Package 'dynamicGraph'

February 19, 2015

Title dynamicGraph

Namespace dynamicGraph

Description Interactive graphical tool for manipulating graphs

Version 0.2.2.6

Date 2010/01/30

Author Jens Henrik Badsberg <coco@badsberg.eu>

Maintainer Jens Henrik Badsberg <coco@badsberg.eu>

Depends R (>= 1.8.1), methods, ggm, tcltk

License GPL (>= 2)

URL <http://www.badsberg.eu>

Address Andreas Bjorns Gade 21, 2. tv, 1428 Kobenhavn K, Denmark

Repository CRAN

Date/Publication 2010-02-01 19:26:39

NeedsCompilation no

R topics documented:

---

dynamicGraph-package   *dynamicGraph*

Description

The package provides an interactive graphical tool for manipulating graphs by a Tcl/tk interface.

Details

Package:	dynamicGraph
Type:	Package
Version:	0.2.2.5
Date:	2009-04-03
License:	GPL (>= 2)

Author(s)

Jens Henrik Badsberg

Maintainer: Jens Henrik Badsberg <coco@badsberg.eu>

---

blockTreeToList	*Extract the list of blocks from a block tree*

Description

Returns the list of blocks from a block tree. The block tree is an (intermediate) result of the function setTreeBlocks. setTreeBlocks is used in simpleGraphToGraph on the slot block.tree of the object of class dg.simple.graph-class.

Usage

```
blockTreeToList(tree)
```

Arguments

tree   The BlockTree part of the result of setTreeBlocks.

Value

A list of blocks, each block of class dg.Block.

Author(s)

Jens Henrik Badsberg

Examples

```
Block.tree <- list(label = "W", Vertices = c("contry"),
                   X = list(Vertices = c("sex", "race"),
                           A = list(Vertices = c("hair", "eye"),
                                   horizontal = FALSE),
                           B = list(Vertices = c("age"),
                                   C = list(Vertices = c("education")))))
Names <- unlist(Block.tree)
Names <- Names[grep("Vertices", names(Names))]
Types <- rep("Discrete", length(Names))
vertices <- returnVertexList(Names, types = Types)
blocktree <- setTreeBlocks(Block.tree, vertices)
blocklist <- blockTreeToList(blocktree$BlockTree)
Labels(blocklist)
str(Parents(blocklist))
str(Children(blocklist))
str(NodeAncestors(blocklist))
str(NodeDescendants(blocklist))
parent(blocklist[[5]])
children(blocklist[[1]])
ancestors(blocklist[[5]])
descendants(blocklist[[1]])
parent(blocklist[[3]]) <- 4
children(blocklist[[2]])
checkBlockList(blocklist)
```

---

dg.Block-class   *Class dg.Block*

Description

The class for the blocks.

Slots

stratum: Object of class "numeric", the stratum of the block.

index: Object of class "numeric" with (minus) the index of the block, the position of the block in a block list.

parent: Object of class "numeric" with the single parent of the block. The slots children, ancestors, and descendants are computed from parent. When conflicts between these four slots in dynamicGraphMain the tree other slots are computed from parent.

children: Object of class "numeric" Integer vector for the children blocks of the block.

ancestors: Object of class "numeric". Integer vector for the ancestor blocks of the block. The slots with the children and descendant blocks is set after the creation of the block (in setTreeBlocks).

descendants: Object of class "numeric". Integer vector for the descendants blocks of the block. The slot with the descendant blocks is set after the creation of the block.

position: Object of class "matrix", the position of the block, the two diagonal opposite corners.

closed: Object of class "logical", if TRUE then the block is closed, e.i. drawn as a "vertex".

visible: Object of class "logical", if TRUE then the block is drawn, else it is hidden in a closed block.

color: Object of class "character", see "dg.Node".

label: Object of class "character", see "dg.Node".

label.position: Object of class "numeric", see "dg.Node".

Extends

Class "dg.Node", directly.

Methods

ancestors<- signature(x = "dg.Block"): ...

ancestors signature(object = "dg.Block"): ...

closed<- signature(x = "dg.Block"): ...

closed signature(object = "dg.Block"): ...

children<- signature(x = "dg.Block"): ...

children signature(object = "dg.Block"): ...

descendants<- signature(x = "dg.Block"): ...

descendants signature(object = "dg.Block"): ...

draw signature(object = "dg.Block"): Method for drawing the closed block.

index<- signature(x = "dg.Block"): ...

index signature(object = "dg.Block"): ...

initialize signature(.Object = "dg.Block"): ...

name signature(object = "dg.Block"): Extract the label of the block.

parent<- signature(x = "dg.Block"): ...

parent signature(object = "dg.Block"): ...

position<- signature(x = "dg.Block"): ...

position signature(object = "dg.Block"): ...

stratum<- signature(x = "dg.Block"): ...

stratum signature(object = "dg.Block"): ...

visible<- signature(x = "dg.Block"): ...

visible signature(object = "dg.Block"): ...

Note

The dg.Block class has the methods name, label, labelPosition, position, color, stratum, index, visible, closed, parent, children, ancestors, and descendants, for extracting values and the replacement methods

label<-, labelPosition<-, position<-, color<-, stratum<-, index<-, visible<-, closed<-, parent<-, children<-, ancestors<-, and descendants<-. Items are added to the pop up menu of a block by the method addToPopups.

The methods draw, and propertyDialog is also avaliable.

Author(s)

Jens Henrik Badsberg

See Also

setBlocks, setTreeBlocks, dg.Node-class.

Examples

```
b <- new("dg.Block")

str(b)

color(b)
label(b)
labelPosition(b)
name(b)
index(b)
position(b)
stratum(b)
ancestors(b)
descendants(b)
visible(b)

color(b) <- "grey"
label(b) <- "NameAndLabel"
labelPosition(b) <- c(1, 2, 3)
# name(b) <- "NameAndLabel" # Not possible!!!
index(b) <- 3
position(b) <- matrix(c( 10,  20,  30,  40,
                        110, 120, 130, 140), byrow = TRUE, ncol = 4)
stratum(b) <- 1
ancestors(b) <- c(1, 2)
descendants(b) <- c(4, 5)
visible(b) <- FALSE

str(b)
```

---

dg.BlockEdge-class *Class dg.BlockEdge*

Description

The class for edges between blocks and for edges between vertices and blocks.

Details

The function is used in returnBlockEdgeList. dynamicGraphMain will automatic update block edges when vertices are moved between blocks.

Slots

oriented: Object of class "logical", see "dg.VertexEdge".

vertex.indices: Object of class "numeric", see also "dg.Edge". Vector with abs(vertex.indices) the indices of the nodes of the block edge. If the index is positiv then the node is a vertex, else it is a block.

width: Object of class "numeric", see "dg.Edge".

dash: Object of class "character", see "dg.Edge".

color: Object of class "character", see "dg.Edge".

label: Object of class "character", see "dg.Edge".

label.position: Object of class "numeric", see "dg.Edge".

Extends

Class "dg.Edge", directly. Class "dg.Node", directly.

Methods

nodeTypesOfEdge signature(object = "dg.BlockEdge"): Extract the types ("super classes": "Vertex" or "Block") of the vertices (nodes) of the edge.

oriented<- signature(x = "dg.BlockEdge"): ...

oriented signature(object = "dg.BlockEdge"): ...

Note

The methods of dg.Edge also applies for dg.BlockEdge.

The method new also accepts the argument vertices or vertexList. The label is then extracted from these vertices. The length of vertices should match vertex.indices, where vertex.indices is used to select vertices form vertexList.

Author(s)

Jens Henrik Badsberg

See Also

    [returnBlockEdgeList](#), [dg.Edge-class](#).

Examples

```
vertices  <- returnVertexList(paste("V", 1:4, sep = ""))
block     <- new("dg.Block", stratum = 1)
blockedge <- new("dg.BlockEdge", vertex.indices = c(4, -1),
                 vertices = new("dg.VertexList", list(vertices[[1]], block)))

str(blockedge)

color(blockedge)
label(blockedge)
labelPosition(blockedge)
width(blockedge)
nodeIndicesOfEdge(blockedge)
nodeTypesOfEdge(blockedge)

color(blockedge) <- "Black"
label(blockedge) <- "V1~1"
labelPosition(blockedge) <- c(0, 1, 2)
width(blockedge) <- 1
nodeIndicesOfEdge(blockedge) <- c(1, -1)

str(blockedge)
```

---

  dg.control   *Options of dynamicGraphMain and simpleGraphtoGraph*

Description

    Allow the user to set some characteristics of 'dynamicGraphMain' and 'DynamicGraph'.

Usage

```
dg.control(label = "dynamicGraph",
           width = 400, height = 400, w = 6, margin = 100,
           closeenough = 2, background = "white", transformation = NULL,
           permitZoom = TRUE, UserMenus = NULL, constrained = FALSE,
           vertexColor = "red", extraVertexColor = "white",
           edgeColor = "black",
           factorVertexColor = "default", factorEdgeColor = "brown",
           blockEdgeColor = "default", blockColors = NULL,
           extraEdgeColor = "peru",
           drawblocks = TRUE, right.to.left = FALSE,
           nested.blocks = FALSE, overlaying = TRUE,
```

```
fixedFactorPositions = FALSE, diagonal = TRUE, N = 3,
vertexClasses = validVertexClasses(),
factorClasses = validFactorClasses(),
edgeClasses = validEdgeClasses(),
viewClasses = validViewClasses(),
drawBlockFrame = TRUE, drawBlockBackground = FALSE,
useNamesForLabels = TRUE, namesOnEdges = TRUE,
updateEdgeLabels = TRUE, enterLeaveUpdate = TRUE,
updateAllViews = TRUE,
saveTkReferences = TRUE, saveFunctions = TRUE,
returnNull = FALSE, hasMethods = TRUE, variableFrame = TRUE,
debug.strata = FALSE, debug.edges = FALSE,
debug.position = FALSE, debug.update = FALSE, ...)
```

Arguments

`label`	Text string with the title set on the graph window. ([dynamicGraphMain](#))
`width`	Integer with the width of the plot canvas. ([dynamicGraphMain](#))
`height`	Integer with the height of the plot canvas. ([dynamicGraphMain](#))
`w`	The radius of the vertices. Send as argument to the draw method for vertices, and edges are shortened by the quantity in each end. ([dynamicGraphMain](#))
`margin`	Integer, the width of the margin round the canvas. ([dynamicGraphMain](#))
`closeenough`	Parameter for whether the mouse is close enough when clicking an object of the graph window, see [tkcanvas](#). ([dynamicGraphMain](#))
`background`	The color of the canvas of the graph window, default: "white". ([dynamicGraphMain](#))
`transformation`	NULL, or rotation matrix for projecting the positions of the vertices, blocks, etc. onto the canvas. ([dynamicGraphMain](#))
`permitZoom`	Logical. If FALSE then zooming is disabled, and no margin around the canvas is set. ([dynamicGraphMain](#))
`vertexColor`	Single text string. Colors of new vertices created in [dynamicGraphMain](#) (by [newVertex](#)). The colors of the vertices of `vertexList` are given in this list. `vertexColor` is given as argument to the draw method of vertices, but this argument are by default not used in the draw method.
`extraVertexColor`	Single text string. As `vertexColor`, but for the vertices of `extraList`.
`edgeColor`	Single text string. `edgeColor` is similar to `vertexColor`.
`factorVertexColor`	Single text string. `factorVertexColor` is similar to `vertexColor`. If `factorVertexColor` is "default" then the color of a factor vertex will depend on the type of the generator of the factor.
`factorEdgeColor`	Single text string. `factorEdgeColor` is similar to `edgeColor`.
`blockEdgeColor`	"default", or list with two text strings for colors. `blockEdgeColor` is similar to `edgeColor`. The two colors are used for respectively edges between two blocks and for edges between blocks and vertices.

blockColors	List of colors of blocks. Similar to vertexColor: Only used when creating new blocks, else the colors set in blockList and blockTree are used.
extraEdgeColor	Single text string. extraEdgeColor is similar to edgeColor.
drawblocks	Logical. If drawblocks is set to FALSE, then the blocks are not drawn. The strata of the vertices are then not updated when the vertices are moved. ([simpleGraphToGraph](#))
right.to.left	Logical. If right.to.left is set to TRUE then the explanatory blocks are drawn to the right. See [setBlocks](#). ([simpleGraphToGraph](#))
nested.blocks	Logical. If nested.blocks is set to TRUE then the blocks are drawn nested. See [setBlocks](#). ([simpleGraphToGraph](#))
overlaying	Logical. If overlaying is set to FALSE then children of a block are not drawn inside the block. See [setTreeBlocks](#). ([simpleGraphToGraph](#))
fixedFactorPositions	
	Logical. If fixedFactorPositions is set to TRUE then the factor vertices will not follow the moved vertices. ([simpleGraphToGraph](#))
diagonal	Logical. If diagonal is set to TRUE then the extra vertices are by default positioned along a diagonal. ([simpleGraphToGraph](#))
N	The number, N > 1, of coordinates for the positions of the vertices and block corners. ([simpleGraphToGraph](#))
vertexClasses	Returned value from [validVertexClasses](#), or extension of this matrix. Used when creating new vertices in [dynamicGraphMain](#).
factorClasses	Returned value from [validFactorClasses](#), or extension of this matrix. Used when creating new factor vertices in [dynamicGraphMain](#).
edgeClasses	Returned value from [validEdgeClasses](#), or extension of this matrix. Used when creating new edges in [dynamicGraphMain](#).
viewClasses	Returned value from [validViewClasses](#), or extension of this matrix. Used when creating new views in [dynamicGraphMain](#).
drawBlockFrame	Logical. If TRUE then frames are drawn around blocks. ([dynamicGraphMain](#))
drawBlockBackground	
	Logical. If TRUE then a block canvas is drawn, with color set by blockColors. ([dynamicGraphMain](#))
useNamesForLabels	
	Logical. If useNamesForLabels is TRUE then names are sued for labels. ([dynamicGraphMain](#))
namesOnEdges	Logical. If FALSE then the names of the vertices are not set on the edge as label. ([dynamicGraphMain](#))
updateEdgeLabels	
	Logical. If FALSE then the edge labels are not cleared when the model is updated. ([dynamicGraphMain](#))
enterLeaveUpdate	
	Logical. If FALSE then the graph window is not redrawn when the mouse enters and leaves the graph window. ([dynamicGraphMain](#))
UserMenus	List with user defined menu items for main menu and pop up menus. See [dg.graphedges-class](#) for an example of a user specified menu. ([dynamicGraphMain](#))
constrained	Logical. If constrained is then the vertices can not be dragged out of blocks.

hasMethods	Logical. If TRUE then the object should have the methods modifyModel and testEdge. (I do not know why the R-function hasMethod does not work on objects (defined outside the package dynamicGraph) inside dynamicGraphMain). (dynamicGraphMain)
saveTkReferences	
	Logical, if saveTkReferences is TRUE then references to Tk-variables of the dynamic graph window are saved in environments in the returned object of dynamicGraphMain (if not returnNull is TRUE). (dynamicGraphMain)
saveFunctions	Logical, if saveFunctions is TRUE then draw and update functions of the dynamic graph window are saved in environments in the returned object of dynamicGraphMain (if not returnNull is TRUE). (dynamicGraphMain)
returnNull	Logical, if returnNull is TRUE then NULL is returned. (dynamicGraphMain)
updateAllViews	Logical. If TRUE then all windows are updated when one is changed. (dynamicGraphMain)
variableFrame	Logical. If variableFrame is TRUE the a frame/box/panel for variables is made left in the graph window. (dynamicGraphMain)
debug.strata	Logical for tracing the strata of the vertices (also in plot). (dynamicGraphMain)
debug.edges	Logical for tracing edges (also by labels in plot). (dynamicGraphMain)
debug.position	Logical for tracing positions of the vertices. (dynamicGraphMain)
debug.update	Logical for tracing redrawing of the graph window. (dynamicGraphMain)
...	Additional deprecated arguments, e.g., returnLink.

Value

A 'list' with components with meanings as explained under 'Arguments'.

Note

The arguments of dg.control can also be given to dynamicGraphMain and DynamicGraph (for backward compatibility). But if the argument control is used for dynamicGraphMain or DynamicGraph then these arguments are ignored.

Author(s)

Jens Henrik Badsberg

See Also

dynamicGraphMain and DynamicGraph

Examples

```
require(dynamicGraph)
str(dg.control())
```

```
dg.DiscreteVertex-class
```
Classes   dg.DiscreteVertex,   dg.ContinuousVertex,   and
dg.OrdinalVertex

Description

The class for vertices for discrete variables, continuous variables, and ordinal variables.

Objects from the Class

Objects has the methods for extracting and setting the slots for vertices, and the method for drawing the vertex.

Slots

name: Object of class "character", see "dg.Vertex".

index: Object of class "numeric", see "dg.Vertex".

position: Object of class "numeric", see "dg.Vertex".

blockindex: Object of class "numeric", see "dg.Vertex".

stratum: Object of class "numeric", see "dg.Vertex".

constrained: Object of class "logical", see "dg.Vertex".

color: Object of class "character", see "dg.Vertex".

label: Object of class "character", see "dg.Vertex".

label.position: Object of class "numeric", see "dg.Vertex".

Extends

Class "dg.Vertex", directly. Class "dg.Node", directly.

Methods

draw signature(object = "dg.DiscreteVertex"): Method for drawing the vertex. The symbol will be a 'd'ot for 'd'iscrete variable.

draw signature(object = "dg.OrdinalVertex"): Method for drawing the vertex. The symbol will be a filled square for ordinal variable.

draw signature(object = "dg.ContinuousVertex"): Method for drawing the vertex. The symbol will be a 'c'circle for 'c'ontinuous variable.

Author(s)

Jens Henrik Badsberg

See Also

dg.Vertex-class, returnVertexList, dg.Vertex-class.

`dg.Edge-class` *Class dg.Edge*

Description

A skeleton class for the classes of edges.

Objects from the Class

Objects has the methods for extracting and setting the slots for edges, and the method for drawing the edge.

Slots

`vertex.indices`**:** Object of class `"numeric"`: The `vertex.indices` of the vertices of the edge.

`width`**:** Object of class `"numeric"`: The `width` of the edge.

`dash`**:** Object of class `"character"`: The `dash` pattern of the edge.

From the Tcl/tk Reference Manual:

"DASH PATTERNS

Many items support the notion of an dash pattern for outlines.

The first possible syntax is a list of integers. Each element represents the number of pixels of a line segment. Only the odd segments are drawn using the "outline" color. The other segments are drawn transparant.

The second possible syntax is a character list containing only 5 possible characters $[.,-\_]$. The space can be used to enlarge the space between other line elements, and can not occur as the first position in the string. Some examples: -dash . = -dash {2 4} -dash - = -dash {6 4} -dash -. = -dash {6 4 2 4} -dash -.. = -dash {6 4 2 4 2 4} -dash {. } = -dash {2 8} -dash , = -dash {4 4}

The main difference of this syntax with the previous is that it it shape-conserving. This means that all values in the dash list will be multiplied by the line width before display. This assures that "." will always be displayed as a dot and "-" always as a dash regardless of the line width.

On systems which support only a limited set of dash patterns, the dash pattern will be displayed as the closest dash pattern that is available. For example, on Windows only the first 4 of the above examples are available. The last 2 examples will be displayed identically to the first one. "

`color`**:** Object of class `"character"`: The `color` of the edge.

`label`**:** Object of class `"character"`: The `label` of the edge.

`label.position`**:** Object of class `"numeric"`: The `label.position` of the edge.

Extends

Class `"dg.Node"`, directly.

