

Package: geoGraph (via r-universe)

September 12, 2024

Type Package

Title Walking through the geographic space using graphs

Version 1.1.1

Maintainer Andrea Manica <am315@cam.ac.uk>

Description Classes and methods for spatial graphs interfaced with support for GIS shapefiles.

License GPL (>=2)

Language en-GB

URL <https://github.com/EvolEcolGroup/geograph>,
<https://evolecolgroup.github.io/geograph/>

BugReports <https://github.com/EvolEcolGroup/geograph/issues>

Encoding UTF-8

LazyData true

LazyDataCompression xz

VignetteBuilder knitr

Config/testthat/edition 3

Roxygen list(markdown = TRUE)

Depends R (>= 3.5.0), methods, graph

Imports fields, RBGL, sp

Suggests testthat, knitr, rmarkdown, sf

RoxygenNote 7.2.3

Collate 'classes.R' 'accessors.R' 'auxil.R' 'basicMethods.R'
'buffer.R' 'closestNode.R' 'combineCosts.R' 'connectivity.R'
'datasets.R' 'dijkstra.R' 'dropDead.R' 'extractFromLayer.R'
'findLand.R' 'geograph.R' 'globals.R' 'interact.R' 'isInArea.R'
'makeGrid.R' 'plot.R' 'rebuild.R' 'setCosts.R' 'setDistCosts.R'
'zoom.R' 'zzz.R'

Repository <https://evolecolgroup.r-universe.dev>

RemoteUrl <https://github.com/EvolEcolGroup/geograph>

RemoteRef main

RemoteSha 6cd7ee502102816ac0b60efed72e90cd7857619d

Contents

geoGraph-package	2
auxiliary	5
buffer	6
closestNode	8
combineCosts	10
connectivity	11
dijkstra-methods	13
dropDeadEdges	15
extractFromLayer	16
findLand	18
gData-class	19
geo.add.edges	20
geo.change.attr	22
getColors	23
getCosts	24
getEdges	26
getNodesAttr	27
gGraph-class	28
hgdp	30
isInArea	31
makeGrid	33
plot-gData	34
plot-gGraph	36
setCosts	38
setDistCosts	40
setEdges	41
worldgraph	42
zoom	43
Index	45

geoGraph-package *The geoGraph package*

Description

This package implements classes and methods for large-scale georeferenced data handled through spatial graphs.

Details

Main functionalities of geoGraph are summarized below.

=== DATA HANDLING ===

In geoGraph, data are stored as a particular formal class named `gGraph`. This class contains spatial coordinates of a set of nodes (`@coords`), attributes for these nodes (`@nodes.attr`), meta-information about nodes attributes (`@meta`), and a graph of connections between nodes of class `graphNEL` (`@graph`).

Several functions are available for handling `gGraph` data:

- some accessors allow to access slots of an object, sometimes with additional treatment of information: `getGraph`, `getNodesAttr`, `getCoords`, `getNodes`, `getEdges`, `getCosts`.
- `setEdges`: add/remove edges specified edges.
- `setCosts`: set costs of edges.
- `hasCosts`: tests if the graph is weighted (i.e., has non-uniform costs).
- `isInArea`: finds which nodes are in the currently plotted area.
- `areConnected`: tests if nodes are directly connected.
- `connectivityPlot`: plot connected components with different colors.
- `dropDeadEdges`: suppress edges whose weight is null.
- `closestNode`: given a longitude and a latitude, finds the closest node; specific values of node attribute can be provided, for instance, to find the closest node on land.
- `show`: printing of `gGraph` objects.
- `extractFromLayer`: extract information from GIS layers.
- `findLand`: checks which nodes are on land.
- `setCosts`: define edges weights according to rules specified in the `@meta` slot.
- `geo.add.edges`, `geo.remove.edges`: graphical functions for adding or removing edges.
- `geo.change.attr`: graphical functions for changing attributes of nodes.

==== GRAPHICS ====

geoGraph aims at providing advanced graphical facilities, such as zooming in or out particular area, moving the plotted area, or visualizing connectivity between nodes.

- `plot`: plot method with various options, allowing to display a shapefile (by default, the map of the world), using color according to attributes, showing connectivity between nodes, etc.
- `points`: similar to plot method, except that a new plot is not created.
- `plotEdges`: the specific function plotting edges. It detects if the object is a weighted graph, and plots edges accordingly.
- `geo.zoomin`, `geo.zoomout`: zoom in and out a plot.
- `geo.back`: replot the previous screens.
- `geo.slide`: slide the plotted area toward the indicated direction.
- `geo.bookmark`, `geo.goto`: set and goto a bookmarked area.

==== DATASETS ====

Datasets occupy a central place in geoGraph, since they provide the spatial models used in later operations.

Two main datasets are proposed, each being a `gGraph` resulting from the splitting of the earth into cells of (almost perfectly) equal sizes. Two different resolutions are provided:

- `worldgraph.10k`: coverage using about 10,000 nodes
- `worldgraph.40k`: coverage using about 40,000 nodes

Other datasets are:

- `worldshape`: shapefile containing world countries.

To cite geoGraph, please use the reference given by `citation("geoGraph")`.

Examples

```
## the class gGraph
worldgraph.10k

## plotting the object
plot(worldgraph.10k, reset = TRUE)

## zooming in
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))
title("Europe")

## to play interactively with graphics, use:
```

```
# geo.zoomin()
# geo.zoomout()
# geo.slide()
# geo.back()

## defining a new object restrained to visible nodes
x <- worldgraph.10k[isInArea(worldgraph.10k)]
plot(x, reset = TRUE, edges = TRUE)
title("x does just contain these visible nodes.")

## define weights for edges
x <- setCosts(x, attr.name = "habitat", method = "prod")
plot(x, edges = TRUE)
title("connectivity defined by habitat (land/land=1, other=0)")

## drop 'dead edges' (i.e. with weight 0)
x <- dropDeadEdges(x)
plot(x, edges = TRUE)
title("after dropping edges with null weight")
```

auxiliary

Auxiliary methods for geoGraph

Description

These methods are low-level functions called by other procedures of geoGraph. Some can, however, be useful in themselves. Note that unlike other functions in geoGraph, these functions do not generally test for the validity of the provided arguments (for speed purposes).

Usage

```
hasCosts(x)

geo.segments(
  x0,
  y0,
  x1,
  y1,
  col = graphics::par("fg"),
  lty = graphics::par("lty"),
  lwd = graphics::par("lwd"),
  ...
)
```

Arguments

x	a valid gGraph .
x0, y0	coordinates of points <i>from</i> which to draw.
x1, y1	coordinates of points <i>to</i> which to draw.
col	a character string or an integer indicating the color of the segments.
lty	a character string or an integer indicating the type of line.
lwd	an integer indicating the line width.
...	further graphical parameters (from 'par') passed to the segments function.

Details

- `hasCosts`: tests whether a [gGraph](#) has costs associated to its edges.
- `geo.segments`: a substitute to `segments` which correctly draws segments between locations distant by more than 90 degrees of longitude.
- `rebuild`: in development.

Value

For `hasCost`, a logical value is returned. `geo.segments` returns `NULL`.

Examples

```
hasCosts(worldgraph.10k)
```

buffer

Compute buffers around locations for gGraph and gData objects

Description

The generic function `buffer` finds buffers around specified locations of a [gGraph](#) or a [gData](#) object. Different format for the output are available.

Usage

```
## S4 method for signature 'gGraph'
buffer(x, nodes, d, res.type = c("nodes", "gGraph"), ...)

## S4 method for signature 'gData'
buffer(x, d, res.type = c("nodes", "gData", "gGraph"), ...)
```

Arguments

<code>x</code>	a valid gGraph or gData object.
<code>nodes</code>	a character vector identifying the nodes around which buffers should be computed.
<code>d</code>	the radius of the buffer, in km.
<code>res.type</code>	the type of result that should be returned (see section value).
<code>...</code>	further arguments passed to specific methods.

Details

The computed buffers are sets of nodes lying within a given distance of specified locations. All nodes of a buffer need to be connected to the location they surround.

Value

The output depends on the value of the argument `res.type`:

- `nodes`: a vector of characters identifying the nodes of the buffers.
- `gGraph`: a [gGraph](#) object with a new attribute "buffer" (TRUE: within buffers; FALSE: outside buffers), and new color rules for this attribute in `@meta$buf.colors`.
- `gData`: a [gData](#) object including all the nodes of the buffers.

