

Package: geoGraph (via r-universe)

October 8, 2026

Type Package

Title Walking Through the Geographic Space Using Graphs

Version 2.0.1

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

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

License MIT + file LICENSE

Language en-US

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 dplyr, fields, RBGL, rnaturalearth, sf, magrittr, dggridR, spdep, tibble, tidyr, terra, FNN, ggplot2

Suggests elevatr, progress, knitr, rmarkdown, rnaturalearthdata, spelling, pastclim, testthat, lintr

Collate 'classes.R' 'accessors.R' 'areConnected.R' 'areNeighbours.R' 'assignByPolygon.R' 'assignByRaster.R' 'basicMethods.R' 'buffer.R' 'closestNode.R' 'combineCosts.R' 'connectivityPlot.R' 'datasets.R' 'dijkstraBetween.R' 'dijkstraBuffer.R' 'dijkstraFrom.R' 'dropDead.R' 'findLand.R' 'gPath2dist.R' 'geo.segments.R' 'geoGraph-deprecated.R' 'geograph-package.R' 'globals.R' 'hasCosts.R' 'interact.R'

'isConnected.R' 'isInArea.R' 'isReachable.R'
 'keepMaxConnected.R' 'makeHexGrid.R' 'makeSquareGrid.R'
 'plot_gData.R' 'plot_gGraph.R' 'plot_gPath.R' 'plot_ggplot.R'
 'polygonBetween.R' 'setCosts.R' 'setDistCosts.R' 'setEdges.R'
 'setNodesAttr.R' 'utils-pipe.R' 'zoom.R' 'zzz.R'

Config/roxygen2/version 8.1.0

Config/pak/sysreqs libabsl-dev cmake libgdal-dev gdal-bin libgeos-dev libicu-dev libssl-dev libproj-dev libsqlite3-dev libudunits2-dev

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

Date/Publication 2026-10-08 18:46:47 UTC

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

RemoteRef main

RemoteSha 4ee04335dda71a64e310564f6719b652b56d4368

Contents

areConnected	3
areNeighbours	4
assignByPolygon	5
assignByRaster	7
autoplot.gData	8
autoplot.gGraph	9
buffer	10
closestNode	11
combineCosts	13
connectivityPlot	15
dijkstraBetween	16
dijkstraBuffer	17
dijkstraFrom	18
dropCosts	19
dropDeadEdges	20
findLand	21
fst_hgdp	22
gData-class	23
geo.add.edges	24
geo.change.attr	25
geo.segments	27
geom_gdata	28
geom_ggraph	29
geom_gpath	30
getColors	31
getCoords	32
getCosts	33
getData	35
getEdges	36
getGraph	37

<i>areConnected</i>	3
getNodes	38
getNodesAttr	39
gGraph-class	40
gPath-class	41
gPath2dist	42
hasCosts	43
hgdp	44
isConnected,gData-method	45
isInArea	46
isReachable	49
keepMaxConnectedSet	50
makeHexGrid	50
makeSquareGrid	51
plot-gData	52
plot-gGraph	55
plot.gPath	58
polygonBetween	59
setColors	60
setCosts	61
setDistCosts	62
setEdges	63
setGraph	65
setNodesAttr	66
subset-gData	67
subset-gGraph	68
worldgraph	69
zoom	70
Index	72

<i>areConnected</i>	<i>Test if a set of nodes form a connected set</i>
---------------------	--

Description

This function tests if a set of nodes form a connected set on a [gGraph](#) object.

Usage

```
areConnected(x, nodes)
```

Arguments

x	a gGraph object
nodes	a vector of node names

Details

@details This function is very similar to `isConnected()`, but it allows the user to specify a subset of nodes to test for connectivity, whereas `isConnected()` tests if all nodes in the object form a connected set. Note that `isConnected()` is a method for both `gGraph` and `gData` objects, whereas `areConnected()` is only a method for `gGraph` objects.

This is an implementation of the `graph::isConnected()` function for data classes in `geoGraph`.

Value

a single logical value, being TRUE if nodes form a connected set.

See Also

Other connectivity_functions: `areNeighbours()`, `isConnected`, `gData-method`, `isReachable()`

Examples

```
# create a small square graph
test_graph <- makeSquareGrid(25, lon.range = c(1, 5), lat.range = c(1, 5))
# test that the function correctly identifies connected sets
# 1, 9, and 10 are connected
areConnected(test_graph, nodes = c("1", "9", "10"))
# even though they are not neighbours
areNeighbours(V1 = "1", V2 = "9", graph = test_graph)
areNeighbours(V1 = "1", V2 = "10", graph = test_graph)
```

areNeighbours	<i>Tests connectivity between pairs of nodes</i>
---------------	--

Description

Tests connectivity between pairs of nodes. This function tests if two nodes are directly connected with an edge (i.e. if they are neighbours).

Usage

```
areNeighbours(V1, V2, graph)
```

Arguments

V1	A vector of node names.
V2	A vector of node names of the same length as V1.
graph	An object of class <code>gGraph</code> or <code>graph::graphNEL</code> (usually in the <code>@graph</code> slot of a <code>gGraph</code> object).