Methods

dash<- signature(x = "dg.Edge") : Set the dash pattern of the edge.

dash signature(object = "dg.Edge"): Return the dash pattern of the edge.

draw signature(object = "dg.Edge"): ...

initialize signature(.Object = "dg.Edge"): ...

label<- signature(x = "dg.Edge"): Set the label of the edge.

label signature(object = "dg.Edge"): Return the label of the edge.

name signature(object = "dg.Edge"): Return the name, equal to the label, of the edge.

nodeIndicesOfEdge<- signature(x = "dg.Edge"): Set the indices of the vertices of the edge.

nodeIndicesOfEdge signature(object = "dg.Edge"): Return the indices of the vertices of
    the edge.

width<- signature(x = "dg.Edge"): Set the width of the edge.

width signature(object = "dg.Edge"): Return the width of the edge.

Author(s)

Jens Henrik Badsberg

See Also

dg.VertexEdge-class, dg.BlockEdge-class, dg.FactorEdge-class.

---

dg.ExtraEdge-class   *Class dg.ExtraEdge*

Description

The class for the edges between vertices and extra vertices.

Slots

vertex.indices: Object of class "numeric", see "dg.Edge". If the index is positiv then the node
    is a vertex, else it is the extra vertex.

width: Object of class "numeric", see "dg.Edge".

dash: Object of class "character", see "dg.Edge".

color: Object of class "character", see "dg.Edge".

label: Object of class "character", see "dg.Edge".

label.position: Object of class "numeric", see "dg.Edge".

Extends

Class "dg.Edge", directly. Class "dg.Node", directly.

Methods

nodeTypesOfEdge signature(object = ″dg.ExtraEdge″): Extract the types ("super classes": ″Vertex″ or ″Extra″) of the vertices (nodes) of the edge.

Note

The methods (except [oriented](#)) of [dg.Edge](#) also applies for dg.ExtraEdge.

The method new also accepts the argument vertices or vertexList. The label is then extracted from these vertices. The length of vertices should match vertex.indices, where vertex.indices is used to select vertices form vertexList.

Extra vertices and nodes are used in demo(dg.USArrests) to display the loadings in a biplot.

Author(s)

Jens Henrik Badsberg

See Also

[returnExtraEdgeList](#), [dg.Edge-class](#), and [dg.TextVertex-class](#).

Examples

```
vertices  <- returnVertexList(paste(″V″, 1:4, sep = ″″))
extra     <- returnVertexList(paste(″E″, 1:4, sep = ″″))
extraedge <- new(″dg.ExtraEdge″, vertex.indices = c(3, -2),
                 vertices = new(″dg.VertexList″,
                               c(vertices[3], extra[2])))

str(extraedge)

color(extraedge)
label(extraedge)
labelPosition(extraedge)
width(extraedge)
nodeIndicesOfEdge(extraedge)
nodeTypesOfEdge(extraedge)

color(extraedge) <- ″Black″
label(extraedge) <- ″Gryf″
labelPosition(extraedge) <- c(0, 1, 2)
width(extraedge) <- 1
nodeIndicesOfEdge(extraedge) <- c(1, -1)
str(extraedge)
```

dg.FactorEdge-class   *Class dg.FactorEdge*

Description

The class for the bipartite graph edges between vertices and factors.

Slots

vertex.indices: Object of class "numeric", see also "dg.Edge". If the index is positiv then the node is a vertex, else it is the factor vertex.

width: Object of class "numeric", see "dg.Edge".

dash: Object of class "character", see "dg.Edge".

color: Object of class "character", see "dg.Edge".

label: Object of class "character", see "dg.Edge".

label.position: Object of class "numeric", see "dg.Edge".

Extends

Class "dg.Node", directly. Class "dg.Edge", directly.

Methods

nodeTypesOfEdge signature(object = "dg.FactorEdge"): Extract the types ("super classes": "Vertex" or "Factor") of the vertices (nodes) of the edge.

Note

The methods (except [oriented](#)) of [dg.Edge](#) also applies for dg.FactorEdge.

The method new also accepts the argument vertices or vertexList. The label is then extracted from these vertices. The length of vertices should match vertex.indices, where vertex.indices is used to select vertices form vertexList.

Author(s)

Jens Henrik Badsberg

See Also

[returnFactorEdgeList](#), [dg.Edge-class](#), and [dg.FactorVertex-class](#).

Examples

```
vertices   <- returnVertexList(paste("V", 1:4, sep = ""))
factor     <- new("dg.FactorVertex", vertex.indices = 1:3,
                  vertexList = vertices)
factoredge <- new("dg.FactorEdge", vertex.indices = c(1, -1),
                  vertices = new("dg.VertexList", list(vertices[[1]], factor)))

str(factoredge)

color(factoredge)
label(factoredge)
labelPosition(factoredge)
width(factoredge)
nodeIndicesOfEdge(factoredge)
nodeTypesOfEdge(factoredge)

color(factoredge) <- "Black"
label(factoredge) <- "V1~V1:2:3"
labelPosition(factoredge) <- c(0, 1, 2)
width(factoredge) <- 1
nodeIndicesOfEdge(factoredge) <- c(1, -1)
str(factoredge)
```

---

dg.FactorVertex-class   *Class dg.FactorVertex*

Description

A skeleton class for the classes of factor vertices.

Slots

fixed.positions: Object of class "logical": If FALSE, then the factor will follow the vertices
when the positions of the vertices are changed.

vertex.indices: Object of class "numeric": The vertex.indices of the vertices of the factor.

name: Object of class "character", see "dg.Vertex".

index: Object of class "numeric", see "dg.Vertex".

position: Object of class "numeric", see "dg.Vertex".

blockindex: Object of class "numeric", see "dg.Vertex".

stratum: Object of class "numeric", see "dg.Vertex".

constrained: Object of class "logical", see "dg.Vertex".

color: Object of class "character", see "dg.Vertex".

label: Object of class "character", see "dg.Vertex".

label.position: Object of class "numeric", see "dg.Vertex".

Extends

Class `"dg.Vertex"`, directly. Class `"dg.Node"`, directly.

Methods

fixed.positions<- signature(x = "dg.FactorVertex"): ...

fixed.positions signature(object = "dg.FactorVertex"): ...

index<- signature(x = "dg.FactorVertex"): ...

index signature(object = "dg.FactorVertex"): ...

initialize signature(.Object = "dg.FactorVertex"): The method new also accepts the argument vertices or vertexList. The name, label, and position is then extracted from these vertices. The length of vertices should match vertex.indices, where vertex.indices is used to select vertices form vertexList.

nodeIndices<- signature(x = "dg.FactorVertex"): ...

nodeIndices signature(object = "dg.FactorVertex"): ...

Note

The methods (except `stratum`) of `dg.Vertex` also applies for dg.FactorVertex.

Author(s)

Jens Henrik Badsberg

See Also

`returnFactorVerticesAndEdges`, `dg.FactorEdge-class`, `dg.Vertex-class`, `dg.Node-class`, and `validFactorClasses`.

Examples

```
vertices       <- returnVertexList(paste("V", 1:4, sep = ""),
                                    types = rep("Discrete", 4))
vertex.indices <- c(1, 2, 3)
vertices       <- new("dg.VertexList", vertices[c(1, 2, 3)])
name           <- paste(Labels(vertices), collapse = ":")

factor <- new("dg.Generator", vertex.indices = vertex.indices,
              position = apply(Positions(vertices), 2, mean),
              index = 0, color = "yellow", name = name, label = name)

factor <- new("dg.FactorVertex", vertex.indices = 1:3, vertices = vertices)

factor <- new("dg.FactorVertex", vertex.indices = 1:3, vertexList = vertices)

str(factor)

color(factor)
label(factor)
```

```
labelPosition(factor)
name(factor)
index(factor)
position(factor)
nodeIndices(factor)

color(factor) <- "green"
label(factor) <- "v-1-2-3"
labelPosition(factor) <- c(1, 2, 3)
name(factor) <- "V-123"
index(factor) <- 3
position(factor) <- c( 10,  20,  30,  40)

str(factor)
```

---

dg.Generator-class   *Classes dg.Generator, dg.DiscreteGenerator, dg.Linear\-Generator,*
                              and dg.QuadraticGenerator

Description

The class for factor vertices for general terms, discrete terms, linear terms, and quadratic terms.

The class adds the draw method to the [dg.FactorVertex-class](#).

The objects of the classes are usually created by the function [returnFactorVerticesAndEdges](#).

Slots

vertex.indices: Object of class "numeric", see "dg.FactorVertex".

fixed.positions: Object of class "logical", see "dg.FactorVertex".

name: Object of class "character", see "dg.Vertex".

index: Object of class "numeric", see "dg.Vertex".

position: Object of class "numeric", see "dg.Vertex".

blockindex: Object of class "numeric", see "dg.Vertex".

stratum: Object of class "numeric", see "dg.Vertex".

constrained: Object of class "logical", see "dg.Vertex".

color: Object of class "character", see "dg.Vertex". Default is yellow for general, cyan for
    discrete, magenta for linear, and blue for quadratic terms.

label: Object of class "character", see "dg.Vertex".

label.position: Object of class "numeric", see "dg.Vertex".

Extends

Class "dg.FactorVertex", directly. Class "dg.Vertex", directly. Class "dg.Node", directly.

Methods

draw signature(object = "dg.Generator"): ...

draw signature(object = "dg.DiscreteGenerator"): ...

draw signature(object = "dg.LinearGenerator"): ...

draw signature(object = "dg.QuadraticGenerator"): ...

Author(s)

Jens Henrik Badsberg

See Also

newFactor, returnFactorVerticesAndEdges, dg.FactorVertex-class, dg.Vertex-class, dg.Node-class.

---

dg.graph-class   *Class dg.graph*

Description

The representation of a graph for dynamicGraph. Vertices, blocks, viewType, edges, etc. are here the dynamicGraph objects.

Usage

```
dg(object,

    ...)
```

Arguments

object   The graph.
...   Additional arguments.

Objects from the Class

Objects can be created by calls of the form new("dg.graph", ...).

Slots

vertexList: Object of class "dg.VertexList": List of vertices (each of class containing the class dg.Vertex) created by returnVertexList or exported from dynamicGraphMain.

blockList: Object of class "dg.BlockList": List of blocks (each of class dg.Block) created by setBlocks or exported from dynamicGraphMain.

viewType: Object of class "character", See dg.graphedges-class.

visibleVertices: Object of class "numeric", See dg.graphedges-class.

visibleBlocks: Object of class "numeric", See dg.graphedges-class.

oriented: Object of class "logical", See dg.graphedges-class.

edgeList: Object of class "dg.VertexEdgeList", See dg.graphedges-class.

blockEdgeList: Object of class "dg.BlockEdgeList", See dg.graphedges-class.

factorVertexList: Object of class "dg.FactorVertexList", See dg.graphedges-class.

factorEdgeList: Object of class "dg.FactorEdgeList", See dg.graphedges-class.

extraList: Object of class "dg.VertexList", See dg.graphedges-class.

extraEdgeList: Object of class "dg.ExtraEdgeList", See dg.graphedges-class.

Extends

Class "dg.graphedges", directly.

Methods

coerce signature(from = "dg.simple.graph", to = "dg.graph"): ...

dg signature(object = "dg.graph"): ...

addModel signature(object = "dg.graph"): ...

addView signature(object = "dg.graph"): ...

replaceModel signature(object = "dg.graph"): ...

replaceView signature(object = "dg.graph"): ...

Author(s)

Jens Henrik Badsberg

See Also

dg.simple.graph-class, and dynamicGraphMain.

Examples

```
from <- c("contry", "contry", "race", "race",         "sex",  "sex")
to   <- c(   "sex",   "race", "hair",  "eye", "education",  "age")
vertexnames <- unique(sort(c(from, to)))
vertices <- returnVertexList(vertexnames)
edge.list <- vector("list", length(to))
for (j in seq(along = to)) edge.list[[j]] <- c(from[j], to[j])
edges <- returnEdgeList(edge.list, vertices, color = "red", oriented = TRUE)

graph <- new("dg.graph", vertexList = vertices, edgeList = edges); str(graph)
dg(graph)
```

---

dg.graphedges-class   *Class dg.graphedges*

Description

The representation of a graph for dynamicGraph. Vertices, blocks, viewType, edges, etc. are here the dynamicGraph objects.

Objects from the Class

Objects can be created by calls of the form new("dg.graphedges", ...).

Slots

viewType: Object of class "character" with the type of view.

visibleVertices: Object of class "numeric": Numeric vector of the indices of the vertices of vertexList to plot.

visibleBlocks: Object of class "numeric": Numeric vector of the indices of the blocks of blockList to plot.

oriented: Object of class "logical": If TRUE (or FALSE) then edges are oriented (or not), also when blocks are missing. If NA then the edges are directed according to the blocks of the edge.

edgeList: Object of class "dg.VertexEdgeList": List of edges (of class containing dg.Edge) created by returnEdgeList or exported from dynamicGraphMain.

blockEdgeList: Object of class "dg.BlockEdgeList": List of blockedges (of class containing the class dg.BlockEdge) created by returnBlockEdgeList or exported from dynamicGraphMain.

factorVertexList: Object of class "dg.FactorVertexList": List of secondary vertices, called factor vertices, for, e.g., the generators of the model, (each of class containing dg.FactorVertex) created by returnFactorVerticesAndEdges or exported from dynamicGraphMain.

factorEdgeList: Object of class "dg.FactorEdgeList": List of bipartite graph edges, called factor edges, (each of class containing dg.FactorEdge) created by the function returnFactorEdgeList or exported from dynamicGraphMain. Factor edges are edges between vertices and factor vertices.

extraList: Object of class "dg.VertexList": List of vertices (of class containing the class dg.Vertex) created by the function returnVertexList or exported from the function dynamicGraphMain, for, e.g., additional titles in the plot.

extraEdgeList: Object of class "dg.ExtraEdgeList": List of edges between extra vertices and vertices.

Methods

addModel signature(object = "dg.graphedges"): ...

addView signature(object = "dg.graphedges"): ...

replaceModel signature(object = "dg.graphedges"): ...

replaceView `signature(object = "dg.graphedges"):` ...

show `signature(object = "dg.graphedges"):` ...

Str `signature(object = "dg.graphedges"):` ...

blockEdgeList<- `signature(x = "dg.graphedges"):` ...

blockEdgeList `signature(object = "dg.graphedges"):` ...

blockList<- `signature(x = "dg.graphedges"):` ...

blockList `signature(object = "dg.graphedges"):` ...

edgeList<- `signature(x = "dg.graphedges"):` ...

edgeList `signature(object = "dg.graphedges"):` ...

extraEdgeList<- `signature(x = "dg.graphedges"):` ...

extraEdgeList `signature(object = "dg.graphedges"):` ...

extraList<- `signature(x = "dg.graphedges"):` ...

extraList `signature(object = "dg.graphedges"):` ...

factorEdgeList<- `signature(x = "dg.graphedges"):` ...

factorEdgeList `signature(object = "dg.graphedges"):` ...

factorVertexList<- `signature(x = "dg.graphedges"):` ...

factorVertexList `signature(object = "dg.graphedges"):` ...

oriented `signature(object = "dg.graphedges"):` ...

viewType<- `signature(x = "dg.graphedges"):` ...

viewType `signature(object = "dg.graphedges"):` ...

visibleBlocks<- `signature(x = "dg.graphedges"):` ...

visibleBlocks `signature(object = "dg.graphedges"):` ...

visibleVertices<- `signature(x = "dg.graphedges"):` ...

visibleVertices `signature(object = "dg.graphedges"):` ...

Author(s)

Jens Henrik Badsberg

See Also

`dg.simple.graph-class`, `dg.graph-class`, and `dynamicGraphMain`.