Examples

```
#### gGraph example ####
## zoom in to an area
plot(worldgraph.10k, reset = TRUE)
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))

## identify one node
oneNodeXY <- c(getCoords(worldgraph.10k)[9299, 1], getCoords(worldgraph.10k)[9299, 2])
points(oneNodeXY[1], oneNodeXY[2], col = "red")

## find some buffers
buffer(worldgraph.10k, "9299", 100) # nothing around 100km
buffer(worldgraph.10k, "9299", 500)
buf500km <- buffer(worldgraph.10k, "9299", 500, res = "gGraph")
plot(buf500km, col.rules = buf500km@meta$buf.colors)
buf1000km <- buffer(worldgraph.10k, "9299", 1000, res = "gGraph")
plot(buf1000km, col.rules = buf1000km@meta$buf.colors)

#### gData example ####
x <- hgdp[27:30] # retain a subset of hgdp
plot(x, reset = TRUE, col.g = "lightgrey", pch.node = 20)
```

```

buf.200 <- buffer(x, 200, res = "gData")
buf.400 <- buffer(x, 400, res = "gData")
buf.600 <- buffer(x, 600, res = "gData")
buf.1000 <- buffer(x, 1000, res = "gData")
points(buf.1000, col.node = "black")
points(buf.600, col.node = "yellow")
points(buf.400, col.node = "gold")
points(buf.200, col.node = "orange")
title("Different buffers for a gData \n(100km, 200km, 500km)")

```

closestNode

Find the closest node to a given location

Description

The function `closestNode` searches for the closest node in a [gGraph](#) or a [gData](#) object to a given location. It is possible to restrain the research to given values of a node attribute. For instance, one can search the closest node on land to a given location.

Usage

```
closestNode(x, ...)
```

```
## S4 method for signature 'gGraph'
```

```
closestNode(x, loc, zoneSize = 5, attr.name = NULL, attr.values = NULL)
```

```
## S4 method for signature 'gData'
```

```
closestNode(x, zoneSize = 5, attr.name = NULL, attr.values = NULL)
```

Arguments

<code>x</code>	a valid gGraph or gData object. In the latter case, the gGraph to which the gData is linked has to be in the current environment.
<code>...</code>	further arguments passed to specific methods.
<code>loc</code>	locations, specified as a list with two components indicating longitude and latitude of locations. Alternatively, this can be a <code>data.frame</code> or a matrix with longitude and latitude in columns, in this order. Note that <code>locator()</code> can be used to specify interactively the locations.
<code>zoneSize</code>	a numeric value indicating the size of the zone (in latitude/longitude units) where the closest node is searched for. Note that this only matters for speed purpose: if no closest node is found inside a given zone, the zone is expanded until nodes are found.
<code>attr.name</code>	the optional name of a node attribute. See details.
<code>attr.values</code>	an optional vector giving values for <code>attr.names</code> . See details.

Details

This function is also used to match locations of a [gData](#) object with nodes of the [gGraph](#) object to which it is linked.

When creating a [gData](#) object, if the `gGraph.name` argument is provided, then locations are matched with the [gGraph](#) object automatically, by an internal call to `closestNode`. Note, however, that it is not possible to specify node attributes (`attr.names` and `attr.values`) this way.

Value

If `x` is a [gGraph](#) object: a vector of node names.

If `x` is a [gData](#) object: a [gData](#) object with matching nodes stored in the `@nodes.id` slot. Note that previous content of `@nodes.id` will be erased.

Functions

- `closestNode(gGraph)`: Method for [gGraph](#)
- `closestNode(gData)`: Method for [gData](#)

See Also

[geo.add.edges](#) and [geo.remove.edges](#) to interactively add or remove edges in a [gGraph](#) object.

Examples

```
## Not run:
## interactive example ##
plot(worldgraph.10k, reset = TRUE)

## zooming in
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))
title("Europe")

## click some locations
myNodes <- closestNode(worldgraph.10k, locator(), attr.name = "habitat", attr.value = "land")
myNodes

## here are the closestNodes
points(getCoords(worldgraph.10k)[myNodes, ], col = "red")

## End(Not run)

## example with a gData object ##
myLoc <- list(x = c(3, -8, 11, 28), y = c(50, 57, 71, 67)) # some locations
obj <- new("gData", coords = myLoc) # new gData object
obj

obj@gGraph.name <- "worldgraph.10k" # this could be done when creating obj
obj <- closestNode(obj, attr.name = "habitat", attr.value = "land")
```

```
## plot the result (original location -> assigned node)
plot(obj, method = "both", reset = TRUE)
title("'x'=location, 'o'=assigned node")
```

 combineCosts

Combine the costs of two gGraph objects

Description

The function `combineCosts` combines the edge costs of two `gGraph` objects. The first object is used as a template to generate the objects with the combined costs. Two `gGraph` objects must have the same edges.

Usage

```
combineCosts(x1, x2, method = c("sum", "product", "function"), FUN = NULL, ...)
```

Arguments

<code>x1</code>	The first <code>gGraph</code> (which will be used as a template to build the combined <code>gGraph</code>)
<code>x2</code>	The second <code>gGraph</code> from which costs will be combined
<code>method</code>	a character string indicating which method should be used to combined edge cost from the two <code>gGraph</code> . Currently available options are 'sum', 'prod' and 'function', where the combined costs are computed as the sum, the product or a custom function (defined in <code>FUN</code>) of the costs of its nodes.
<code>FUN</code>	a function used to compute the cost between two nodes (needed if <code>method="function"</code>).
<code>...</code>	additional parameters to be passed to <code>FUN</code> .

Details

Note that costs are inversely proportional to connectivity between edges: the larger the cost associated to an edge, the lower the connectivity between the two concerned nodes.

Also note that 'costs' defined in `geoGraph` are equivalent to 'weights' as defined in `graph` and `RBGL` packages.

Value

A `gGraph` object with the newly defined costs, based on the combination of the two `gGraph` objects, used as weightings of edges.

Examples

```

data("worldgraph.40k")
# new graph with custom cost function
exp.cost <- function(x1, x2, cost.coeff) {
  exp(-abs(x1 - x2) * cost.coeff)
}
# create a set of node costs
worldgraph.40k@nodes.attr$meanProd <- runif(n = 40962)
new_costs_graph <-
  setCosts(
    worldgraph.40k,
    node.values = worldgraph.40k@nodes.attr$meanProd,
    method = "function",
    FUN = exp.cost,
    cost.coeff = 0.5
  )
# combine costs from the original graph with the new costs
combine_costs_graph <- combineCosts(worldgraph.40k, new_costs_graph, method = "sum")

```

connectivity

Check connectivity of a gGraph object

Description

The functions `areNeighbours`, `areConnected` and the method `isConnected` test connectivity in different ways.

Usage

```

areNeighbours(V1, V2, graph)

areConnected(x, nodes)

## S4 method for signature 'gData'
isConnected(object, ...)

isReachable(x, loc)

connectivityPlot(x, ...)

## S4 method for signature 'gGraph'
connectivityPlot(x, ..., seed = NULL)

## S4 method for signature 'gData'
connectivityPlot(x, col.gGraph = 0, ..., seed = NULL)

```

Arguments

V1	a vector of node names
V2	a vector of node names
graph	a valid graphNEL object.
x	a valid gGraph object.
nodes	a vector of node names
object	a valid gData object.
...	other arguments passed to other methods.
loc	location, specified as a list of two components giving respectively the longitude and the latitude. Alternatively, it can be a matrix-like object with one row and two columns.
seed	an optional integer giving the seed to be used when randomizing colors. One given seed will always give the same set of colors. NULL by default, meaning colors are randomized each time a plot is drawn.
col.gGraph	a character string or a number indicating the color of the nodes to be used when plotting the gGraph object. Defaults to '0', meaning that nodes are invisible.

Details

- `areNeighbours`: tests connectivity between couples of nodes on an object inheriting graph class (like a [graphNEL](#) object).
- `areConnected`: tests if a set of nodes form a connected set on a [gGraph](#) object.
- `isConnected`: tests if the nodes of a [gData](#) object form a connected set. Note that this is a method for [gData](#), the generic being defined in the graph package.
- `isReachable`: tests if one location (actually, the closest node to it) is reachable from the set of nodes of a [gData](#) object.
- `connectivityPlot`: plots connected sets of a [gGraph](#) or a [gData](#) object with different colors.

In `connectivityPlot`, isolated nodes (i.e. belonging to no connected set of size > 1) are plotted in light grey.

Value

- `areNeighbours`: a vector of logical, having one value for each couple of nodes.
- `areConnected`: a single logical value, being TRUE if nodes form a connected set.
- `isConnected`: a single logical value, being TRUE if nodes of the object form a connected set.

Examples

```
connectivityPlot(rawgraph.10k)
connectivityPlot(worldgraph.10k)
```

dijkstra-methods *Shortest path using Dijkstra algorithm*

Description

The methods `dijkstraFrom` and `dijkstraBetween` are wrappers of procedures implemented in RBGL package, designed for [gGraph](#) and [gData](#) object.