Value

a vector of logical, having one value for each pair of nodes.

See Also

Other connectivity_functions: [areConnected\(\)](#), [isConnected](#), [gData-method](#), [isReachable\(\)](#)

Examples

```
# create a small square graph
test_graph <- makeSquareGrid(25, lon.range = c(1, 5), lat.range = c(1, 5))
# get the coordinates of the first 10 nodes
getCoords(test_graph)[1:10, ]
# test that the function correctly identifies neighbours
# 1 and 2 are neighbours, but 1 and 9 are not
areNeighbours(V1 = c("1", "1"), V2 = c("2", "9"), graph = test_graph)
```

assignByPolygon

Assign node attributes from a polygon layer

Description

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

Usage

```
assignByPolygon(x, ...)

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

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

## S4 method for signature 'numeric'
assignByPolygon(x, layer = "world", attr = "all", ...)

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

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

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

Arguments

x a matrix, data.frame, list, valid [gGraph](#), or valid [gData](#) object. For matrix and data.frame, input must have two columns giving longitudes and latitudes. For list, input must have two components being vectors of longitudes and latitudes.

...	further arguments passed to other methods (currently unused).
layer	a shapefile of class <code>sf</code> (see <code>sf::st_read</code> to import a GIS shapefile). Alternatively, "world" to use the built-in world shapefile.
attr	a character vector of variable names to extract from the layer. Use "all" to extract all available variables.

Details

Nodes can be specified as a matrix, `data.frame`, list, `gGraph`, or `gData` object. Outputs match the input format.

The `gGraph` method can be memory-intensive for large graphs since it assigns attributes to all nodes.

Value

For matrix, `data.frame`, or list input: a `data.frame` with one row per location and one column per requested variable. For `gGraph` input: a `gGraph` object with new node attributes added to `@nodes.attr`. For `gData` input: a `gData` object with new data added to `@data`.

Functions

- `assignByPolygon(matrix)`: Method for matrix input
- `assignByPolygon(data.frame)`: Method for `data.frames` input
- `assignByPolygon(numeric)`: Method for numeric vector input
- `assignByPolygon(list)`: Method for numeric list input
- `assignByPolygon(gGraph)`: Method for `gGraph` objects
- `assignByPolygon(gData)`: Method for `gData` objects

Note

The `gGraph` method should be carefully used, output is going to be heavy.

See Also

`findLand` to find which locations are on land. `assignByRaster` to assign attributes from raster data.

Examples

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

## retrieve continent info for all nodes
## (might take a few seconds)
x <- assignByPolygon(worldgraph.10k, layer = "world", 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)
```

assignByRaster

Assign raster values to graph nodes

Description

The function `assignByRaster` takes a [gGraph](#) object and a `SpatRaster`, assigns each raster cell to the nearest graph node, and collapses the values to a single scalar per node using a summary function.

Usage

```
assignByRaster(
  graph,
  raster,
  layer.name = "raster_points",
  fun = "mean",
  na.rm = TRUE,
  ...
)
```

Arguments

<code>graph</code>	a gGraph object.
<code>raster</code>	a <code>SpatRaster</code> object (from the <code>terra</code> package).
<code>layer.name</code>	a character string giving the name of the new node attribute. Defaults to <code>"raster_points"</code> .
<code>fun</code>	a function or character string specifying how to summarize raster values within each node. Built-in options are <code>"mean"</code> , <code>"max"</code> , <code>"min"</code> , <code>"median"</code> , <code>"sd"</code> , <code>"any"</code> , and <code>"all"</code> . Alternatively, pass any function that takes a numeric vector and returns a scalar. Defaults to <code>"mean"</code> .
<code>na.rm</code>	logical. Whether to remove NA values before summarizing. Defaults to <code>TRUE</code> .
<code>...</code>	additional arguments passed to <code>fun</code> .

Details

This function is memory-intensive for high-resolution rasters or large graphs. Once values are assigned, further node attributes can be derived using [setNodesAttr](#).

Value

A [gGraph](#) object with a new scalar node attribute named `layer.name`.

See Also

[setNodesAttr](#) to set node attributes manually. [assignByPolygon](#) to assign attributes from GIS shapefiles.

Examples

```
## Make a new gGraph without any nodes attribute
geo.box <- c(xmin = -10, xmax = 30, ymin = 35, ymax = 60)
ggraph <- makeHexGrid(geo.box, spacing = 1000)

## Create a synthetic raster of random elevation values over the same region.
set.seed(42)
r <- terra::rast(
  xmin = geo.box["xmin"], xmax = geo.box["xmax"],
  ymin = geo.box["ymin"], ymax = geo.box["ymax"],
  resolution = 5, crs = "EPSG:4326"
)
terra::values(r) <- runif(terra::ncell(r))

## assign mean raster value per node
ggraph <- assignByRaster(ggraph, r, layer.name = "elevation", fun = "mean")

## assign standard deviation per node (useful for ruggedness)
ggraph <- assignByRaster(ggraph, r, layer.name = "ruggedness", fun = "sd")
```

autoplot.gData

Default ggplot for a [gData](#)

Description

Default ggplot for a [gData](#)

Usage

```
## S3 method for class 'gData'
autoplot(object, show.gGraph = TRUE, edges = FALSE, ...)
```

Arguments

object	a gData .
show.gGraph	logical; overlay the linked gGraph if available.
edges	logical; whether to draw edges.
...	unused.