Examples

```
# The use of "addModel"  and "addModel"
# in the example "usermenus" of demo:

your.DrawModel <- function(object, slave = FALSE, viewType = "Simple", ...) {

    dots <- list(...)
    localArguments    <- dots$Arguments
```

```
      # Here you should make your new model.
      # A copy is made by the following:

      ModelObject <- object

      title <- ModelObject@name

      # and compute graph edges:

      dgEdges <- graphEdges(ModelObject, viewType, Arguments = localArguments)

      show(dgEdges)

      if (slave) {
        # Drawing ''an other model'' in a new window:
        addModel(dgEdges,
                 frameModels =       localArguments$frameModels,
                 modelObject =       ModelObject,
                 modelObjectName =   ModelObject@name)
      }
      else {
        # Overwriting with ''an other model'' in same view:
        replaceModel(dgEdges,
                     frameModels     = localArguments$frameModels,
                     modelIndex      = localArguments$modelIndex,
                     viewIndex       = localArguments$viewIndex,
                     modelObject     = ModelObject,
                     modelObjectName = ModelObject@name)
      }
  }

 your.LabelAllEdges <- function(object, slave = FALSE, ...)
  {

   dots          <- list(...)
   localArguments <-      dots$Arguments
   dg             <-      localArguments$dg

   # browser()

   getNodeName <- function(index, type)
     if (type == "Vertex")
       name(localArguments$frameModel@vertices[[index]])
     else if (type == "Factor")
       name(localArguments$dg@factorVertexList[[abs(index)]])
     else if (type == "Extra")
       name(localArguments$dg@extraList[[abs(index)]])
     else if (type == "Block")
       label(localArguments$dg@blockList[[abs(index)]])
     else
       NULL
```

```
visitEdges <- function(edges) {
  for (i in seq(along = edges)) {
    vertices <- nodeIndicesOfEdge(edges[[i]])
    types    <- nodeTypesOfEdge(edges[[i]])

    name.f <- getNodeName(vertices[1], types[1])
    name.t <- getNodeName(vertices[2], types[2])

    R <- testEdge(object, action = "remove",
                  name.1 = name.f, name.2 = name.t,
                  from = vertices[1], to = vertices[2],
                  from.type = types[1], to.type = types[2],
                  edge.index = i, force = force, Arguments = localArguments)

    if (!is.null(R)) {
      if (TRUE || (hasMethod("label", class(R))))
        label(edges[[i]]) <- label(R)
      if (TRUE || (hasMethod("width", class(R))))
        width(edges[[i]]) <- width(R)
    }
  }
  return(edges)
}

dg@edgeList       <- visitEdges(dg@edgeList)
dg@factorEdgeList <- visitEdges(dg@factorEdgeList)
dg@blockEdgeList  <- visitEdges(dg@blockEdgeList)

if (slave) {
  # Drawing ''an other model'' in a new window:
  addModel(dg,
           frameModels     = localArguments$frameModels,
           modelObject     = localArguments$object,
           modelObjectName = localArguments$object@name)
}
else {
  # Overwriting with ''an other model'' in same view:
  replaceModel(dg,
               frameModels     = localArguments$frameModels,
               frameViews      = localArguments$frameViews,
               graphWindow     = localArguments$graphWindow,
               modelObject     = localArguments$object,
               modelObjectName = localArguments$object@name)
}

}

test.function <- function(...) print("Test.Function")
test.function <- function(...) print(list(...)$Arguments$object@name)

Menus <-
list(MainUser =
     list(label =
```

```
       "Transformation by 'prcomp' on position of \"vertices\", and redraw",
           command = function(object, ...) {
             localArguments <- list(...)$Arguments
             transformation <-
                 t(prcomp(Positions(localArguments$vertexList))$rotation)
             control <- localArguments$control
             control$transformation <- transformation
             replaceControls(control,
                             frameModels = localArguments$frameModels,
                             modelIndex  = localArguments$modelIndex,
                             viewIndex   = localArguments$viewIndex,
                             Arguments   = localArguments)
           }),
     MainUser =
     list(label = "Position of \"vertices\" by 'cmdscale', and redraw",
           command = function(object, ...) {
             localArguments <- list(...)$Arguments
             Vertices <- localArguments$dg@vertexList
             Edges <- localArguments$dg@edgeList
             positions <- Positions(localArguments$dg@vertexList)
             N <- dim(positions)[2]
             e <- NodeIndices(Edges)
             n <- Names(Vertices)
             X <- matrix(rep(-1, length(n)^2), ncol = length(n))
             for (i in 1:length(e)) {
               suppressWarnings(w <- as.numeric(names(e)[i]))
               if (is.na(w)) w <- .5
               X[e[[i]][1], e[[i]][2]] <- w
               X[e[[i]][2], e[[i]][1]] <- w
             }
             dimnames(X) <- list(n, n)
             d <- 1.25
             X[X==-1] <- d
             X <- X - d * diag(length(n))
             mdsX <- cmdscale(X, k = N, add = TRUE, eig = TRUE, x.ret = TRUE)
             # mdsX <- isoMDS(X, k = N)
             M <- max(abs(mdsX$points))
             Positions(localArguments$dg@vertexList) <<- mdsX$points / M * 45

             # replaceVertexList(localArguments$vertexList,
             #                   frameModels = localArguments$frameModels,
             #                   modelIndex  = localArguments$modelIndex,
             #                   viewIndex   = localArguments$viewIndex,
             #                   Arguments   = localArguments)

             vertices(localArguments$frameModels) <-
                                               localArguments$dg@vertexList
           }),
     MainUser =
     list(label = "Position of \"vertices\"",
           command = function(object, ...)
             print(Positions(list(...)$Arguments$vertexList))),
     MainUser =
```

```
            list(label = "Label all edges, in this window",
                 command = function(object, ...)
                              your.LabelAllEdges(object, slave = FALSE, ...)),
            MainUser =
            list(label = "Label all edges, in slave window",
                 command = function(object, ...)
                              your.LabelAllEdges(object, slave = TRUE, ...)),
            MainUser =
            list(label = "Draw model, in this window",
                 command = function(object, ...)
                              your.DrawModel(object, slave = FALSE, ...)),
            MainUser =
            list(label = "Draw model, in slave window",
                 command = function(object, ...)
                              your.DrawModel(object, slave = TRUE, ...)),
            MainUser =
            list(label = "Call of function 'modalDialog', result on 'title' at top",
                 command = function(object, ...)
                 {
                   localArguments <- list(...)$Arguments
                   ReturnVal <- modalDialog("Test modalDialog Entry",
                                            "Enter name",
                                            localArguments$control$title,
                                            top = localArguments$top)
                   print(ReturnVal)
                   if (ReturnVal == "ID_CANCEL")
                     return()
                   tktitle(localArguments$top) <- ReturnVal
                 }
               ),
            MainUser =
            list(label =
                 "Call of function 'test.function', result on 'viewLabel' at bottom",
                 command = function(object, ...)
                 {
                   localArguments <- list(...)$Arguments
                   tkconfigure(localArguments$viewLabel,
                               text = paste(localArguments$dg@viewType, " | ",
                                            test.function(...))) } ),
            MainUser =
            list(label =
                 "Test of 'Arguments'",
                 command = function(object, ...)
                 {

                   dots             <- list(...)
                   localArguments   <- list(...)$Arguments

                   print(names(dots))
                   print(names(localArguments))

                   # Only the nine values
                   # "frameModels", "modelIndex", "viewIndex",
```

```
                # "object", "objectName",
                # "selectedNodes", "selectedEdges",
                # "closedBlock", and "hiddenBlock "
                # should be extracted from "list(...)$Arguments":

                frameModels      <- localArguments$frameModels

                modelIndex       <- localArguments$modelIndex
                viewIndex        <- localArguments$viewIndex

                object           <- localArguments$object
                objectName       <- localArguments$objectName

                selectedNodes    <- localArguments$selectedNodes
                selectedEdges    <- localArguments$selectedEdges
                closedBlock      <- localArguments$closedBlock
                hiddenBlock      <- localArguments$hiddenBlock

                #  "control" can alse be extracted from "framModels"

                Control          <- localArguments$control

                control          <- frameModels@control

                # These 3 are 'deprecated':

                drawModel        <- localArguments$drawModel    #
                redrawView       <- localArguments$redrawView   #

                envir            <- localArguments$envir        #

                # The following are currently retained in "list(...)$Arguments",
                # but can be extracted from the above teen values:

                frameViews       <- frameModels@models[[modelIndex]]
                graphWindow      <- frameViews@graphs[[viewIndex]]

                vertexList       <- frameModels@vertices
                blockList        <- frameModels@blocks

                dg               <- graphWindow@dg

                visibleVertices  <- visibleVertices(dg)
                visibleBlocks    <- visibleBlocks(dg)
                edgeList         <- edgeList(dg)
                oriented         <- oriented(dg)
                blockEdgeList    <- blockEdgeList(dg)
                factorVertexList <- factorVertexList(dg)
                factorEdgeList   <- factorEdgeList(dg)
                extraList        <- extraList(dg)
                extraEdgeList    <- extraEdgeList(dg)

                viewType         <- viewType(dg)
```

```
        top               <- top(graphWindow)
        box               <- vbox(graphWindow)
        canvas            <- canvas(graphWindow)
        viewLabel         <- viewLabel(graphWindow)
        tags              <- tags(graphWindow)

        # The values now in 'control' are no longer
        # avaliable in "list(...)$Arguments":

        title             <- control$label # bad example since the
                                            # name "title" has been
                                            # change to "label".

        vertexColor       <- control$vertexColor

        print(names(control))

        browser()

        } ),

    Vertex =
    list(label = "Test of user popup menu for vertices: Label",
        command = function(object, name, ...)
        {
          # print(name)
          dots <- list(...)
          # print(names(dots))
          # print(c(dots$type))
          # print(c(dots$index))
          localArguments <- dots$Arguments
          # print(names(localArguments))
          # str(localArguments$selectedNodes)
          # str(localArguments$selectedEdges)
          print(localArguments$dg@vertexList[[dots$index]]@label)
        }
      ),
    Edge =
    list(label = "Test of user popup menu for edges: Class",
        command = function(object, name1, name2, ...)
        {
          dots          <- list(...)
          localArguments <-     dots$Arguments

          # print(c(name1, name2))
          # print(c(dots$edge.index, dots$which.edge, dots$from, dots$to))
          # print(c(dots$from.type, dots$to.type, dots$edge.type))
          # print(names(localArguments))
          # str(localArguments$selectedNodes)
          # str(localArguments$selectedEdges)

          ReturnVal <- selectDialog("Test selectDialog Entry",
```

```
                                    "Select name",
                                    localArguments$control$edgeClasses[,1],
                                    top = localArguments$top)
            print(ReturnVal)
            if (ReturnVal == "ID_CANCEL")
              return()
            if ((dots$from > 0) && (dots$to > 0)) {
              dg <- localArguments$dg
              class(dg@edgeList[[dots$edge.index]]) <-
                        localArguments$control$edgeClasses[ReturnVal, 2]
              # replaceView ?
              replaceModel(dg,
                  frameModels     = localArguments$frameModels,
                  modelIndex      = localArguments$modelIndex,
                  viewIndex       = localArguments$viewIndex,
                  modelObject     = localArguments$object,
                  modelObjectName = localArguments$objectName)
            }
          }
        ),
     ClosedBlock =
     list(label = "Test of user popup menu for blocks",
          command = function(object, name, ...)
          {
            print(name)
            print(c(list(...)$index))
          }
          )
     )
```

---

dg.list-class   *Classes dg.list, dg.NodeList, and dg.EdgeList*

Description

A class for lists of vertices, factors, blocks, and edges.

Objects from the Class

Appropriate slots of [DynamicGraph-class](#) and [DynamicGraphView-class](#) should be of this class.

The lists returned from [returnVertexList](#), [returnEdgeList](#), [returnFactorEdgeList](#), and [returnBlockEdgeList](#), and appropriate components of the returned values from [setBlocks](#) and [returnFactorVerticesAndEdges](#) are of this class.

Objects can be created by calls of the form new("dg.list", ...).

Slots

.Data: Object of class "list": The list.

Extends

Class "dg.EdgeList":

Class "dg.NodeList", directly. Class "dg.list", by class "dg.NodeList". Class "list", by class "dg.NodeList".

Methods

asDataFrame signature(objectlist = "dg.list"): ...

Blockindices signature(objectlist = "dg.list"): ...

Closed<- signature(objectlist = "dg.list"): ...

Closed signature(objectlist = "dg.list"): ...

Constrained<- signature(objectlist = "dg.list"): ...

Constrained signature(objectlist = "dg.list"): ...

Colors<- signature(objectlist = "dg.list"): ...

Colors signature(objectlist = "dg.list"): ...

Dashes<- signature(objectlist = "dg.list"): ...

Dashes signature(objectlist = "dg.list"): ...

FixedPositions<- signature(objectlist = "dg.list"): ...

FixedPositions signature(objectlist = "dg.list"): ...

Indices signature(objectlist = "dg.list"): ...

LabelPositions<- signature(objectlist = "dg.list"): ...

LabelPositions signature(objectlist = "dg.list"): ...

Labels<- signature(objectlist = "dg.list"): ...

Labels signature(objectlist = "dg.list"): ...

Names<- signature(objectlist = "dg.list"): ...

Names signature(objects = "dg.list"): ...

Parents<- signature(objectlist = "dg.list"): ...

Parents signature(objectlist = "dg.list"): ...

Children<- signature(objectlist = "dg.list"): ...

Children signature(objectlist = "dg.list"): ...

NodeAncestors<- signature(objectlist = "dg.list"): ...

NodeAncestors signature(objectlist = "dg.list"): ...

NodeDescendants<- signature(objectlist = "dg.list"): ...

NodeDescendants signature(objectlist = "dg.list"): ...

NodeIndices signature(objectlist = "dg.list"): ...

NodeTypes signature(objectlist = "dg.list"): ...

Oriented<- signature(objectlist = "dg.list"): ...

Oriented signature(objectlist = "dg.list"): ...

Positions<- `signature(objectlist = "dg.list")`: ...

Positions `signature(objectlist = "dg.list")`: ...

show `signature(object = "dg.list")`: ...

Str `signature(objectlist = "dg.list")`: ...

Strata<- `signature(objectlist = "dg.list")`: ...

Strata `signature(objectlist = "dg.list")`: ...

Visible<- `signature(objectlist = "dg.list")`: ...

Visible `signature(objectlist = "dg.list")`: ...

Widths<- `signature(objectlist = "dg.list")`: ...

Widths `signature(objectlist = "dg.list")`: ...

Author(s)

Jens Henrik Badsberg

See Also

[dynamicGraphMain](#), and [DynamicGraphModel-class](#).

---

dg.Model-class   *Class dg.Model*

Description

An example class for the model object of dynamicGraph.

Arguments

name   Text string with the `name` of the model object.

Value

An object of class `dg.Model`.

Objects from the Class

This is an example of the object for interface between [dynamicGraphMain](#) and your models.

The model object of the call of [dynamicGraphMain](#) should have the methods `modifyModel`, `testEdge`, `graphEdges` and `setGraphEdges`.

When the graph is modified, by adding or dropping vertices or edge, the method `modifyModel` is called on the argument object of [dynamicGraphMain](#).

If a value different from NULL is returned from `modifyModel` at the key event "add edge" then the edge is added to the the view of the model, the graph window.

If NULL is returned from modifyModel then the methods addModel and replaceModel can be used to draw the new graph inside modifyModel. If a value different from NULL is returned from modifyModel then the methods addView ,addModel, replaceView, and replaceModel should not be called from modifyModel for the action add edge.

If the edge is to be added in a slave window then the edges to draw can be returned in a component with name edgeList (or newEdges$vertexEdges) of the returned structure.

If an object is returned in the list of the returned value from modifyModel then object in dynamicGraphMain is replaced by this object (and the object is also assigned in the top level environment, if objectName was given to dynamicGraphMain).

If a factor edge is added then the component edgeList with the edges of the new model should be available in the returned structure. New factor edges and factor vertices should also be avaliable in the components FactorEdges and FactorVertices.

Similar for the key events "dropEdge", "addVertex" and "dropVertex".

The methods graphEdges and setGraphEdges are used to communicate the graph components between several views of the same model in dynamicGraphMain.

The method graphEdges of the model object is for returning an object of class dg.graphedges-class to draw in the view, depending on the viewType.

[[UPDATE: If NULL is returned for a component in the returned list from graphEdges then the corresponding value of Arguments is used in redrawView. To force an empty list, return the value list() (or numeric(0) for VisibleVertices and VisibleBlocks). See also the argument factorVertexList of drawModel.]]

The method setGraphEdges of the model object is called on the model object when the model is modified.

The methods testEdge of object should return an object with the methods label and width for labeling edges, see dg.Test-class.

Slots

dg: Object of class "dg.graphedges": The graphedges of the model.

name: Object of class "character": The name of the model.

Methods

modifyModel signature(object = "dg.Model"): ...

graphEdges signature(object = "dg.Model"): ...

initialize signature(.Object = "dg.Model"): ...

setGraphEdges signature(object = "dg.Model"): ...

Str signature(object = "dg.Model"): ...

testEdge signature(object = "dg.Model"): ...

Author(s)

Jens Henrik Badsberg

References

CoCo, with a guide at http://www.jstatsoft.org/v06/i04/, has an interface to dynamicGraph.

See Also

dg.Test-class.

Examples

```
# Part of the example "defaultObjects" of demo:


# Edit the following to meet your needs:
#
# - Change the name "your.Model"
#
# - Work out how the get names, types and edges from your model object.
#
# - At "message", insert the relevant code for testing and modifying the model.
#
# - The slots visibleVertices, visibleBlocks, extraVertices, edges,
#     blockEdges, factorVertices, factorEdges should be eliminated,
#     and you should in "graphEdges" return relevant lists.
#

setClass("your.Model",
         representation(name = "character",
                        dg   = "dg.graphedges"))

# "newDefaultModelObject"<-
#   function(name)
#   {
#     result <- new("your.Model", name = name, dg = new("dg.graphedges"))
#     return(result)
#   }

setMethod("setSlots", "your.Model",
          function(object, arguments) {
            for (i in seq(along = arguments)) {
              name <- names(arguments)[i]
              if (is.element(name, slotNames(object)))
                slot(object, name) <- arguments[[i]]
              else
                message(paste("Argument '", name, "' not valid slot of '",
                              class(object), "', thus ignored.",
                              sep = "")) }
            return(object)
          })

setMethod("initialize", "your.Model",
          function(.Object, ...) {
              # print(c("initialize", "your.Model", class(.Object)))
```

```
                    Args <- list(...)
                    .Object <- setSlots(.Object, Args)
                    return(.Object)
                 }
             )

  setMethod("graphEdges", "your.Model",
            function(object, viewType = NULL, ...)
            { # print(viewType); print ("graphEdges")

                dots          <- list(...)
                localArguments <-      dots$Arguments
                Vertices       <-      localArguments$vertexList

                Edges          <- object@dg@edgeList
                VisibleVertices <- object@dg@visibleVertices

                if (viewType == "Factor") {
                  factors <- .cliquesFromEdges(Edges, Vertices, VisibleVertices)
                  # print(factors)
                  if (is.null(factors) || (length(factors) == 0)) {
                    FactorVertices  <- new("dg.FactorVertexList")
                    FactorEdges     <- new("dg.FactorEdgeList")
                  } else {
                    result <- returnFactorVerticesAndEdges(Vertices, factors)
                    FactorVertices  <- result$FactorVertices
                    FactorEdges     <- result$FactorEdges
                  }
                  new("dg.graphedges",
                      viewType        = viewType,
                      oriented        = object@dg@oriented,
                      edgeList        = object@dg@edgeList,
                      blockEdgeList   = object@dg@blockEdgeList,
                      factorVertexList = FactorVertices,
                      factorEdgeList  = FactorEdges,
                      visibleVertices = object@dg@visibleVertices,
                      visibleBlocks   = object@dg@visibleBlocks,
                      extraList       = object@dg@extraList,
                      extraEdgeList   = object@dg@extraEdgeList)
                } else if (viewType == "Moral") {
                  message("Moral view not implemented; ")
                  new("dg.graphedges",
                      viewType        = viewType,
                      oriented        = object@dg@oriented,
                      edgeList        = object@dg@edgeList,
                    # blockEdgeList   = new("dg.BlockEdgeList"),
                    # factorVertexList = new("dg.FactorVertexList"),
                    # factorEdgeList  = new("dg.FactorEdgeList"),
                      visibleVertices = object@dg@visibleVertices,
                      visibleBlocks   = numeric(),
                      extraList       = object@dg@extraList,
                      extraEdgeList   = object@dg@extraEdgeList)
                } else if (viewType == "Essential") {
```

```
                     message("Essential view not implemented; ")
                     new("dg.graphedges",
                             viewType         = viewType,
                             oriented         = object@dg@oriented,
                             edgeList         = object@dg@edgeList,
                         # blockEdgeList    = new("dg.BlockEdgeList"),
                         # factorVertexList = new("dg.FactorVertexList"),
                         # factorEdgeList   = new("dg.FactorEdgeList"),
                             visibleVertices  = object@dg@visibleVertices,
                             visibleBlocks    = numeric(),
                             extraList        = object@dg@extraList,
                             extraEdgeList    = object@dg@extraEdgeList)
                 } else if (viewType == "Simple") {
                     new("dg.graphedges",
                             viewType         = viewType,
                             oriented         = object@dg@oriented,
                             edgeList         = object@dg@edgeList,
                             blockEdgeList    = object@dg@blockEdgeList,
                         # factorVertexList = new("dg.FactorVertexList"),
                         # factorEdgeList   = new("dg.FactorEdgeList"),
                             visibleVertices  = object@dg@visibleVertices,
                             visibleBlocks    = object@dg@visibleBlocks,
                             extraList        = object@dg@extraList,
                             extraEdgeList    = object@dg@extraEdgeList)
                 } else
                     message("View type not implemented; ")
             })

    setMethod("setGraphEdges", signature(object = "your.Model"),
             function(object, dg = NULL, ...)
     {
        if (!is.null(dg)) object@dg <- dg
        return(object)
     })


    setMethod("dg", signature(object = "your.Model"),
             function(object,
                     modelObject = NULL,
                     modelObjectName = NULL,
                     control = dg.control(...),
                     ...)
      {

        Names <- Your.function.for.extracting.variable.names.from.object(
                 object = object)
        Types <- Your.function.for.extracting.variable.types.from.object(
                 object = object)
        Edges <- Your.function.for.extracting.variable.edges.from.object(
                 object = object)

        simpleGraph <- new("dg.simple.graph", vertex.names = Names,
                         types = Types, # edge.list = Edges,
```

```
                             from = Edges[,1], to = Edges[,2])

     graph <- simpleGraphToGraph(simpleGraph)

     dg(graph, object = object, ...)

 })


 setMethod("testEdge", signature(object = "your.Model"),
           function(object, action, name.1, name.2, ...)
 {
     dots <- list(...)
     from.type <- dots$from.type
     to.type <- dots$to.type
     f <- function(type) if(is.null(type)) "" else paste("(", type, ")")
     message(paste("Should return an object with the edge from",
                   name.1, f(from.type), "to", name.2, f(to.type),
                   "deleted from the argument object"))
     return(new("your.Test", name = "TestObject"))
 })


 setMethod("modifyModel", signature(object = "your.Model"),
          function(object, action, name, name.1, name.2, ...)
 {
     dots            <- list(...)
     localArguments <-      dots$Arguments
     Edges           <-      dots$newEdges$vertexEdges
     Vertices        <-      localArguments$vertexList

     viewType <- "Simple"

     DoFactors <- FALSE
     if (!is.null(dots$Arguments)
         && !is.null(dots$Arguments$factorVertexList)
         && (length(dots$Arguments$factorVertexList) > 0)
         && !is.null(dots$Arguments$vertexList))
       DoFactors <- TRUE

     if (DoFactors)
       viewType <- "Factor"

     # print(names(dots))
     # str(dots)

     # print(names(localArguments))

     # print(localArguments$visibleVertices)

     # str(localArguments$selectedNodes)
     # if (length(dots$selectedNodes) > 0)
     #   str(dots$selectedNodes)
```

```
# str(localArguments$selectedEdges)
# if (length(dots$selectedEdges) > 0)
#    str(dots$selectedEdges)

FactorVertices  <- new("dg.FactorVertexList")
FactorEdges     <- new("dg.FactorEdgeList")
BlockEdges      <- new("dg.BlockEdgeList")
VisibleVertices <- localArguments$visibleVertices
VisibleBlocks   <- localArguments$visibleBlocks
ExtraVertices   <- new("dg.VertexList")
ExtraEdges      <- new("dg.ExtraEdgeList")

f <- function(type) if (is.null(type)) "" else paste("(", type, ")")
g <- function(type) if (is.null(type)) "" else type
if (action == "dropEdge") {
  message(paste("Should return an object with the edge from",
                name.1, f(dots$from.type), "to", name.2, f(dots$to.type),
                "deleted from the argument object"))
  if ((g(dots$from.type) == "Factor") || (g(dots$from.type) == "Factor"))
    return(NULL)
} else if (action == "addEdge") {
   message(paste("Should return an object with the edge from",
                name.1, f(dots$from.type), "to", name.2, f(dots$to.type),
                "added to the argument object"))
  if ((g(dots$from.type) == "Factor") || (g(dots$from.type) == "Factor"))
    return(NULL)
} else if (action == "dropVertex")  {
   message(paste("Should return an object with the vertex",
                name, f(dots$type),
                "deleted from the argument object"))
  if ((g(dots$type) == "Factor"))
    return(NULL)
  VisibleVertices <- VisibleVertices[VisibleVertices != dots$index]
  if (DoFactors && (dots$index > 0)) {
    x <- (localArguments$factorVertexList)
    factors <- lapply(x, function(i) i@vertex.indices)
    types   <- lapply(x, function(i) class(i))
    factors <- lapply(factors,
                      function(x) {
                        y <- x[x != dots$index]
                        if (length(y) > 0) return(y) else return(NULL) } )


    if (!is.null(factors)) {
      types   <- types[unlist(lapply(factors, function(i) !is.null(i)))]
      factors <- .removeNull(factors)
    }
    if (!is.null(factors)) {
      subset <- function(x)
        lapply(x, function(a)
                   any(unlist(lapply(x,
                                     function(A)
```

```
                                              all(!is.na(match(a, A))) &&
                                              (length(a) < length(A))))))
            s <- subset(factors)
            types   <- types[!unlist(s)]
            factors <- factors[!unlist(s)]
            if (!(is.null(factors))) {
              result <- returnFactorVerticesAndEdges(
                              localArguments$vertexList, factors, types,
                              factorClasses = validFactorClasses())
              FactorVertices <- result$FactorVertices
              FactorEdges <- result$FactorEdges
            }
          } else {
            DoFactors <- FALSE
            FactorVertices <- new("dg.FactorVertexList")
            FactorEdges     <- new("dg.FactorEdgeList")
          }
        }
    } else if (action == "addVertex") {
      VisibleVertices <- c(VisibleVertices, dots$index)
      message(paste("Should return an object with the vertex",
                    name, f(dots$type), dots$index,
                    "added to the argument object"))
      if (DoFactors && (dots$index > 0)) {
        x <- (localArguments$factorVertexList)
        factors <- lapply(x, function(i) i@vertex.indices)
        types   <- lapply(x, function(i) class(i))
        if (!is.null(factors))
          factors <- .removeNull(factors)
        if (is.null(factors)) {
          factors <- list(dots$index)
          types   <- validFactorClasses()[1, 1]
        } else {
          n <- length(types)
          factors <- append(factors, list(dots$index))
          types   <- append(types, types[n])
        }
        if (!(is.null(factors))) {
          result <- returnFactorVerticesAndEdges(
                          localArguments$vertexList, factors, types,
                          factorClasses = validFactorClasses())
          FactorVertices <- result$FactorVertices
          FactorEdges <- result$FactorEdges
        }
      }
    }
    if (is.null(FactorVertices) && DoFactors && !is.null(Edges)) {

      factors <- .cliquesFromEdges(Edges, Vertices, VisibleVertices)

      if (is.null(factors) || (length(factors) == 0)) {
        FactorVertices <- new("dg.FactorVertexList")
        FactorEdges     <- new("dg.FactorEdgeList")
```

```
      } else {
        result <- returnFactorVerticesAndEdges(Vertices, factors)
        FactorVertices  <- result$FactorVertices
        FactorEdges     <- result$FactorEdges
      }
    }
    dg <- new("dg.graphedges",
              edgeList        = Edges,
              viewType        = viewType,
          #   oriented        = oriented,
              blockEdgeList   = BlockEdges,
              factorVertexList = FactorVertices,
              factorEdgeList  = FactorEdges,
              visibleVertices = VisibleVertices,
              visibleBlocks   = VisibleBlocks,
              extraList       = ExtraVertices,
              extraEdgeList   = ExtraEdges)
    ".IsEmpty" <- function(x) {
      if (is.null(x) || (length(x) == 0) ||
          (length(x) == 1) && is.null(x[[1]]))
        return(TRUE)
      else
        return(FALSE)
      }
    if (.IsEmpty(FactorEdges) && (viewType == "Factor")) {
      object <- setGraphEdges(object, dg = dg)
      graphContent <- graphEdges(object, viewType = viewType,
                                 Arguments = localArguments)
      dg <- graphContent
    }
    return(list(object  = object, dg = dg))
 })

setMethod("Str", "your.Model",
          function(object, setRowLabels = FALSE, title = "", ...) {
              message(object@name) })


new("your.Model", name = "YourModelObject")
```