Usage

```
dijkstraBetween(x, ...)

## S4 method for signature 'gGraph'
dijkstraBetween(x, from, to)

## S4 method for signature 'gData'
dijkstraBetween(x)

dijkstraFrom(x, ...)

## S4 method for signature 'gGraph'
dijkstraFrom(x, start)

## S4 method for signature 'gData'
dijkstraFrom(x, start)

## S3 method for class 'gPath'
plot(x, col = "rainbow", lwd = 3, ...)

gPath2dist(m, diag = FALSE, upper = FALSE, res.type = c("dist", "vector"))
```

Arguments

<code>x</code>	a gGraph or a gData object. For plotting method of <code>gPath</code> objects, a <code>gPath</code> object.
<code>...</code>	further arguments passed to the <code>segments</code> method.
<code>from</code>	a vector of character strings giving node names.
<code>to</code>	a vector of character strings giving node names.
<code>start</code>	a character string naming the 'source' node.

<code>col</code>	a character string indicating a color or a palette of colors to be used for plotting edges.
<code>lwd</code>	a numeric value indicating the width of edges.
<code>m</code>	a <code>gPath</code> object obtained by <code>dijkstraBetween</code> .
<code>diag, upper</code>	unused parameters added for consistency with <code>as.dist</code> .
<code>res.type</code>	a character string indicating what type of result should be returned: a <code>dist</code> object (' <code>dist</code> '), or a vector of distances (' <code>vector</code> '). Note that ' <code>dist</code> ' should only be required for pairwise data, as output by <code>dijkstraBetween</code> (as opposed to <code>dijkstraFrom</code>).

Details

`dijkstraFrom` finds minimum costs paths to nodes from a given 'source' node.

`dijkstraBetween` finds minimum costs paths between all possible pairs of nodes given two sets of nodes.

All these functions return objects with S3 class "gPath". These objects can be plotted using `plot.gPath`.

`gPath2dist` extracts the pairwise distances from the `gPath` returned by `dijkstraBetween` and returns a `dist` object. Note that if the `gPath` does not contain pairwise information, a warning will be issued, but the resulting output will likely be meaningless.

In '`dijkstraBetween`', paths are seeked all possible pairs of nodes between 'from' and 'to'.

Value

A "gPath" object. These are basically the outputs of RBGL's `sp.between` function (see `?sp.between`), with a class attribute set to "gPath", and an additional slot 'xy' containing geographic coordinates of the nodes involved in the paths.

Examples

```
## Not run:

## plotting
world <- worldgraph.40k
par(mar = rep(.1, 4))
plot(world, reset = TRUE)

## check connectivity
isConnected(hgdp) # must be ok

## Lowest cost path from an hypothetical origin
ori.coord <- list(33, 10) # one given location long/lat
points(data.frame(ori.coord), pch = "x", col = "black", cex = 3) # an 'x' shows the putative origin
ori <- closestNode(world, ori.coord) # assign it the closest node
```

```
myPath <- dijkstraFrom(hgdp, ori) # compute shortest path

## plotting
plot(world, pch = "") # plot the world
points(hgdp, lwd = 3) # plot populations
points(data.frame(ori.coord), pch = "x", col = "black", cex = 3) # add origin
plot(myPath) # plot the path

## End(Not run)
```

dropDeadEdges

Get rid of some 'dead' edges or nodes

Description

The functions `dropDeadEdges` and `dropDeadNodes` are used to remove 'dead edges' and 'dead nodes'.

Usage

```
dropDeadEdges(x, thres)
```

Arguments

<code>x</code>	a valid gGraph .
<code>thres</code>	a numeric value indicating the threshold cost for an edge to be removed. All costs strictly greater than <code>thres</code> will be removed.

Details

Dead edges are edges associated to a prohibitive cost, that is, edges that no longer imply connectivity between two nodes.

Dead nodes are nodes that are not connected to any other node, thus not having any role in the connectivity of a graph.

Value

A [gGraph](#) object.

Examples

```
## Not run:
plot(worldgraph.10k, reset = TRUE)
x <- dropDeadNodes(worldgraph.10k)
plot(x)

## End(Not run)
```

extractFromLayer	<i>Retrieves node attributes from a layer</i>
------------------	---

Description

The generic function `extractFromLayer` uses information from a GIS shapefile to define node attributes. For each node, information is retrieved from the layer and assigned to that node.

Usage

```
extractFromLayer(x, ...)
```

```
## S4 method for signature 'matrix'
extractFromLayer(x, layer = "world", attr = "all", ...)
```

```
## S4 method for signature 'data.frame'
extractFromLayer(x, layer = "world", attr = "all", ...)
```

```
## S4 method for signature 'list'
extractFromLayer(x, layer = "world", attr = "all", ...)
```

```
## S4 method for signature 'gGraph'
extractFromLayer(x, layer = "world", attr = "all", ...)
```

```
## S4 method for signature 'gData'
extractFromLayer(x, layer = "world", attr = "all", ...)
```

Arguments

<code>x</code>	a matrix, a data.frame, a list, a valid gGraph , or a valid gData object. For matrix and data.frame, input must have two columns giving longitudes and latitudes of locations being considered. For list, input must have two components being vectors giving longitudes and latitudes of locations.
<code>...</code>	further arguments to be passed to other methods. Currently not used.
<code>layer</code>	a shapefile of the class <code>SpatialPolygonsDataFrame</code> (see <code>readShapePoly</code> in <code>mapproj</code> package to import such data from a GIS shapefile). Alternatively, a character string indicating one shapefile released with <code>geoGraph</code> ; currently, only 'world' is available (see <code>?data(worldshape)</code>).

`attr` a character vector giving names of the variables to be extracted from the layer. If 'all', all available variables are extracted. In case of problem, available names are displayed with the error message. Available data are also stored in `layer@data`.

Details

Nodes can be specified in different ways, including by providing a [gGraph](#) or a [gData](#) object. Outputs match the input formats.

Value

The output depends on the nature of the input:

- `matrix`, `data.frame`, `list`: a `data.frame` with one row per location, and as many columns as requested variables ('attributes').

- `gGraph`: a [gGraph](#) object with new node attributes (`@nodes.attr` slot). If nodes attributes already existed, new attributes are added as new columns.
- `gData`: a [gData](#) object with new data associated to locations (`@data` slot). New information is merge to older information according to the type of data being stored.

See Also

[findLand](#), to find which locations are on land.

Examples

```
## Not run:

plot(worldgraph.10k, reset = TRUE)

## see what info is available
names(worldshape@data)
unique(worldshape@data$CONTINENT)

## retrieve continent info for all nodes
## (might take a few seconds)
x <- extractFromLayer(worldgraph.10k, layer = worldshape, attr = "CONTINENT")
x
table(getNodesAttr(x, attr.name = "CONTINENT"))

## subset Africa
temp <- getNodesAttr(x, attr.name = "CONTINENT") == "Africa"
temp[is.na(temp)] <- FALSE
x <- x[temp]
plot(x, reset = TRUE)
```

```
## End(Not run)
```

findLand	<i>Find which nodes are on land</i>
----------	-------------------------------------

Description

The generic function `findLand` uses information from a GIS shapefile to define which nodes are on land, and which are not. Strickly speaking, being 'on land' is in fact being inside a polygon of the shapefile.

Usage

```
findLand(x, ...)

## S4 method for signature 'matrix'
findLand(x, shape = "world", ...)

## S4 method for signature 'data.frame'
findLand(x, shape = "world", ...)

## S4 method for signature 'gGraph'
findLand(x, shape = "world", attr.name = "habitat", ...)
```

Arguments

<code>x</code>	a matrix, a data.frame, or a valid gGraph object. For matrix and data.frame, input must have two columns giving longitudes and latitudes of locations being considered.
<code>...</code>	further arguments to be passed to other methods. Currently not used.
<code>shape</code>	a shapefile of the class <code>SpatialPolygonsDataFrame</code> (see <code>readShapePoly</code> in <code>maptools</code> package to import such data from a GIS shapefile). Alternatively, a character string indicating one shapefile released with <code>geoGraph</code> ; currently, only 'world' is available (see <code>?data(worldshape)</code>).
<code>attr.name</code>	a character string giving the name of the node attribute in which the output is to be stored.

Details

Nodes can be specified either as a matrix of geographic coordinates, or as a [gGraph](#) object.

Value

The output depends on the nature of the input:

- `matrix, data.frame`: a factor with two levels being 'land' and 'sea'.

- `gGraph`: a [gGraph](#) object with a new node attribute, possibly added to previously existing node attributes (`@nodes.attr` slot).

See Also

[extractFromLayer](#), to retrieve any information from a GIS shapefile.

Examples

```
## create a new gGraph with random coordinates
myCoords <- data.frame(long = runif(1000, -180, 180), lat = runif(1000, -90, 90))
obj <- new("gGraph", coords = myCoords)
obj # note: no node attribute
plot(obj)

## find which points are on land
obj <- findLand(obj)
obj # note: new node attribute

## define rules for colors
temp <- data.frame(habitat = c("land", "sea"), color = c("green", "blue"))
temp
obj@meta$color <- temp

## plot object with new colors
plot(obj)
```

`gData-class`

Formal class "gData"

Description

The class `gData` is a formal (S4) class storing georeferenced data, consisting in a set of locations (longitude and latitude) where one or several variables have been measured. These data are designed to be matched against a [gGraph](#) object, each location being assigned to the closest node of the [gGraph](#) object.

