge is *to*.

If the graph is “undirected” then the `duplicates` option can be used to indicate whether reciprocal edges are wanted. The default is to leave them out. In this case the notions of *from* and *to* are not relevant.

Usage

```
edgeMatrix(object, duplicates=FALSE)
eWV(g, eM, sep = ifelse(edgemode(g) == "directed", "->",
                 "--"), useNNames=FALSE)
pathWeights(g, p, eM=NULL)
```

Arguments

`object`	An object that inherits from `graph`.
`g`	An object that inherits from `graph`.
`duplicates`	Whether or not duplicate edges should be produced for “undirected” graphs.
`eM`	An edge matrix
`sep`	a character string to concatenate node labels in the edge label
`useNNames`	a logical; if TRUE, node names are used in the edge label; if FALSE, node indices are used
`p`	a vector of node names constituting a path in graph g
`...`	arguments passed to `edgeMatrix`.

Details

Implementations for `graphNEL`, `clusterGraph` and `distGraph` are available.

Value

`edgeMatrix` returns a matrix with two rows, *from* and *to*, and as many columns as there are edges. Entries indicate the index in the node vector that corresponds to the appropriate end of the edge.

`eWV` uses the edge matrix to create an annotated vector of edge weights.

`pathWeights` returns an annotated vector of edge weights for a specified path in a graph.

Note

A path through an undirected graph may have several representations as a named vector of edges. Thus in the example, when the weights for path b-a-i are requested, the result is the pair of weights for edges a–b and a–i, as these are the edge labels computed for graph `g1`.

Author(s)

R. Gentleman

See Also

`edges`

Examples

```
set.seed(123)
g1 <- randomGraph(letters[1:10], 1:4, p=.3)
edgeMatrix(g1)
g2 <- new("clusterGraph", clusters=list(a=c(1,2,3), b=c(4,5,6)))
em2 <- edgeMatrix(g2)
eWV(g1, edgeMatrix(g1))
eWV(g1, edgeMatrix(g1), useNNames=TRUE)
pathWeights(g1, c("b", "a", "i"))
```

`edgeSets` *MultiGraph edgeSet data*

Description

C57BL/6J and C3H/HeJ mouse strains exhibit different cardiovascular and metabolic phenotypes on the hyperlipidemic apolipoprotein E (Apoe) null background. The interaction data for the genes from adipose, brain, liver and muscle tissue samples from male and female mice are included as a list of `data.frames`. Each `data.frame` contains information for the `from-gene`, `to-gene` and the strength of interaction (`weight`) for each of the tissues studied.

Usage

```
data(esetsFemale)
data(esetsMale)
```

Source

Sage Commons Repository `http://sagebase.org/commons/dataset1.php#UCLA1`

Examples

```
data(esetsFemale)
data(esetsMale)
```

`edgeWeights` *Retrieve the edge weights of a graph*

Description

A generic function that returns the edge weights of a graph. If `index` is specified, only the weights for the edges from the specified nodes are returned. The user can control which edge attribute is interpreted as the weight, see the Details section.

Usage

```
edgeWeights(object, index, ..., attr = "weight", default = 1, type.checker = is.numeric)
```

Arguments

`object`	A graph, any object that inherits from the `graph` class.
`index`	If supplied, a character or numeric vector of node names or indices.
`...`	Unused.
`attr`	The name of the edge attribute to use as a weight. You can view the list of defined edge attributes and their default values using `edgeDataDefaults`. The default attribute name is `"weight"`, see the Details section.
`default`	The value to use if `object` has no edge attribute named by the value of `attr`. The default is the value 1 (double).
`type.checker`	A function that will be used to check that the edge weights are of the correct type. This function should return TRUE if the input vector is of the right type and FALSE otherwise. The default is to check for numeric edge weights using `is.numeric`. If no type checking is desired, specify `NULL`.

Details

If `index` is suppled, then edge weights from these nodes to all adjacent nodes are returned. If `index` is not supplied, then the edge weights for all nodes are returned. The value for nodes without any outgoing edges will be a zero-length vector of the appropriate mode.

The `edgeWeights` method is a convenience wrapper around `edgeData`, the general-purpose way to access edge attribute information for a `graph` instance. In general, edge attributes can be arbitary R objects. However, for `edgeWeights` to make sense, the values must be vectors of length not more than one.

By default, `edgeWeights` looks for an edge attribute with name `"weight"` and, if found, uses these values to construct the edge weight list. You can make use of attributes stored under a different name by providing a value for the `attr` argument. For example, if `object` is a graph instance with an edge attribute named `"WTS"`, then the call `edgeWeights(object, attr="WTS")` will attempt to use those values.

The function specified by `type.checker` will be given a vector of edge weights; if the return value is not `TRUE`, then an error will be signaled indicating that the edge weights in the graph are not of the expected type. Type checking is skipped if `type.checker` is `NULL`.

If the graph instance does not have an edge attribute with name given by the value of the `attr` argument, `default` will be used as the weight for all edges. Note that if there is an attribute named by `attr`, then its default value will be used for edges not specifically customized. See `edgeData` and `edgeDataDefaults` for more information.

Because of their position after the `...`, no partial matching is performed for the arguments `attr`, `default`, and `type.checker`.

Value

A named list of named edge weight vectors. The names on the list are the names of the nodes specified by `index`, or all nodes if `index` was not provided. The names on the weight vectors are node names to identify the edge to which the weight belongs.

Author(s)

R. Gentleman and S. Falcon

See Also

`nodes` `edges` `edgeData` `edgeDataDefaults` `is.numeric` `is.integer` `is.character`

Examples

```
V <- LETTERS[1:4]
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")
edgeWeights(gR2, "C")
edgeWeights(gR2)
edgeWeights(gR2, attr="foo", default=5)
edgeData(gR2, attr="weight")
edgeData(gR2, from="C", attr="weight")
```

`fromGXL-methods` *Methods for GXL manipulations in package graph*

Description

GXL `http://www.gupro.de/GXL` is "an XML sublanguage designed to be a standard exchange format for graphs". This document describes tools in the graph package for importing GXL data to R and for writing graph data out as GXL.

Value

`fromGXL`	currently returns a graphNEL when possible. This function is based on `xmlEventParse` with handlers defined in the function NELhandler. The dump() element of this handler should emit information on all children of nodes and edges; the asGraphNEL() element will return a `graphNEL` object with weights if child `<attr>` with name attribute "weights" is present for each edge element.
`toGXL`	for an input of class "graphNEL", returns an object of class c("XMLInternalDOM", "XMLOutputStream"); see the example for how to convert this to a text stream encoding XML
`dumpGXL`	returns an R list with all the node, edge, and named attribute information specified in the GXL stream
`validateGXL`	returns silently (invisibly returns the parsed tree) for a DTD-compliant stream, or is otherwise very noisy

Methods

fromGXL con = connection: returns a graphNEL based on a parsing of the GXL stream on the connection

dumpGXL con = connection: returns an R list based on a parsing of the GXL stream on the connection

validateGXL con = connection: checks the GXL stream against its DTD

toGXL object = graphNEL: creates an XMLInternalDOM representing the graph in GXL

Note

At present, toGXL does not return a validating GXL stream because XML package does not properly handle the dtd and namespaces arguments to xmlTree. This is being repaired. To fix the stream, add `<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.1.dtd">` as second record in the output.

Some structures in a graphNEL and some tags in GXL may not be handled at this time.

Author(s)

Vince Carey <stvjc@channing.harvard.edu>

Examples

```
sf <- file(system.file("GXL/simpleExample.gxl", package="graph"))
show(fromGXL(sf))
print(dumpGXL(sf))
close(sf)
#validateGXL(sf)
# bad <- file(system.file("GXL/c2.gxl", package="graph"))
# heres how you can check if the GXL is well-formed, if
# you have a libxml2-based version of R XML package
#
# try( validateGXL(bad) )
#
```

```
gR <- graphNEL(nodes=letters[1:4], edgeL=list(
 a=list(edges=4), b=list(edges=3), c=list(edges=c(2,1)), d=list(edges=1)),
 edgemode="directed")
#
# following requires that you are using XML bound with recent libxml2
#
#an <- as.numeric
#if (an(libxmlVersion()$major)>=2 && an(libxmlVersion()$minor)>=4)
## since toGXL returns an XML object, we need to attach the XML
## package.
library("XML")
cat(saveXML(toGXL(gR)$value()))
wtd <- file(system.file("GXL/kmstEx.gxl", package="graph"))
wtdg <- fromGXL(wtd)
close(wtd)
print(edgeWeights(wtdg))
```

`graph-class` *Class "graph"*

Description

A virtual class that all graph classes should extend.