---

dg.Node-class   *Class dg.Node*

Description

A skeleton class for the classes of vertices, edges and block objects.

Objects from the Class

The objects has the slots and methods relevant to vertices, edges and blocks.

Slots

color: Object of class "character" with the color of the object.

label: Object of class "character" with the label of the object.

label.position: Object of class "numeric" with the label.positions of the object.

Methods

addToPopups signature(object = "dg.Node"): Add items to the pop up menu (nodePop\-upMenu) of the object by tkadd.

color<- signature(x = "dg.Node"): Set the color of the object.

color signature(object = "dg.Node"): Return the color of the object.

label<- signature(x = "dg.Node"): Set the label of the object.

label signature(object = "dg.Node"): Return the label of the object.

labelPosition<- signature(x = "dg.Node"): Set the label.position of the object.

labelPosition signature(object = "dg.Node"): Return the label.position of the object.

propertyDialog signature(object = "dg.Node"): Open a Tk/tcl-window with the slots of the node.

setSlots signature(object = "dg.Node"): ...

Author(s)

Jens Henrik Badsberg

See Also

`dg.Vertex-class`, `dg.Edge-class`, `dg.Block-class`.

---

dg.simple.graph-class *Class dg.simple.graph*

Description

A simple representation of a graph for dynamicGraph. Vertices are represented by text-strings for names or numbers for indices, edges by indices or names of vertices, etc.

Objects from the Class

Objects can be created by calls of the form new("dg.simple.graph", ...).

Slots

> `viewType`: Object of class `"character"`: A text string with the type of view.
>
> `vertex.names`: Object of class `"vector"`: A vector with text strings for the `names` of the vertices.
>
> `types`: Object of class `"character"`: A vector with text strings for the `types`, labels of `dg.Vertex`, of the vertices.
>
> `labels`: Object of class `"vector"`: A vector with text strings for the `labels` of the vertices.
>
> `from`: Object of class `"vector"`: If not `edge.list` is given: The indices of the first endpoints of the edges.
>
> `to`: Object of class `"vector"`: If not `edge.list` is given: The indices of the second endpoints of the edges.
>
> `edge.list`: Object of class `"list"`: If not `from` and `to` are given: A list where each item specifies an edge by a vector of the indices or names of the vertices.
>
> `edge.types`: Object of class `"character"`: A vector of text strings giving the `types` of the edges, identify which classes the edges should be of, containing the dg.VertexEdge.
>
> `blocks`: Object of class `"list"`: A list defining the `blocks`: Each item is the vector of the indices of the vertices of the block, or the vector with text strings for the names of the vertices of the block. The arguments `right.to.left`, `nested.blocks` and `blockColors` are here used in `setBlocks` to control the layout of the blocks.
>
> `block.tree`: Object of class `"list"`: If not the argument `blocks` is used: A structure with the blocks in a `block.tree`. The arguments `overlaying` and `blockColors` are here used in `setTreeBlocks` to control the layout of the blocks.
>
> `oriented`: Object of class `"logical"`: Logical. If `oriented` is set to TRUE then the edges are oriented, also when no block structure is given.
>
> `factors`: Object of class `"list"`: A list defining the `factor vertices`: Each item is the vector of the indices of the vertices of a factor.
>
> `texts`: Object of class `"character"`: A vector of text strings, for additional labels. These labels will be set by 'ExtraVertices' of class `dg.TextVertex-class`.
>
> `extra.from`: Object of class `"vector"`: If not `extra.edge.list` is given: The indices of the first endpoints of the extra edges, negative for extra vertices.
>
> `extra.to`: Object of class `"vector"`: If not `extra.edge.list` is given: The indices of the second endpoints of the extra edges, negative for extra vertices.
>
> `extra.edge.list`: Object of class `"list"`: If not `extra.from` and `extra.to` are given: A list where each item specifies an extra edge by a vector of the indices or names of the vertices or extra vertices, negative indices for extra vertices.

Methods

> **addModel** signature(object = "dg.simple.graph"): ...
>
> **addView** signature(object = "dg.simple.graph"): ...
>
> **coerce** signature(from = "dg.simple.graph", to = "dg.graph"): ...
>
> **dg** signature(object = "dg.simple.graph"): ...
>
> **replaceModel** signature(object = "dg.simple.graph"): ...
>
> **replaceView** signature(object = "dg.simple.graph"): ...

Author(s)

Jens Henrik Badsberg

See Also

[dg.graph-class](), and [DynamicGraph]().

Examples

```
x <- new("dg.simple.graph", vertex.names = c("a", "b"),
         from = 1, to = 2) ; str(x)
dg(new("dg.simple.graph", vertex.names = c("a", "b"),
       edge.list = list(c(1, 2))))
```

---

dg.Test-class   *Class dg.Test*

Description

An example class for the test object for the model object of dynamicGraph.

Value

An object of class dg.Test.

Objects from the Class

The methods label and width should be implemented by you for your test object returned by the method testEdge of your model object.

Slots

deviance: Object of class "numeric": The deviance of the test.

df: Object of class "numeric": The df of the test.

p: Object of class "numeric": The p-value of the test.

Methods

label signature(object = "dg.Test"): Return the label of the test.

width signature(object = "dg.Test"): Return the width of the test.

initialize signature(.Object = "dg.Test"): ...

setSlots signature(object = "dg.Test"): ...

Author(s)

Jens Henrik Badsberg

See Also

dg.Model-class.

Examples

```
# Part of the example "defaultObjects" of demo:

setClass("your.Test",
         representation(name = "character",
                        deviance = "numeric", df = "numeric", p = "numeric"))

setMethod("setSlots", "your.Test",
          function(object, arguments) {
            for (i in seq(along = arguments)) {
              name <- names(arguments)[i]
              if (is.element(name, slotNames(object)))
                slot(object, name) <- arguments[[i]]
              else
                message(paste("Argument '", name, "' not valid slot of '",
                              class(object), "', thus ignored.",
                              sep = "")) }
            return(object)
          })

setMethod("initialize", "your.Test",
          function(.Object, ...) {
              # print(c("initialize", "your.Test", class(.Object)))
              Args <- list(...)
              if (!is.element("df", names(Args)) ||
                  !is.element("deviance", names(Args))) {
                Args <- (Args[!names(Args) == "df"])
                Args <- (Args[!names(Args) == "deviance"])
                .Object@df <- round(runif(1, 1, 25))
                .Object@deviance <- rchisq(1, .Object@df)
                .Object@p <- 1 - pchisq(.Object@deviance, .Object@df)
                message("Just generating a random test!!!!")
              }
              .Object <- setSlots(.Object, Args)
              return(.Object)
            }
          )


if (!isGeneric("label") && !isGeneric("label", where = 2)) {
  if (is.function("label"))
    fun <- label
  else
    fun <- function(object) standardGeneric("label")
  setGeneric("label", fun)
}

setMethod("label", "your.Test",
```

```
          function(object) format(object@p, digits = 4))

if (!isGeneric("width") && !isGeneric("width", where = 2)) {
  if (is.function("width"))
    fun <- width
  else
    fun <- function(object) standardGeneric("width")
  setGeneric("width", fun)
}

setMethod("width", "your.Test",
          function(object) round(2 + 5 * (1 - object@p)))

new("your.Test", name = "TestObject")
```

---

dg.TextVertex-class   *Class dg.TextVertex*

Description

The class for vertices for setting text string in the graph window. These vertices are given by the argument extraList to [dynamicGraphMain](#).

Objects from the Class

The vertices of nodes of this class are drawn with very small symbols. Objects has the methods for extracting and setting the slots for vertices.

Slots

name: Object of class "character" with the name of the vertex.

index: Object of class "numeric" with the index of the vertex, the position of the vertex in the extra list.

position: Object of class "numeric" with the position of the vertex. Vertices in the same dynamic graph should have the same number of coordinates. A small dot is placed at the position of the vertex. This dot can be move outside the window.

blockindex: Object of class "numeric" with the blockindex of the vertex.

stratum: Object of class "numeric" with the stratum of the vertex.

constrained: Object of class "logical", see "dg.Vertex".

color: Object of class "character" with the color of the vertex.

label: Object of class "character" with the label of the vertex.

label.position: Object of class "numeric" with the label.position of the vertex. Labels of vertices in the same dynamic graph should have the same number of coordinates.

Extends

Class "dg.Node", directly. Class "dg.Vertex", directly.

Methods

draw signature(object = ″dg.TextVertex″): ...

Author(s)

Jens Henrik Badsberg

See Also

dg.Vertex-class, returnVertexList, dg.Node-class, and dg.ExtraEdge-class.

---

dg.Vertex-class   *Class dg.Vertex*

Description

A skeleton class for the classes of vertices.

Objects from the Class

Objects has the methods for extracting and setting the slots for vertices.

Slots

name: Object of class ″character″ with the name of the vertex. Should be a valid name on variables for your model object.

index: Object of class ″numeric″ with the index of the vertex, the position of the vertex in a vertex list.

position: Object of class ″numeric″ with the position of the vertex. Vertices in the same dynamic graph should have the same number of coordinates.

blockindex: Object of class ″numeric″ with the blockindex of the vertex.

stratum: Object of class ″numeric″ with the stratum of the vertex.

constrained: Object of class ″logical″: If TRUE, then the vertex can not be dragged out of the block of the vertex.

color: Object of class ″character″ with the color of the vertex.

label: Object of class ″character″ with the label of the vertex.

label.position: Object of class ″numeric″ with the label.position of the vertex. Labels of vertices in the same dynamic graph should have the same number of coordinates.

Extends

Class ″dg.Node″, directly.

Methods

> **blockindex<-** signature(x = "dg.Vertex"): ...
>
> **blockindex** signature(object = "dg.Vertex"): ...
>
> **index<-** signature(x = "dg.Vertex"): ...
>
> **index** signature(object = "dg.Vertex"): ...
>
> **initialize** signature(.Object = "dg.Vertex"): ...
>
> **name<-** signature(x = "dg.Vertex"): ...
>
> **name** signature(object = "dg.Vertex"): ...
>
> **position<-** signature(x = "dg.Vertex"): ...
>
> **position** signature(object = "dg.Vertex"): ...
>
> **stratum<-** signature(x = "dg.Vertex"): ...
>
> **stratum** signature(object = "dg.Vertex"): ...
>
> **visible<-** signature(x = "dg.Vertex"): ...
>
> **visible** signature(object = "dg.Vertex"): ...
>
> **constrained<-** signature(x = "dg.Vertex"): ...
>
> **constrained** signature(object = "dg.Vertex"): ...
>
> **propertyDialog** signature(object = "dg.Node"): ...
>
> **ancestors<-** signature(x = "dg.Vertex"): Not implemented.
>
> **ancestors** signature(object = "dg.Vertex"): Not implemented.
>
> **descendants<-** signature(x = "dg.Vertex"): Not implemented.
>
> **descendants** signature(object = "dg.Vertex"): Not implemented.

Note

The dg.Vertex class has the methods name, label, labelPosition, position, stratum, blockindex, constrained,

color, and index for extracting values of the object and the replacement methods name<-, label<-, labelPosition<-, position<-, stratum<-, blockindex<-, constrained<-,

color<-, and index<-. The method draw is used to draw a vertex, and items can be added to the pop up menu of the vertex by the method addToPopups.

Some of these methods also applies for edges (dg.Edge), blocks (dg.Block),

block edges (dg.BlockEdge), factor vertices (dg.FactorVertex) and edges from vertices to factors (dg.FactorEdge).

Author(s)

Jens Henrik Badsberg

See Also

returnVertexList, dg.Node-class.

Examples

```
a <- new("dg.DiscreteVertex", name = "a", label = "A",
         index = 1, position = c(0, 0, 0))

str(a)

color(a)
label(a)
labelPosition(a)
name(a)
index(a)
position(a)
stratum(a)

color(a) <- "red"
label(a) <- "A vertex"
labelPosition(a) <- c(1, 2, 3)
name(a) <- "Capital.A"
index(a) <- -1
position(a) <- c(10, 20, 30)
stratum(a) <- 1

str(a)
```

---

dg.VertexEdge-class   *Class dg.VertexEdge*

Description

The class for edges between vertices.

Details

Edges are not constrained to have two vertices.

Objects from the Class

Objects has the methods for extracting and setting the slots for edges, and the method for drawing the edge.

Slots

oriented: Object of class "logical". If oriented is NA then the edge is drawn as an arrow if the
    vertices of the edge are in different blocks, oriented according to the strata of the blocks. If
    oriented is NA and the vertices of the edge are in the same block, then an undirected edge is
    drawn. If oriented is TRUE then an arrow is drawn from the first vertex of the edge to the
    second. If oriented is FALSE then an undirected edge is drawn, also between blocks.

vertex.indices: Object of class "numeric", see "dg.Edge". These are the indices of the vertices
    in the list of vertices.

width: Object of class "numeric", see "dg.Edge".

dash: Object of class "character", see "dg.Edge".

color: Object of class "character", see "dg.Edge".

label: Object of class "character", see "dg.Edge".

label.position: Object of class "numeric", see "dg.Edge".

Extends

Class "dg.Edge", directly. Class "dg.Node", directly.

Methods

nodeTypesOfEdge signature(object = "dg.VertexEdge"): ...

oriented<- signature(x = "dg.VertexEdge"): ...

oriented signature(object = "dg.VertexEdge"): ...

propertyDialog signature(object = "dg.VertexEdge"): ...

Note

The dg.Edge class has beside the methods of dg.Vertex the methods oriented and oriented<-.

The method nodeIndicesOfEdge will extract the indices of the vertices of the edge, and the method nodeTypesOfEdge will extract the types ("super classes": vertex, factor or block) of the vertices (nodes) of an edge. The method draw is used to draw the edge, and items are added to the pop up menu of an edge by the method addToPopups.

Some of these methods also applies for block edges (dg.BlockEdge) and factor edges (dg.FactorEdge).

Author(s)

Jens Henrik Badsberg

See Also

newVertexEdge, returnEdgeList, dg.Edge-class.

