Package: rviewgraph (via r-universe)

October 4, 2026

Type Package

Title Animated Graph Layout Viewer

Version 1.4.2

Date 2023-04-25

Author Alun Thomas

Maintainer Alun Thomas <Alun.Thomas@utah.edu>

Description Provides 'Java' graphical user interfaces for viewing, manipulating and plotting graphs. Graphs may be directed or undirected.

Suggests knitr, rmarkdown

Enhances igraph, Matrix

Depends rJava

SystemRequirements Java (>= 8)

License GPL-2

LazyLoad yes

Collate 'onLoad.R' 'rviewgraph-package.R' 'rViewGraph.R' 'vg.R'

Encoding UTF-8

RoxygenNote 7.1.2

VignetteBuilder knitr

NeedsCompilation no

Config/pak/sysreqs default-jdk

Repository https://cran.r-universe.dev

Date/Publication 2023-05-10 16:50:02 UTC

RemoteUrl https://github.com/cran/rviewgraph

RemoteRef HEAD

RemoteSha 67fa4edc7f2f546681dfc5484191ce1601b23d0e

Contents

rViewGraph 2
vg 7

Index **14**

`rViewGraph` *This is a function to create and start a 'Java' graph animation GUI.*

Description

Creates and starts an animated graphical user interface (GUI) for positioning the vertices of a graph in 2 dimensions.

Usage

```
rViewGraph(object, names, cols, shapes, layout, directed, running, ...)

## Default S3 method:
rViewGraph(
  object,
  names = NULL,
  cols = "yellow",
  shapes = 0,
  layout = NULL,
  directed = FALSE,
  running = TRUE,
  ...
)

## S3 method for class 'igraph'
rViewGraph(
  object,
  names = igraph::V(object)$name,
  cols = "yellow",
  shapes = 0,
  layout = igraph::layout.random(object),
  directed = igraph::is.directed(object),
  running = TRUE,
  ...
)
```

Arguments

`object`	the object specifying the graph. This can be specified in various ways:

- A square n = dim(object)[1] by n real valued incidence matrix. This will create a graph with n vertices indexed by 1:n and edges between vertices with indices i and j if object[i,j] != 0. If the graph is directed edges are directed from i to j if the entry is positive, and from j to i if the entry is negative.
- An m = dim(object)[1] by 2 matrix of positive integers specifying the indexes of the vertices at the ends of m edges. This will create a graph with n = max(object) vertices indexed by 1:max(object) and edges connecting the vertex indexed by object[i,1] to the vertex indexed by object[i,2] for each i in 1:m. If the graph is directed, the edges are directed from object[i,1] to object[i,2]. NOTE: A 2 by 2 matrix will be interpreted as an incidence matrix, not an array of edges.
- A vector of 2*m positive integers specifying the indexes of the vertices at the ends of m = length(object)/2 edges. This is the way in which igraph specifies edges. If x is such a vector, calling rViewGraph{x} is equivalent to calling rViewGraph(matrix(x,ncol=2,byrow=F)).
- An igraph graph object.

names — the names of the vertices. This is an object that can be interpreted as a vector of strings that will be used to label the vertices. If the length is less than the number of vertices, the names will be cycled. The default is names = 1:n, where n is the number of vertices. If unlabeled vertices are required use, for example, names=" ". The size of the string is used to determine the size of the vertex so, for instance, names = " A " will produce bigger vertices than names = "A".

cols — the colours of the vertices. This is on object that can be interpreted as a vector of colours specified in the usual R ways. If the length is less than the number of vertices, the colours will be cycled. The default is cols = "yellow".

shapes — the shapes of the vertices. This is a vector of integers specifying the shapes of the vertices. The available shapes are:

- 0 = rectangle
- 1 = oval
- 2 = diamond
- any other values are taken as 0.

The default is shapes = 0.

layout — the starting positions of the vertices. This is an n by 2 array of reals with layout[i,] specifying the horizontal and vertical coordinates for the starting point of the ith vertex. By default this is set to NULL in which case random starting points are used.

directed — indicates whether or not the graph is directed.

running — indicates whether or not to start with the animation running.