Details

`degree` returns either a named vector (names correspond to the nodes in the graph) containing the degree for undirected graphs or a list with two components, `inDegree` and `outDegree` for directed graphs.

`connComp` returns a list of the connected components. Each element of this list contains the labels of all nodes in that component.

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. This is the default for `connComp`. A *digraph* is *strongly connected* if every two vertices are mutually reachable. A *strongly connected* component of a *digraph*, **D**, is a maximal *strongly connected* subdigraph of **D**. See the **RBGL** package for an implementation of Trajan's algorithm to find *strongly connected* components (`strongComp`).

In the **graph** implementation of `connComp` *weak connectivity* is used. If the argument to `connComp` is a directed graph then `ugraph` is called to create the underlying undirected graph and that is used to compute connected components. Users who want different behavior are encouraged to use **RBGL**.

Objects from the Class

A virtual Class: No objects may be created from it.

Slots

`edgeData`**:** An `attrData` instance for edge attributes.

`nodeData`**:** An `attrData` instance for node attributes.

`graphData`**:** A `list` for graph-level attributes. Only mandatory list item is `edgemode` which indicates whether edges are `"directed"` or `"undirected"`

`renderInfo`**:** A `list` of graph rendering information.

Methods

nodes return a character vector containing the names of the nodes of the graph

nodes<- rename the nodes of the graph

show `signature(object = "graph")`:A print method for the graph.

acc `signature(object = "graph")`: find all nodes accessible from the specified node.

complement `signature(x = "graph")`: compute the complement of the supplied graph. The complement is defined with respect to the complete graph on the nodes in `x`. Currently this returns an object of class `graphNEL`.

connComp `signature(object = "graph")`: find the connected components of a graph.

degree `signature(object = "graph", Nodes = "missing")`: find the `degree` of a node (number of coincident edges).

degree `signature(object = "graph", Nodes = "ANY")`: as above.

degree `signature(object = "MultiGraph", Nodes = "missing")`: find the `degree` of a node (number of coincident edges).

dfs `signature(object = "graph")`: execute a depth first search on a graph starting with the specified node.

edges `signature(object="graph", which="character")`: return the edges indicated by `which`. `which` can be missing in which case all edges are returned or it can be a character vector with the node labels indicating the nodes whose edge lists are wanted.

edgeDataDefaults Get and set default attributes for the edges in the graph.

edgeData Get and set attributes for edges in the graph

edgemode `signature(object="graph")`: return the `edgemode` for the graph. Currently this can be either `directed` or `undirected`.

edgemode<- `signature(object="graph",value="character")`: set the `edgemode` for the graph. Currently this can be either `directed` or `undirected`.

edgeWeights Return a list of edge weights in a list format similar to the `edges` method.

intersection `signature(x = "graph", y = "graph")`: compute the intersection of the two supplied graphs. They must have identical nodes. Currently this returns an object of class `graphNEL`. With edge weights of 1 for any matching edge.

isAdjacent `signature(from="character", to="character")`: Determine if edges exists between nodes.

isConnected `signature(object = "graph")`: A boolean that details if a graph is fully connected or not.

isDirected Return `TRUE` if the graph object has directed edges and `FALSE` otherwise.

join `signature(x = "graph", y = "graph")`: returns the joining of two graphs. Nodes which are shared by both graphs will have their edges merged. Note that edgeWeights for the resulting graph are all set to 1. Users wishing to preserve weights in a join operation must perform addEdge operations on the resulting graph to restore weights.

nodes<- A generic function that allows different implementations of the `graph` class to reset the node labels

nodeDataDefaults Get/set default attributes for nodes in the graph.

nodeData Get/set attributes for nodes in the graph.

numEdges `signature(object = "graph")`: compute the number of edges in a graph.

numNodes `signature(object = "graph")`: compute the number of nodes in a graph.

plot Please see the help page for the `plot,graph-method` method in the `Rgraphviz` package

union `signature(x = "graph", y = "graph")`: compute the union of the two supplied graphs. They must have identical nodes. Currently this returns an object of class `graphNEL`.

edgeNames `signature(object = "graph")`: Returns a vector of the edge names for this graph, using the format `tail~head`, where `head` is the name of the tail node and `head` is the name of the head node.

updateGraph `signature(object = "graph")`: Updates old instances of graph objects.

Author(s)

R. Gentleman and E. Whalen.

References

Graph Theory and its Applications, J. Gross and J. Yellen.

See Also

`graphNEL-class`, `graphAM-class`, `distGraph-class`.

Examples

```
set.seed(123)
g1 <- randomGraph(letters[1:10], 1:4, p= 0.3)
numEdges(g1)
edgeNames(g1)
edges(g1)
edges(g1, c("a","d")) # those incident to a or d
```

`graph-defunct` *Defunct Functions in Package* **graph**

Description

The functions or variables listed here are no longer part of the graph package.

Usage

```
buildRepDepGraph()
pkgInstOrder()
ugraphOld()
```

See Also

`Defunct`

`graph.par` *Graphical parameters and other settings*

Description

Functions providing an interface to persistent graphical parameters and other settings used in the package.

Usage

```
graph.par(...)
graph.par.get(name)
```

Arguments

`...`	either character strings naming parameters whose values are to be retrieved, or named arguments giving values that are to be set.
`name`	character string, giving a valid parameter name.

Details

`graph.par` works sort of like `par`, but the details are yet to be decided.

`graph.par.get(name)` is equivalent to `graph.par(name)[[1]]`

Value

In query mode, when no parameters are being set, `graph.par` returns a list containing the current values of the requested parameters. When called with no arguments, it returns a list with all parameters. When a parameter is set, the return value is a list containing previous values of these parameters.

Author(s)

Deepayan Sarkar, `<deepayan.sarkar@r-project.org>`

See Also

`par`

`graph2SparseM` *Coercion methods between graphs and sparse matrices*

Description

These functions provide coercions between objects that inherit from the `graph` class to sparse matrices from the `SparseM` package.

Usage

```
graph2SparseM(g, useweights=FALSE)
sparseM2Graph(sM, nodeNames, edgemode=c("directed", "undirected"))
```

Arguments

`g`	An instance of the `graph` class.
`useweights`	A logical value indicating whether to use the edge weights in the graph as values in the sparse matrix.
`sM`	A sparse matrix.
`nodeNames`	A `character` vector of the node names.
`edgemode`	Specifies whether the graph to be created should have directed (default) or undirected edges. If undirected, the input matrix `sM` must be symmetric.

Details

A very simple coercion from one representation to another.

Currently it is presumed that the matrix is square. For other graph formats, such as bipartite graphs, some improvements will be needed; patches are welcome.

Value

`graph2SparseM` takes as input an instance of a subclass of the `graph` class and returns a sparse matrix.

`sparseM2Graph` takes a sparse matrix as input and returns an instance of the `graphNEL` class. By default, the `graphNEL` returned will have directed edges.

Author(s)

R. Gentleman

See Also

`graph-class`, `graphNEL-class`, and for other conversions, `aM2bpG` and `ftM2adjM`

Examples

```
set.seed(123)
g1 <- randomGraph(letters[1:10], 1:4, p=.3)
s1 <- graph2SparseM(g1, useweights=TRUE)
g2 <- sparseM2Graph(s1, letters[1:10], edgemode="undirected")
## consistency check
stopifnot(all.equal(g1, g2))
```

`graphAM-class` *Class "graphAM"*

Description

A graph class where node and edge information is represented as an adjacency matrix. The adjacency matrix is square and element `adjMat[i, j]` is one if there is an edge from node i to node j and zero otherwise.

Details

The non-zero matrix values can be used to initialize an edge attribute. If this is desired, use the `values` argument in the call to `new` and provide a list with a single named element. The name determines the attributes and the value provides the default value for that attribute.