Value

a ggplot.

See Also

Other ggplot_methods: [autoplot.gGraph\(\)](#), [geom_gdata\(\)](#), [geom_ggraph\(\)](#), [geom_gpath\(\)](#)

Examples

```
autoplot(hgdp)
```

autoplot.gGraph	<i>Default ggplot for a gGraph</i>
-----------------	--

Description

Default ggplot for a [gGraph](#)

Usage

```
## S3 method for class 'gGraph'  
autoplot(object, edges = TRUE, ...)
```

Arguments

object	a gGraph .
edges	logical; whether to draw edges.
...	unused.

Value

a ggplot.

See Also

Other ggplot_methods: [autoplot.gData\(\)](#), [geom_gdata\(\)](#), [geom_ggraph\(\)](#), [geom_gpath\(\)](#)

Examples

```
autoplot(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 ####
## Subset gGraph to Europe
x <- rawgraph.10k[isInArea(worldgraph.10k, reg = list(x = c(-10, 50), y = c(35, 70)), quiet = TRUE)]

## identify one node
node <- closestNode(x, data.frame(lon = 12, lat = 50))

## find a buffer
buffer(x, node, 1000)
buf500km <- buffer(x, node, 1000, res.type = "gGraph")
plot(buf500km, col.rules = buf500km@meta$buf.colors, reset = TRUE)

#### gData example ####

## retain a subset of hgdp
x <- hgdp[27:30]
plot(x, reset = TRUE, col.g = "lightgrey", pch.node = 20)
buf.400 <- buffer(x, 400, res.type = "gData")
points(buf.400, col.node = "gold")
```

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,
  method = "knn",
  zoneSize = 5,
  attr.name = NULL,
  attr.values = NULL
)
```

```
## S4 method for signature 'gData'
closestNode(
  x,
```

```

    method = "knn",
    zoneSize = 5,
    attr.name = NULL,
    attr.values = NULL
  )

```

Arguments

<code>x</code>	a valid <code>gGraph</code> or <code>gData</code> object. In the latter case, the <code>gGraph</code> to which the <code>gData</code> 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 the locations interactively.
<code>method</code>	the method to use for finding the closest node. The default is "knn", which uses a fast FNN-based spatial index over unit-sphere Cartesian coordinates. The legacy "inArea" method is also available, but it is slower.
<code>zoneSize</code>	The initial size of the search area (in degrees). Only needs to be specified if <code>method = "inArea"</code> . The search zone will be expanded until at least 3 candidate 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.name</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.name` 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`

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, , drop = FALSE], 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, type = "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

x1	The first gGraph (which will be used as a template to build the combined gGraph)
x2	The second gGraph from which costs will be combined

method	a character string indicating which method should be used to combined edge cost from the two gGraph. 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 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, based on the combination of the two gGraph objects, used as weightings of edges.

See Also

[setCosts](#) to set costs for a single gGraph object

Other cost_functions: [dropCosts\(\)](#), [getCosts\(\)](#), [hasCosts\(\)](#), [setCosts\(\)](#), [setDistCosts\(\)](#)

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")
```

connectivityPlot	<i>Plot connected sets of a gGraph object</i>
------------------	---

Description

The function `connectivityPlot` plots the connected sets of a `gGraph` or `gData` object with different colors. Isolated nodes (i.e. belonging to no connected set of size > 1) are plotted in light gray.

Usage

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

Arguments

<code>x</code>	a valid <code>gGraph</code> or <code>gData</code> object.
<code>...</code>	other arguments passed to other methods.
<code>seed</code>	an optional integer giving the seed to be used when randomizing colors. A given seed will always produce the same set of colors. NULL by default, meaning colors are randomized each time a plot is drawn.

Value

A named character vector of colors, one per node, returned invisibly.

Functions

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

Examples

```
# plot connected sets of a gGraph object  
connectivityPlot(worldgraph.10k)  
  
# plot connected sets of a gData object  
connectivityPlot(hgdp)
```

dijkstraBetween	<i>Find the shortest path between nodes in a graph</i>
-----------------	--

Description

This function finds the shortest path between nodes in a graph using Dijkstra's algorithm. It can be applied to both `gGraph` and `gData` objects.

Usage

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

Arguments

<code>x</code>	A gGraph or gData object.
<code>...</code>	Additional arguments passed to other methods (currently not used).
<code>from</code>	A character vector of starting node IDs.
<code>to</code>	A character vector of ending node IDs.