... — passed along extra arguments.

Details

Creates and starts a 'Java' GUI showing a real time animation of a Newton-Raphson optimization of a force function specified between the vertices of an arbitrary graph. There are attractive forces

between adjacent vertices and repulsive forces between all vertices. The repulsions go smoothly to zero in a finite distance between vertices so that, unlike some other methods, different components don't send each other off to infinity.

The program is controlled by a slide bar, some buttons, the arrow, home and shift keys, but mostly by mouse operations. All three mouse buttons are used. The interactive mouse, key and slide bar operations are described below.

Value

`rViewGraph` is intended only for interactive use. When used in a non-interactive environment it immediately exits returning the value `NULL`. Otherwise, all versions of `rViewGraph` return a list of functions that control the actions of the interactive viewer.

`run()` Starts the GUI running if it's not already doing so.

`stop()` Stops the GUI running if it's running.

`hide()` Stops the GUI and hides it.

`show()` Shows the GUI. If it was running when `hide` was called, it starts running again.

`getLayout()` Returns the coordinates of the vertices as currently shown in the GUI. These are given as an `n` by 2 array as required for the `layout` parameter of `rViewGraph` itself.

`setLayout(layout = NULL)`
Sets the coordinates of the vertices to the given values. `layout` is specified in the same way as required for the `layout` parameter of `rViewGraph` itself. The default has `layout` set to `NULL`, and new random coordinates are generated.

`hidePaper()` By default the GUI indicates, with a different colour, the portion of the plane that corresponds to the current choice of paper for printing. This function removes that area.

`showPaper(size = "letter", landscape = TRUE)`
Indicates, with a different colour, the portion of the plane corresponding to a choice of paper for printing. `size` can be any of `letter`, `A4`, `A3`, `A2`, `A1`, `A0`, `C1`, or `C0`. `landscape` can be either `TRUE` or `FALSE`, in which case portrait orientation is used. The default is to show the portion of the plane that would be printed on US letter in landscape orientation.

`hideAxes()` By default, axes are shown to indicate the origin. This function hides them.

`showAxes()` Shows the axes if they are hidden.

`writePostScript()`
This starts a `Java` PostScript print job dialog box that can be used send the current view of the graph to a printer or to write a PostScript file. The plot produced should closely match what is indicated by `showPaper`.

`ps()` Alias for `writePostScript`.

Interactive mouse, key and slide bar controls

- Slide bars at the bottom of the GUI control the repulsive force in the energy equation used to set the coordinates. If the graph is undirected, there is a single 'Repulsion' parameter, if directed, there are 'X-Repulsion' and 'Y-Repulsion' parameters, and a 'Gravity' parameter that influences how these are combined.

- Mouse operations without shift key and without control key pressed.
  1. Left mouse: Drags a vertex. Vertex is free on release.
  2. Middle mouse: Drags a vertex. Vertex is fixed at release position.
  3. Right mouse: Translates the view by the amount dragged. A bit like putting your finger on a piece of paper and moving it.
  4. Double click with any mouse button in the background: Resets the vertices to new random positions.
- Mouse operations with shift key but without control key pressed.
  1. Left mouse: Drags a vertex and the component it is in. Vertex and component free on release.
  2. Middle mouse: Drags a vertex and the component it is in. Vertex and component are fixed at release positions.
  3. Right mouse: Translates the positions of the vertices relative to the position of the canvas by the amount dragged. This is useful to center the picture on the canvas ready for outputting.
- Mouse operations without shift key but with control key pressed.
  1. Left mouse: Click on a vertex to un-hide any hidden neighbours.
  2. Middle mouse: Click on a vertex to hide it.
  3. Double click left mouse: Un-hides all hidden vertices.
  4. Double click middle mouse: Hides all vertices.
- Mouse operations with shift key and with control key pressed.
  1. Left mouse: Click on a vertex to un-hide all vertices in the same component.
  2. Middle mouse: Click on a vertex to hide it and the component it is in.
- Key functions without shift key pressed. Mouse has to be in the picture canvas.
  1. Up arrow: Increases the scale of viewing by 10%.
  2. Down arrow: Decreases the scale of viewing by 10%.
  3. Left arrow: Rotates the view by 15 degrees clockwise.
  4. Right arrow: Rotates the view by 15 degrees anticlockwise.
  5. Home key: Undoes all scalings and rotations and places the origin at the top left corner of the canvas.
- Key functions with shift key pressed. Mouse has to be in the picture canvas.
  1. Up arrow: Increases the vertex positions by 10% relative to the scale of the canvas.
  2. Down arrow: Decreases the vertex positions by 10% relative to the scale of the canvas.
  3. Left arrow: Rotates the vertex positions by 15 degrees clockwise relative to the canvas orientation.
  4. Right arrow: Rotates the vertex positions by 15 degrees anticlockwise relative to the canvas orientation.