Objects from the Class

Objects can be created by calls of the form `graphAM(adjMat, edgemode, values)`.

Slots

`adjMat`: An adjacency `"matrix"` describing the graph structure. The `colnames` of the matrix will be used as node names for the graph if present.

`edgeData`: Storage for edge attributes.

`nodeData`: Storage for node attributes.

Extends

Class `"graph"`, directly.

Constructor

`graphAM(adjMat=matrix(integer(), 0, 0), edgemode=undirected, values=NA)` creates a graphAM instance.

adjMat An `integer` matrix specifying which nodes have edges between them.

edgemode Either "directed" or "undirected".

values A named list of length 1, used (rather obscurely) to specify that non-zero adjMat values initialize an edge attribute. The name of the single element in that list becomes the name of that attribute, with the specified default value. This default value is, however, never used: the specified edge attribute always has the value contained in the adjacency matrix, which is traditionally 1, but can be any positive number.

Methods

addEdge `signature(from = "character", to = "character", graph = "graphAM", weights = "missing")`: ...

addNode `signature(object = "graphAM", nodes = "character")`: ...

clearNode `signature(node = "character", object = "graphAM")`: ...

coerce `signature(from = "graphAM", to = "graphNEL")`: ...

coerce `signature(from = "graphAM", to = "graphBAM")`: ...

coerce `signature(from = "graphAM", to = "matrix")`: In converting to a `matrix`, if an edge attribute named `"weight"` is defined, the non-zero elements of the matrix will contain the corresponding attribute value. For more flexible matrix conversion, see `toMatrix`.

coerce `signature(from = "matrix", to = "graphAM")`: This coerce method exists for symmetry. In most cases, creating a new `graphAM` instance using `new` gives one more control over the resulting graph.

edges `signature(object = "graphAM", which = "missing")`: ...

edges `signature(object = "graphAM", which = "character")`: ...

initialize `signature(.Object = "graphAM")`: ...

inEdges `signature(node = "character", object = "graphNEL")`: Return the incoming edges for the specified nodes. See `inEdges`.

isAdjacent `signature(object = "graphAM", from = "character", to = "character")`: ...

nodes<- `signature(object = "graphAM", value = "character")`: ...

nodes `signature(object = "graphAM")`: ...

numEdges `signature(graph = "graphAM")`: ...

numNodes `signature(object = "graphAM")`: ...

removeEdge `signature(from = "character", to = "character", graph = "graphAM")`: ...

removeNode `signature(node = "character", object = "graphAM")`: ...

Author(s)

Seth Falcon

See Also

graph-class, graphNEL-class

Examples

```
mat <- rbind(c(0, 0, 1, 1),
             c(0, 0, 1, 1),
             c(1, 1, 0, 1),
             c(1, 1, 1, 0))
rownames(mat) <- colnames(mat) <- letters[1:4]
g1 <- graphAM(adjMat=mat)
stopifnot(identical(mat, as(g1, "matrix")), validObject(g1))

## now with weights:
mat[1,3] <- mat[3,1] <- 10
gw <- graphAM(adjMat=mat, values=list(weight=1))

## consistency check:
stopifnot(identical(mat, as(gw, "matrix")),
          validObject(gw),
          identical(gw, as(as(gw, "graphNEL"), "graphAM")))
```

`graphBAM-class` *EXPERIMENTAL class "graphBAM"*

Description

The graphBAM class represents a graph as an adjacency matrix. The adjacency matrix is stored as a bit array using a `raw` vector to reduce the memory footprint and speed operations like `graphIntersection`. This class is EXPERIMENTAL and its API is subject to change.

Usage

```
graphBAM(df, nodes=NULL, edgemode="undirected", ignore_dup_edges = FALSE)
```

Arguments

`df`	A `data.frame` with three columns: "from", "to" and "weight". Columns "from" and "to" can be either factors or character vectors. Each row of `df` describes an edge in the resulting graph. The "weight" column must be numeric.
`nodes`	A character vector of node labels. Use this to add degree zero nodes to the graph. If `NULL`, the set of nodes found in `from` and `to` will be used.
`edgemode`	A string, one of "directed" or "undirected".
`ignore_dup_edges`	If `FALSE` (default), specifying duplicate edges in the input is an error. When set to `TRUE` duplicate edges are ignored. Edge weight values are ignored when determining duplicates. This is most useful for graph import and conversion.

Constructors

The `GraphBAM` function is used to create new `graphBAM` instances. Edges are specified in a `data.frame`. For undirected graphs, reciprical edges should not be includes unless `ignoe_dup_edges` is `TRUE`.

Extends

Class `"graph"`, directly.

Methods for graphBAM objects

`addEdge(from, to, graph, weights)` Return a new `graphBAM` object with the specified edge(s) added. The `from` and `to` arguments must either be the same length or one of them must be of length one. Each time an edge is added, the entire graph is copied. For the purpose of building a graph it will often be more efficient to build up the list of edges and call `GraphBAM`.

`addNode(node, object)` Return a new `graphBAM` object with the specified node(s) added.

`clearNode(node, object)` This operation is not currently supported.

`edges(object, which)` Returns an adjacency list representation of the graph. The list will have an entry for each node with a vector of adjacent node labels or `character(0)`. For undirected graphs, `edges` returns the reciprocal edges. The optional argument `which` can be a character vector of node labels. When present, only entries for the specified nodes will be returned.

`inEdges(node, object)` (Not yet supported) Similar to the `edges` function, but the adjacency list maps nodes that have an edge to the given node instead of from the given node.

`isAdjacent(object, from, to)` Returns a logical vector indicating whether there is an edge corresponding to the elements in `from` and `to`. These vectors must have the same length, unless one has length one.

`nodes(object)` Return the node labels for the graph

`numEdges(object)` Returns the number of edges in the graph.

`numNodes(object)` Returns the number of nodes in the graph

`removeEdge(from, to, graph)` Return a new `graphBAM` object with the specified edges removed. The `from` and `to` arguments must be the same length unless one of them has length one.

`removeNode(node, object)` Returns a new `graphBAM` object with the specified node removed. Node and edge attributes corresponding to that node are also removed.

`edgeData(self, from, to, attr)` Access edge attributes. See help for `edgeData`.

`edgeDataDefaults(self, attr)` Access edge data default attributes .

`nodeDataDefaults(self, attr)` Access node data default attributes .

`edgeWeights(object, index)` Return the edge weights for the graph in adjacency list format. The optional argument `index` specified a character vector of nodes. In this case, only the weights for the specified nodes will be returned.

`extractFromTo(g)` Returns a data frame with column names "from", "to", and "weight" corresponding to the connected nodes in the graphBAM object.

`graphIntersect(x, y, nodeFun, edgeFun)` When given two `graphBAM` objects, `graphIntersect` returns a new `graphBAM` containing the nodes and edges in common between the two graphs. Both x and y should either be directed or undirected. The intersection is computed by first finding the intersection of the node sets, obtaining the resulting subgraphs, and finding the intersection of the resulting edge sets. Node/Edge attributes that are equal are carried over to the result. Non equal edge/node attributes will result in the corresponding attribute being set to NA. The user has the option of providing a named list of functions correspoding to the names of the edge attributes for resolving conflicting edge attributes. For resolving any of the conflicting node attributes the user has the option of providing a named `list` of functions corresponding to the node attribute names.

`graphUnion(x, y, nodeFun, edgeFun)` When given two `graphBAM` objects, `graphUnion` returns a new `graphBAM` containing the union of nodes and edges between the two graphs. The union is compted by first finding the union of the nodesets. Both x and y should be either directed or undirected. Node/Edge attributes that are equal are carried over to the result. Non equal edge/node attributes will result in the corresponding attribute being set to NA. The user has the option of providing a named list of functions correspoding to the names of the edge attributes for resolving conflicting edge attributes. For resolving any of the conflicting node attributes the user has the option of providing a named `list` of functions corresponding to the node attribute names.

`edgemode(object) <- value` Set the edgemode for the graph ("directed" or "undirected"). If the specified edgemode is the same, the object is returned without changes. Otherwise, a directed graph is converted to an undirected graph via `ugraph` and an undirected graph is returned such that each edge is interpreted as two edges, one in each direction.