Details

Note that for several operations on a `gData` object, the [gGraph](#) object to which it is linked will have to be present in the same environment.

Slots

`coords` a matrix of spatial coordinates with two columns, being respectively longitude (from -180 to 180) and latitude. Positive numbers are intended as 'east' and 'north', respectively.

`nodes.id` a vector of character strings giving the name of the nodes (of the `gGraph` object) associated to the locations.

`data` any kind of data associated to the locations in `coords`. For matrix-like objects, rows should correspond to locations.

`gGraph.name` a character string the name of the `gGraph` object against which the object is matched. Note that none of these is mandatory: `new("gData")` would work, and create an empty `gGraph` object. Also note that a finer matching of locations against the nodes of a `gGraph` object can be achieved after creating the object, for instance using the `closestNode` method.

Objects from the class gData

`gData` objects can be created by calls to `new("gData", ...)`, where '...' can be the following arguments:

See Also

Related class:

- `gGraph`

Examples

```
hgdp

## plot data
plot(worldgraph.40k, pch = "")
points(hgdp)

## subset and plot data
onlyNorth <- hgdp[hgdp@data$Latitude > 0] # only northern populations

plot(worldgraph.40k, reset = TRUE)
abline(h = 0) # equator
points(onlyNorth, pch.node = 20, cex = 2, col.node = "purple")
```

geo.add.edges

Add and remove edges from a gGraph object

Description

The functions `geo.add.edges` and `geo.remove.edges` allow one to add or remove edges interactively with a `gGraph` object. When adding edges, two approaches are possible:

- click vertices defining new edges (`mode="points"`)
- select an area in which all edges from a reference graph are added (`mode="area"`).

Usage

```
geo.add.edges(x, mode = c("points", "area", "all"), refObj = "rawgraph.40k")
```

Arguments

x	a valid gGraph object.
mode	a character string indicating the mode for addition or removal of edges. 'points': user is expected to click vertices to indicate edges. 'area': user is expected to click two points defining a rectangular area within which all edges are selected. 'all': all edges from the reference graph are added to the current object.
refObj	a valid gGraph object, used as a reference when adding edges. When selecting an area inside which edges are added, all edges existing in this area in refObj are added to x. Alternatively, a character string can be provided, corresponding to one of the following datasets: 'rawgraph.10k', 'rawgraph.40k'.

Value

A [gGraph](#) object with newly added or removed edges.

See Also

[setEdges](#) to non-interactively add or remove edges in a [gGraph](#) object.

Examples

```
## Not run:
plot(worldgraph.10k, reset = TRUE)

## zooming in
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))
title("Europe")

## remove edges
geo.remove.edges(worldgraph.10k) # points mode
geo.remove.edges(worldgraph.10k, mode = "area") # area mode

## add edges
geo.add.edges(worldgraph.10k) # points mode
geo.add.edges(worldgraph.10k, mode = "area") # area mode

## End(Not run)
```

geo.change.attr *Change values of a node attribute*

Description

The functions `geo.change.attr` changes values of a given node attribute for a set of selected nodes of a [gGraph](#) object.

Usage

```
geo.change.attr(
  x,
  mode = c("points", "area"),
  attr.name,
  attr.value,
  only.name = NULL,
  only.value = NULL,
  newCol = "black",
  restore.edges = FALSE,
  refObj = "rawgraph.40k"
)
```

Arguments

<code>x</code>	a valid gGraph object.
<code>mode</code>	a character string indicating whether selected nodes are clicked one by one ('points') or by defining a rectangular area ('area').
<code>attr.name</code>	the name of the node attribute to be modified.
<code>attr.value</code>	the new value of attribute assigned to selected nodes.
<code>only.name</code>	(optional) in area mode, the name of a node attribute to add an extra selection criterion. See details.
<code>only.value</code>	(optional) in area mode, and if <code>only.name</code> is specified, the values of <code>only.name</code> that can be selected. See details.
<code>newCol</code>	a character string giving the new color for the attribute value.
<code>restore.edges</code>	a logical indicating whether edges stemming from the modified nodes should be re-added to the graph, using <code>refObj</code> as a reference. This is useful when connectivity is to be redefined using setCosts for nodes that were previously disconnected.
<code>refObj</code>	a character string or a gGraph object, used as reference when re-adding edges. If a character string is provided, it must match one of the following dataset: 'rawgraph.10k', 'rawgraph.40k'.

Details

The argument `only.name` allows one to perform a more accurate selection of nodes whose attribute is changed, by specifying values (`only.value`) of an attribute (`only.name`) that can be selected. For instance, one may want to define new attributes for nodes of `worldgraph.10k` that are exclusively on land: this would be done by specifying `only.name="habitat"` and `only.value="land"`.

Value

A `gGraph` object with modified node attributes.

Examples

```
## Not run:
plot(worldgraph.10k, reset = TRUE)

## have to click here for an area
## all nodes are modified in the area
x <- geo.change.attr(worldgraph.10k, mode = "area", attr.name = "habitat", attr.value = "fancy
habitat", newCol = "pink") # modify selected area

plot(x, reset = TRUE) # modification in the whole selected area

## have to click here for an area
## only nodes on land are modified
x <- geo.change.attr(x, mode = "area", attr.name = "habitat", attr.value = "fancy2
habitat", newCol = "purple", only.name = "habitat", only.value = "land")

plot(x, reset = TRUE) # modification in the whole selected area

## End(Not run)
```

getColors

Get colors associated to edges of a gGraph object

Description

The function `getColors` returns the colors associated to the nodes of a `gGraph` object, based on a specified node attribute.

Usage

```
getColors(x, ...)

## S4 method for signature 'gGraph'
getColors(x, nodes = "all", attr.name, col.rules = NULL, ...)
```

Arguments

x	a valid gGraph .
...	other arguments passed to other methods.
nodes	a vector of character strings or of integers identifying nodes by their name or their index. Can be "all", in which case all nodes are considered.
attr.name	a character string indicating the name of node attribute to be used to define colors.
col.rules	a matrix giving the rules for plotting attribute values with different colors. See details.

Details

Colors are based on a node attribute, that is, on a column of the `nodes.attr` data.frame. This attribute should have a finite number of values, and would most likely be a factor. Correspondence between values of this variable and colors must be provided in the `@meta$color` slot, or as `col.rules` argument. Color rules must be provided as a two-column matrix; the first column contains values of a node attribute, and is named after this attribute; the second must be named "color", and contain valid colors.

See example section to know how this slot should be designed.

Value

A vector of characters being valid colors.

Functions

- `getColors(gGraph)`: Method for `gGraph` objects

Examples

```
worldgraph.10k # there is a node attribute 'habitat'
worldgraph.10k@meta$color

head(getNodes(worldgraph.10k))
head(getColors(worldgraph.10k, res.type = "vector", attr.name = "habitat"))
```

getCosts

Get costs associated to edges of a gGraph object

Description

The function `getCosts` returns the costs associated to the edges of a `gGraph` object using different possible outputs. These outputs are designed to match possible outputs of `getEdges` function.

Usage

```

getCosts(x, ...)

## S4 method for signature 'gGraph'
getCosts(x, res.type = c("asIs", "vector"), unique = FALSE, ...)

getNodeCosts(x, ...)

## S4 method for signature 'gGraph'
getNodeCosts(x, attr.name, ...)

```

Arguments

x	a valid gGraph .
...	other arguments passed to other methods (currently unused).
res.type	a character string indicating which kind of output should be used. See value.
unique	a logical indicating whether the costs should be returned for unique edges (TRUE), or if duplicate edges should be considered as well (TRUE, default).
attr.name	the name of the node attribute used to define node costs.

Details

getNodeCosts returns the costs associated to nodes based on one node attribute.

The notion of 'costs' in the context of [gGraph](#) objects is identical to the concept of 'weights' in [graph](#) (and thus [graphNEL](#)) objects. The larger it is for an edge, the less connectivity there is between the couple of concerned nodes.

Value

The output depends on the value of the argument `res.type`:

- `asIs`: output is a named list of weights, each slot containing weights associated to the edges stemming from one given node. This format is that of the `weights` accessor for [graphNEL](#) objects.
- `vector`: a vector of weights; this output matches matrix outputs of [getEdges](#).

Functions

- `getCosts(gGraph)`: Method for [gGraph](#) object
- `getNodeCosts()`: Function to get the costs values for nodes
- `getNodeCosts(gGraph)`: Method to get node costs for [gGraph](#) object

See Also

Most other accessors are documented in [gGraph](#) manpage.

Examples

```
head(getEdges(worldgraph.10k, res.type = "matNames", unique = TRUE))
head(getCosts(worldgraph.10k, res.type = "vector", unique = TRUE))
```

getEdges

Get edges from a gGraph object

Description

The function `getEdges` returns the edges of a [gGraph](#) object using different possible outputs.