Author(s)

Alun Thomas

Source

A full description of the force function and algorithm used is given by C Cannings and A Thomas, Inference, simulation and enumeration of genealogies. In D J Balding, M Bishop, and C Cannings, editors, The Handbook of Statistical Genetics. Third Edition, pages 781-805. John Wiley & Sons, Ltd, 2007.

Examples

```
require(rviewgraph)

# First generate the random edges of an Erdos Renyi random graph.
f = sample(100,size=200,replace=TRUE)
t = sample(100,size=200,replace=TRUE)

# The following should all show the same graph:
# ... specified as a two column matrix.
v1 = rViewGraph(cbind(f,t))

# ... in 'igraph' preferred format.
v2 = rViewGraph(c(f,t))

# ... as an adjacency matrix.
x = matrix(0,ncol=max(f,t),nrow=max(f,t))
for (i in 1:length(f)) x[f[i],t[i]] = 1
v3 = rViewGraph(x)


# Specifying names, colours and shapes.

# Use unlabeled vertices, as red, green and blue diamonds.
v4 = rViewGraph(cbind(f,t), names = "  ", cols = c(2,3,4), shapes=2)

# Use yellow vertices with random shapes, labeled with capital letters.
y = matrix(sample(1:26,100,TRUE),ncol=2)
v5 = rViewGraph(y,names=LETTERS,cols="cyan",shapes=sample(0:2,26,TRUE))


# Controlling a currently active GUI.
if (!is.null(v5))
{
 # Shift the coordinates, although this is more
# easily done with mouse controls.
 v5$setLayout(100 + v5$getLayout())

# Reset the coordinates to random values.
v5$setLayout()

 # Pepare a plot for printing, fix it, and start a PostScript print job.
 v5$hideAxes()
 v5$showPaper("A3",F)
 v5$stop()
 v5$writePostScript()
```

```
}
```

---

vg *Animated graph viewer*

Description

vg creates and starts an animated graphical user interface for positioning the vertices of a graph in 2 dimensions.

Usage

```
vg(grob, directed, running)

## S3 method for class 'list'
vg(grob, directed = grob$directed, running = grob$running)

## S3 method for class '`NULL`'
vg(grob = NULL, directed = FALSE, running = TRUE)

## Default S3 method:
vg(grob = NULL, directed = FALSE, running = TRUE)
```

Arguments

grob	is a graphical object state saved as a list from a previous run of vg using the save() function. If grob is NULL or not a list, vg starts with an empty graph.
directed	indicates whether or not the graph is directed. By default directed = FALSE. If the graph is directed, the edges have arrows indicating direction, and there are three slide bars to control the repulsion parameters. For an undirected graph there is a single control.
running	indicates whether or not the viewer is started with the animation running. By default running = TRUE.

Details

Creates and starts a 'Java' graphical user interface (GUI) showing a real time animation of a Newton-Raphson optimization of a force function specified between the vertices of an arbitrary graph. There are attractive forces between adjacent vertices and repulsive forces between all vertices. The repulsions go smoothly to zero in a finite distance between vertices so that, unlike some other methods, different components don't send each other off to infinity.