Details

The function uses the `RBGL` package to compute the shortest paths. It checks for the connectivity of the graph and handles cases where there are duplicated paths.

Value

A [gPath](#) object.

Functions

- `dijkstraBetween(gGraph)`: Method for `gGraph`
- `dijkstraBetween(gData)`: Method for `gData`

See Also

Other `dijkstra_methods`: [dijkstraBuffer\(\)](#), [dijkstraFrom\(\)](#), [gPath2dist\(\)](#)

Examples

```
## select a few populations from the HGDP dataset
hgdp.sub <- hgdp[getData(hgdp)$Population %in%
  c("French", "Balochi", "BantuKenya", "Papuan", "Pima")]
hgdp.path <- dijkstraBetween(hgdp.sub) # compute shortest path
```

dijkstraBuffer	<i>Find nodes reachable within a cost threshold</i>
----------------	---

Description

The function `dijkstraBuffer` identifies all nodes reachable from a given origin within a specified maximum cumulative cost, where the distance is defined as the cumulative least-cost path over weighted edges.

Usage

```
dijkstraBuffer(x, origin, d, res.type = c("nodes", "gGraph"), ...)
```

Arguments

<code>x</code>	a valid gGraph object with edge costs defined (see setCosts).
<code>origin</code>	either a character string naming a node in <code>x</code> , or a <code>data.frame</code> , list, or numeric vector of length 2 giving longitude and latitude of the origin location.
<code>d</code>	numeric. Maximum cumulative cost defining the reachable area.
<code>res.type</code>	a character string indicating the output format: • "nodes": a character vector of reachable node names (default). • "gGraph": the input gGraph with a new logical node attribute <code>reachable</code> indicating which nodes fall within the cost threshold.
<code>...</code>	further arguments passed to other methods (currently unused).

Details

Internally, the function computes single-source shortest paths using Dijkstra's algorithm from the origin node and returns all nodes whose minimum path cost does not exceed the specified threshold.

Value

A character vector of reachable node names when `res.type = "nodes"`, or a [gGraph](#) object with a new logical node attribute `reachable` when `res.type = "gGraph"`.

See Also

[dijkstraFrom](#) and [gPath2dist](#) for the underlying path computations. [buffer](#) for geographic distance-based buffers. [setCosts](#) to define edge costs before running `dijkstraBuffer`.

Other `dijkstra_` methods: [dijkstraBetween\(\)](#), [dijkstraFrom\(\)](#), [gPath2dist\(\)](#)

Examples

```
## Subset gGraph to Europe
x <- rawgraph.10k[isInArea(worldgraph.10k, reg = list(x = c(-10, 50), y = c(35, 70)), quiet = TRUE)]

## Get all nodes reachable within cost 10 from node the origin (here Zurich)
zurich <- data.frame(lon = 8.55, lat = 47.37)

## Set costs for the graph and calculate the buffer
graph <- setCosts(x, attr.name = "habitat", method = "mean")
x2 <- dijkstraBuffer(graph, origin = zurich, d = 10, res.type = "gGraph")

## Plot reachable nodes in dark blue, all others transparent
col.rules <- data.frame(
  reachable = c(TRUE, FALSE),
  color      = c("darkblue", "transparent")
)
plot(x2, col.rules = col.rules, reset = TRUE)
```

dijkstraFrom

Find the minimum cost path

Description

This function finds the shortest path from a given 'source' node to all other nodes in the graph using Dijkstra's algorithm. It can be applied to both `gGraph` and `gData` objects.

Usage

```
dijkstraFrom(x, start)

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

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

Arguments

<code>x</code>	A <code>gGraph</code> or <code>gData</code> object.
<code>start</code>	a character string naming the 'source' node.

Details

The function uses the `RBGL` package to compute the shortest paths. It checks for the connectivity of the graph and handles cases where there are duplicated paths.

Value

A [gPath](#) object.

Functions

- `dijkstraFrom(gGraph)`: method for `gGraph`
- `dijkstraFrom(gData)`: method for `gData`

See Also

Other `dijkstra_methods`: [dijkstraBetween\(\)](#), [dijkstraBuffer\(\)](#), [gPath2dist\(\)](#)

Examples

```
# Using a gData object:

# select a few populations from the HGDP dataset
hgdp.sub <- hgdp[getData(hgdp)$Population %in%
  c("Orcadian", "Adygei", "Russian", "Basque")]

# select a location of another HGDP population
french.hgdp <- hgdp[getData(hgdp)$Population %in%
  c("French")]

# Choose an origin node
french <- getNodes(french.hgdp)

my.path <- dijkstraFrom(hgdp.sub, french)
```

dropCosts

Remove all costs from a gGraph object

Description

The function `dropCosts` removes all edge weights (costs) from a [gGraph](#) object, returning an un-weighted graph.

Usage

```
dropCosts(x, ...)
```

S4 method for signature 'gGraph'