`ugraph(graph)` Return an undirected version of the current graph. Conceptually, the arrows of a graph's directed edges are removed.

`nodes(object) <- value` Replacement of a `graphBAM` object's node labels is currently not supported. An error is raised if this method is called.

Coercion

`graphBAM` objects can be coerced to `graphAM`, `graphNEL`, and `matrix` instances via `as(g, CLASS)`.

Author(s)

N. Gopalakrishnan, S. Falcon

Examples

```
f <- c("a", "a", "b", "c", "d")
t <- c("b", "c", "c", "d", "a")
weight <- c(2.3, 2.3, 4.3, 1.0, 3.0)
df <- data.frame(from=f, to=t, weight= weight)
g <- graphBAM(df)
nd <- nodes(g)
nodeDataDefaults(g, attr ="color") <- "green"
nodeData(g,n=c("b", "c"), attr ="color") <- "red"
w1 <- edgeWeights(g)
w2 <- edgeWeights(g,"a")
w3 <- edgeWeights(g,1)
d1 <- edges(g)
d2 <- edges(g,c("a", "b"))
e1 <- edgeData(g)
```

```
e2 <- edgeData(g, "a", "c",attr="weight")
em <- edgeMatrix(g)
id <- isDirected(g)
sg <- subGraph(c("a","c","d"), g)
ft <- extractFromTo(g)
am <- as(g,"graphAM")
nl <- as(g,"graphNEL")
mt <- as(g,"matrix")
k  <- graphIntersect(g,g)
k <- graphUnion(g,g)
e <- removeEdgesByWeight(g,lessThan= 3.0)
f <- removeNode("a", g)
g
```

`graphExamples` *A List Of Example Graphs*

Description

This data set contains a list of example `graphNEL` objects, which can then be used for plotting.

Usage

```
data(graphExamples)
```

Source

Various sources, primarily from `randomGraph` and `randomEGraph`

Examples

```
data(graphExamples)
a <- graphExamples[[1]]
a
```

`graphNEL-class` *Class "graphNEL"*

Description

This is a class of graphs that are represented in terms of nodes and an edge list. This is a suitable representation for a graph with a large number of nodes and relatively few edges.

Details

The `graphNEL` class provides a very general structure for representing graphs. It will be reasonably efficient for lists with relatively more nodes than edges. Although this representation can support multi-edges, such support is not implemented and instances of `graphNEL` are assumed to be simple graphs with at most one edge between any pair of nodes.

The `edgeL` is a named `list` of the same length as the node vector. The names are the names of the nodes. Each element of `edgeL` is itself a list. Each element of this (sub)list is a vector (all must be the same length) and each element represents an edge to another node. The sublist named `edges` holds index values into the node vector. And each such entry represents an edge from the node which has the same name as the component of `edgeL` to the node with index provided. Another component that is often used is named `weights`. It represents edge weights. The user can specify any other edge attributes (such as types etc). They are responsible for any special handling that these might require.

For an `undirected` instance all edges are reciprocated (there is an edge from A to B and from B to A).

Note that the reason for using indices to represent the `to` end of a node is so that we can easily support permutation of the node labels as a way to generate randomizations of the graph.

Slots

- `nodes`**:** Object of class `"vector"`.
- `edgeL`**:** Object of class `"list"`. The `edgeL` must be the same length as `nodes`. The elements of this vector correspond to the same element in `nodes`. The elements are themselves lists. If the node has any edges then this list will have an element named `edges`. This will eventually change. Since edge weights are now stored in the edge attributes construct, we do not need the extra level of list.

Extends

Class `"graph"`, directly.

Constructor

`graphNEL(nodes=character(), edgeL=list(), edgemode=undirected)` creates a graphNEL instance.

- **nodes** A character vector of node labels.
- **edgeL** A named list either in the format returned by the `edges` method or a list of lists where each inner list has an element named `edges` and optionally an element named `weights`. If `weights` is present, it must be the same length as the `edges` element.
- **edgemode** Either "directed" or "undirected".

Methods

- **adj** `signature(object = "graphNEL")`: A method for finding nodes adjacent to the suplied node.
- **edgeL** `signature(graph = "graphNEL")`: A method for obtaining the edge list.
- **edgeWeights** `signature(object = "graphNEL")`: A method for obtaining the edge weights.

edges `signature(object = "graphNEL")`: A method for obtaining the edges.

inEdges `signature(node = "character", object = "graphNEL")`: Return the incoming edges for the specified nodes. See `inEdges`.

nodes `signature(object = "graphNEL")`: A method for obtaining the nodes.

numNodes `signature(object = "graphNEL")`:A method for determining how many nodes are in the graph.

subGraph `signature(snodes="character", graph = "graphNEL")`:A method for obtaining the induced subgraph based on the set of supplied nodes and the supplied graph.

plot Please see the help page for `plot.graphNEL` in the `Rgraphviz` package

graph2graphviz `signature(object = "graphNEL")`: A method that will convert a `graphNEL` object into a matrix suitable for interaction with `Rgraphviz`. Not intended to be called directly. This function will insure that no NA's (or other undesired values) are in the graph, or created by coersion.

nodes<- `signature(object="graphNEL",value="character")`: A method for replacing the nodes in a graph object. It checks to be sure the values are the right length and unique.

coerce `signature(from = "graphNEL", to = "graphAM")`: Called via `as`, the method converts to an adjacency matrix representation. See `graphAM-class`.

coerce `signature(from = "graphNEL", to = "graphBAM")`: Called via `as`, the method converts to an bit array representation. See `graphBAM-class`.

Author(s)

R. Gentleman

See Also

`graphAM-class`, `distGraph-class`, `clusterGraph-class`

Examples

```
set.seed(123)
V <- LETTERS[1:4]
edL <- vector("list", length=4)
names(edL) <- V
for(i in 1:4)
   edL[[i]] <- list(edges=5-i, weights=runif(1))
gR <- graphNEL(nodes=V, edgeL=edL)
edges(gR)
edgeWeights(gR)
```

`inEdges` *Generic Method inEdges*

Description

Returns a list of all incoming edges for the specified nodes.

Usage

```
inEdges(node, object)
```

Arguments

`node`	character vector of node names
`object`	a `graph` object

Details

If no `node` argument is specified, `inEdges` returns the incoming edges for all nodes in the graph.

For an undirected graph, `inEdges` returns all edges for the specified nodes.

Value

A list with length matching the length of `node`. If `node` was missing, a list containing an element for each node in the graph.

Each list element contains a character vector of node names giving the nodes that have outgoing edges to the node given by the name of the list element.

Author(s)

R. Gentleman

See Also

`removeNode`, `clearNode`

Examples

```
V <- LETTERS[1:4]
edL3 <- vector("list", length=4)
for(i in 1:4)
  edL3[[i]] <- list(edges=(i%%4)+1, weights=i)
names(edL3) <- V
gR3 <- graphNEL(nodes=V, edgeL=edL3, "directed")
inEdges(c("A", "B"), gR3)
```

`integrinMediatedCellAdhesion`

KEGG Integrin Mediated Cell Adhesion graph

Description

A graph representing the integrin-mediated cell adhesion pathway from KEGG, as well as a list of attributes for use in plotting the graph with `Rgraphviz`.

Usage

```
data(integrinMediatedCellAdhesion)
```

Details

The `integrinMediatedCellAdhesion` data set contains two objects:

The first is `IMCAGraph`, which is an object of class `graph-NEL` and represents the hsa04510 graph from `KEGG`.

The second is `IMCAAttrs`, which is a list of four elements. The first element, `defAttrs` corresponds to the `attrs` arguments of `agopen` and `plot.graph`. The second element is `nodeAttrs` which corresponds to the `nodeAttrs` argument in the same two functions from `Rgraphviz`. The third element, `subGList` corresponds to the `subGList` argument in those functions. Lastly, the fourth element, `LocusLink` provides a named list where the names are the nodes and the values are vectors of LocusLink ID values which correspond to those nodes.

The values from `defAttrs`, `nodeAttrs` and `subGList` in the `IMCAAttrs` list are part of an ongoing attempt by Bioconductor to provide the set of options to most accurately recreate the actual visual image of the pathway from the KEGG site using `Rgraphviz`. Users may try out their own combination of attributes and settings for their own needs, but these represent our own efforts at as closely recreating the image as possible.