The program is controlled by a slide bar, some buttons, the arrow, home and shift keys, but mostly by mouse operations. All three mouse buttons are used. These operations are described below.

vg will replace rViewGraph, although the latter is still currently available. vg allows for far more control of the graph than rViewGraph, including adding and removing vertices and edges, and

changing the appearance of the vertices, all of which can be done while the animation is running. It has a different set of force parameter controls that are useful for directed acyclic graphs (DAGs) specifically, but also, to a lesser extent, for arbitrary directed graphs. It also provides functions for saving and restoring a graphical state including vertices and edges, vertex positions, and vertex appearances.

vg is intended primarily for interactive use. When used in a non-interactive environment it will run without a visible GUI, however, a graph structure can be created and saved and viewed in future, interactive, R sessions.

Value

vg returns a list of functions that specify and query the graph, coordinates and appearance, and control the viewer.

`add(i)` If the graph does not contain vertices indexed by `i`, they are added to it. `i` may be any integer or integer vector.

`remove(i)` If the graph contains vertices indexed by `i`, they are disconnected from their neighbours and removed from the graph. `i` may be any integer or integer vector.

`connect(i, j)` If the graph does not have edges between vertices indexed by `i` and `j`, they are made. If `i` and `j` are of different lengths, only the first `min(length(i),length(j))` values are used. If the relevant vertices are are not already in the graph, they are first made and added. `i` and `j` may be any integers or integer vectors.

`disconnect(i, j)` If the graph has edges between vertices indexed by `i` and `j` they are removed. If `i` and `j` are of different lengths, only the first `min(length(i),length(j))` values are used. The vertices themselves are not removed, even if they have no other neighbours. `i` and `j` may be any integers or integer vectors.

`clear()` All vertices and edges are removed from the graph, however, any customizations made to the appearance of the vertices will persist until `clearMap()` is called.

`isDirected()` Returns `TRUE` if the graph's edges are directed, `FALSE` otherwise.

`getIds()` Returns a vector of all the vertex indices that the viewer has encountered, whether or not they are currently in the graph.

`contains(id)` Returns a vector of booleans indicating whether the vertices corresponding to the given indices are currently in the graph.

`connects(i, j)` Returns a vector of booleans indicating whether the vertices with the indices in vector `i` are connected to the corresponding vertices in `j`. If `i` and `j` are of different lengths, only first `min(length(i),length(j))` values are queried.

`neighbours(i)` Returns a list of vectors containing the neighbours of the vertices indexed by `i`. If the graph is directed, both in and out neighbours are included. If a vertex is unconnected, an empty vector is returned. If an index is for a vertex that is not currently in the graph, `NULL` is returned.

`neighbors(i)` Alias for `neighbours(i)`.

`inNeighbours(i)` Returns a list of vectors containing the indices of vertices with edges from them to the ones in `i`. If the graph is undirected, this is the same as `neighbours(i)`.

`inNeighbors(i)` Alias for `inNeighbours(i)`.

`outNeighbours(i)`
: Returns a list of vectors containing the indices of vertices with edges to them from the ones in `i`. If the graph is undirected, this is the same as `neighbours(i)`.

`outNeighbors(i)`
: Alias for `outNeighbours(i)`.

`getVertices()` Returns a vector of the indices of all vertices currently in the graph.

`getEdges()` Returns a matrix with 2 columns and a row for each edge in the current graph. Each row gives the indices of the vertices that the edge connects. If the edges are directed, the connections are oriented from vertices in the first column to those in the second.

`getX(id)` Returns the current horizontal coordinates of the vertices with indices in `id`. Returns 0 if an index has not previously been seen.

`getY(id)` Returns the current vertical coordinates of the vertices with indices in `id`. Returns 0 if an index has not previously been seen.

`setXY(id, x, y)` Sets the horizontal and vertical coordinates for the vertices with indices in `id`. If `x` or `y` are not as long as `id` their values are repeated cyclically to get vectors of the right length.

`label(i, lab=i)` Sets the strings shown on the vertices indexed by `i` to those specified by `lab`. If `lab` is not as long as `i`, its values are repeated cyclically to get a vector of the right length. An error will occur if `lab` can't be interpreted as a vector of strings.