Examples

```
vertices <- returnVertexList(paste("V", 1:4, sep = ""))
e <- new("dg.VertexEdge", vertex.indices = c(1, 2, 3),
         vertices = new("dg.VertexList", vertices[1:3]))

str(e)

color(e)
label(e)
labelPosition(e)
width(e)
oriented(e)
nodeIndicesOfEdge(e)
```

```
nodeTypesOfEdge(e)

color(e) <- "Black"
label(e) <- "1-2"
labelPosition(e) <- c(10, 20, 30)
width(e) <- 1
oriented(e) <- TRUE
nodeIndicesOfEdge(e) <- c(1, 2)

str(e)
```

---

drawModel   *DEPRECATED: Draw the dynamicGraph window and slaves*

Description

The functions drawModel and redrawView within dynamicGraph is for adding models to dynamicGraph,
for adding new views of a model, and for overwriting an existing view with an other model.

The functions can not be found at top level.

The functions are called by the methods [addModel](#), [addView](#), [replaceModel](#), and [replaceView](#).

Arguments

frameModels	An object of class [DynamicGraph-class](#). NULL, or frameModels of list(...)$Arguments.
frameViews	An object of class [DynamicGraphModel-class](#). NULL, or frameViews of list(...)$Arguments. If frameViews is set to NULL, the default value, then a new model frame will be created by drawModel.
graphWindow	An object of class [DynamicGraphView-class](#). If graphWindow is set to the value of list(...)$Arguments$graphWindow then the calling graph window will be redrawn. If graphWindow is set to NULL, the default value, then a new slave graph window will be drawn.
dg	As for [dynamicGraphMain](#). If dg is given (set to a value different from NULL) then this value is used, else the value extracted from list(...)$Arguments is used.
object	As for [dynamicGraphMain](#). If object is given then this value is used, else the value extracted from list(...)$Arguments is used.
frameModelsEnv	frameModelsEnv is then environment for storing hidden values of the frameModels. Extracted from frameModels by default.
initialWindow returnNewMaster	Logical, if initialWindow is TRUE then the labels of the edges are updated.
	Logical, if returnNewMaster is TRUE then ....
redraw setUpdateCountModelMain	Logical, if redraw is TRUE then ....
	Logical. If setUpdateCountModelMain is TRUE then views of the same model will be updated.

returnFrameModel

        Logical, if `returnFrameModel` is TRUE then ....

control   Options for `dynamicGraphMain`, see `dg.control`.

...   Used to porting `list(...)$Arguments`.

Details

The `drawModel` and `redrawView` functions can be called from the functions of menus (main menu and pop up menus) of `dynamicGraphMain`, from .GlobalEnv in `DynamicGraph`) via returned values from `dynamicGraphMain` (and from the methods of the model object in the scope of the function `dynamicGraphMain`). As a result the graph window will be redrawn with an other view of the model, possible with, e.g., other edges, an other model is drawn, or a new slave graph window will appear.

If the value of a argument to `drawModel` or `redrawView` is set, then this value is used, else the value from the calling window is used. The value of the calling window is given in the argument `Arguments` in the call of the function of the menu item.

Below is an example, where items for labeling all the edges of the graph are added to the menu. The edges are visited, a test is computed for each edge, the label and width of the edge is updated, and the graph is drawn with the updated edge list.

Value

The returned value from `dynamicGraphMain`.

Note

The functions can not be called from top level, that is, the functions does not exists at .GlobalEnv, but only in returned values from `dynamicGraphMain`.

It is recommended that the functions not are called, but that `DynamicGraph` is used with the arguments `frameModels`, `frameViews`, `graphWindow`, `addModel`, `addView`, and/or `overwrite` to call the functions.

Author(s)

Jens Henrik Badsberg

See Also

See also `dynamicGraphMain`, `DynamicGraph` `DynamicGraph-class`, and `DynamicGraphModel-class`.

DynamicGraph   *DEPRECATED: Simple interface to dynamicGraph*

Description

A simple interface to dynamicGraph in the sense that the graph should not be given as an object as to dynamicGraphMain. Here vertices can be specified by a vector of text strings with names, and/or edges by pairs of the indices of the vertices.

The function can also be used to add models and views to an existing dynamicGraph.

The interface is deprecated: Use the method dg on an object of class dg.simple.graph-class instead, or the methods addModel, addView, replaceModel, or replaceView.

Usage

```
DynamicGraph(names = character(), types = character(),
             from = vector(), to = vector(), edge.list = list(NULL),
             labels = names, edge.types = character(),
             blocks = list(NULL), block.tree = list(NULL), oriented = NA,
             factors = list(NULL), texts = character(),
             extra.from = vector(), extra.to = vector(),
             extra.edge.list = list(NULL),
             object = NULL, viewType = "Simple",
             frameModels = NULL, frameViews = NULL, graphWindow = NULL,
             addModel = FALSE, addView = FALSE, overwrite = FALSE,
             returnNewMaster = FALSE, redraw = FALSE,
             control = dg.control(...), ...)
```

Arguments

names	See vertex.names of dg.simple.graph-class.
types	See dg.simple.graph-class.
from	See dg.simple.graph-class.
to	See dg.simple.graph-class.
edge.types	See dg.simple.graph-class.
edge.list	See dg.simple.graph-class.
labels	See dg.simple.graph-class.
blocks	See dg.simple.graph-class.
block.tree	See dg.simple.graph-class.
oriented	See dg.simple.graph-class.
factors	See dg.simple.graph-class.
texts	See dg.simple.graph-class.
extra.from	See dg.simple.graph-class.

extra.to	See `dg.simple.graph-class`.
extra.edge.list	
	See `dg.simple.graph-class`.
viewType	See `dg.simple.graph-class`.
object	The model object, or NULL, see `dg.Model-class`.
frameModels	An object of class `DynamicGraph-class`. frameModels is the object for a dataset and the models on that dataset.
frameViews	An object of class `DynamicGraphModel-class`. frameViews is the object for a model and the views of that model.
graphWindow	An object of class `DynamicGraphView-class`. graphWindow is the object for a view of a model.
addModel	Logical, if addModel then a model is added to the argument frameModels, and a view of the model is drawn. If the argument overwrite is TRUE and the argument graphWindow is given then the model of graphWindow is replaced by the model argument object. If the argument overwrite is TRUE and the argument frameViews is given then the model of frame\-Views is replaced by the model argument object.
addView	Logical, if addView then a view of type set by the argument viewType for the model of the argument frameViews is added.
overwrite	Logical, see the argument addModel.
redraw	Logical. If TRUE then the dynamicGraph of the arguments frameModels is 'redrawn'. New instances of the windows are made.
returnNewMaster	
	Logical. Alternative implementation of addModel, using the code of redraw. As redraw, but the windows of frameModels exists, and a new model is added.
control	Options for `DynamicGraph` and `dynamicGraphMain`, see `dg.control`.
...	Additional arguments to `dynamicGraphMain`.

Details

After converting the arguments for the graph first to an object of class `dg.simple.graph-class` then to an object of class `dg.graph-class` the function `dynamicGraphMain` does all the work.

The list of objects can be exported from `dynamicGraphMain`, also after modifying the graph.

Value

The returned value from `dynamicGraphMain`.

Author(s)

Jens Henrik Badsberg

Examples

```
require(tcltk); require(dynamicGraph)

# Example 1:

W <- dg(as(new("dg.simple.graph", vertex.names = 1:5), "dg.graph"),
        control = dg.control(title = "Very simple"))

# Example 2:

W <- dg(new("dg.simple.graph", from = 1:4, to = c(2:4, 1)),
        control = dg.control(title = "Simply edges"))

# Example 3:

V.Types <- c("Discrete", "Ordinal", "Discrete",
             "Continuous", "Discrete", "Continuous")

V.Names  <- c("Sex", "Age", "Eye", "FEV", "Hair", "Shosize")
V.Labels <- paste(V.Names, 1:6, sep ="/")

From <- c(1, 2, 3, 4, 5, 6)
To   <- c(2, 3, 4, 5, 6, 1)

W <- dg(new("dg.simple.graph", vertex.names = V.Names, types = V.Types,
            labels = V.Labels, from = From, to = To),
        control = dg.control(title = "With labels (extraVertices)"))


# Example 4: Oriented (cyclic) edges, without causal structure:

W <- dg(new("dg.simple.graph", vertex.names = V.Names, types = V.Types,
            labels = V.Labels, from = From, to = To, oriented = TRUE),
        control = dg.control(title = "Oriented edges"))


# Example 5: A factor graph:

Factors <- list(c(1, 2, 3, 4), c(3, 4, 5), c(4, 5, 6))

W <- dg(new("dg.simple.graph", vertex.names = V.Names, types = V.Types,
          labels = V.Labels, factors = Factors, viewType = "Factor"),
        control = dg.control(title = "Factorgraph", namesOnEdges = FALSE))


# Example 6: Edges with more than two vertices:

EdgeList <- list(c(1, 2, 3, 4), c(3, 4, 5), c(4, 5, 6))

W <- dg(new("dg.simple.graph", vertex.names = V.Names, types = V.Types,
          labels = V.Labels, edge.list = EdgeList),
        control = dg.control(title = "Multiple edges", namesOnEdges = FALSE))
```

W

DynamicGraph-class   *Class DynamicGraph*

Description

The class of the object of dynamicGraph.

Objects from the Class

An object of this class DynamicGraph-class is the returned value from the method dg and from the function dynamicGraphMain.

The object has the lists of vertices and blocks of the dynamicGraph, and a list of models, each (of class DynamicGraphModel-class) with the model and the views of the model.

Each view (of class DynamicGraphView-class) of a model will hold the edges (edges between vertices, factors and blocks) and factor- and extra-vertices of the view, together with vectors of indices of vertices and blocks visible in the view.

Objects of the class DynamicGraph-class is the outer frame holding the "data" and the models, each model is of the class DynamicGraphModel-class with one or more views of the class DynamicGraphView-class in a sub-frame for the model.

The input to dynamicGraphMain should be of the class dg.graph-class.

Slots

id.env: Object of class "character": Internal identification of the object.

label: Object of class "character": The value of the argument label of dynamicGraphMain, the header label, tktitle, of the window, also printed by Str.

vertices: Object of class "dg.VertexList": The vertexList of the graph window.

blocks: Object of class "dg.BlockList": The blockList of the graph window.

control: Object of class "list" with the options of the functions dynamicGraphMain and simpleGraphToGraph.

models: Object of class "list": A list of objects, each of class DynamicGraphModel-class.

Methods

label<- signature(x = "DynamicGraph"): ...

label signature(object = "DynamicGraph"): ...

vertices<- signature(x = "DynamicGraph"): ...

vertices signature(object = "DynamicGraph"): ...

blocks<- signature(x = "DynamicGraph"): ...

blocks signature(object = "DynamicGraph"): ...

control<- signature(x = "DynamicGraph"): ...

control signature(object = "DynamicGraph"): ...

models<- signature(x = "DynamicGraph"): ...

models signature(object = "DynamicGraph"): ...

Str signature(object = "DynamicGraph"): Compactly display the internal *str*ucture of a dynamicGraph object, using [asDataFrame](#) on each list of node objects.

show signature(object = "DynamicGraph"): calls the method [Str](#).

dg signature(object = "DynamicGraph"): Redraw the object.

Author(s)

Jens Henrik Badsberg

See Also

[dynamicGraphMain](#), [DynamicGraphModel-class](#), and [DynamicGraphView-class](#).

---

dynamicGraphMain   *Dynamic Graph*

Description

Interactive plot for manipulating graphs.

Usage

```
dynamicGraphMain(vertexList = NULL, blockList = NULL,
                 dg = NULL, object = NULL, objectName = NULL,
                 control = dg.control(...), ...)
```

Arguments

vertexList	List of vertices (each of class containing the class dg.Vertex) created by [returnVertexList](#) or exported from [dynamicGraphMain](#).
blockList	List of blocks (each of class dg.Block) created by [setBlocks](#) or exported from [dynamicGraphMain](#).
dg	dg is an object of class [dg.graphedges-class](#).
object	NULL, or object with the methods [modifyModel](#) and [testEdge](#) - for respectively updating the object when the graph is updated, and for computing the test statistic when an edge is labeled. The returned object from [testEdge](#) should have the methods label and width
	for extracting the label of a test for putting the label on the edge and for extracting the width for the edge. See [dg.Model-class](#).
objectName	If set to a text string then the object is assigned with this name in .Global.Env when the object is updated.
control	Options for [dynamicGraphMain](#), see [dg.control](#).
...	Additional arguments.

Details

This is a dynamic plotting tool for handling graphs. The lay out of the graph can be edited by moving the vertices of the graph by the mouse, and edges can be added by clicking vertices and dropping by clicking the edges.

The function is incremental in the sense that the user can add a method for updating the model object of the window when the graph is updated, and a method for computing the test of an edge when the edge is clicked by the mouse.

Edges can be oriented, drawn by arrows.

Blocks can be used to define a causal structure of the variables represented by vertices, and edges between blocks are then oriented. Blocks can be given in a structure such that descendant blocks of a blocks also are closed when a block is closed.

A secondary set of vertices, factor vertices, can be used to represent hypergraphs.

"Slave graph windows" can be created: The windows will share vertices and blocks, thus when a vertex is moved in one window, the position of the vertex will also change in all slave windows. The edges are not shared among windows, since the individual windows will typical represent different models. Thus factors (vertices and edges) are not shared between graph windows.

Value

An object of class `DynamicGraph-class` (if not `returnNull` is TRUE) with the lists of vertices and blocks (block trees) of the dynamicGraph, and list of models, each (of class `DynamicGraphModel-class`) with the views of the model.

Each view (of class `DynamicGraphView-class`) of a model will hold the edges (edges between vertices, factors and blocks) and factor- and extra-vertices of the view, together with which vertices and blocks are visible in the view.

All object in the graph window:

- Left click, hold and drag:The object will move.
- Left click:Action to the object: Vertices, edges, and blocks will highlight, at vertices and blocks edges are added after highlighting, tests are computed for edge labels.
- Left or right click, with SHIFT or/and CONTROL:The object will be marked.
- Double left click:Action to object: Vertices and edges are deleted, blocks will close, closed blocks will open.
- Right click:The pop up menu of the object will appear.

Vertices (vertices and factor vertices):

Right click the vertex to get the pop up menu of the vertex:

- Highlight a vertex:For adding an edge - Left click the vertex.
- Highlight a vertex:For adding to "selectedNodes" - Left or right click the vertex while holding SHIFT or/and CONTROL down.
- Mark a vertex:For adding edges, etc. - Left or right click the vertex while holding SHIFT or/and CONTROL down.

- Cancel highlighting:Left click (or drag) the vertex.
- Add an edge:After first highlighting a vertex - Left click the other vertex.
- Move a vertex:Left click and drag the vertex.
- Move a vertex label:Left click and drag the label of the vertex.
- Delete a vertex:Double left click the vertex.
- Create new graph:A slave window with the vertex delete - Select "Drop vertex" from pop up menu at the vertex.
- Change a vertex label:Double left click the label, and enter the new label in the appearing dialog window.
- Delete a vertex label:Select "Delete vertex label" from the pop up menu at the vertex or at the vertex label.
- Create new vertex:At mouse position - Middle click the canvas.
- Create new vertex:At mouse position with the edge to last vertex - Double middle click the canvas.

The main menu "Variables":

- Create new variable:With setting the class of the new vertex and defining an expression for updating the `object` - Select "Create new variable".
- Display, add, a vertex:In the current window - Select "Select vertex among variables not displayed (here)".
- Display, add, a vertex:In a slave window - Select "Select vertex among variables not displayed (slave)".
- Export the vertex list:Select "Assign 'vertexList' in .GlobalEnv".
- Export the labels:Select "Assign 'extraList' in .GlobalEnv".

Edges (edges to/from vertices, blocks and factors):

Right click the edge to get the pop up menu of the edge:

- Highlight a edge:Left click the edge.
- Highlight a edge:For adding to "selectedEdges" - Left or right click the edge while holding SHIFT or/and CONTROL down.
- Add an edge:Left click first the vertex (or block) to highlight, and then left click other vertex (or block).
- Delete an edge:And update in the current window - Double left click the edge.
- Delete an edge:And create a slave graph with the resulting graph - Select "Drop edge" from the pop up menu at the edge.
- Move an edge (2 vertices):Left click the edge and drag the edge.
- Move an edge label:Left click the edge label and drag the label.
- Set an edge label:Select "Set edge label" from the edge pop up menu.
- Compute an edge label:Left click the edge label, or select "Compute edge label" from pop up menu at the edge.

- Compute an edge label:Force computation for "harder models" - Double left click the edge label, or select "Force compute edge label" from pop up menu of the edge.

- Delete an edge label:Triple left click the edge label, or select "Delete label of edge" from the pop up menu at the edge.

The main menu "Edges":

- Delete all edge labels:Select "Delete all edge labels".

- Export the edge list:Select "Assign 'edgeList' in .GlobalEnv".

Blocks, opened:

Right click the block label (or colored block canvas if `drawBlockBackground` is set to TRUE) to get the pop up menu of opened block:

- Move a block:With its sub blocks and vertices - Left click the block label or colored block canvas and drag the block. [Slow !!!]

- Resize a block:Left click a block corner or a block edge and drag.

- Minimize a block:With its sub blocks and vertices - Double left click the block label or the colored block canvas.

- Maximize a block:Zoom to the block - Right click the block label or colored the block canvas and select "Maximize" in the appering popup menu.

- Zoom out to the full graph:Right click the block label or colored block canvas and select "Redraw (Zoom to) full graph" in the appearing block pop up menu.

Blocks, closed:

Right click the block to get the pop up menu of the closed block:

- Highlight a block:For adding edges from all vertices of the block to a vertex or a block - Left click the block.

- Highlight a block:For adding to "selectedNodes" - Left or right click the block while holding SHIFT or/and CONTROL down.

- Mark a block:For adding edges, etc. - Left or right click the block while holding SHIFT or/and CONTROL down.

- Cancel highlighting:Of a block - Left click (or drag) the block.

- Add edges:To all vertices of block after highlighting first a block or a vertex - Click the other block.

- Move a closed block:Left click and drag the block.

- Move a block label:Left click and drag the label of the block.

- Open a closed block:Double left click the block.

- Change a block label:Double left click the label of the closed block, and enter the new label in the appearing dialog window.

- Delete a block label:Select "Delete label of block" from the pop up menu at the block or the block label.

The main menu "Blocks":

- Export the block list:Select "Assign 'blockList' in .GlobalEnv".
- Export the block edges:Select "Assign 'blockEdgeList' in .GlobalEnv".

Factor vertices:

Right click the factor vertex to get the pop up menu of the factor: Actions as for vertices.
The main menu "Generators":

- Export the factor vertices:Select "Assign 'factorVertexList' in .GlobalEnv".
- Export the factor edges:Select "Assign 'factorEdgeList' in .GlobalEnv".

Factor edges:

Right click the factor edge to get the pop up menu of the factor edge: Actions are as for edges.

The panel for vertices:

- Highlight vertex name for adding or deleting vertex: Left click the vertex name.
- Delete or add vertices: Double left click a vertex name.
- Popup menu for selected vertex: Click "Popup selected in panel" in "File Menu".
- Dialog window for properties: Middle click vertex name.

The panel for blocks in tree and vertices:

- Move vertex to other block: Left click the vertex name and drag to the other block.
- Move block to other block: Left click the block name and drag to the other block.
- Popup menu for selected vertex or block : Click "Popup selected in panel" in "File Menu".