Source

`http://www.genome.ad.jp/kegg/pathway/hsa/hsa04510.html`

Examples

```
data(integrinMediatedCellAdhesion)
if (require("Rgraphviz") & interactive())
  plot(IMCAGraph, attrs=IMCAAttrs$defAttrs,
       nodeAttrs=IMCAAttrs$nodeAttrs, subGList=IMCAAttrs$subGList)
```

`isAdjacent-methods` *Determine if nodes share an edge in a graph*

Description

For a given subclass of `graph-class`, returns `TRUE` if the graph contains an edge from node specified by `from` to the node specified by `to`.

The appropriate logical vector will be returned as long as `from` and `to` have the same length and contain nodes in the graph object specified by `object`.

Usage

```
isAdjacent(object, from, to, ...)
```

Arguments

`object`	An instance of a subclass of `graph-class`.
`from`	A `character` vector of nodes in the graph.
`to`	A `character` vector of nodes in the graph
`...`	May be used by methods called on subclasses of `graph`

`isDirected-methods` *Determine if a graph has directed or undirected edges*

Description

The edges of a `graph-class` object are either directed or undirected. This function returns `TRUE` if the edges are directed and `FALSE` otherwise.

Usage

```
isDirected(object)
```

Arguments

`object`	A `graph-class` instance

`leaves` *Find the leaves of a graph*

Description

A leaf of an undirected graph is a node with degree equal to one. A leaf of a directed graph is defined with respect to in-degree or out-degree. The leaves of a directed graph with respect to in-degree (out-degree) are those nodes with in-degree (out-degree) equal to zero.

Usage

```
leaves(object, degree.dir)
```

Arguments

`object`	A `graph` object
`degree.dir`	One of `"in"` or `"out"`. This argument is ignored when `object` is undirected and required otherwise. When `degree.dir="in"` (`degree.dir="out"`), nodes have no in coming (out going) edges will be returned.

Value

A character vector giving the node labels of the leaves.

Author(s)

Seth Falcon

Examples

```
data(graphExamples)
graphExamples[[1]]
leaves(graphExamples[[1]])

data(apopGraph)
leaves(apopGraph, "in")
leaves(apopGraph, "out")
```

`listEdges` *List the Edges of a Graph*

Description

A list where each element contains all edges between two nodes, regardless of orientation. The list has names which are node pairs, in lexicographic order, and elements all edges between those nodes.

Usage

```
listEdges(object, dropNULL=TRUE)
```

Arguments

`object`	An instance of the `graphNEL-class` class.
`dropNULL`	Should those node pairs with no edges be dropped from the returned list.

Details

The function is currently only implemented for graphs of the `graphNEL-class`. The edges in the returned list are instances of the `simpleEdge-class`.

Value

A named list of `simpleEdge-class` objects.

Author(s)

R. Gentleman

See Also

`simpleEdge-class`, `edges`

Examples

```
set.seed(123)
V <- LETTERS[1:4]
edL <- vector("list", length=4)
names(edL) <- V
toE <- LETTERS[4:1]
for(i in 1:4)
   edL[[i]] <- list(edges=5-i, weights=runif(1))
gR <- graphNEL(nodes=V, edgeL=edL)
listEdges(gR)
```

`MAPKsig` *A graph encoding parts of the MAPK signaling pathway*

Description

A graph encoding parts of the MAPK signaling pathway

Usage

```
data(MAPKsig)
```

Format

The format is: Formal class 'graphNEL' [package "graph"] with edgemode "directed".

Source

The KEGG pancreatic cancer pathway was visually inspected on 17 Sept 2007, and the subgraph associated with MAPK signaling was isolated. The HUGO names for each symbol in the KEGG visualization were obtained and checked for existance on hgu95av2. Some abbreviated terms for processes are also included as nodes.

Examples

```
data(MAPKsig)
if (require(Rgraphviz)) {
   nat = rep(FALSE, length(nodes(MAPKsig)))
   names(nat) = nodes(MAPKsig)
   plot(MAPKsig, nodeAttrs=list(fixedsize=nat))
   }
```

`mostEdges` *Find the node in a graph with the greatest number of edges*

Description

`mostEdges` finds the node that has the most edges in the graph. This is the node with the highest degree.

Usage

```
mostEdges(objGraph)
```

Arguments

`objGraph`	the graph object

Value

`index`	the index of the node with the most edges
`id`	the node value with the most edges; may be affy id, locus link id, or genename depending on the node type
`maxLen`	the number of edges for that node

Author(s)

Elizabeth Whalen

See Also

`numEdges`, `aveNumEdges`, `numNoEdges`

Examples

```
set.seed(123)
g1 <- randomGraph(11:30, letters[20:26], p=.4)
mostEdges(g1)
```

`MultiGraph-class` *EXPERIMENTAL class "MultiGraph"*

Description

The MultiGraph class represents a single node set and a set of edge sets. Each edge set is either directed or undirected. We can think of an edge in a MultiGraph as a 4-tuple (from-node, to-node, edge-type, weight), where the edge-type field in the tuple identifies the edge set, the weight is a numeric value, and the order of the nodes only matters in the case of a directed edge set. Unlike some of the graph representations, self-loops are allowed (from-node == to-node).

There is support for arbitrary edge attributes which is primarily useful for rendering plots of MultiGraphs. These attributes are stored separately from the edge weights to facilitate efficient edge weight computation.

Usage

```
MultiGraph(edgeSets, nodes = NULL, directed = TRUE, ignore_dup_edges = FALSE)
eweights(object, names.sep = NULL)
edgeSetIntersect0(g, edgeFun = NULL)
edgeSetIntersect0(g, edgeFun = NULL)
extractGraphAM(g, edgeSets)
extractGraphBAM(g, edgeSets)
```

Arguments

`edgeSets`	A named list of `data.frame` objects each representing an edge set of the multigraph. Each `data.frame` must have three columns: "from", "to", and "weight". Columns "from" and "to" can be either factors or character vectors. The "weight" column must be numeric.
`nodes`	A character vector of node labels. Nodes with zero degree can be included in a graph by specifying the node labels in `nodes`. The node set of the resulting multigraph is the union of the node labels found in `edgeSets` and `nodes`.
`directed`	A logical vector indicating whether the edge sets specified in `edgeSets` represent directed edges. If this argument has length one, the value applies to all edge sets in `edgeSets`. Otherwise, this argument must have the same length as `edgeSets`, values are aligned by position.
`object`	A `MultiGraph` instance
`g`	A `MultiGraph` instance
`names.sep`	The string to use as a separator between from and to node labels. If `NULL` no names will be attached to the returned vector.
`ignore_dup_edges`	If `FALSE` (default), specifying duplicate edges in the input is an error. When set to `TRUE` duplicate edges are ignored. Edge weight values are ignored when determining duplicates. This is most useful for graph import and conversion.
`edgeFun`	A user specified named list of functions to resolve edge attributes in a union or intersection operation

Constructors

`MultiGraph`

Methods

nodes Return the nodes of the multigraph.

numEdges Return an integer vector named by edge set containing edge counts for each edge set.

numNodes Return the number of nodes in the multigraph.

eweights Return a list named by edge set; each element is a numeric vector of edge weights for the corresponding edge set.

isDirected Return a logical vector named by the edge sets in `object` with a `TRUE` indicating a directed edge set and `FALSE` for undirected.

edges Returns a list named by edge set; for the edges in the MultiGraph

edgeNames Returns a list named by the edge set; for the names of the edges in the MultiGraph

extractFromTo Return a list named by the edge sets; each element is a data frame with column names from, to and weight corresponding to the connected nodes in the edge set.

subsetEdgeSets Return a new `MultiGraph` object representing the subset of edge sets from the original `MultiGraph`.

extractGraphAM Return a named `list` of `graphAM` objects corresponding to the edge sets from the original `MultiGraph`.

extractGraphBAM Return a named `list` of `graphBAM` objects corresponding to the edge sets from the original `MultiGraph`.

ugraph Return a new `MultiGraph` object in which all edge sets have been converted to undirected edge sets. This operation sets all edge weights to one and drops other edge attributes.

edgeSetIntersect0 Return a new `MultiGraph` object representing the intersection of edges across all edge sets within `g`. The return value will have a single edge set if the edge sets in `g` are disjoint. Otherwise, there will be a single edge set containing the shared edges. The node set is preserved. Edge weights and edge attributes are transferred over to the output if they have the same value, else user has the option of providing a function to resolve the conflict.

edgeSetUnion0 Return a new `MultiGraph` object representing the union of edges across all edge sets within `g`. The node set is preserved. Edge weights and edge attributes are transferred over to the output if they have the same value, else user has the option of providing a function to resolve the conflict.