`colour(i, col="yellow")`
: Sets the colours of the vertices indexed by `i` to those specified by `col`. If `col` is not as long as `i`, its values are repeated cyclically to get a vector of the right length. Colours can be specified in the usual `R` ways.

`color(i, col="yellow")`
: Alias for `colours(i,col)`.

`shape(i, shp=0)` Sets the shapes of the vertices indexed by `i` to those specified by `shp`. If `shp` is not as long as `i`, its values are repeated cyclically to get a vector or the right length. Shapes are specified as integers:

- 0 = rectangle
- 1 = oval
- 2 = diamond
- any other values are taken as 0.

`size(i, width=-1, height=width)`
: Sets the sizes of the vertices indexed by `i` to those specified by `width` and `height`. If, for a certain index, both `width` and `height` are non-negative, the size of the vertex is fixed. If either of `width` or `height` is negative, the size of the vertex is chosen adaptively to fit the current label. If `width` or `height` are not as long as `i`, their values are repeated cyclically to get vectors of the right length.

`map(i, lab=i, col="yellow", shp=0, width=-1, height=width`
: Combines `label()`, `colour()`, `shape()` and `size()` in a single index-to-appearance mapping function. These functions can be called before or after the vertices

have been added to the graph, and the representations will persist for vertices removed from, and replaced in, the graph until until they are explicitly changed or `clearMap()` is called.

`clearMap()` Resets all vertices to the default appearance of yellow rectangle labeled with the index.

`run()` Starts the GUI running if it's not already doing so.

`stop()` Stops the GUI running if it's running.

`isRunning()` Returns `TRUE` if the animation is running, `FALSE` otherwise.

`show()` Shows the GUI. If it was running when `hide()` was called, it starts running again.

`hide()` Stops the GUI and hides it.

`showPaper(size = "letter", landscape = TRUE)`
Indicates, with a different colour, the portion of the plane corresponding to a choice of paper for printing. `size` can be any of `letter`, `A4`, `A3`, `A2`, `A1`, `A0`, `C1`, or `C0`. `landscape` can be either `TRUE` or `FALSE`, in which case portrait orientation is used. The default is to show the portion of the plane that would be printed on US letter in landscape orientation.

`hidePaper()` By default the GUI indicates, with a different colour, the portion of the plane that corresponds to the current choice of paper for printing. This function removes that area.

`showAxes()` Shows the axes if they are hidden.

`hideAxes()` By default, axes are shown to indicate the origin. This function hides them.

`ps()` Starts a `Java` PostScript print job dialog box that can be used send the current view of the graph to a printer or to write a PostScript file. The plot produced should closely match what is indicated by `showPaper`.

`save()` Returns a `list` specifying the current state of the graph structure, vertex coordinates, and vertex appearance map.

`restore(grob)` Restores a saved graph and map state.

Interactive mouse, key and slide bar controls

- Slide bars at the bottom of the GUI control the repulsive force in the energy equation used to set the coordinates. If the graph is undirected, there is a single 'Repulsion' parameter, if directed, there are 'X-Repulsion' and 'Y-Repulsion' parameters, and a 'Gravity' parameter that influences how these are combined.
- Mouse operations without shift key and without control key pressed.
  1. Left mouse: Drags a vertex. Vertex is free on release.
  2. Middle mouse: Drags a vertex. Vertex is fixed at release position.
  3. Right mouse: Translates the view by the amount dragged. A bit like putting your finger on a piece of paper and moving it.
  4. Double click with any mouse button in the background: Resets the vertices to new random positions.
- Mouse operations with shift key but without control key pressed.

1. Left mouse: Drags a vertex and the component it is in. Vertex and component free on release.
2. Middle mouse: Drags a vertex and the component it is in. Vertex and component are fixed at release positions.
3. Right mouse: Translates the positions of the vertices relative to the position of the canvas by the amount dragged. This is useful to center the picture on the canvas ready for outputting.