```
dropCosts(x)
```

Arguments

`x` a valid [gGraph](#) object.

`...` additional arguments passed to other methods (currently unused).

Value

A [gGraph](#) object with all edge costs removed.

Functions

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

See Also

[getCosts](#) to retrieve edge costs, [setCosts](#) to set edge costs. [hasCosts](#) to check if a graph has costs defined.

Other cost_functions: [combineCosts\(\)](#), [getCosts\(\)](#), [hasCosts\(\)](#), [setCosts\(\)](#), [setDistCosts\(\)](#)

Examples

```
hasCosts(rawgraph.10k)
x <- dropCosts(worldgraph.10k)
hasCosts(x)
```

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)
```

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. Strictly 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>sf</code> (see sf::st_read() to import a GIS shapefile). Alternatively, a character string indicating one shapefile released with <code>geoGraph</code> ; currently, only 'world' is available.
<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

[assignByPolygon](#), 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$colors <- temp

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

fst_hgdp

Pairwise FST matrix for HGDP populations

Description

A matrix of pairwise F_{ST} values between the HGDP populations, used in the `geoGraph` case studies to relate genetic differentiation to geographic and least-cost distances.

Format

A numeric matrix with one row and one column per population. Row and column names are population identifiers matching those in [hgdp](#).

Examples

```
data(fst_hgdp)
dim(fst_hgdp)
fst_hgdp[1:5, 1:5]
```

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

<code>x</code>	a valid gGraph object.
<code>mode</code>	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.
<code>refObj</code>	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 <code>refObj</code> are added to <code>x</code> . 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)
```

 geo.segments

Plot segments correctly when crossing the antimeridian

Description

A substitute to `segments` which correctly draws segments between locations distant by more than 90 degrees of longitude (i.e. from one hemisphere to the other). It is used instead of `segments`, but it is slower.

Usage

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

Arguments

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

Details

This low-level function is designed to be called by other procedures of [geoGraph](#). However, it can sometimes be useful by itself. Note that unlike other functions in `geoGraph`, this functions does not test for the validity of the provided arguments (for speed purposes).

Value

NULL.

Examples

```
plot(worldgraph.10k, reset = TRUE)
geo.segments(x0 = -170, y0 = 60, x1 = 170, y1 = 55, col = "red", lwd = 2)
```

geom_gdata	<i>ggplot layer for a gData</i>
------------	---

Description

Adds sample localities of a [gData](#) to a ggplot as a `ggplot2::geom_sf()` layer.

Usage

```
geom_gdata(
  data = NULL,
  mapping = ggplot2::aes(),
  original = FALSE,
  stat = "sf",
  position = "identity",
  na.rm = FALSE,
  show.legend = NA,
  ...
)
```

Arguments

data	a gData object.
mapping	aesthetics.
original	logical. If TRUE, plot at the original sample locations; if FALSE (default), plot at the assigned-node coordinates on the linked gGraph .
stat, position, na.rm, show.legend, ...	forwarded to ggplot2::geom_sf() .

Value

a ggplot layer.

See Also

Other ggplot_methods: [autoplot.gData\(\)](#), [autoplot.gGraph\(\)](#), [geom_ggraph\(\)](#), [geom_gpath\(\)](#)

Examples

```
library(ggplot2)
ggplot() +
  geom_gdata(data = hgdp, color = "black", size = 1.5) +
  theme_void()
```

geom_ggraph

ggplot layer for a [gGraph](#)

Description

Adds nodes (and optionally edges) of a [gGraph](#) to a ggplot as `ggplot2::geom_sf()` layers.

Usage

```
geom_ggraph(  
  data = NULL,  
  mapping = ggplot2::aes(),  
  edges = FALSE,  
  stat = "sf",  
  position = "identity",  
  na.rm = FALSE,  
  show.legend = NA,  
  ...  
)
```

Arguments

data a [gGraph](#) object.

mapping aesthetics. Variables from the [gGraph](#)'s node attributes may be used (e.g. `aes(color = habitat)`).

edges logical; whether to draw edges. Defaults to FALSE.

stat, position, na.rm, show.legend, ...
forwarded to `ggplot2::geom_sf()`.

Value

one or two ggplot layers (edges under nodes).

See Also

Other ggplot_methods: [autoplot.gData\(\)](#), [autoplot.gGraph\(\)](#), [geom_gdata\(\)](#), [geom_gpath\(\)](#)

Examples

```
library(ggplot2)  
ggplot() +  
  geom_ggraph(data = worldgraph.10k, aes(color = habitat), edges = TRUE, size = 0.3) +  
  scale_color_manual(values = c(land = "grey70", sea = "lightblue", coast = "grey70")) +  
  coord_sf(crs = "+proj=ortho +lat_0=40 +lon_0=-80") +  
  theme_void()
```

geom_gpath	<i>ggplot layer for a gPath</i>
------------	---------------------------------

Description

Adds a [gPath](#) (from [dijkstraBetween\(\)](#) or [dijkstraFrom\(\)](#)) to a ggplot as [ggplot2::geom_sf\(\)](#) linestrings.

Usage