`graphIntersect(x, y, nodeFun, edgeFun)` When given two `MultiGraph` objects, `graphIntersect` returns a new `MultiGraph` containing the nodes and edges in common between the two graphs. The intersection is computed by first finding the intersection of the node sets, obtaining the induced subgraphs, and finding the intersection of the resulting edge sets. The corresponding named edgeSets in `x` and `y` should both be either directed or undirected.Node/Edge attributes that are equal are carried over to the result. Non equal edge/node attributes will result in the corresponding attribute being set to NA. The user has the option of providing a named list(names of edgeSets) of list (names of edge attributes) of edge functions correspoding to the names of the edge attributes for resolving conflicting edge attributes (`edgeFun`). For resolving any of the conflicting node attributes, the user has the option of providing a named `list` of functions corresponding to the node attribute names (`nodeFun`).

`graphUnion(x, y, nodeFun, edgeFun)` When given two `MultiGraph` objects, `graphUnion` returns a new `MultiGraph` containing the union of nodes and edges between the two graphs. The corresponding pairs of named edgeSets in `x` and `y` should both be either directed or undirected. Non equal edge/node attributes will result in the corresponding attribute being set to NA. The user has the option of providing a named list(names of edgeSets) of list (names of edge attributes) of edge functions correspoding to the names of the edge attributes for resolving conflicting edge attributes (`edgeFun`). For resolving any of the conflicting node attributes, the user has the option of providing a named `list` of functions corresponding to the node attribute names (`nodeFun`).

`edgeSets(object, ...)` Returns the names of the edgeSets in the MultiGraph `object` as a character vector.

show Prints a short summary of a MultiGraph object

Author(s)

S. Falcon, Gopalakrishnan N

Examples

```
ft1 <- data.frame(from=c("a", "a", "a", "b", "b"),
                    to=c("b", "c", "d", "a", "d"),
                  weight=c(1, 3.1, 5.4, 1, 2.2))
```

```
ft2 <- data.frame(from=c("a", "a", "a", "x", "x", "c"),
                    to=c("b", "c", "x", "y", "c", "a"),
                  weight=c(3.4, 2.6, 1, 1, 1, 7.9))

esets <- list(es1=ft1, es2=ft2)

g <- MultiGraph(esets)

nodes(g)
numEdges(g)
eweights(g)
eweights(g, names.sep = "=>")
isDirected(g)
edges(g, edgeSet ="es1")
edges(g, "a", "es1")
edgeNames(g, "es2")
edgeSets(g)
ug <- ugraph(g)
isDirected(ug)
numEdges(ug)
edgeSetIntersect0(g)
subsetEdgeSets(g, "es1")
extractFromTo(g)
extractGraphAM(g)
extractGraphAM(g, "es1")
extractGraphBAM(g, "es1")
graphIntersect(g, g)
graphUnion(g,g)
mgEdgeDataDefaults(g, "es1", attr = "color" ) <- "white"
mgEdgeData(g, "es1", from = "a", to = c("b", "c"), attr = "color") <- "red"
mgEdgeData(g, "es1", from = "a", to = c("b", "c"), attr = "color")
nodeDataDefaults(g, attr ="shape") <- "circle"
nodeData(g, n = c("a", "b", "c"), attr = "shape") <- "triangle"
nodeData(g, n = c("a", "b", "x", "y"), attr = "shape")
```

`multiGraph-class` *Class "multiGraph"*

Description

A collection of classes to model multigraphs. These include the multiGraph class as well as classes to contain edge sets.

Objects from the Class

Objects can be created from the `multiGraph` class, the `edgeSet` class is virtual, and particular variants should be used.

Slots

These slots are for the multiGraph class.

nodes The names of the nodes.

edgeL A list of edge lists.

nodeData An instance of the `attrData` class.

graphData A list.

These slots are for the `edgeSet` class, or one of its sublcasses.

edgeData An instance of the `attrData` class.

edgemode A character vector, one of directed, or undirected.

edgeL A list of the edges (graphNEL)

adjMat An adjacency matrix (graphAM)

Methods

show Print a multigraph.

isDirected A vector indicating which of the edgeSets is directed.

nodes Retrieve the node names

numNodes Return the number of nodes

edges Return either all edges, or a subset of them, depending on the arguments supplied.

numEdges Return a vector with the number of edges, for each edge set.

`nodeData-methods` *Get and set attributes for the nodes of a graph object*

Description

Attributes of the nodes of a graph can be accessed using `nodeData`. The attributes must be defined using `nodeDataDefaults`. You can ommit the `n` argument to retrieve attributes for all nodes in the graph. You can ommit the `attr` argument to retrieve all attributes.

Usage

```
nodeData(self, n, attr)
nodeData(self, n, attr) <- value
```

Arguments

`self`	A `graph-class` instance
`n`	A `character` vector of node names
`attr`	A `character` vector of length one specifying the name of a node attribute
`value`	An R object to store as the attribute value

---

nodeDataDefaults-methods *Get and set default attributes for the nodes of a graph*

Description

You can associate arbitrary attributes with the nodes of a graph. Use `nodeDataDefaults` to specify the set of attributes that describe nodes. Each attribute must have a default value. You can set the attribute for a particular node or set of nodes using `nodeData`.

Usage

```
nodeDataDefaults(self, attr)
nodeDataDefaults(self, attr) <- value
```

Arguments

`self`	A `graph-class` instance
`attr`	A `character` vector of length one giving the name of an attribute
`value`	An R object to set as the default value for the given attribute

---

numNoEdges *Calculate the number of nodes that have an edge list of NULL*

Description

`numNoEdges` calculates the number of nodes that have an edge list of NULL (i.e. no edges).

Usage

```
numNoEdges(objGraph)
```

Arguments

`objGraph`	the graph object

Value

An integer representing the number of NULL edge lists in the graph.

Author(s)

Elizabeth Whalen

See Also

`numEdges`, `aveNumEdges`, `mostEdges`

Examples

```
set.seed(999)
g1 <- randomEGraph(letters, .01)
numNoEdges(g1)
```

`pancrCaIni` *A graph encoding parts of the pancreatic cancer initiation pathway*

Description

A graph encoding parts of the pancreatic cancer initiation pathway

Usage

```
data(pancrCaIni)
```

Format

The format is: Formal class 'graphNEL' [package "graph"] with edgemode "directed".

Source

The KEGG pancreatic cancer pathway was visually inspected on 17 Sept 2007 and a subgraph was isolated. The HUGO names for each symbol in the KEGG visualization were obtained and checked for existance on hgu95av2. Some abbreviated terms for processes are also included as nodes.

Examples

```
data(pancrCaIni)
if (require(Rgraphviz)) {
   nat = rep(FALSE, length(nodes(pancrCaIni)))
   names(nat) = nodes(pancrCaIni)
   plot(pancrCaIni, nodeAttrs=list(fixedsize=nat))
   }
```

`randomEGraph` *Random Edge Graph*

Description

A function to create random graphs according to a random edge model. The user supplies the set of nodes for the graph as `V` and either a probability, `p`, that is used for each edge or the number of edges, `edges` they want to have in the resulting graph.

Usage

```
randomEGraph(V, p, edges)
```

Arguments

`V`	The nodes for the graph.
`p`	The probability of an edge being selected.
`edges`	The number of edges wanted.

Details

The user must specify the set of nodes and either a probability for edge selection or the number of edges wanted, but not both. Let `nV` denote the number of nodes. There are `choose(nV, 2)` edges in the complete graph. If `p` is specified then a biased coin (probability of heads being `p`) is tossed for each edge and if it is heads that edge is selected. If `edges` is specified then that many edges are sampled without replacement from the set of possible edges.