The graph:

- Create a slave window:Select "Make slave window: Same model".
- Create a slave window:Select "Make slave window: Copy model".
- Switch class of view:Select "Set class of graph window".
- Refresh:Faster fix of "corrupted" window: Select "Refresh view (set positions as 'stored')".
- Redraw the view (zoom out):Select "Redraw graph window (more refreshing)".
- Update model:Select "Update model and redraw (total refreshing)".
- Enable rotation:Select "Rest (enable) rotation".
- Disable rotation:Select "Disable rotation".
- Zoom in:Select "Zoom in" from main menu, or hit <F1> in canvas.
- Zoom out:Select "Zoom out" from main menu, or hit <F2> in canvas.
- Export current arguments:Select "Assign 'Args' in .GlobalEnv".
- Export the model lattice:Select "Assign 'frameModels' in .GlobalEnv".
- Export the graph lattice:Select "Assign 'frameViews' in .GlobalEnv".
- Export the graph window:Select "Assign 'graphWindow' in .GlobalEnv".
- Export the object:Select "Assign 'object' in .GlobalEnv".

Rotation:

- Enable rotation:Select "Enable (reset) transformation" from the main menu "Graph".

- Rotate the graph:Middle click the canvas, and drag.

- Disable rotation:Select "Disable transformation" from the main menu "Graph".

- Export transformation:Export the projection matrix - Select "Assign 'transformation' in .GlobalEnv" from the main menu "Graph".

Acknowledgments

Many thanks to the gR-group for useful discussions, especially to Claus Dethlefsen for testing early versions of this package on DEAL.

Note

Vertices, edges, blocks, block edges, factors, and factor edges are objects of the following classes: dg.Vertex, dg.Edge, and dg.Block contains dg.Node, dg.FactorVertex contains dg.Vertex, and dg.VertexEdge, dg.BlockEdge, dg.FactorEdge contains dg.Edge.

The methods draw, color, color<-, label, label<-, labelPosition, labelPosition<-, name, name<-, index, index<-, position, position<-, stratum, stratum<-, visible, visible<-, addToPopups, oriented, oriented<-, width, width<-, nodeIndicesOfEdge, nodeIndicesOfEdge<-, nodeTypesOfEdge,

ancestors, ancestors<-, descendants, and descendants<-, are implemented for objects of these classes.

For lists of vertices, edges, blocks, block edges, factors, and factor edges the methods Names, Names<-, Colors, Colors<-, Labels, Labels<-, LabelPositions, LabelPositions<-, Positions, Positions<-, Strata, Strata<-, Indices,

NodeAncestors, NodeAncestors<-, NodeDescendants, and NodeDescendants<- are available.

The model object of the call of dynamicGraphMain should have the methods modifyModel, testEdge, graphEdges, and setGraphEdges. When the graph is modified, by adding or dropping vertices or edge, the method modifyModel is called on the argument object of dynamicGraphMain. If an object is returned in the list of the returned value from modifyModel then object in dynamicGraphMain is replaced by this object, and the object is also assigned in the top level environment, if objectName was given to dynamicGraphMain.

The method testEdge of object should return an object with the methods label and width for labeling edges, see dg.Test-class.

The methods graphEdges and setGraphEdges are used to communicate the graph components between several views of the same model. The method graphEdges of the model object is for returning an object of class dg.graphedges-class to draw in the view, depending on the viewType. The method setGraphEdges of the model object is called on the model object when the model is modified.

Author(s)

Jens Henrik Badsberg

References

CoCo, with a guide at http://www.jstatsoft.org/v06/i04/, has an interface to dynamicGraph.

See Also

See also DynamicGraph and all the other functions of this package.

An example has been divided on the following 4 manual pages: dg.Model-class and dg.Test-class gives an example of a model object with test object. The pages of dg.graphedges-class show how the user can add menu items with actions that redraws the graph after modification of edges. Finally, validVertexClasses show how to create a new vertex class with a new symbol for drawing the vertex and an item added to the pop up menu of the new vertex class. The demo demo(Circle.newClass) of dynamicGraph

will do this example collected from these 4 pages.

Examples

```
require(tcltk)
require(dynamicGraph)

V.Names <- paste(c("Sex", "Age", "Eye", "FEV", "Hair", "Shosize"),
                 1:6, sep ="/")

V.Types <- c("Discrete", "Ordinal", "Discrete",
             "Continuous", "Discrete", "Continuous")

Vertices <- returnVertexList(V.Names, types = V.Types, color = "red")

From <- c(1, 2, 3, 4, 5, 6)
To   <- c(2, 3, 4, 5, 6, 1)

EdgeList <- vector("list", length(To))
for (j in seq(along = To)) EdgeList[[j]] <- c(From[j], To[j])
Edges <- returnEdgeList(EdgeList, Vertices, color = "black")

# Z <- dynamicGraphMain(Vertices, edgeList = Edges, control = dg.control(w = 4))

graph <- new("dg.graph", vertexList = Vertices, edgeList = Edges)
W <- dg(graph, control = dg.control(w = 4))
```

DynamicGraphModel-class

Class DynamicGraphModel

Description

The class for the models of the dynamicGraph.

Objects from the Class

AnObject of this class is created for each model in dynamicGraphMain.

Objects of class DynamicGraphModel-class will be a part of the returned value of the function dynamicGraphMain, and will store the model, with the views of the model, each of class DynamicGraphView-class.

Slots

id.env: Object of class "character": Internal identification of the object.

label: Object of class "character": The value of the argument title of dynamicGraphMain, combined with the slot name of the model object, printed by Str.

index: Object of class "numeric": The index of the model.

model: Object of class "list": The model of the graph windows.

graphs: Object of class "list": A list of view objects, each of class DynamicGraphView-class.

Methods

label<- signature(x = "DynamicGraphModel"): ...

label signature(object = "DynamicGraphModel"): ...

index signature(object = "DynamicGraphModel"): ...

model<- signature(x = "DynamicGraphModel"): ...

model signature(object = "DynamicGraphModel"): ...

graphs<- signature(x = "DynamicGraphModel"): ...

graphs signature(object = "DynamicGraphModel"): ...

Str signature(object = "DynamicGraphModel"): Compactly display the internal *str*ucture of a dynamicGraph model object.

show signature(object = "DynamicGraph"): calls the method Str.

control<- signature(x = "DynamicGraphModel"): See DynamicGraph-class.

Author(s)

Jens Henrik Badsberg

See Also

dynamicGraphMain, DynamicGraph-class, and DynamicGraphView-class.

DynamicGraphView-class
                          Class DynamicGraphView

Description

The class for views of the models of the dynamicGraph.

Objects from the Class

An object of the class DynamicGraphView-class is created for each view of an model in dynamicGraphMain.

Objects of this class will be a part of each model in the list of models of the returned value from dynamicGraphMain, and will store the view specific items of a model.

Methods are avaliable for returning references to the Tcl/tk window.

Slots

id.env: Object of class "character": Internal identification of object.

id: Object of class "numeric": (internal) Integer, increased for each redraw of the graph window.

title: Object of class "character": The value of the argument title of dynamicGraphMain, combined with the viewType of the model object, printed by Str.

index: Object of class "numeric": The index of the view.

dg: Object of class "dg.graphedges" ~~

Methods

label<- signature(x = "DynamicGraphView"): ...

label signature(object = "DynamicGraphView"): ...

index signature(object = "DynamicGraphView"): ...

dg<- signature(x = "DynamicGraphView"): (This could be a handle for, e.g., adding edges.)

dg signature(object = "DynamicGraphView"): ...

Str signature(object = "DynamicGraphView"): Compactly display the internal *str*ucture of a dynamicGraph view object.

show signature(object = "DynamicGraph"): calls the method Str.

canvas signature(object = "DynamicGraphView"): ...

tags signature(object = "DynamicGraphView"): ...

top signature(object = "DynamicGraphView"): ...

vbox signature(object = "DynamicGraphView"): ...

viewLabel signature(object = "DynamicGraphView"): ...

control<- signature(x = "DynamicGraphView"): See DynamicGraph-class.

Author(s)

Jens Henrik Badsberg

See Also

[dynamicGraphMain](), [DynamicGraph-class](), and [DynamicGraphModel-class]().

---

modalDialog	*Modal dialog window for returning a text string*

Description

Ask for a text string in a pop up window.

Usage

```
modalDialog(title, question, entryInit, top = NULL, entryWidth = 20,
            returnValOnCancel = "ID_CANCEL", do.grab = FALSE)
```

Arguments

title	Text string for the title bar of the appering window.
question	Text string for the question.
entryInit	Default value of answer.
top	Text string for the TclTk top.
entryWidth	Integer for the entryWidth.
returnValOnCancel	
	Text string for the returned value on Cancel.
do.grab	Logical. tkgrab.set resulted in fail for some systems.

Value

The text string entered, or returnValOnCancel.

Author(s)

From the examples compiled by James Wettenhall.

References

<http://bioinf.wehi.edu.au/~wettenhall/RTclTkExamples/modalDialog.html>

Examples

```
Menus <-
 list(MainUser =
      list(label = "Test of user drag down menu - Position of \"vertices\"",
           command = function(object, ...)
             print(Positions(list(...)$Arguments$vertexList))),
      MainUser =
      list(label = "Test of user drag down menu - modalDialog",
           command = function(object, ...) {
             Args <- list(...)$Arguments
             ReturnVal <- modalDialog("Test modalDialog Entry", "Enter name",
                                      Args$control$title,
                                      graphWindow = Args$graphWindow)
             print(ReturnVal)
             if (ReturnVal == "ID_CANCEL")
               return() } )
      )
```

---

nameToVertexIndex   *The indices of vertices*

Description

For each name, find in a list of vertices the index of the vertex with that name.

Usage

```
nameToVertexIndex(vertexnames, vertices)
```

Arguments

vertexnames	Vector of text strings of the vertexnames of the vertices for which the indices should be found.
vertices	A list of vertices, of each of class containing the class dg.Node.

Value

Integer vector with the indices of the vertices.

Author(s)

Jens Henrik Badsberg

Examples

```
Names <- c("Sex", "Age", "Eye", "FEV", "Hair", "Shosize")
Types <- rep("Discrete", 6)
vertices <- returnVertexList(Names, types = Types)
nameToVertexIndex(c("Sex", "Eye"), vertices)
```

---

replaceBlockList *Replace the block list of model frame*

Description

Replace the block list of model frame.

Usage

```
replaceBlockList(blockList, frameModels = NULL,
                 frameViews = frameModels@models[[modelIndex]],
                 modelIndex = 1,
                 graphWindow = frameViews@graphs[[viewIndex]],
                 viewIndex = 1, ...)
```

Arguments

blockList	Object of class "dg.BlockList": blockList is the new block list.
frameModels	See replaceVertexList.
frameViews	See replaceVertexList.
modelIndex	See replaceVertexList.
graphWindow	See replaceVertexList.
viewIndex	See replaceVertexList.
...	See replaceVertexList.

Author(s)

Jens Henrik Badsberg

---

replaceControls *Replace the controls of model frame*

Description

Replace the controls of model frame.

Usage

```
replaceControls(control, frameModels = NULL,
                frameViews = frameModels@models[[modelIndex]],
                modelIndex = 1,
                graphWindow = frameViews@graphs[[viewIndex]],
                viewIndex = 1, ...)
```

Arguments

control	Structure as returned from dg.controlc.
frameModels	See replaceVertexList.
frameViews	See replaceVertexList.
modelIndex	See replaceVertexList.
graphWindow	See replaceVertexList.
viewIndex	See replaceVertexList.
...	See replaceVertexList.

Author(s)

Jens Henrik Badsberg

---

replaceVertexList   *Replace the vertex list of model frame*

Description

Replace the vertex list of model frame

Usage

```
replaceVertexList(vertexList, frameModels = NULL,
                  frameViews = frameModels@models[[modelIndex]],
                  modelIndex = 1,
                  graphWindow = frameViews@graphs[[viewIndex]],
                  viewIndex = 1, ...)
```

Arguments

vertexList	Object of class "dg.VertexList": vertexList is the new vertex list.
frameModels	The vertex list is replaced in frameModels, and frameModels is redrawn in graphWindow.
frameViews	The frameViews to redraw.
modelIndex	The modelIndex of the model to redraw.
graphWindow	The graphWindow to redraw.
viewIndex	The viewIndex of the view to redraw.
...	Optional arguments.

Note

The models is redrawn as stored in frameModels. If you by interaction with the graph window has changed the model of graphWindow, please export the frameModels from the graph window by selection "Assign 'frameModels' .." from the "Export" menu of the graph window before replacing.

Author(s)

Jens Henrik Badsberg

---

returnBlockEdgeList   *Class dg.BlockEdgeList: The block edge list*

Description

Return a list of block edges, each of class dg.BlockEdge.

Objects can be created by calls of the form new("dg.BlockEdgeList", ...).

Usage

```
returnBlockEdgeList(edge.list, vertices, blocks,
                    visibleBlocks = 1:length(blocks), width = 2,
                    color = "default", N = 3, oriented = NA, type = NULL)
```

Arguments

edge.list	A list of vectors identifying the edges (between vertices). Each vector of edge.list should be a vector of integers giving the indices of the vertices of an edge, or a vector of text strings with the names of the vertices.
vertices	The list of vertices, each of a class containing dg.Vertex.
blocks	The list of blocks, each of a class dg.Block.
visibleBlocks	A numeric vector with the indices of the visibleBlocks. The argument is for view where some blocks are not visible in the view, but the vertices of the blocks are drawn.
width	A numeric with the initial width of block edges.
color	"default", or list with one or two text strings for colors giving the initial color of block edges. The two colors are used for respectively edges between blocks and for edges between blocks and vertices.
N	Integer, N is the number of coordinates of the vertices.
oriented	Logical, if TRUE then the edges are oriented.
type	A text string giving the type of block edges.

Value

A list of block edges, each of class dg.BlockEdge.

Slots

.Data: Object of class "list".

Extends

Class ″dg.EdgeList″, directly. Class ″dg.list″, directly. Class ″list″, from data part. Class ″dg.NodeList″, by class ″dg.EdgeList″.

Class ″vector″, by class ″dg.EdgeList″. Class ″vector″, by class ″dg.list″. Class ″vector″, by class ″list″.

Methods

initialize signature(.Object = ″dg.BlockEdgeList″): …

ancestorsBlockList signature(blockList = ″dg.BlockList″): …

descendantsBlockList signature(blockList = ″dg.BlockList″): …

checkBlockList signature(blockList = ″dg.BlockList″): …

Note

The methods of the edge list, `returnEdgeList`, also applies for block edge lists.

Author(s)

Jens Henrik Badsberg

Examples

```
Block.tree <- list(label = "W", Vertices = c("country"),
                   X = list(Vertices = c("sex", "race"),
                            A = list(Vertices = c("hair", "eye"),
                                     horizontal = FALSE),
                            B = list(Vertices = c("age"),
                                     C = list(Vertices = c("education")))))
Names <- unlist(Block.tree)
Names <- Names[grep("Vertices", names(Names))]
Types <- rep("Discrete", length(Names))
vertices <- returnVertexList(Names, types = Types)
blocktree <- setTreeBlocks(Block.tree, vertices)
blocks <- blockTreeToList(blocktree$BlockTree)
from <- c("country", "country", "race", "race",        "sex", "sex")
to   <- c(    "sex",    "race", "hair",  "eye", "education", "age")
from <- match(from, Names)
to <- match(to, Names)
edge.list <- vector("list", length(to))
for (j in seq(along = to)) edge.list[[j]] <- c(from[j], to[j])
edges <- returnEdgeList(edge.list, vertices, color = "red", oriented = TRUE)
vertices <- blocktree$Vertices
blockedges <- returnBlockEdgeList(edge.list, vertices, blocks,
                                  color = "red", oriented = TRUE)
blockedges <- new("dg.BlockEdgeList", edge.list = edge.list,
                                      vertices = vertices, blocks = blocks,
                                      color = "red", oriented = TRUE)

Names(blockedges)
```

```
Colors(blockedges)
Labels(blockedges)
LabelPositions(blockedges)
# Positions(blockedges)
# Strata(blockedges)
# Indices(blockedges)
str(NodeTypes(blockedges))
str(NodeIndices(blockedges))
Widths(blockedges)
Oriented(blockedges)
Widths(blockedges) <- rep(1,  7)
Widths(blockedges) <- rep(1, 14)
Widths(blockedges)
asDataFrame(blockedges)
```

---

returnEdgeList   *Class dg.VertexEdgeList: The edge list*

Description

Return a list of edges, each of class containing dg.VertexEdge.

Objects can be created by calls of the form new("dg.VertexEdgeList", ...).

Usage

```
returnEdgeList(edge.list, vertices, width = 2, color = "DarkSlateGrey", N = 3,
               oriented = NA, types = NULL, edgeClasses = validEdgeClasses())
```

Arguments

edge.list	A list of vectors identifying the edges. Each vector of edge.list should be a vector of integers giving the indices of the vertices of an edge, or a vector of text strings with the names of the vertices.
vertices	The list of vertices, each of a class containing dg.Vertex. vertices are used to set the initial labels of the edges.
width	A single numeric with the initial width of the edges.
color	A single text string giving the color of the edges.
oriented	Logical, if TRUE then the edges are oriented.
types	A vector of text strings giving the types of the edges, identify which classes the edges should be of, containing the dg.VertexEdge.
N	Integer, N is the number of coordinates of the vertices.
edgeClasses	Returned value from [validEdgeClasses](#), or extension of this matrix.

Value

A list of edges, each of class containing dg.VertexEdge.

Slots

.Data: Object of class "list".

Extends

Class "dg.EdgeList", directly. Class "dg.list", directly. Class "list", from data part. Class "dg.NodeList", by class "dg.EdgeList".

Class "vector", by class "dg.EdgeList". Class "vector", by class "dg.list". Class "vector", by class "list".

Methods

initialize signature(.Object = "dg.VertexEdgeList"): ...

Note

Beside the methods of the vertex list, vertexList, (except Positions, Indices and Strata) the edge list also has the methods NodeTypes,

NodeIndices,

Widths, Widths<-, Dashes, Dashes<-, Oriented, and Oriented<-.

Author(s)

Jens Henrik Badsberg

See Also

vertexList and dg.VertexEdge-class.

Examples

```
from <- c("contry", "contry", "race", "race",        "sex",  "sex")
to   <- c(   "sex",    "race", "hair",  "eye", "education",  "age")
vertexnames <- unique(sort(c(from, to)))
vertices <- returnVertexList(vertexnames)
# from <- match(from, vertexnames)
# to <- match(to, vertexnames)
edge.list <- vector("list", length(to))
for (j in seq(along = to)) edge.list[[j]] <- c(from[j], to[j])

edges <- returnEdgeList(edge.list, vertices, color = "red", oriented = TRUE)

edges <- new("dg.VertexEdgeList", edge.list = edge.list,
             vertices = vertices, color = "red", oriented = TRUE)

Names(edges)
Colors(edges)
Labels(edges)
LabelPositions(edges)
# Positions(edges)
```

```
# Strata(edges)
# Indices(edges)
str(NodeTypes(edges))
str(NodeIndices(edges))
Dashes(edges)
Widths(edges)
Oriented(edges)
Widths(edges) <- rep(1, 7)
Widths(edges) <- rep(1, 6)
Widths(edges)
asDataFrame(edges)
```

---

returnExtraEdgeList   *Class dg.ExtraEdgeList: The extra edge list*

Description

Return a list of extra edges, each of class dg.ExtraEdge.

Objects can be created by calls of the form new("dg.ExtraEdgeList", ...).

Usage

```
returnExtraEdgeList(edge.list, vertices, extravertices = NULL,
                    width = 2, color = "DarkSlateGrey", N = 3, type = NULL)
```

Arguments

edge.list	A list of vectors identifying the edges (between vertices). Each vector of edge.list should be a vector of integers giving the indices of the vertices of an edge, positive for vertices, negative for extra vertices, or a vector of text strings with the names of the vertices.
vertices	The list of vertices, each of a class containing dg.Vertex.
extravertices	The list of extravertices, each of a class containing dg.ExtraVertex.
width	A numeric with the initial width of the extra edges.
color	A text string giving the initial color of the extra edges.
N	Integer, N is the number of coordinates of the vertices.
type	A text string giving the type of the extra edges.