```
geom_gpath(
  data = NULL,
  mapping = ggplot2::aes(),
  stat = "sf",
  position = "identity",
  na.rm = FALSE,
  show.legend = NA,
  ...
)
```

Arguments

`data` a [gPath](#) object.
`mapping` aesthetics.
`stat`, `position`, `na.rm`, `show.legend`, ...
 forwarded to [ggplot2::geom_sf\(\)](#).

Value

a ggplot layer.

See Also

Other ggplot_methods: [autoplot.gData\(\)](#), [autoplot.gGraph\(\)](#), [geom_gdata\(\)](#), [geom_ggraph\(\)](#)

Examples

```
library(ggplot2)
addis <- list(lon = 38.74, lat = 9.03)
addis_node <- closestNode(worldgraph.40k, addis)
myPath <- dijkstraFrom(hgdp, addis_node)
ggplot() +
  geom_ggraph(
    data = worldgraph.40k, aes(color = habitat),
    edges = FALSE, size = 1, show.legend = FALSE
  ) +
  scale_color_manual(values = c(land = "grey70", sea = "lightblue", coast = "grey70")) +
  geom_gpath(data = myPath, color = "firebrick", linewidth = 0.4) +
```

```
geom_gdata(data = hgdp, color = "black", size = 1.5) +
coord_sf(crs = "+proj=ortho +lat_0=40 +lon_0=30") +
theme_void()
```

getColors

*Get colors associated to nodes of a gGraph object***Description**

The function `getColors` returns either the color rules stored in a [gGraph](#) object (`res.type = "rules"`) or a vector of colors for each node based on a specified node attribute (`res.type = "colors"`).

Usage

```
getColors(x, ...)

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

Arguments

<code>x</code>	a valid gGraph object.
<code>...</code>	other arguments passed to other methods (currently unused).
<code>nodes</code>	a vector of node names or indices, or "all" for all nodes (default). Only used when <code>res.type = "colors"</code> .
<code>attr.name</code>	a character string giving the name of the node attribute to use for color assignment. Required when <code>res.type = "colors"</code> .
<code>col.rules</code>	a two-column data.frame mapping attribute values to colors. If NULL, uses <code>x@meta\$colors</code> . Only used when <code>res.type = "colors"</code> .
<code>res.type</code>	a character string indicating the output type: "colors": a named character vector of colors, one per node. "rules": the color rules data.frame stored in <code>x@meta\$colors</code>.

Details

Color rules are stored as a two-column data.frame in `x@meta$colors`. The first column is named after the node attribute and contains its possible values; the second column is named "color" and contains valid R color strings.

Value

A named character vector of colors when `res.type = "colors"`, or a `data.frame` of color rules when `res.type = "rules"`.

Functions

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

See Also

[setColors](#) to set color rules. [getNodesAttr](#) to retrieve node attributes.

Other accessor_methods: [getCoords\(\)](#), [getCosts\(\)](#), [getData\(\)](#), [getEdges\(\)](#), [getGraph\(\)](#), [getNodes\(\)](#), [getNodesAttr\(\)](#), [setColors\(\)](#), [setGraph\(\)](#)

Examples

```
## get color rules
getColors(worldgraph.10k, res.type = "rules")

## get node colors based on habitat attribute
head(getColors(worldgraph.10k, attr.name = "habitat"))
```

getCoords

Get coordinates of nodes in a gGraph or gData object

Description

The function `getCoords` returns the coordinates (longitude and latitude) of nodes in a [gGraph](#) or [gData](#) object.

Usage

```
getCoords(x, ...)

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

## S4 method for signature 'gData'
getCoords(x, original = TRUE, ...)
```

Arguments

<code>x</code>	a valid gGraph or gData object.
<code>...</code>	additional arguments passed to other methods (currently unused).
<code>original</code>	logical. If TRUE (default), returns the original location coordinates; if FALSE, returns the coordinates of the matched nodes on the gGraph grid.

Value

A matrix with two columns, lon and lat, giving the longitude and latitude of each node or location.

Functions

- getCoords(gGraph): Method for gGraph objects
- getCoords(gData): Method for gData objects

See Also

[getNodeIds](#), [getGraph](#)

Other accessor_methods: [getColors\(\)](#), [getCosts\(\)](#), [getData\(\)](#), [getEdges\(\)](#), [getGraph\(\)](#), [getNodeIds\(\)](#), [getNodeIdsAttr\(\)](#), [setColors\(\)](#), [setGraph\(\)](#)

Examples

```
head(getCoords(worldgraph.10k))

## for a gData object
getCoords(hgdp)

## coordinates of matched grid nodes instead of original locations
getCoords(hgdp, original = FALSE)
```

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 in different formats. getNodeCosts returns the costs associated to nodes based on a node attribute and the cost rules stored in @meta\$costs.