Value

An object of class `graphNEL-class` that contains the nodes and edges.

Author(s)

R. Gentleman

See Also

`randomGraph`

Examples

```
set.seed(123)
V <- letters[14:22]
g1 <- randomEGraph(V, .2)

g2 <- randomEGraph(V, edges=30)
```

`randomGraph` *Random Graph*

Description

This function generates a random graph according to a model that involves a latent variable. The construction is to randomly assign members of the set `M` to the nodes, `V`. An edge is assigned between two elements of `V` when they both have the same element of `M` assigned to them. An object of class `graphNEL` is returned.

Usage

```
randomGraph(V, M, p, weights=TRUE)
```

Arguments

`V`	The nodes of the graph.
`M`	A set of values used to generate the graph.
`p`	A value between 0 and 1 that indicates the probability of selecting an element of `M`
`weights`	A logical indicating whether to use the number of shared elements of `M` as weights.

Details

The model is quite simple. To generate a graph, `G`, the user supplies the list of nodes, `V` and a set of values `M` which will be used to create the graph. For each node in `V` a logical vector with length equal to the length of `M` is generated. The probability of a `TRUE` at any position is determined by `p`. Once valus from `M` have been assigned to each node in `V` the result is processed into a graph. This is done by creating an edge between any two elements of `V` that share an element of `M` (as chosen by the selection process).

The sizes of `V` and `M` and the values of `p` determine how dense the graph will be.

Value

An object of class `graphNEL-class` is returned.

Author(s)

R. Gentleman

See Also

`randomEGraph`, `randomNodeGraph`

Examples

```
set.seed(123)
V <- letters[1:10]
M <- 1:4
g1 <- randomGraph(V, M, 0.2)
numEdges(g1) # 16, in this case
edgeNames(g1)# "<from> ~ <to>"  since undirected
```

`randomNodeGraph` *Generate Random Graph with Specified Degree Distribution*

Description

`randomNodeGraph` generates a random graph with the specified degree distribution. Self-loops are allowed. The resultant graph is directed (but can always be coerced to be undirected).

Usage

```
randomNodeGraph(nodeDegree)
```

Arguments

`nodeDegree`	A named integer vector specifying the node degrees.

Details

The input vector must be named, the names are taken to be the names of the nodes. The sum must be even (there is a theorem that says we require that to construct a graph). Self-loops are allowed, although patches to the code that make this a switchable parameter would be welcome.

Value

An instance of the `graphNEL` class. The graph is directed.

Author(s)

R. Gentleman

References

Random Graphs as Models of Networks, M. E. J. Newman.

See Also

`randomGraph`, `randomEGraph`

Examples

```
set.seed(123)
c1 <- c(a = 1, b = 1, c = 2, d = 4)

(g1 <- randomNodeGraph(c1))
stopifnot(validObject(g1))
```

`removeEdge` *removeEdge*

Description

A function to remove the specified edges from a graph.

Usage

```
removeEdge(from, to, graph)
```

Arguments

`from`	from edge labels
`to`	to edge labels
`graph`	a `graph` object

Details

A new graph instance is returned with the edges specified by corresponding elements of the `from` and `to` vectors removed. If `from` and `to` are not the same length, one of them should have length one. All edges to be removed must exist in `graph`.

Value

A new instance of a graph with the same class as `graph` is returned with the specified edges removed.

Author(s)

R. Gentleman

See Also

`addNode`,`addEdge`,`removeNode`

Examples

```
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)

gX <- removeEdge("A", "B", gR)

set.seed(123)
g <- randomEGraph(V=letters[1:5],edges=5)
g2 <- removeEdge(from=c("a","b"), to=c("d","c"), g)
```

`removeNode` *removeNode*

Description

A function to remove a node from a graph. All edges to and from the node are also removed.

Usage

```
removeNode(node, object)
```

Arguments

`node`	The label of the node to be removed.
`object`	The graph to remove the node from.

Details

The specified node is removed from the graph as are all edges to and from that node. A new instance of the same class as `object` with the specified node(s) is returned.

Note, node can be a vector of labels, in which case all nodes are removed.

This is similar to `subGraph`.

Value

A new instance of a graph of the same class as `object` but with all specified nodes removed.

Author(s)

R. Gentleman

See Also

`removeEdge`, `addEdge`, `addNode`,`subGraph`

Examples

```
V <- LETTERS[1:4]
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")
gX <- removeNode("C", gR2)
```

`renderInfo-class` *Class "renderInfo"*

Description

A container class to manage graph rendering attributes.

Objects from the Class

Objects can be created by calls of the form `new("renderInfo")` or by using the initializer `.renderInfoPrototype`.

Slots

`pars`**:** List of default rendering attributes with two items `nodes` and `edges`. When not set further down the parameter hierarchy, these defaults will be used for all nodes/edges in the graph.

`nodes`**:** Named list of attributes specific to nodes.

`edges`**:** Named list of attributes specific to edges.

`graph`**:** Named list of graph-wide attributes.

Each item of `nodes` and `edges` can take arbitrary vectors, the only restriction is that they have to be of either length 1 or length equal to the number of nodes or edges, respectively.

`pars` and `graph` can take arbitrary skalars, the latter for both edges and nodes.

Methods

The following are functions rather than methods and build the API to control the graphical output of a graph when it is plotted using `renderGraph`.

parRenderInfo, parRenderInfo<- getter and setter for items of slot `pars`

nodeRenderInfo, nodeRenderInfo<- getter and setter for items of slot `nodes`

edgeRenderInfo, edgeRenderInfo<- getter and setter for items of slot `edges`

graphRenderInfo, graphRenderInfo<- getter and setter for items of slot `graph`

The getters all take two arguments: `g` is a graph object and `name` is a character giving the name of one of the item in the respective slot. When `name` is missing this will give you the whole list.

The setters are a bit more complex: `nodeRenderInfo<-` and `edgeRenderInfo<-` can take

named list of named vectors where the names have to match the node or edge names. Items in the vector that don't match a valid edge or node name will be silently ignored. For undirected edges the order of head nodes and tail nodes in edge names is ignored, i.e. `a~b` is equivalent to codeb~a

named list of skalars which will set all the attribute for all edges or nodes in the graph `parRenderInfo<-`will only take a list with items `nodes`, `edges` and `graph`. The content of these list items can be arbitrary named vectors. `parRenderInfo<-`takes an arbitrary list

Available rendering parameters for nodes are:

col: the color of the line drawn as node border. Defaults to `black`.

lty: the type of the line drawn as node border. Defaults to `solid`. Valid values are the same as for the R's base graphic parameter `lty`.

lwd: the width of the line drawn as node border. Defaults to `1`. Note that the underlying low level plotting functions do not support vectorized `lwd` values. Instead, only the first item of the vector will be used.

fill: the color used to fill a node. Defaults to `transparent`.

textCol: the font color used for the node labels. Defaults to `black`.

fontsize: the font size for the node labels in points. Defaults to `14`. Note that the fontsize will be automatically adjusted to make sure that all labels fit their respective nodes. You may want to increase the node size by supplying the appropriate layout parameters to `Graphviz` in order to allow for larger fontsizes.

cex: Expansion factor to further control the fontsize. As default, this parameter is not set, in which case the fontsize will be clipped to the node size. This mainly exists to for consistency with the base graphic parameters and to override the clipping of fontsize to nodesize.

Available rendering parameters for edges are:

col: the color of the edge line. Defaults to `black`.

lty: the type of the edge line. Defaults to `solid`. Valid values are the same as for the R's base graphic parameter `lty`.

lwd: the width of the edge line. Defaults to `1`.

textCol: the font color used for the edge labels. Defaults to `black`.

fontsize: the font size for the edge labels in points. Defaults to `14`.

cex: Expansion factor to further control the fontsize. This mainly exists to be consistent with the base graphic parameters.