Usage

```
getEdges(x, ...)

## S4 method for signature 'gGraph'
getEdges(x, res.type = c("asIs", "matNames", "matId"), unique = FALSE, ...)
```

Arguments

<code>x</code>	a valid gGraph .
<code>...</code>	other arguments passed to other methods (currently unused).
<code>res.type</code>	a character string indicating which kind of output should be used. See value.
<code>unique</code>	a logical indicating whether all returned edges should be unique (TRUE) or if duplicated edges should be allowed (TRUE, default).

Value

The output depends on the value of the argument `res.type`:

- `asIs`: output is a named list of nodes, each slot containing nodes forming an edge with one given node. This format is that of the edges accessor for [graphNEL](#) objects.
- `matNames`: a matrix with two columns giving couples of node names forming edges.
- `matId`: a matrix with two columns giving couples of node indices forming edges.

Functions

- `getEdges(gGraph)`: Method for `gGraph` objects

See Also

Most other accessors are documented in [gGraph](#) manpage.

See [setEdges](#) to add/remove edges, or [geo.add.edges](#) and [geo.remove.edges](#) for interactive versions.

Examples

```
example(gGraph)

getEdges(x)
getEdges(x, res.type = "matNames")
getEdges(x, res.type = "matId")
```

getNodesAttr

Get nodes attributes from gGraph/gData object

Description

The function `getNodesAttr` returns the values of a set of variables associated to the nodes (i.e. node attributes) of a `gGraph` or `gData` object.

Usage

```
getNodesAttr(x, ...)

## S4 method for signature 'gGraph'
getNodesAttr(x, nodes = NULL, attr.name = NULL, ...)

## S4 method for signature 'gData'
getNodesAttr(x, attr.name = NULL, ...)
```

Arguments

<code>x</code>	a valid <code>gGraph</code> or <code>gData</code> object.
<code>...</code>	other arguments passed to other methods (currently unused).
<code>nodes</code>	an optional integer, logical, or character string indicating the subset of nodes to be used. If <code>NULL</code> , all nodes are used.
<code>attr.name</code>	an optional character string indicating which node attributes should be returned. If provided, it must match at least one of the columns of <code>x@nodes.attr</code> .

Value

A data.frame with the requested nodes attributes. Nodes are displayed in rows, variables in columns.

Functions

- `getNodeAttr(gGraph)`: Method for gGraph objects
- `getNodeAttr(gData)`: Method for gData objects

See Also

Most other accessors are documented in [gGraph](#) and [gData](#) manpages.

Examples

```
## gGraph method
head(getNodeAttr(worldgraph.40k))
```

```
## gData method
getNodeAttr(hgdp)
```

gGraph-class

Formal class "gGraph"

Description

The class gGraph is a formal (S4) class storing geographic data. Such data are composed of a set of geographic coordinates of vertices (or 'nodes'), and a graph describing connectivity between these vertices. Data associated to the nodes can also be stored ('nodes attributes'), as well as meta-information used when plotting the object, or when computing weights associated to the edges based on nodes attributes.

Details

In all slots, nodes are uniquely identified by their name (reference is taken from the row names of @coords slot).

Slots

`coords` a matrix of spatial coordinates with two columns, being respectively longitude (from -180 to 180) and latitude. Positive numbers are intended as 'east' and 'north', respectively.

`nodes.attr` a data.frame whose rows are nodes, and whose columns are different variables associated to the nodes.

`meta` list, most likely containing named data.frames (see Slots).

graph an object of the class [graphNEL](#), from the graph package (see `class?graphNEL`), describing connectivity among nodes.

Note that none of these is mandatory: `new("gGraph")` would work, and create an empty gGraph object.

Objects from the class gGraph

gGraph objects can be created by calls to `new("gGraph", ...)`, where '...' can be the following arguments:

See Also

Related classes are:

% - [graphNEL](#) (graph package): `slot @graph` in gGraph.

Examples

```
## create an empty object
new("gGraph")

## plotting the object
plot(rawgraph.10k, reset = TRUE)

## zooming in
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))
title("Europe")

## to play interactively with graphics, use:
# geo.zoomin()
# geo.zoomout()
# geo.slide()
# geo.back()

## defining a new object restrained to visible nodes
x <- rawgraph.10k[isInArea(rawgraph.10k)]
plot(x, reset = TRUE, edges = TRUE)
title("x does just contain these visible nodes.")

## define weights for edges
x <- setCosts(x, attr.name = "habitat", method = "prod")
plot(x, edges = TRUE)
title("costs defined by habitat (land/land=1, other=100)")

## drop 'dead edges' (i.e. with weight 0)
x <- dropDeadEdges(x, thres = 10)
plot(x, edges = TRUE)
title("after dropping edges with null weight")
```

hgdp

*Human genome diversity panel - georeferenced data***Description**

The datasets hgdp and hgdpPlus provides genetic diversity several human populations worldwide. Both datasets are [gData](#) objects, interfaced with the [gGraph](#) object [worldgraph.40k](#).

Format

hgdp is a [gGraph](#) object with the following data: %
%

@nodes.attr\$habitat habitat corresponding to each % vertice; currently 'land' or 'sea'.%

@meta\$color a matrix assigning a color for plotting % vertices (second column) to different values of habitat (first % column).%

Details

hgdp describes 52 populations from the original Human Genome Diversity Panel.

hgdpPlus describes hgdp populations plus 24 native American populations.

References

Authors *Journal*, YEAR, **nb**: pp-pp.

Examples

```
## check object
hgdp

## plotting the object
plot(hgdp)

## results from Handley et al.
## Not run:
## Addis Ababa
addis <- list(lon = 38.74, lat = 9.03)
addis <- closestNode(worldgraph.40k, addis) # this takes a while

## shortest path from Addis Ababa
myPath <- dijkstraFrom(hgdp, addis)

## plot results
plot(worldgraph.40k, col = 0)
```

```

points(hgdp)
points(worldgraph.40k[addis], psize = 3, pch = "x", col = "black")
plot(myPath)

## correlations distance/genetic div.
geo.dist <- sapply(myPath[-length(myPath)], function(e) e$length)
gen.div <- getData(hgdp)[, "Genetic.Div"]
plot(gen.div ~ geo.dist)
lm1 <- lm(gen.div ~ geo.dist)
abline(lm1, col = "blue") # this regression is wrong
summary(lm1)

## End(Not run)

```

isInArea

Find which nodes fall in a given area

Description

The generic function `isInArea` finds which nodes fall in a given area. Nodes can be specified in different ways, including by providing a `gGraph` or a `gData` object. Different format for the output are also available.

Usage

```

isInArea(x, ...)

## S4 method for signature 'matrix'
isInArea(
  x,
  reg = "current",
  res.type = c("logical", "integer", "character"),
  buffer = 0
)

## S4 method for signature 'data.frame'
isInArea(
  x,
  reg = "current",
  res.type = c("logical", "integer", "character"),
  buffer = 0
)

## S4 method for signature 'gGraph'
isInArea(
  x,
  reg = "current",

```

```

    res.type = c("logical", "integer", "character"),
    buffer = 0
  )

## S4 method for signature 'gData'
isInArea(
  x,
  reg = "current",
  res.type = c("logical", "integer", "character"),
  buffer = 0
)

```

Arguments

<code>x</code>	a matrix, a data.frame, a valid gGraph , or a valid gData object. For matrix and data.frame, input must have two columns giving longitudes and latitudes of locations being considered.
<code>...</code>	further arguments passed to specific methods.
<code>reg</code>	a character string or a list indicating the area ('reg' stands for 'region'). Character strings can be "current" (current user window, default) or "zoom" (current zoom). If the argument is a list, it has to have two components, both being numeric vectors of length two, giving x and y limits of the area. Note that such list can be produced by <code>locator</code> , so <code>locator(1)</code> is a valid value for <code>reg</code> .
<code>res.type</code>	a character string indicating what kind of output should be produced. See value.
<code>buffer</code>	a numeric value giving a buffer adding extra space around the area, as a proportion of current area's dimensions.

Value

The output depends on the value of the argument `res.type`:

- `logical`: a vector of logicals having one value for each node of the input.
- `integer`: a vector of integers corresponding to the indices of nodes falling within the area.
- `character`: a vector of characters corresponding to the names of the nodes falling within the area.

Functions

- `isInArea(matrix)`: Method for matrix
- `isInArea(data.frame)`: Method for data.frame
- `isInArea(gGraph)`: Method for gGraph object
- `isInArea(gData)`: Method for gData object

Examples

```

plot(worldgraph.10k, reset = TRUE)

## zooming in
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))
title("Europe")

## different outputs of isInArea
head(isInArea(worldgraph.10k)) # logical
length(isInArea(worldgraph.10k))
sum(isInArea(worldgraph.10k))
head(which(isInArea(worldgraph.10k))) # which nodes are TRUE ?

head(isInArea(worldgraph.10k, res.type = "integer")) # node indices

head(isInArea(worldgraph.10k, res.type = "character")) # node names

## use isInArea to have a subset of visible nodes
x <- worldgraph.10k[isInArea(worldgraph.10k)]
plot(x, reset = TRUE)

```

makeGrid

Build a regular grid gGraph

Description

The function `makeGrid` builds a [gGraph](#) using a regular grid for a given area. If no area is specified, currently plotted area is used. Note that such grid is only valid for small scales, for cases in which curvature of the surface of the earth can be neglected.