Usage

```
getCosts(x, ...)

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

getNodeCosts(x, ...)

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

Arguments

<code>x</code>	a valid gGraph object.
<code>...</code>	other arguments passed to other methods (currently unused).
<code>res.type</code>	a character string indicating the output format: • <code>"asIs"</code>: a named list of weights for each node's edges (default). • <code>"vector"</code>: a named numeric vector of all edge weights. • <code>"rules"</code>: the cost rules data.frame stored in <code>x@meta\$costs</code>, with one row per node attribute value and a column named <code>"cost"</code>.
<code>unique</code>	logical. If TRUE, only unique edge weights are returned. Defaults to FALSE. Only used when <code>res.type</code> is <code>"asIs"</code> or <code>"vector"</code> .
<code>attr.name</code>	character string. The name of the node attribute used to look up costs from <code>x@meta\$costs</code> (for <code>getNodeCosts</code> only).

Details

In `geoGraph`, costs are equivalent to weights in the `graph` package: the larger the cost of an edge, the lower the connectivity between its two nodes.

Value

For `getCosts`:

- `"asIs"`: a named list of edge weights, one element per node.
- `"vector"`: a named numeric vector of edge weights.
- `"rules"`: a data.frame of cost rules from `x@meta$costs`.

For `getNodeCosts`: a numeric vector of costs, one per node.

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

[setCosts](#) to set edge costs. [dropCosts](#) to remove all costs. [hasCosts](#) to check if a graph has costs defined.

Other accessor_methods: [getColors\(\)](#), [getCoords\(\)](#), [getData\(\)](#), [getEdges\(\)](#), [getGraph\(\)](#), [getNodes\(\)](#), [getNodesAttr\(\)](#), [setColors\(\)](#), [setGraph\(\)](#)

Other cost_functions: [combineCosts\(\)](#), [dropCosts\(\)](#), [hasCosts\(\)](#), [setCosts\(\)](#), [setDistCosts\(\)](#)

Examples

```
## get edge costs as a vector
head(getCosts(worldgraph.10k, res.type = "vector", unique = TRUE))

## get cost rules
getCosts(worldgraph.10k, res.type = "rules")

## get node costs based on habitat attribute
head(getNodeCosts(worldgraph.10k, attr.name = "habitat"))
```

getData

Get the data component of a gData object

Description

The function `getData` returns the data stored in the `@data` slot of a [gData](#) object.

Usage

```
getData(x, ...)

## S4 method for signature 'gData'
getData(x, ...)
```

Arguments

`x` a valid [gData](#) object.

`...` additional arguments passed to other methods (currently unused).

Value

The data stored in the `@data` slot of the input object, typically a `data.frame` where each row corresponds to a sampled location.

Functions

- `getData(gData)`: Method for `gData` objects

See Also

[getCoords](#) to retrieve coordinates. [getNodes](#) to retrieve matched node identifiers. [getNodesAttr](#) to retrieve node attributes from the underlying [gGraph](#).

Other accessor_methods: [getColors\(\)](#), [getCoords\(\)](#), [getCosts\(\)](#), [getEdges\(\)](#), [getGraph\(\)](#), [getNodes\(\)](#), [getNodesAttr\(\)](#), [setColors\(\)](#), [setGraph\(\)](#)

Examples

```
## get the data stored in the hgdp dataset
head(getData(hgdp))
```

getEdges	<i>Get edges from a gGraph object</i>
----------	---------------------------------------

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 [graph: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

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

Other accessor_methods: [getColors\(\)](#), [getCoords\(\)](#), [getCosts\(\)](#), [getData\(\)](#), [getGraph\(\)](#), [getNodes\(\)](#), [getNodesAttr\(\)](#), [setColors\(\)](#), [setGraph\(\)](#)

Examples

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

getGraph

Get the graph component of a gGraph or gData object

Description

The function `getGraph` returns the `graph:graphNEL` object stored in a `gGraph` or `gData` object.

Usage

```
getGraph(x, ...)

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

## S4 method for signature 'gData'
getGraph(x, ...)
```

Arguments

`x` a valid `gGraph` or `gData` object.
`...` additional arguments passed to other methods (currently unused).

Value

A `graph:graphNEL` object.

Functions

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

See Also

[getNodes](#), [getEdges](#), [getCoords](#)

Other accessor_methods: [getColors\(\)](#), [getCoords\(\)](#), [getCosts\(\)](#), [getData\(\)](#), [getEdges\(\)](#), [getNodes\(\)](#), [getNodesAttr\(\)](#), [setColors\(\)](#), [setGraph\(\)](#)

Examples

```
getGraph(worldgraph.10k)
```

getNodes

Get nodes of a gGraph or gData object

Description

The function `getNodes` returns the names of nodes in a [gGraph](#) or [gData](#) object.

Usage

```
getNodes(x, ...)