Author(s)

Deepayan Sarkar, Florian Hahne

Examples

```
g <- randomGraph(letters[1:4], 1:3, p=0.8)
nodeRenderInfo(g) <- list(fill=c("a"="red", "b"="green"))
edgeRenderInfo(g) <- list(lwd=3)
edgeRenderInfo(g) <- list(lty=3, col="red")
parRenderInfo(g) <- list(edges=list(lwd=2, lty="dashed"),
nodes=list(col="gray", fill="gray"))
nodeRenderInfo(g)
edgeRenderInfo(g, "lwd")
edgeRenderInfo(g, c("lwd", "col"))
parRenderInfo(g)
```

`reverseEdgeDirections` *Reverse the edges of a directed graph*

Description

Return a new directed graph instance with each edge oriented in the opposite direction relative to the corresponding edge in the input graph.

Usage

```
reverseEdgeDirections(g)
```

Arguments

g	A `graph` subclass that can be coerced to `graphAM`

Details

WARNING: this doesn't handle edge attributes properly. It is a preliminary implementation and subject to change.

Value

A `graphNEL` instance

Author(s)

S. Falcon

Examples

```
g <- graphNEL(nodes=c("a", "b", "c"),
         edgeL=list(a=c("b", "c"), b=character(0), c=character(0)),
         edgemode="directed")

stopifnot(isAdjacent(g, "a", "b"))
stopifnot(!isAdjacent(g, "b", "a"))
```

```
grev <- reverseEdgeDirections(g)
stopifnot(!isAdjacent(grev, "a", "b"))
stopifnot(isAdjacent(grev, "b", "a"))
```

`simpleEdge-class` *Class "simpleEdge".*

Description

A simple class for representing edges in graphs.

Objects from the Class

Objects can be created by calls of the form `new("simpleEdge", ...)`.

Slots

`edgeType`**:** Object of class `"character"`; the type of the edge.

`weight`**:** Object of class `"numeric"`; the edge weight.

`directed`**:** Object of class `"logical"`; is the edge directed.

`bNode`**:** Object of class `"character"`; the beginning node.

`eNode`**:** Object of class `"character"`; the ending node.

Methods

No methods defined with class "simpleEdge" in the signature.

Note

All slots are length one vectors (this is not currently checked for). If the edge is not directed there is no real meaning to the concepts of beginning node or ending node and these should not be interpreted as such.

Author(s)

R. Gentleman

Examples

```
new("simpleEdge", bNode="A", eNode="D")
```

`Standard labeling of edges with integers`

Standard labeling of edges with integers

Description

Functions to convert between from-to representation and standard labeling of the edges for undirected graphs with no self-loops.

Usage

```
ftM2int(ft)
int2ftM(i)
```

Arguments

`i`	Numeric vector.
`ft`	Numeric nx2 or 2xn matrix.

Details

A standard 1-based node labeling of a graph G=(V,E) is a one-to-one mapping between the integers from 1 to |V| and the nodes in V. A standard 1-based edge labeling of an undirected graph G=(V,E) with no self-loops is *the* one-to-one mapping between the integers from 1 to |V| choose 2 = |V|*(|V|-1)/2 such that the edge labeled 1 is between nodes 2 and 1, the edge labeled 2 is between nodes 3 and 1, the edge labeled 3 is between nodes 3 and 2, and so on.

Value

For `ftM2int`, a numeric vector of length n. For `int2ftM`, a `length(i) x 2` matrix.

Author(s)

Wolfgang Huber

Examples

```
nNodes <- 200
nEdges <- choose(nNodes, 2)
i <- 1:nEdges
ft <- int2ftM(i)
ft[1:6,]
stopifnot(all(ft[,1]>ft[,2])) ## always from higher to lower
stopifnot(!any(duplicated(paste(ft[,1], ft[,2]))))
stopifnot(ft[nEdges, 1]==nNodes, ft[nEdges, 2]==nNodes-1)

j <- ftM2int(ft)
stopifnot(all(i==j))
```

`subGraph` *Create a Subgraph*

Description

Given a set of nodes and a graph this function creates and returns subgraph with only the supplied nodes and any edges between them.

Usage

```
subGraph(snodes, graph)
```

Arguments

`snodes`	A `character` vector of node labels.
`graph`	A graph object, it must inherit from the `graph` class.

Details

The returned subgraph is a copy of the graph. Implementations for Implementations for `graphNEL`, `distGraph` and `clusterGraph`.

Value

A graph of the same class as the `graph` argument but with only the supplied nodes.

Author(s)

R. Gentleman

See Also

`nodes`,`edges`

Examples

```
set.seed(123)
x <- rnorm(26)
names(x) <- letters
library(stats)
d1 <- dist(x)
g1 <- new("distGraph", Dist=d1)
subGraph(letters[1:5], g1)
```

`toDotR-methods` *Methods for Function toDotR, using R to generate a dot serialization*

Description

There are two basic methods of generating dot (`http://www.graphviz.org`) language serializations of R `graph-class` structures. First, using the `toDot` methods of the Rgraphviz package, the native graphviz agraph-associated methods can be employed to create the dot serialization. Second, with the methods described here, R functions can be used to perform the serialization directly from the graph data structure, without Rgraphviz.

Methods

G = "graphNEL", outDotFile = "character", renderList = "list", optList = "list" create dot language descriptionof graph

G = "graphNEL", outDotFile = "character", renderList = "missing", optList = "missing" create dot language descriptionof graph

G = "graphNEL", outDotFile = "character", renderList = "missing", optList = "list" create dot language descriptionof graph

G = "graphNEL", outDotFile = "missing", renderList = "missing", optList = "missing" create dot language descriptionof graph

G = "graphNEL", outDotFile = "missing", renderList = "missing", optList = "list" create dot language descriptionof graph

G = "graphNEL", outDotFile = "missing", renderList = "character", optList = "missing" create dot language descriptionof graph

G = "graphNEL", outDotFile = "missing", renderList = "list", optList = "list" create dot language descriptionof graph

G = "graphNEL", outDotFile = "missing", renderList = "list", optList = "missing" create dot language descriptionof graph

G = "compoundGraph", outDotFile = "character", renderList = "list", optList = "missing" create dot language descriptionof graph

G = "compoundGraph", outDotFile = "character", renderList = "list", optList = "list" create dot language descriptionof graph

G = "compoundGraph", outDotFile = "missing", renderList = "list", optList = "missing" create dot language descriptionof graph

See Also

`toDot-methods`

Examples

```
example(randomGraph)
tmp <- tempfile()
toDotR( g1, tmp )
readLines(tmp)
unlink(tmp)
```

`ugraph` *Underlying Graph*

Description

For a *directed* graph the underlying graph is the graph that is constructed where all edge orientation is ignored. This function carries out such a transformation on `graphNEL` instances.

Usage

```
ugraph(graph)
```

Arguments

`graph`	a `graph` object.

Details

If `graph` is already *undirected* then it is simply returned.

If `graph` is a multi-graph (has multiple edges) an error is thrown as it is unclear how to compute the underlying graph in that context.

The method will work for any `graph` subclass for which an `edgeMatrix` method exists.

Value

An instance of `graphNEL` with the same nodes as the input but which is `undirected`.

Author(s)

R. Gentleman

References

Graph Theory and its Applications, J. Gross and J. Yellen.

See Also

`connComp` `edgeMatrix`

Examples

```
V <- letters[1:4]
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")

ugraph(gR2)
```

`validGraph` *Test whether graph object is valid*

Description

validGraph is a validating function for a graph object.

Usage

```
validGraph(object, quietly=FALSE)
```

Arguments

`object`	a graph object to be tested
`quietly`	`TRUE` or `FALSE` indicating whether output should be printed.

Value

If the graph object is valid, `TRUE` is returned otherwise `FALSE` is returned. If `object` is not a valid graph and `quietly` is set to `FALSE` then descriptions of the problems are printed.

Author(s)

Elizabeth Whalen

See Also

`graph-class`

Examples

```
  testGraph<-graphNEL()
  testGraph@nodes<-c("node1","node2","node3")
  validGraph(testGraph)
```

`write.tlp` *Write a graph object in a file in the Tulip format*

Description

Write a graph object in a file in the Tulip format.

Usage

```
write.tlp(graph, filename)
```

Arguments

`graph`	a `graph` object
`filename`	Name of the output file

Details

The Tulip format is used by the program Tulip.

Author(s)

Laurent Gautier <laurent@cbs.dtu.dk>

References

http://www.tulip-software.org/

Index