Value

A list of edges, each of class dg.ExtraEdge.

Slots

.Data: Object of class "list".

Extends

Class "dg.EdgeList", directly. Class "dg.list", directly. Class "list", from data part. Class "dg.NodeList", by class "dg.EdgeList". Class "vector", by class "dg.EdgeList".

Class "vector", by class "dg.list". Class "vector", by class "list".

Methods

initialize signature(.Object = "dg.ExtraEdgeList"): ...

Note

The methods of the edge list, returnEdgeList, also applies for extra edge lists.

The application is similar to returnBlockEdgeList, see example here.

Author(s)

Jens Henrik Badsberg

See Also

returnFactorEdgeList and returnBlockEdgeList.

---

returnFactorEdgeList   *Class dg.FactorEdgeList: The factor edge list*

Description

Return a list of factor edges, each of class dg.FactorEdge.

Objects can be created by calls of the form new("dg.FactorEdgeList", ...).

Usage

```
returnFactorEdgeList(edge.list, vertices, factorvertices = NULL,
                     width = 2, color = "DarkSlateGrey", N = 3,

                     type = NULL)
```

Arguments

edge.list	A list of vectors identifying the edges (between vertices). Each vector of edge.list should be a vector of integers giving the indices of the vertices of an edge, or a vector of text strings with the names of the vertices.
vertices	The list of vertices, each of a class containing dg.Vertex.
factorvertices	The list of factorvertices, each of a class containing dg.FactorVertex.
width	A numeric with the initial width of the factor edges.

color	A text string giving the initial color of the factor edges.
N	Integer, N is the number of coordinates of the vertices.
type	A text string giving the type of the factor edges.

Value

A list of edges, each of class dg.FactorEdge.

Slots

.Data: Object of class "list".

Extends

Class "dg.EdgeList", directly. Class "dg.list", directly. Class "list", from data part. Class "dg.NodeList", by class "dg.EdgeList".

Class "vector", by class "dg.EdgeList". Class "vector", by class "dg.list". Class "vector", by class "list".

Methods

initialize signature(.Object = "dg.FactorEdgeList"): ...

Note

The methods of the edge list, returnEdgeList, also applies for factor edge lists.

No example is given here since the factor edge list usually will be returned by the function returnFactorVerticesAndEdges. The application is similar to returnBlockEdgeList, see example here.

Author(s)

Jens Henrik Badsberg

See Also

returnFactorVerticesAndEdges and returnBlockEdgeList.

---

returnFactorVerticesAndEdges

Class dg.FactorVertexList: The factor vertex list

Description

Create factor vertex and factor edge lists.

Objects can be created by calls of the form new("dg.FactorVertexList", ...).

Usage

```
returnFactorVerticesAndEdges(Vertices, factors = NULL, types = "Generator",
                             factorVertexColor = "default",
                             factorEdgeColor = "DarkOliveGreen",

                             fixedFactorPositions = FALSE,
                             factorClasses = validFactorClasses())
```

Arguments

Vertices   The list of `Vertices`, each containing the class dg.`Vertex`.

factors   The list of vectors identifying the `factors`. Each item in the list is a vector of
                  the indices of vertices of a factor.

types   The `types` of the factors. Either a single `type` or a list of the same length as
                  `factors`. Each item of types should match the labels of [factorClasses](#), and
                  is used to set the class of the factor vertex.

factorVertexColor
                  The `factorVertexColor` of the factor vertices.

factorEdgeColor
                  The `factorEdgeColor` of the factor edges.

fixedFactorPositions
                  Logical. If `fixedFactorPositions` is setto TRUE then the factor vertices will
                  not follow the moved vertices.

factorClasses   The valid [factorClasses](#).

Details

The argument `factors` is a list of vectors identifying the `factors`, or generators. Each item in the
list is a vector with of the indices (or names) of the vertices of a factor, or variables of a generator.
A factor vertex is made for each factor, and factor edges from this factor vertex to the vertices of
the factor or added to the factor edge list. Also the edges between pairs of the vertices in the factors
are returned.

Value

A list with components

FactorVertices   The list of factor vertices, each of class containing [dg.FactorVertex](#).

FactorEdges   The list of factor edge, each of class containing [dg.FactorEdge](#).

PairEdges   A matrix with the edges of the graph, two columns with the indices of the ver-
                 tices of two ends of the edges.

Slots

`.Data`: Object of class `"list"`.

Extends

Class ″dg.NodeList″, directly. Class ″dg.list″, directly. Class ″list″, from data part. Class ″vector″, by class ″dg.NodeList″.

Class ″vector″, by class ″dg.list″. Class ″vector″, by class ″list″.

Methods

No methods defined with class "dg.FactorVertexList" in the signature.

Note

The methods of the vertex list, returnVertexList, also applies for factor lists, and the methods of the edge list, returnEdgeList, also applies for factor edge lists.

Your modifyModel should compute the new factors, generators, when modifying the model. See dg.Model-class for examples on use of returnFactorVerticesAndEdges.

Author(s)

Jens Henrik Badsberg

---

returnVertexList   *Class dg.VertexList: The vertex list*

Description

Return a list of vertices of classes containing the class dg.Vertex.

Objects can be created by calls of the form new(″dg.VertexList″, ...).

Usage

```
returnVertexList(names, labels = NULL, types = NULL,
                 strata = NULL, line = FALSE, N = 3,
                 colors = ifelse(types == "TextVertex",
                                 "FloralWhite", "DarkRed"),
                 vertexClasses = validVertexClasses())
```

Arguments

names	Vector of text strings with the names of the vertices.
labels	Vector of text strings with the labels of the vertices.
types	Vector of text strings with the types of the vertices.
strata	Vector of integers with the strata of the vertices.
line	Logical, if TRUE then the vertices are positioned on a line, else in a regular polygone, in a circle.
N	Integer, N is the number of coordinates of the vertices.
colors	Vector of text strings with colors of the vertices.
vertexClasses	The valid vertexClasses.

Value

A list of vertices of classes containing the class dg.Vertex.

Slots

.Data: Object of class "list".

Extends

Class "dg.NodeList", directly. Class "dg.list", directly. Class "list", from data part.

Class "vector", by class "dg.NodeList". Class "vector", by class "dg.list". Class "vector", by class "list".

Methods

initialize signature(.Object = "dg.VertexList"): ...

Note

The methods Names, Names<-, Colors, Colors<-, Labels, Labels<-, LabelPositions, LabelPositions<-, Positions, Positions<-, Strata, Strata<-, Indices, and asDataFrame are available for vertex lists.

Author(s)

Jens Henrik Badsberg

Examples

```
vertices <- returnVertexList(c("A", "B", "C", "D"),
                              labels = c("OrdinalVertex", "TextVertex",
                                "ContinuousVertex", "DiscreteVertex"),
                              types = c("Ordinal", "TextVertex",
                                          "Continuous", "Discrete"), N = 2)

vertices <- new("dg.VertexList", names = c("A", "B", "C", "D"),
                              labels = c("OrdinalVertex", "TextVertex",
                                "ContinuousVertex", "DiscreteVertex"),
                              types = c("Ordinal", "TextVertex",
                                          "Continuous", "Discrete"), N = 2)
Names(vertices)
Colors(vertices)
Labels(vertices)
LabelPositions(vertices)
Positions(vertices)
Strata(vertices)
Indices(vertices)
Names(vertices) <- c("a", "b", "c", "d")
Colors(vertices) <- rep("Blue", 4)
Labels(vertices) <- c("A", "B", "C", "D")
LabelPositions(vertices) <- matrix(rep(0, 12), ncol = 3)
```

```
Positions(vertices) <- matrix(rep(0, 12), ncol = 3)
Strata(vertices) <- rep(1, 4)
Names(vertices)
Colors(vertices)
Labels(vertices)
LabelPositions(vertices)
Positions(vertices)
Strata(vertices)
Indices(vertices)
asDataFrame(vertices)
```

---

selectDialog   *Dialog window for selection between items*

Description

Dialog window for selection between items.

Usage

```
selectDialog(title = "Selection entry", question = "Select item",
             itemNames, top = NULL, returnValOnCancel = "ID_CANCEL",
             do.grab = FALSE)
```

Arguments

title	Text string for the title bar of the appering window.
question	Text string for the question.
itemNames	Default value of answer.
top	Text string for the TclTk top.
returnValOnCancel	
	Text string for the returned value on Cancel.
do.grab	Logical. tkgrab.set resulted in fail for some systems.

Value

An integer, or returnValOnCancel.

Author(s)

From the examples compiled by James Wettenhall.

References

<http://bioinf.wehi.edu.au/~wettenhall/RTclTkExamples/modalDialog.html>

See Also

[modalDialog](#).

setBlocks   *Class dg.BlockList: The block list*

Description

Create a block list with positioning the vertices in the blocks.

Objects can be created by calls of the form new("dg.BlockList", ...).

Usage

```
setBlocks(block.list, vertices, labels = NULL,
          right.to.left = FALSE, nested.blocks = FALSE,

          blockColors = NULL, color = "Grey", N = 3)
```

Arguments

block.list	The list of vectors identifying the blocks. Each item in the list is a vector with of the indices (or text strings for names) of vertices of a block.
vertices	The list of vertices, each containing the class dg.Vertex. Returned with positions set in the interval of the blocks.
labels	List of text strings with the labels of the blocks. If labels is set to NULL then labels are found as names(block.list).
right.to.left	Logical. If right.to.left is set to TRUE then the explanatory blocks are drawn to the right.
nested.blocks	Logical. If nested.blocks then the blocks are drawn nested.
blockColors	Vector of text string with the blockColors of the blocks.
color	Single text string with color of the blocks. Only used when blockColors is not given.
N	Integer, N is the number of coordinates of the vertices and block corners.

Value

A list with components

Blocks	The list of blocks, each of class [dg.Block]().
Vertices	The list of vertices, with the positions of the vertices updated such the vertices has positions within the blocks.

Slots

.Data: Object of class "list".

Extends

Class "dg.NodeList", directly. Class "dg.list", directly. Class "list", from data part.

Class "vector", by class "dg.NodeList". Class "vector", by class "dg.list". Class "vector", by class "list".

Methods

checkBlockList signature(blockList = "dg.BlockList"): ...

ancestorsBlockList signature(blockList = "dg.BlockList"): ...

descendantsBlockList signature(blockList = "dg.BlockList"): ...

Author(s)

Jens Henrik Badsberg

Examples

```
require(tcltk)

require(dynamicGraph)

V.Types <- c("Discrete", "Ordinal", "Discrete",
             "Continuous", "Discrete", "Continuous")

V.Names <- c("Sex", "Age", "Eye", "FEV", "Hair", "Shosize")
V.Names <- paste(V.Names, 1:6, sep ="/")

From <- c(1, 2, 3, 4, 5, 6)
To   <- c(2, 3, 4, 5, 6, 1)

# A block recursive model:

Blocks <- list(Basic = c(2, 1), Intermediate = c(5, 4, 3),  Now = c(6))

V.Names <- paste(V.Names, c(1, 1, 2, 2, 2, 3), sep =":")

graph <- new("dg.simple.graph", vertex.names = V.Names, types = V.Types,
             from = From, to = To, blocks = Blocks)

W <- dg(graph,
        control = dg.control(width = 600, height = 600, drawblocks = TRUE,
                             drawBlockBackground = FALSE, title = "DrawBlocks",
                             namesOnEdges = FALSE))

# A block recursiv model, without drawing blocks:

W <- dg(simpleGraphToGraph(graph, control = dg.control(drawblocks = FALSE)),
        control = dg.control(width = 600, height = 600, title = "No blocks drawn"))

# A block recursive model with nested blocks:
```

```
W <- dg(simpleGraphToGraph(graph,
          control = dg.control(nested.blocks = TRUE,
                               blockColors =
                               paste("Grey", 100 - 2 * (1:10), sep = ""))),
        control = dg.control(width = 600, height = 600, title = "Nested blocks"))

# The block list of the last example:

vertices <- returnVertexList(V.Names, types = V.Types)
blockList <- setBlocks(Blocks, vertices = vertices,  nested.blocks = TRUE,
                       blockColors = paste("Grey", 100 - 2 * (1:10), sep = ""))

names(blockList)
str(blockList$Blocks[[1]])

names(blockList$Blocks)
Names(blockList$Blocks)
Labels(blockList$Blocks)
LabelPositions(blockList$Blocks)
Positions(blockList$Blocks)
Strata(blockList$Blocks)
Colors(blockList$Blocks)
NodeAncestors(blockList$Blocks)
NodeDescendants(blockList$Blocks)
Visible(blockList$Blocks)
Indices(blockList$Blocks)

names(blockList$Vertices)
Names(blockList$Vertices)
Labels(blockList$Vertices)
LabelPositions(blockList$Vertices)
Positions(blockList$Vertices)
Strata(blockList$Vertices)
Colors(blockList$Vertices)
Indices(blockList$Vertices)

asDataFrame(blockList$Vertices)
asDataFrame(blockList$Blocks)
```

---

setTreeBlocks   *The block tree*

Description

Create a block tree with positioning the vertices in to blocks.

Usage

```
setTreeBlocks(block.tree, vertices, root.label = "", N = 3,
              delta = ifelse(overlaying, 1, 0),
              Delta = ifelse(overlaying, 0, 1.5),
              d = 5, f = 1/4, blockColors = NULL, overlaying = FALSE)
```

Arguments

block.tree	A structure with the blocks in a `block.tree`. See below.
vertices	The list of `vertices`, each containing the class `dg.Vertex`. Returned with positions set in the interval of the blocks.
root.label	A text string with the `root.label` of the root block.
N	Integer, `N` is the number of coordinates of the vertices and block corners.
delta	Numeric. Decrement of block size for nested blocks, and space between blocks when `overlaying` is TRUE. The decrement is `delta` divided by 100, times the the size of the window canvas, width or height.
Delta	Numeric. Decrement of block size for nested blocks, and space between blocks when `overlaying` is FALSE. The decrement is `Delta` divided by 100, times the the size of the window canvas, width or height.
d	Numeric. If not d is given in `block.tree`, see below: The heading bar (with the label) has a height of $(d + 2)$ divided by 100, times height of the window canvas.
f	Numeric. If not f or g is given in `block.tree`, see below: The the vertices of the block are placed in an array with a height (width if `horizontal` is set to FALSE) of f divided by 100, times height (width) of the block. Thus this size is relative to the block size.
blockColors	Vector of text string with the `blockColors` of the blocks.
overlaying	Logical. If `overlaying` is set to FALSE then children blocks of a block are not drawn inside the block.

Details

A recursive definition: Block.tree is a list with the vertices of the "current" blocks, some parameters for controlling the layout, and possible some `block.trees`:

- ...$Vertices The `vertices` of the block.
- ...$label A text string for the label of the block. Will overwrite "block-name" and `root.label`.
- ...$d Numeric. The heading bar (with the label) has a height of $(d + 2)$ divided by 100, times the height of the window canvas.
- ...$g Numeric. The vertices of the block are placed in an array with a height (width if `horizontal` is set to FALSE) of g divided by 100, times the height (width) of the window canvas. Thus this size will not decrease with the block size.
- ...$f Numeric. If not g is given: The the vertices of the block are placed in an array with a height (width if `horizontal` is set to FALSE) of f divided by 100, times the height (width) of the block. Thus this size is relative to the block size.

- ...$G Numeric. (If the height of the block is 100 we are now left with `100 - 2 * delta - d - 2 - g` for the blocks.) The sub blocks (apart from `common.children`) then have a of height (width, if `horizontal` is set to FALSE) of G divided by 100, times the height (width) of the window canvas. Thus the sub block size will not decrease with the block size. (If the height of the block is 100 we are now left with `100 - 2 * delta - d - 2 - g - G` for the `common.children`.)

- ...$F Numeric. If not G is given: The proportion G of the remaining space are used for sub blocks (apart from `common.children`) and the proportion G of the space for blocks are used for `common.children`.

- ...$horizontal Logical. If `horizontal` is set to TRUE, then the sub blocks, but `common.children`, are placed side by side, else the blocks are placed vertically.

- ...$closed Logical. If `closed` is set to TRUE, then the block is initially drawn "closed", and the vertices and sub blocks of the block are not visible.

- ...$vertices.last Logical. If `vertices.last` then the vertices of the block are placed after the sub blocks.

- ...$"block-name"= list(...) Repeated zero, one or more times for sub blocks. `"block-name"` is the label of the block, and `list(...)` is a `Block.tree`.

- ...$common.children= list(...) Omitted, or a list with common children of the other sub blocks of the block. The list is again a `Block.tree`.

Value

A list with components

BlockTree	A tree of blocks, each of class [dg.Block]().
Vertices	The list of vertices, with the positions updated such the vertices has positions within the blocks.

Note

Ancestors and descendants are set in [setTreeBlocks](). Ancestors are used in the function [returnBlockEdgeList]() to find the edges between blocks and between blocks and vertices.

Descendants are used in [dynamicGraphMain]() when closing, opening and moving blocks, and when adding or dropping edges from and to blocks.

The methods [NodeAncestors](), [NodeAncestors<-]() [NodeDescendants]() and [NodeDescendants<-]() can be used on the block list resulting of [blockTreeToList]() on the block tree.