Usage

```

makeGrid(
  size = NULL,
  n.lon = NULL,
  n.lat = NULL,
  lon.range = NULL,
  lat.range = NULL
)

```

Arguments

<code>size</code>	an integer giving the approximate number of nodes of the grid. The function will attempt to make a square grid of (approximately) this size.
<code>n.lon</code>	the number of longitude coordinates of the grid (i.e., width of the grid, in number of cells)

n.lat the number of latitude coordinates of the grid (i.e., height of the grid, in number of cells)

lon.range, lat.range vectors of length two giving the range covered by the grid, in longitude and latitude, respectively.

Value

A [gGraph](#) object.

Examples

```
## zoom in to a smaller area
plot(worldgraph.10k)
geo.zoomin(c(-10, 0, 50, 54))

## make a new gGraph
newGraph <- makeGrid(1e3)
newGraph <- findLand(newGraph)
newGraph@meta$colors <- data.frame(
  habitat = c("sea", "land"),
  color = c("blue", "green")
)

## plot the new gGraph
plot(newGraph, reset = TRUE, edge = TRUE)
```

plot-gData

Plot a gData object.

Description

Various functions to plot a [gData](#) object: `plot` opens a device and plots the object, while `points` plots the object on the existing device. Plotting of [gData](#) object relies on plotting the [gGraph](#) object to which it is linked, and then represent the locations of the [gData](#) and/or the associated nodes.

Usage

```
## S4 method for signature 'gData,missing'
plot(
  x,
  type = c("nodes", "original", "both"),
  pch.ori = 4,
  pch.nodes = 1,
  col.ori = "black",
  col.nodes = "red",
```

```

    col.gGraph = NULL,
    reset = FALSE,
    sticky.points = TRUE,
    ...
)

## S4 method for signature 'gData'
points(
  x,
  type = c("nodes", "original", "both"),
  pch.ori = 4,
  pch.nodes = 1,
  col.ori = "black",
  col.nodes = "red",
  sticky.points = TRUE,
  ...
)

```

Arguments

x	a valid gData object. The gData object to which it is linked must exist in the global environment.
type	a character string indicating which information should be plotted: original locations ('original'), associated nodes ('nodes', default), or both ('both'). In the latter case, an arrow goes from locations to nodes.
pch.ori	a numeric or a character indicating the type of point for locations.
pch.nodes	a numeric or a character indicating the type of point for nodes.
col.ori	a character string indicating the color to be used for locations.
col.nodes	a character string indicating the color to be used for nodes.
col.gGraph	a (recycled) color vector for the associated gGraph object. If NULL, default color is used. Set to NA or "transparent" to avoid plotting the gGraph .
reset	a logical stating whether the plotting area should be reset to fit the gData object (TRUE), or should conserve previous plotting and settings (FALSE, default).
sticky.points	a logical indicating if added points should be kept when replotting (TRUE, default), or not (FALSE). In any case, reset=TRUE will prevent points to be re-drawn.
...	further arguments passed to points.

Details

When `sticky.points` is set to TRUE, all operations performed on the graphics like zooming or sliding the window can be performed without losing the `gData` plot.

Functions

- `points(gData)`: Plot as points

See Also

- Different functions to explore these plots:
[geo.zoomin](#), [geo.zoomout](#), [geo.slide](#), [geo.back](#), [geo.bookmark](#), [geo.goto](#).

Examples

```
myLoc <- list(x = c(3, -8, 11, 28), y = c(50, 57, 71, 67)) # some locations
obj <- new("gData", coords = myLoc) # new gData object
obj

obj@gGraph.name <- "worldgraph.10k"
obj <- closestNode(obj, attr.name = "habitat", attr.value = "land")

## plot the result (original location -> assigned node)
plot(obj, type = "both", reset = TRUE)
title("'x'=location, 'o'=assigned node")

## using different parameters
points(obj, type = "both", pch.ori = 2, col.ori = "red", pch.nodes = 20, col.nodes = "pink")

## only nodes, fancy plot
plot(obj, col.nodes = "red", cex = 1, pch.node = 20)
points(obj, col.nodes = "red", cex = 2)
points(obj, col.nodes = "orange", cex = 3)
points(obj, col.nodes = "yellow", cex = 4)
```

plot-gGraph

Plot a gGraph object.

Description

Various functions to plot a [gGraph](#) object: `plot` opens a device and plot the object, while `points` plots the object on the existing device. `plotEdges` only plots the edges of the graph: it can be called directly, or via arguments passed to `plot` and `points`.

Arguments

<code>x</code>	a gGraph object.
<code>shape</code>	a shapefile used as background to the object. Must be of the class <code>SpatialPolygonsDataFrame</code> (see <code>readShapePoly</code> in <code>maptools</code> package to import such data from a GIS shapefile). Alternatively, a character string indicating one shapefile released with <code>geoGraph</code> .
<code>psize</code>	a numeric giving the size of points.
<code>pch</code>	a numeric or a character indicating the type of point.

<code>col</code>	a character string indicating the color to be used.
<code>edges</code>	a logical indicating if edges should be plotted (TRUE) or not (FALSE).
<code>reset</code>	a logical indicating if plotting parameters should be reset (TRUE) or not (FALSE).
<code>bg.col</code>	a character string indicating the color of the polygons of the shapefile used as background.
<code>border.col</code>	a character string indicating the color of the polygon borders.
<code>lwd</code>	a numeric indicating the width of line (used for edges).
<code>useCosts</code>	a logical indicating if edge width should be inversely proportionnal to edge cost (TRUE) or not (FALSE).
<code>maxLwd</code>	a numeric indicating the maximum edge width (corresponding to the maximum weight).
<code>col.rules</code>	a data.frame with two named columns, the first one giving values of a node attribute, and the second one stating colors to be used for each value. If not provided, this is seeked from the <code>@meta\color</code> slot of the object.
<code>sticky.points</code>	a logical indicating if added points should be kept when replotting (TRUE), or not (FALSE). In any case, <code>reset=TRUE</code> will prevent points to be redrawn.
<code>lty</code>	the type of line (for the edges).
<code>pcol</code>	a character indicating the color to be used for points.
<code>sticky.edges</code>	a logical indicating whether added edges should be kept when replotting (TRUE), or not (FALSE, default). In any case, <code>reset=TRUE</code> will prevent points to be redrawn.
<code>...</code>	further arguments passed to the generic methods (<code>plot</code> , <code>points</code> , and <code>segments</code> , respectively).

Details

Plotting of a `gGraph` object stores some parameters in R; see details for more information.

To be able to zoom in and out, or slide the window, previous plotting information are stored in a particular environment (`.geoGraphEnv`), which is created when loading `geoGraph`. Users should not have to interact directly with objects in this environment.

The resulting plotting behaviour is that when plotting a `gGraph` object, last plotting parameters are re-used. To override this behaviour, specify `reset=TRUE` as argument to `plot`.

See Also

- Different functions to explore these plots:
[geo.zoomin](#), [geo.zoomout](#), [geo.slide](#), [geo.back](#).
- [isInArea](#), to retain a set of visible data.

Examples

```
## just the background
plot(worldgraph.10k, reset = TRUE, type = "n")

## basic plot
plot(worldgraph.10k)

## zooming and adding edges
geo.zoomin(list(x = c(90, 150), y = c(0, -50)))
plot(worldgraph.10k, edges = TRUE)

## display edges differently
plotEdges(worldgraph.10k, col = "red", lwd = 2)

## replot points with different color
points(worldgraph.10k, col = "orange")

## mask points in the sea
inSea <- unlist(getNodesAttr(worldgraph.10k, attr.name = "habitat")) == "sea"
head(inSea)
points(worldgraph.10k[inSea], col = "white", sticky = TRUE) # this will stay

## but better, only draw those on land, and use a fancy setup
par(bg = "blue")
plot(worldgraph.10k[!inSea], bg.col = "darkgreen", col = "purple", edges = TRUE)
```

setCosts

Set friction in a gGraph object

Description

The function `setCosts` define costs for the edges of a `gGraph` object according to a node attribute and some rules defined in the `@meta\costs` slot of the object. Each node has a value for the chosen attribute, which is associated to a costs (a friction). The cost of an edge is computed as a function (see argument `method`) of the costs of its nodes.