- Mouse operations without shift key but with control key pressed.
  1. Left mouse: Click on a vertex to un-hide any hidden neighbours.
  2. Middle mouse: Click on a vertex to hide it.
  3. Double click left mouse: Un-hides all hidden vertices.
  4. Double click middle mouse: Hides all vertices.
- Mouse operations with shift key and with control key pressed.
  1. Left mouse: Click on a vertex to un-hide all vertices in the same component.
  2. Middle mouse: Click on a vertex to hide it and the component it is in.
- Key functions without shift key pressed. Mouse has to be in the picture canvas.
  1. Up arrow: Increases the scale of viewing by 10%.
  2. Down arrow: Decreases the scale of viewing by 10%.
  3. Left arrow: Rotates the view by 15 degrees clockwise.
  4. Right arrow: Rotates the view by 15 degrees anticlockwise.
  5. Home key: Undoes all scalings and rotations and places the origin at the top left corner of the canvas.
- Key functions with shift key pressed. Mouse has to be in the picture canvas.
  1. Up arrow: Increases the vertex positions by 10% relative to the scale of the canvas.
  2. Down arrow: Decreases the vertex positions by 10% relative to the scale of the canvas.
  3. Left arrow: Rotates the vertex positions by 15 degrees clockwise relative to the canvas orientation.
  4. Right arrow: Rotates the vertex positions by 15 degrees anticlockwise relative to the canvas orientation.

Author(s)

Alun Thomas

References

A full description of the force function and algorithm used is given by C Cannings and A Thomas, Inference, simulation and enumeration of genealogies. In D J Balding, M Bishop, and C Cannings, editors, The Handbook of Statistical Genetics. Third Edition, pages 781-805. John Wiley & Sons, Ltd, 2007.

See Also

rViewGraph

Examples

```
# Start the viewer.
# This will print a warning message if not run interactively.
require(rviewgraph)
v = vg()

# Generate some random vertices to connect with edges.
from = sample(1:20,15,TRUE)
to = sample(1:20,15,TRUE)

# Connect these edges in the viewer.
v$connect(from,to)

# Negative vertex indices are also allowed.
v$connect(-from,-to)

# Add some new vertices, unconnected to any others.
v$add(30:35)

# Remove some vertices.
v$rem((-5:5)*2)

# Query some of the structure of the graph.
v$contains(1)
v$contains(-1)
v$connects(1,-1)
v$contains(1:50)
v$connects(from,to)
v$connects(from,-to+2)
v$neighbours(from)

# Change what some of the vertices look like.
v$map(-10:10, lab = "", col=1, shp=1, width=1:15)
v$label(10:36, lab=LETTERS)
v$colour((-1000:-1),"cyan")
v$shape((-1000:-1)*2,2)

# Hide the axes.
v$hideAxes()

# Stop the animation.
# Not necessary for outputting but sometimes helpful.
v$stop()

# Change the paper size and orientation to A4 portrait.
v$showPaper("A4",FALSE)

# Start the print dialog box.
#v$ps()

# Restart the animation, and check that it's running.
v$run()
```

```
v$isRunning()

# Save the application.
s = v$save()

# Change the graph and appearance.
v$colour(-1000:1000,"red")
v$connect(rep(1,100),1:100)

# Then decide you didn't like the changes so restore
# the saved state.
v$restore(s)

# Can also restore a state in a new GUI.
v2 = vg(s)

# Get a vector of all the indices that the viewer has seen.
ids = v$getIds()

# Get a vector of the indices of the vertices currently
# in the graph.
verts = v$getVertices()

# Get a matrix with 2 columns specifying the current edges
# of the graph.
edges = v$getEdges()

# Get the current coordinates of the specified vertices.
x = v$getX(verts)
y = v$getY(verts)

# Change the current coordinates of the vertices.
v$setXY(verts,2*x,0.5*y+2)
```

Index

∗ **animation**
vg, 7
∗ **edges**
vg, 7
∗ **graph**
rViewGraph, 2
vg, 7
∗ **vertices**
vg, 7

rViewGraph, 2

vg, 7