Author(s)

Jens Henrik Badsberg

Examples

```
# Example 1:

Block.tree <- list(label = "W", Vertices = c("country"),
                   X = list(Vertices = c("race", "sex"),
```

```
                              A = list(Vertices = c("hair", "eye"),
                                       horizontal = FALSE),
                              B = list(Vertices = c("education"),
                                       C = list(Vertices = c("age")))))
V.Names <- unlist(Block.tree)
vertices <- returnVertexList(V.Names[grep("Vertices", names(V.Names))])
blocktree <- setTreeBlocks(Block.tree, vertices)

Positions(blockTreeToList(blocktree$BlockTree))
Positions(blocktree$Vertices)
NodeAncestors(blockTreeToList(blocktree$BlockTree))
NodeDescendants(blockTreeToList(blocktree$BlockTree))

vertexStrata <- Strata(blocktree$Vertices)
vertexStrata
vertexNames <- Names(blocktree$Vertices)
names(vertexNames) <- NULL
vertexNames

# Indices of the vertices in blocks:

indicesInBlock <- vector("list", max(vertexStrata))
for (i in seq(along = vertexStrata))
  indicesInBlock[[vertexStrata[i]]] <-
    append(indicesInBlock[[vertexStrata[i]]], i)
str(indicesInBlock)

# Names of the vertices in blocks:

vertexNamesInblock <- vector("list", max(vertexStrata))
for (i in seq(along = vertexStrata))
  vertexNamesInblock[[vertexStrata[i]]] <-
    append(vertexNamesInblock[[vertexStrata[i]]], vertexNames[i])
str(vertexNamesInblock)

# A useful function, replace "k" (block index k)
# in block "i" by "x[k]", the content "x[k]" of block "k":

f <- function(A, x) {
  result <- vector("list", length(A))
  names(result) <- names(A)
  for (i in seq(along = A))
    if ((length(A[[i]]) > 0) && (A[[i]] != 0))
      for (k in A[[i]])
        result[[i]] <- append(result[[i]], x[k])
  return(result)
}

# For each block, names of vertices in ancestor blocks:

vertexAncOfBlock <- f(NodeAncestors(blockTreeToList(blocktree$BlockTree)),
                      vertexNamesInblock)
str(vertexAncOfBlock)
```

```
for (i in seq(along = vertexAncOfBlock))
  if (length(vertexAncOfBlock[[i]]) > 0)
    vertexAncOfBlock[[i]] <- unlist(vertexAncOfBlock[[i]])
str(vertexAncOfBlock)

# For each block, names of vertices in descendant blocks:

vertexDesOfBlock <- f(NodeDescendants(blockTreeToList(blocktree$BlockTree)),
                      vertexNamesInblock)
str(vertexDesOfBlock)

for (i in seq(along = vertexDesOfBlock))
  if (length(vertexDesOfBlock[[i]]) > 0)
    vertexDesOfBlock[[i]] <- unlist(vertexDesOfBlock[[i]])
str(vertexDesOfBlock)


# Example 2:

Block.tree <-
  list(g = 0, G = 54, label = "Pedegree.G",
       Male.Side =
       list(g = 0, G = 33,
            Father =
            list(g = 0, G = 12,
                 P.G.Father = list(Vertices = c("P.G.Father.1")),
                 P.G.Mother = list(Vertices = c("P.G.Mother.1")),
                 common.children = list(g = 0, label = "Father.1",
                                          Vertices = c("Father.1"))),
            Mother =
            list(g = 0, G = 12,
                 M.G.Father = list(Vertices = c("M.G.Father.1")),
                 M.G.Mother = list(Vertices = c("M.G.Mother.1")),
                 common.children = list(g = 0, label = "Mother.1",
                                          Vertices = c("Mother.1"))),
            common.children = list(g = 2, Vertices = c("Male"))),
  Female.Side = list(g = 0, G = 12,
    P.G.Father = list(Vertices = c("P.G.Father.2")),
    P.G.Mother = list(Vertices = c("P.G.Mother.2")),
    M.G.Father = list(Vertices = c("M.G.Father.2")),
    M.G.Mother = list(Vertices = c("M.G.Mother.2")),
    common.children = list(g = 0, G = 12, label = "Female",
      Father = list(Vertices = c("Father.2")),
      Mother = list(Vertices = c("Mother.2")),
      common.children = list(g = 2, Vertices = c("Female")))),
  common.children = list(Vertices = c("Marriage"), g = 3, label = "Children",
    Son = list(Vertices = c("Son"), g = 3,
      P.G.Son = list(Vertices = c("P.G.Son"), g = 2),
      P.G.Dat = list(Vertices = c("P.G.Dat"), g = 1)),
    Dat = list(Vertices = c("Dat"), g = 2,
      M.G.Son = list(Vertices = c("M.G.Son")),
      M.G.Dat = list(Vertices = c("M.G.Dat")))
```

```
     )
  )

  v <- unlist(Block.tree)
  V.Names <- v[grep("Vertices", names(v))]
  rm(v)

  FromTo <- matrix(c("P.G.Father.1", "Father.1", "P.G.Father.2", "Father.2",
                     "P.G.Mother.1", "Father.1", "P.G.Mother.2", "Father.2",
                     "M.G.Father.1", "Mother.1", "M.G.Father.2", "Mother.2",
                     "M.G.Mother.1", "Mother.1", "M.G.Mother.2", "Mother.2",
                     "Father.1",      "Male",      "Father.2",     "Female",
                     "Mother.1",      "Male",      "Mother.2",     "Female",
                     "Male",          "Marriage", "Female",        "Marriage",
                     "Marriage",      "Son",       "Marriage",     "Dat",
                     "Son",           "P.G.Son",   "Dat",          "M.G.Son",
                     "Son",           "P.G.Dat",  "Dat",           "M.G.Dat"),
                   byrow = TRUE, ncol = 2)

  From <- match(FromTo[,1], V.Names)
  To   <- match(FromTo[,2], V.Names)

  V.Types <- rep("Discrete", length(V.Names))

  Object <- NULL

  graph <- new("dg.simple.graph", vertex.names = V.Names, types = V.Types,
               from = From, to = To, block.tree = Block.tree)

  W <- dg(graph,
          control = dg.control(width = 600, height = 600,
                               drawblocks = TRUE, drawBlockFrame = TRUE,
                               overlaying = TRUE, title = "Pedegree.G"))
```

---

simpleGraphToGraph   *simple graph to graph*

Description

Simple graph to graph

Usage

```
simpleGraphToGraph(sdg = NULL, frameModels = NULL, dg = NULL,
                   control = dg.control(...), ...)
```

Arguments

sdg   Object of class `dg.simple.graph-class`.

frameModels	Object of class [DynamicGraph-class](#). If given then vertices (and blocks) are extracted from here.
dg	Object of class [dg.graph-class](#). If given then vertices, edges and blocks are extracted from here.
control	control is a list. See: [dg.control](#)
...	Optional arguments. The list Arguments could hold objects from dg.

Author(s)

Jens Henrik Badsberg

Examples

```
require(tcltk); require(dynamicGraph)

control <- dg.control(width = 150, height = 200, margin = 200)

# modelObject <- newDefaultModelObject("AnModelObject")

modelObject <- new("dg.Model", name = "AnModelObject")

V.Types <- c("Discrete", "Ordinal", "Discrete",
             "Continuous", "Discrete", "Continuous")

V.Names <- c("Sex", "Age", "Eye", "FEV", "Hair", "Shosize")
V.Labels <- paste(V.Names, 1:6, sep ="/")

From <- c(1, 2, 3)
To   <- c(2, 3, 1)

simpleGraphZ.1 <- new("dg.simple.graph", vertex.names = V.Names[1:3],
                      labels = V.Labels[1:3], types = V.Types[1:3],
                      from = From, to = To)
graphZ.1 <- simpleGraphToGraph(simpleGraphZ.1)

Z.0 <- dg(simpleGraphZ.1, modelObject = modelObject,
          control = control, title = "Z")

Z.00 <- wDG(sdg = simpleGraphZ.1, object = modelObject,
            control = control, title = "Z")

Z <- dg(graphZ.1, modelObject = modelObject, control = control, title = "Z")

From <- c(2, 3)
To   <- c(3, 1)

simpleGraphW.1 <- new("dg.simple.graph", from = From, to = To)
graphW.1 <- simpleGraphToGraph(simpleGraphW.1,
                               vertexList = graphZ.1@vertexList,
                               blockList = graphZ.1@blockList)
```

```
W <- addModel(graphW.1, frameModels = Z)


simpleGraphV.1 <- new("dg.simple.graph", from = From, to = To)
graphV.1 <- simpleGraphToGraph(simpleGraphV.1,
                               Vertices = graphZ.1@vertexList,
                               BlockList = graphZ.1@blockList)

V <- addView(graphV.1, frameModels = Z, modelIndex = 1, viewType = "Factor")


From <- 1
To   <- 2

simpleGraphU.1 <- new("dg.simple.graph", from = From, to = To)
graphU.1 <- simpleGraphToGraph(simpleGraphU.1)

graphU.1 <- simpleGraphToGraph(simpleGraphU.1,
                               Vertices = graphZ.1@vertexList,
                               BlockList = graphZ.1@blockList)

graphU.1 <- simpleGraphToGraph(simpleGraphU.1,
                               vertexList = graphZ.1@vertexList,
                               blockList = graphZ.1@blockList)

graphU.1 <- simpleGraphToGraph(from = From, to = To,
                               vertexList = graphZ.1@vertexList,
                               blockList = graphZ.1@blockList)

U <- replaceModel(graphU.1,
                  frameModels = Z, modelIndex = 1, graphIndex = 1,
                  title = "U")
```

---

validEdgeClasses   *Valid edge classes*

Description

Return matrix with labels of valid edge classes and the valid edge classes

Usage

```
validEdgeClasses()
```

Details

The argument edgeClasses to the functions dynamicGraphMain and DynamicGraph, and to dg.VertexEdge-class and returnEdgeList is by default the returned value of this function. If new edge classes are created then edgeClasses should be set to a value with this returned value extended appropriate.

Value

Matrix of text strings with labels (used in dialog windows) of valid edge classes and the valid edge classes (used to create the edges).

Note

The "draw" method for an edge should return a list with the items "lines", "tags", "from", "to", label" and "label.position". "lines" is the "tk"-objects for line objects between pairs of vertices, with coordinates at the vertices. "tags" is the "tk" -objects for objects between pairs of vertices, with coordinates at the middle of the two vertices.

Author(s)

Jens Henrik Badsberg

See Also

validVertexClasses.

Examples

```
require(tcltk)

# Test with new edge class (demo(Circle.newEdge)):

setClass("NewEdge", contains = c("dg.VertexEdge", "dg.Edge", "dg.Node"))

myEdgeClasses <- rbind(validEdgeClasses(),
                       NewEdge = c("NewEdge", "NewEdge"))

setMethod("draw", "NewEdge",
          function(object, canvas, position,
                   x = lapply(position, function(e) e[1]),
                   y = lapply(position, function(e) e[2]),
                   stratum = as.vector(rep(0, length(position)),
                                       mode = "list"),
                   w = 2, color = "green", background = "white",
                   font.edge.label = "8x16")
          {
            f <- function(i, j) {
              dash <- "."
              arrowhead <- "both"
              l <- function(xi, yi, xj, yj)
                tkcreate(canvas, "line", xi, yi, xj, yj, width = w,
                         arrow = arrowhead, dash = dash,
                         # arrowshape = as.list(c(2, 5, 3) * w),
                         fill = color(object), activefill = "DarkSlateGray")
              lines <- list(l(x[[i]], y[[i]], x[[j]], y[[j]]))
              label.position <- (position[[i]] + position[[j]]) / 2
              pos <- label.position + rep(0, length(label.position))
              label <- tkcreate(canvas, "text", pos[1], pos[2],
```

```
                                              text = object@label, anchor = "nw",
        font = "8x16", activefill = "DarkSlateGray")
                    tags <- NULL
                    x. <- mean(unlist(x))
                    y. <- mean(unlist(y))
                    s <- 4 * w * sqrt(4 / pi)
                    p <- tkcreate(canvas, "rectangle",
            x. - s, y. - s, x. + s, y. + s,
                                  fill = color(object), activefill = "SeaGreen")
                    tags <- list(p)
                    return(list(lines = lines, tags = tags,
                                from = object@vertex.indices[i],
                                to = object@vertex.indices[j],
                                label = label, label.position = label.position))
                  }
                  result <- NULL
                  edge <- object@vertex.indices
                  m <- length(edge)
                  for (j in seq(along = edge))
                    if (j < length(edge))
                      for (k in (j+1):length(edge))
                          result <- append(result, list(f(j, k)))
                  return(result)
      })

setMethod("addToPopups", "NewEdge",
          function(object, type, nodePopupMenu, i,
      updateArguments, Args, ...)
            {
                tkadd(nodePopupMenu, "command",
                    label = paste(" --- This is a my new vertex!"),
                    command = function() { print(name(object))})
            })




V.Types <- c("Discrete", "Ordinal", "Discrete",
             "Continuous", "Discrete", "Continuous")
V.Names <- c("Sex", "Age", "Eye", "FEV", "Hair", "Shosize")
V.Labels <- paste(V.Names, 1:6, sep ="/")

From <- c(1, 2, 3, 4, 5, 6, 3)
To   <- c(2, 3, 4, 5, 6, 1, 6)

control <- dg.control(updateEdgeLabels = FALSE,
                      edgeColor = "green", vertexColor = "blue",
                      edgeClasses = myEdgeClasses)

simpleGraph.Z.nE <- new("dg.simple.graph", vertex.names = V.Names,
                        types = V.Types, labels = V.Labels,
                        from = From, to = To,
                        edge.types = c("NewEdge",
                                       "VertexEdge",
```

```
                                  "Dashed",
                                  "Dotted",
                                  "DoubleArrow",
                                  "DoubleConnected",
                                  "TripleConnected"),
                   texts = c("Gryf", "gaf"))

graph.Z.nE <- simpleGraphToGraph(simpleGraph.Z.nE, control = control)

Object <- NULL

Z.nE <- dg(graph.Z.nE, modelObject = Object, control = control, title = "Z")
```

---

validFactorClasses   *Valid factor vertex classes*

Description

Return matrix with labels of valid factor vertex classes and the valid factor vertex classes.

Usage

```
validFactorClasses()
```

Details

The argument factorClasses to [DynamicGraph](#), and to [returnFactorVerticesAndEdges](#) and [newFactor](#) is by default the returned value of this function. If new factor vertex classes are created then factorClasses should be set to a value with this returned value extended appropriate.

Value

Matrix of text strings with labels (used in dialog windows) of valid factor vertex classes and the valid factor vertex classes (used to create the factor vertices).

Author(s)

Jens Henrik Badsberg

See Also

[validVertexClasses](#)

Examples

```
validFactorClasses()
```

---

validVertexClasses   *Valid vertex classes*

Description

Return matrix with labels of valid vertex classes and the valid vertex classes

Usage

```
validVertexClasses()
```

Details

The argument vertexClasses to the functions [dynamicGraphMain](#) and [DynamicGraph](#), and to [dg.Vertex-class](#) and [returnVertexList](#) is by default the returned value of this function. If new vertex classes are created then vertexClasses should be set to a value with this returned value extended appropriate.

Value

Matrix of text strings with labels (used in dialog windows) of valid vertex classes and the valid vertex classes (used to create the vertices).

Author(s)

Jens Henrik Badsberg

See Also

[validEdgeClasses](#).

Examples

```
require(tcltk)

# Test with new vertex class (demo(Circle.newVertex)):

setClass("NewVertex", contains = c("dg.Node", "dg.Vertex"),
         representation(my.text   = "character",
                        my.number = "numeric"),
         prototype(my.text    = "",
                   my.number  = 2))

myVertexClasses <- rbind(validVertexClasses(),
                         NewVertex = c("NewVertex", "NewVertex"))

setMethod("draw", "NewVertex",
          function(object, canvas, position,
                   x = position[1], y = position[2], stratum = 0,
```

```
                          w = 2, color = "green", background = "white")
            {
              s <- w * sqrt(4 / pi) / 2
             p1 <- tkcreate(canvas, "oval",
                              x - s - s, y - s,
                              x + s - s, y + s,
                              fill = color(object), activefill = "IndianRed")
              p2 <- tkcreate(canvas, "oval",
                              x - s + s, y - s,
                              x + s + s, y + s,
                              fill = color(object), activefill = "IndianRed")
              p3 <- tkcreate(canvas, "oval",
                              x - s, y - s - s,
                              x + s, y + s - s,
                              fill = color(object), activefill = "IndianRed")
              p4 <- tkcreate(canvas, "poly",
                              x - 1.5 * s, y + 3 * s,
                              x + 1.5 * s, y + 3 * s,
                              x, y,
                              fill = color(object), activefill = "SteelBlue")
               return(list(dynamic = list(p1, p2, p3, p4), fixed = NULL)) })

setMethod("addToPopups", "NewVertex",
          function(object, type, nodePopupMenu, i,
   updateArguments, Args, ...)
          {
               tkadd(nodePopupMenu, "command",
                   label = paste(" --- This is a my new vertex!"),
                   command = function() { print(name(object))})
          })

if (!isGeneric("my.text")) {
  if (is.function("my.text"))
    fun <- my.text
  else
    fun <- function(object) standardGeneric("my.text")
  setGeneric("my.text", fun)
}
setGeneric("my.text<-",
          function(x, value) standardGeneric("my.text<-"))

setMethod("my.text", "NewVertex",
          function(object) object@my.text)
setReplaceMethod("my.text", "NewVertex",
                 function(x, value) {x@my.text <- value; x} )

if (!isGeneric("my.number")) {
  if (is.function("my.number"))
    fun <- my.number
  else
    fun <- function(object) standardGeneric("my.number")
  setGeneric("my.number", fun)
}
```

```
setGeneric("my.number<-",
           function(x, value) standardGeneric("my.number<-"))

setMethod("my.number", "NewVertex",
          function(object) object@my.number)
setReplaceMethod("my.number", "NewVertex",
                 function(x, value) {x@my.number <- value; x} )


V.Types <- c(rep("NewVertex", 3), "Discrete", "Ordinal", "Continuous")
V.Names <- c("Sex", "Age", "Eye", "FEV", "Hair", "Shosize")
V.Labels <- paste(V.Names, 1:6, sep ="/")

From <- c(1, 2, 3, 4, 5, 6)
To   <- c(2, 3, 4, 5, 6, 1)

control <- dg.control(updateEdgeLabels = FALSE,
                      edgeColor = "green", vertexColor = "blue",
                      vertexClasses = myVertexClasses)

simpleGraph.Z.nV <- new("dg.simple.graph", vertex.names = V.Names,
                        types = V.Types, labels = V.Labels,
                        from = From, to = To, texts = c("Gryf", "gaf"))

graph.Z.nV <- simpleGraphToGraph(simpleGraph.Z.nV, control = control)

Object <- NULL

Z.nV <- dg(graph.Z.nV, modelObject = Object, control = control, title = "Z")
```

---

validViewClasses   *Valid view classes*

Description

Return matrix with labels of valid view classes and the valid view classes

Usage

```
validViewClasses()
```

Details

The argument viewClasses to [dynamicGraphMain](#) and [DynamicGraph](#) is by default the returned value of this function. If new view classes are created then viewClasses should be set to a value with this returned value extended appropriate.

Value

Matrix of text strings with labels (used in dialog windows) of valid view classes and the valid view classes (used to create the views).

Author(s)

Jens Henrik Badsberg

See Also

`DynamicGraphView-class`.

Examples

```
validViewClasses()
```

---

wDG   *DEPRECATED: Interface to dynamicGraph*

Description

A wrapper to `dynamicGraphMain` for adding models and views, represented by a simple dynamic graph, to an existing dynamicGraph.

(The function was a part of the deprecated interface function `DynamicGraph`: Use the method `dg` on object of `dg.simple.graph-class` in stead.)

The wrapper is depricated: use the methods `dg`, `addModel`, `addView`, `replaceModel`, or `replaceView`.

Usage

```
wDG(sdg = NULL, object = NULL,
    frameModels = NULL, frameViews = NULL, graphWindow = NULL,
    dg = NULL, addModel = FALSE, addView = FALSE, overwrite = FALSE,
    returnNewMaster = FALSE, redraw = FALSE, control = dg.control(...), ...)
```

Arguments

sdg	Object of class `dg.simple.graph-class`
object	The model object, or NULL, see `dg.Model-class`.
frameModels	An object of class `DynamicGraph-class`. frameModels is the object for a dataset (defining vertics and blocks) and the models on that dataset.
frameViews	An object of class `DynamicGraphModel-class`. frameViews is the object for a model and the views of that model.
graphWindow	An object of class `DynamicGraphView-class`. graphWindow is the object for a view of a model.

dg	An optional object of class `dg.graphedges-class`. If this agrument is given then edges and factors are extracted from the argument. Similar with an optional argument `Arguments`.
addModel	Logical, if `addModel` then a model is added to the argument `frameModels`, and a view of the model is drawn. If the argument `overwrite` is TRUE and the argument `graphWindow` is given then the model of `graphWindow` is replaced by the model argument `object`. If the argument `overwrite` is TRUE and the argument `frameViews` is given then the model of `frame\-Views` is replaced by the model argument `object`.
addView	Logical, if `addView` then a view of type set by the argument `viewType` for the model of the argument `frameViews` is added.
overwrite	Logical, see the argument `addModel`. The argument `returnLink` must be set to TRUE to overwrite a view.
redraw	Logical. If TRUE then the dynamicGraph of the arguments `frameModels` is 'redrawn'. New instances of the windows are made.
returnNewMaster	
	Logical. Alternative implementation of `addModel`, using the code of `redraw`. As `redraw`, but the windows of `frameModels` exists, and a new model is added.
control	Options for `DynamicGraph` and `dynamicGraphMain`, see `dg.control`.
...	Additional arguments to `dynamicGraphMain`.

Value

The returned value from `dynamicGraphMain`.

Author(s)

Jens Henrik Badsberg

Examples

```
require(tcltk); require(dynamicGraph)
```

Index