Usage

```
setCosts(
  x,
  attr.name = NULL,
  node.values = NULL,
  method = c("mean", "product", "function"),
  FUN = NULL,
```

```
    ...
  )
```

Arguments

x	a gGraph object with a least one node attribute, and a @meta\$costs component (for an example, see worldgraph.10k dataset).
attr.name	the name of the node attribute used to compute costs (i.e., of one column of @nodes.attr).
node.values	a numeric vector giving costs associated to the nodes. If provided, it will be used instead of attr.name.
method	a character string indicating which method should be used to compute edge cost from nodes costs. Currently available options are 'mean', 'prod' and 'function', where the cost associated to an edge is respectively computed as the mean, the product or a custom function (defined in FUN) of the costs of its nodes.
FUN	a function used to compute the cost between two nodes (needed if method="function").
...	additional parameters to be passed to FUN.

Details

Note that costs are inversely proportional to connectivity between edges: the larger the cost associated to an edge, the lower the connectivity between the two concerned nodes.

Also note that 'costs' defined in [geoGraph](#) are equivalent to 'weights' as defined in [graph](#) and [RBGL](#) packages.

Value

A [gGraph](#) object with the newly defined costs used as weightings of edges.

See Also

[dropDeadEdges](#), to get rid of edge whose cost is below a given threshold. [geo.add.edges](#) to add edges to a [gGraph](#) object.

Examples

```
plot(rawgraph.10k, reset = TRUE)

## zooming in
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))
title("Europe")

## defining a new object restrained to visible nodes
x <- rawgraph.10k[isInArea(rawgraph.10k)]

## define weights for edges
x <- setCosts(x, attr.name = "habitat")
plot(x, edges = TRUE)
```

```
title("costs defined by habitat (land/land=1, other=100)")
```

setDistCosts

Set costs associated to edges based on geographic distances

Description

The function `setDistCosts` sets the costs of a `gGraph` object using the geographic distance. The cost associated to an edge is defined as the great circle distance between the two nodes of this edge. `setDistCosts` actually relies on `rdist.earth` of the `fields` package.

Usage

```
setDistCosts(x, ...)  
  
## S4 method for signature 'gGraph'  
setDistCosts(x, ...)
```

Arguments

`x` a valid `gGraph`.
`...` other arguments passed to other methods (currently unused).

Details

The notion of 'costs' in the context of `gGraph` objects is identical to the concept of 'weights' in `graph` (and thus `graphNEL`) objects. The larger it is for an edge, the less connectivity there is between the couple of concerned nodes.

Value

For the `gGraph` method, a `gGraph` object with appropriate weights. Note that former weights will be removed from the object.

Functions

- `setDistCosts(gGraph)`: Method for `gGraph` object

See Also

The `getCosts` accessor, returning costs of the edges of a `gGraph` object in different ways.

Examples

```

if (require(fields)) {
  ## load data
  plot(rawgraph.10k, reset = TRUE)
  geo.zoomin(list(x = c(110, 150), y = c(-10, -40)))
  plotEdges(rawgraph.10k)

  ## compute costs
  x <- rawgraph.10k[isInArea(rawgraph.10k)]
  x <- setDistCosts(x)

  ## replot edges
  plotEdges(x) # no big differences can be seen
  head(getCosts(x))
}

```

setEdges

Add and remove edges from a gGraph object

Description

The function `setEdges` allows one to add or remove edges in a [gGraph](#) by directly specifying the relevant nodes, as a list or a `data.frame`. This low-level function is called by `geo.add.edges` and `geo.remove.edges`.

Usage

```

setEdges(x, ...)

## S4 method for signature 'gGraph'
setEdges(x, add = NULL, remove = NULL, costs = NULL, ...)

```

Arguments

<code>x</code>	a valid gGraph object.
<code>...</code>	other arguments passed to other methods (currently unused).
<code>add</code>	a list or a <code>data.frame</code> containing node names of edges to be added. The first element of the list (or column of the <code>data.frame</code>) gives starting nodes of edges; the second gives ending nodes. Hence, the nodes of the <i>i</i> -th edge are <code>add[[1]][i]</code> and <code>add[[2]][i]</code> if <code>add</code> is a list, and <code>add[i,]</code> if <code>add</code> is a <code>data.frame</code> .
<code>remove</code>	same as <code>add</code> argument, but edges are removed.
<code>costs</code>	a numeric vector providing costs of the edges to be added. <code>costs[i]</code> is the weight of the <i>i</i> -th edge.

Value

A [gGraph](#) object with newly added or removed edges.

Functions

- `setEdges(gGraph)`: Method for `gGraph` object

See Also

`geo.add.edges` and `geo.remove.edges` to interactively add or remove edges in a `gGraph` object.

`getEdges` to retrieve edges in different formats.

worldgraph

Worldwide geographic graphs

Description

The datasets 'rawgraph.10k', 'rawgraph.40k', 'worldgraph.10k', and 'worldgraph.40k' are geographic graphs (`gGraph` objects) of the world, with respective resolutions of 10,242 and 40,962 vertices.

Format

`worldgraph.10k` and `worldgraph.40k` are `gGraph` objects with the following specificities:

@nodes.attr\$habitat habitat corresponding to each vertice; currently 'land' or 'sea'.

@meta\$color a matrix assigning a color for plotting vertices (second column) to different values of habitat (first column).

Details

'rawgraph's are raw graphs as obtained directly from the method provided in references.

'worldgraph's are 'rawgraph's that have been modified manually to rectify connectivity between edges at some places. The most noticeable change is that all edges involving sea vertices have been removed.

'worldshape' is a shapefile of countries of the world (snapshot from 1994).

Source

Graph reconstructed by Andrea Manica.

References

=== On the construction of the graph ===

Randall, D. A.; Ringler, T. D.; Heikes, R. P.; Jones, P. & Baumgardner, J. Climate Modeling with Spherical Geodesic Grids *Computing in science & engineering*, 2002, **4**: 32-41.

Examples

```

worldgraph.10k

## plotting the object
plot(worldgraph.10k, reset = TRUE)
title("Hello world")

## zooming in
geo.zoomin(list(x = c(-12, 45), y = c(33, 75)))
title("Europe")
geo.zoomin(list(x = c(-12, 2), y = c(50, 60)))
plotEdges(worldgraph.10k)
title("United Kingdom")

## zooming out
# geo.zoomout() # needs clicking on device
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))
title("Europe")

## defining the subset of visible points
x <- worldgraph.10k[isInArea(worldgraph.10k)]
plot(x, reset = TRUE, edges = TRUE)
title("One subsetted object.")

## Not run:
## interactive zooming
geo.zoomin()

## End(Not run)

```

zoom

Navigate in the plot of a gGraph object

Description

The functions `geo.zoomin`, `geo.zoomout`, `geo.slide`, `geo.back`, `geo.bookmark` and `geo.goto` are used to navigate interactively in the plot of a [gGraph](#) object.

Arguments

<code>reg</code>	a list of length 2, with its first component being the new x (longitude) boundaries (a vector of length 2), and its second being new y (latitude) boundaries (a vector of length 2).
<code>vec</code>	a numeric vector of length 4 giving the new coordinates of the plotting window, in the order: <code>xmin</code> , <code>xmax</code> , <code>ymin</code> , <code>ymax</code> .
<code>name</code>	a character string giving the name of the bookmark to create (in <code>geo.bookmark</code>) or to get back to (in <code>geo.goto</code>).

Details

`geo.zoomin` and `geo.zoomout` are used to zoom in and out. For zooming in, the user has to delimit the opposite corner of the new plotting area; alternatively, a set of coordinates can be provided. For zooming out, each click on the screen will zoom out further.

`geo.slide` moves the window toward the direction indicated by clicking in the screen.

`geo.back` redraws previous plots each time screen is clicked.

`geo.bookmark` sets a bookmark for the current area. If the name for the bookmark is left to `NULL`, then the list of currently available bookmarks is returned.

`geo.goto` allows the user to get back to a bookmarked area.

`.zoomlog.up` is an auxiliary function used to update the zoom log, by providing new sets of coordinates.

Whenever clicking is needed, a right-click will stop the function.

See Also

[plot.gGraph](#) for plotting of a `gGraph` object.

Examples

```
plot(worldgraph.10k, reset = TRUE)

## zooming in
x.ini <- c(-100, -60)
y.ini <- c(-30, 30)
for (i in 0:3) {
  geo.zoomin(list(x = x.ini + i * 60, y = y.ini))
}

## Not run:
## going back
geo.back() # you have to click !

## zooming in interactively
geo.zoomin() # you have to click !

## zooming out
geo.zoomout() # you have to click !

## moving window
geo.slide() # you have to click !

## End(Not run)
```

Index

- * **classes**
 - gGraph-class, 28
- * **datasets**
 - hgdp, 30
 - worldgraph, 42
- * **graphs**
 - gGraph-class, 28
- * **hplot**
 - plot-gData, 34
 - plot-gGraph, 36
 - zoom, 43
- * **manip**
 - geoGraph-package, 2
- * **methods**
 - auxiliary, 5
 - buffer, 6
 - closestNode, 8
 - connectivity, 11
 - dijkstra-methods, 13
 - dropDeadEdges, 15
 - extractFromLayer, 16
 - findLand, 18
 - getColors, 23
 - getCosts, 24
 - getEdges, 26
 - getNodesAttr, 27
 - isInArea, 31
 - makeGrid, 33
 - plot-gData, 34
 - plot-gGraph, 36
 - setDistCosts, 40
 - setEdges, 41
- * **spatial**
 - dijkstra-methods, 13
 - geoGraph-package, 2
 - gGraph-class, 28
 - plot-gData, 34
 - plot-gGraph, 36
- * **utilities**
 - auxiliary, 5
 - buffer, 6
 - closestNode, 8
 - connectivity, 11
 - dropDeadEdges, 15
 - extractFromLayer, 16
 - findLand, 18
 - geo.add.edges, 20
 - geo.change.attr, 22
 - getColors, 23
 - getCosts, 24
 - getEdges, 26
 - getNodesAttr, 27
 - isInArea, 31
 - makeGrid, 33
 - setCosts, 38
 - setDistCosts, 40
 - setEdges, 41
 - zoom, 43
 - .zoomlog.up (zoom), 43
 - [,gData,ANY,ANY,ANY-method (gData-class), 19
 - [,gData,ANY,ANY-method (gData-class), 19
 - [,gData-method (gData-class), 19
 - [,gGraph,ANY,ANY,ANY-method (gGraph-class), 28
 - [,gGraph,ANY,ANY-method (gGraph-class), 28
 - [,gGraph-method (gGraph-class), 28
 - areConnected, 3
 - areConnected (connectivity), 11
 - areNeighbours (connectivity), 11
 - auxiliary, 5
 - buffer, 6
 - buffer,gData-method (buffer), 6
 - buffer,gGraph-method (buffer), 6
 - buffer-methods (buffer), 6

- closestNode, [3](#), [8](#)
- closestNode,gData-method (closestNode), [8](#)
- closestNode,gGraph-method (closestNode), [8](#)
- closestNode-methods (closestNode), [8](#)
- combineCosts, [10](#)
- connectivity, [11](#)
- connectivityPlot, [3](#)
- connectivityPlot (connectivity), [11](#)
- connectivityPlot,gData-method (connectivity), [11](#)
- connectivityPlot,gGraph-method (connectivity), [11](#)
- connectivityPlot-methods (connectivity), [11](#)

- dijkstra-methods, [13](#)
- dijkstraBetween (dijkstra-methods), [13](#)
- dijkstraBetween,gData-method (dijkstra-methods), [13](#)
- dijkstraBetween,gGraph-method (dijkstra-methods), [13](#)
- dijkstraBetween-methods (dijkstra-methods), [13](#)
- dijkstraFrom (dijkstra-methods), [13](#)
- dijkstraFrom,gData-method (dijkstra-methods), [13](#)
- dijkstraFrom,gGraph-method (dijkstra-methods), [13](#)
- dijkstraFrom-methods (dijkstra-methods), [13](#)
- dropCosts (gGraph-class), [28](#)
- dropCosts,gGraph-method (gGraph-class), [28](#)
- dropDeadEdges, [3](#), [15](#), [39](#)
- dropDeadNodes (dropDeadEdges), [15](#)

- extractFromLayer, [3](#), [16](#), [19](#)
- extractFromLayer,data.frame-method (extractFromLayer), [16](#)
- extractFromLayer,gData-method (extractFromLayer), [16](#)
- extractFromLayer,gGraph-method (extractFromLayer), [16](#)
- extractFromLayer,list-method (extractFromLayer), [16](#)
- extractFromLayer,matrix-method (extractFromLayer), [16](#)

- extractFromLayer-methods (extractFromLayer), [16](#)

- findLand, [3](#), [17](#), [18](#)
- findLand,data.frame-method (findLand), [18](#)
- findLand,gGraph-method (findLand), [18](#)
- findLand,matrix-method (findLand), [18](#)
- findLand-methods (findLand), [18](#)

- gData, [6–9](#), [12](#), [13](#), [16](#), [17](#), [27](#), [28](#), [30–32](#), [34](#), [35](#)
- gData (gData-class), [19](#)
- gData-class, [19](#)
- geo.add.edges, [3](#), [9](#), [20](#), [27](#), [39](#), [42](#)
- geo.back, [4](#), [36](#), [37](#)
- geo.back (zoom), [43](#)
- geo.bookmark, [4](#), [36](#)
- geo.bookmark (zoom), [43](#)
- geo.change.attr, [3](#), [22](#)
- geo.goto, [4](#), [36](#)
- geo.goto (zoom), [43](#)
- geo.remove.edges, [3](#), [9](#), [27](#), [42](#)
- geo.remove.edges (geo.add.edges), [20](#)
- geo.segments (auxiliary), [5](#)
- geo.slide, [4](#), [36](#), [37](#)
- geo.slide (zoom), [43](#)
- geo.zoomin, [4](#), [36](#), [37](#)
- geo.zoomin (zoom), [43](#)
- geo.zoomout, [4](#), [36](#), [37](#)
- geo.zoomout (zoom), [43](#)
- geoGraph (geoGraph-package), [2](#)
- geoGraph-package, [2](#)
- getColors, [23](#)
- getColors,gGraph-method (getColors), [23](#)
- getColors-methods (getColors), [23](#)
- getCoords, [3](#)
- getCoords (gGraph-class), [28](#)
- getCoords,gData-method (gData-class), [19](#)
- getCoords,gGraph-method (gGraph-class), [28](#)
- getCosts, [3](#), [24](#), [40](#)
- getCosts,gGraph-method (getCosts), [24](#)
- getCosts-methods (getCosts), [24](#)
- getData (gData-class), [19](#)
- getData,gData-method (gData-class), [19](#)
- getData-methods (gData-class), [19](#)
- getEdges, [3](#), [24](#), [25](#), [26](#), [42](#)
- getEdges,gGraph-method (getEdges), [26](#)

- getEdges-methods (getEdges), 26
- getGraph, 3
- getGraph (gGraph-class), 28
- getGraph,gData-method (gData-class), 19
- getGraph,gGraph-method (gGraph-class), 28
- getNodeCosts (getCosts), 24
- getNodeCosts,gGraph-method (getCosts), 24
- getNodeCosts-methods (getCosts), 24
- getNodes, 3
- getNodes (gGraph-class), 28
- getNodes,gData-method (gData-class), 19
- getNodes,gGraph-method (gGraph-class), 28
- getNodeAttr, 3, 27
- getNodeAttr,gData-method (getNodeAttr), 27
- getNodeAttr,gGraph-method (getNodeAttr), 27
- getNodeAttr-methods (getNodeAttr), 27
- gGraph, 3, 4, 6–10, 12, 13, 15–28, 30–36, 38–44
- gGraph (gGraph-class), 28
- gGraph-class, 28
- gPath (dijkstra-methods), 13
- gPath2dist (dijkstra-methods), 13
- graph, 25, 40
- graphNEL, 12, 25, 26, 29, 40

- hasCosts, 3
- hasCosts (auxiliary), 5
- hgdp, 30
- hgdpPlus (hgdp), 30

- initialize,gData-method (gData-class), 19
- initialize,gGraph-method (gGraph-class), 28
- is.gData (gData-class), 19
- is.gGraph (gGraph-class), 28
- isConnected,gData-method (connectivity), 11
- isInArea, 3, 31, 37
- isInArea,data.frame-method (isInArea), 31
- isInArea,gData-method (isInArea), 31
- isInArea,gGraph-method (isInArea), 31
- isInArea,matrix-method (isInArea), 31

- isInArea-methods (isInArea), 31
- isReachable (connectivity), 11

- makeGrid, 33

- plot, 4
- plot,gData,missing-method (plot-gData), 34
- plot,gData-method (plot-gData), 34
- plot,gGraph,missing-method (plot-gGraph), 36
- plot,gGraph-method (plot-gGraph), 36
- plot-gData, 34
- plot-gGraph, 36
- plot.gData (plot-gData), 34
- plot.gGraph, 44
- plot.gGraph (plot-gGraph), 36
- plot.gPath (dijkstra-methods), 13
- plotEdges, 4
- plotEdges (plot-gGraph), 36
- points, 4
- points,gData-method (plot-gData), 34
- points,gGraph-method (plot-gGraph), 36
- points.gData (plot-gData), 34
- points.gGraph (plot-gGraph), 36

- rawgraph.10k (worldgraph), 42
- rawgraph.40k (worldgraph), 42
- rdist.earth, 40
- rebuild (auxiliary), 5

- setCosts, 3, 22, 38
- setDistCosts, 40
- setDistCosts,gGraph-method (setDistCosts), 40
- setDistCosts-methods (setDistCosts), 40
- setEdges, 3, 21, 27, 41
- setEdges,gGraph-method (setEdges), 41
- setEdges-methods (setEdges), 41
- show, 3
- show,gData-method (gData-class), 19
- show,gGraph-method (gGraph-class), 28

- worldgraph, 42
- worldgraph.10k, 4
- worldgraph.40k, 4, 30
- worldshape, 4
- worldshape (worldgraph), 42

- zoom, 43