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

## S4 method for signature 'gData'
getNodes(x, ...)
```

Arguments

`x` a valid [gGraph](#) or [gData](#) object.
`...` additional arguments passed to other methods (currently unused).

Value

A character vector of node names.

Functions

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

See Also

[getGraph](#), [getCoords](#), [getNodesAttr](#)

Other accessor_methods: [getColors\(\)](#), [getCoords\(\)](#), [getCosts\(\)](#), [getData\(\)](#), [getEdges\(\)](#), [getGraph\(\)](#), [getNodesAttr\(\)](#), [setColors\(\)](#), [setGraph\(\)](#)

Examples

```
head(getNodes(worldgraph.10k))
getNodes(hgdp)
```

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 gGraph or gData 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

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

See Also

[getNodes](#), [getEdges](#), [getCoords](#)

Other accessor_methods: [getColors\(\)](#), [getCoords\(\)](#), [getCosts\(\)](#), [getData\(\)](#), [getEdges\(\)](#), [getGraph\(\)](#), [getNodes\(\)](#), [setColors\(\)](#), [setGraph\(\)](#)

Examples

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

```
## gData method
getNodesAttr(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 [graph::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: [graph::graphNEL](#): 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")
```

gPath-class

Formal class "gPath"

Description

The class gPath is an S3 class storing the results of shortest path computations between nodes in a [gGraph](#) object. Paths are computed using Dijkstra's algorithm via the RBGL package and represent the minimum-cost routes connecting pairs of nodes in the graph.

Details

gPath objects are primarily created as outputs from [dijkstraFrom](#), [dijkstraBetween](#), and [polygonBetween](#) applied to [gGraph](#) or [gData](#) objects. The structure is based on the output from RBGL's `sp.between` function, enhanced with geographic coordinate information.

Creating gPath objects

gPath objects are created by dijkstra methods. Direct construction is not recommended.

Structure

A named list where each element represents a path between two nodes, containing:

- path_detail: a character vector of node names from source to destination.
- length: a numeric value giving the total cost of the path.

An xy attribute stores a matrix of spatial coordinates (longitude and latitude) for all nodes referenced in the paths, with node identifiers as row names.

See Also

[dijkstraFrom](#), [dijkstraBetween](#) and [polygonBetween](#) to create gPath objects. [plot.gPath](#) to visualize paths. [gPath2dist](#) to extract distances. [gGraph](#) and [gData](#) for related classes.

Examples

```
ori <- closestNode(worldgraph.40k, cbind(33, 10))
myPath <- dijkstraFrom(hgdp, ori)

## examine the structure
length(myPath) # number of paths
myPath[[1]]$path_detail # nodes in first path
myPath[[1]]$length # cost of first path
myPath[[1]]$length_detail # details for each step

## get coordinates of nodes in paths
head(attr(myPath, "xy"))

## plot the paths
plot(worldgraph.40k, col = NA, reset = TRUE)
plot(myPath)
```

gPath2dist

Extract distances from a gPath object

Description

This function extracts distances from a gPath object returned by [dijkstraBetween\(\)](#) or [dijkstraFrom\(\)](#). Depending on res.type, it returns either a [dist](#) object or a numeric vector of distances.

Usage

```
gPath2dist(m, diag = FALSE, upper = FALSE, res.type = NULL)
```

Arguments

<code>m</code>	a <code>gPath</code> object obtained by <code>dijkstraBetween()</code> or <code>dijkstraFrom()</code> .
<code>diag</code>	unused parameter added for consistency with <code>as.dist()</code> .
<code>upper</code>	unused parameter added for consistency with <code>as.dist()</code> .
<code>res.type</code>	deprecated parameter that is now ignored; the function automatically detects whether the input is from <code>dijkstraBetween()</code> or <code>dijkstraFrom()</code> and returns the appropriate output type.

Value

Either a `dist` object containing pairwise distances between nodes the `gPath` object was constructed from `dijkstraBetween()`, or a numeric vector of distances if the `gPath` object was constructed from `dijkstraFrom()`.

See Also

Other `dijkstra_methods`: `dijkstraBetween()`, `dijkstraBuffer()`, `dijkstraFrom()`

Examples

```
## for pairwise distances between multiple a "dist" object is returned
# select a few populations from the HGDP dataset
hgdp.sub <- hgdp[getData(hgdp)$Population %in%
  c("Balochi", "BantuKenya", "Papuan", "Pima")]
hgdp.path <- dijkstraBetween(hgdp.sub) # compute shortest path
gPath2dist(hgdp.path) # extract as dist object
## for distances from a single origin node to multiple the output is a vector of distances
# # choose an origin node
start <- "24988"
hgdp.path <- dijkstraFrom(hgdp.sub, start) # compute shortest path from origin
gPath2dist(hgdp.path) # extract as vector of distances
```

hasCosts

Check if a gGraph has costs

Description

This function tests whether a `gGraph` has heterogeneous costs associated to its edges.

Usage

```
hasCosts(x)
```

Arguments

`x` a valid `gGraph` object.

Details

This low-level function is designed to be called by other procedures of [geoGraph](#). However, it can sometimes be useful by itself. Note that a [gGraph](#) with all costs set uniform will also result in FALSE. Unlike other functions in [geoGraph](#), this function does not test for the validity of the provided arguments (for speed purposes).

Value

TRUE if the graph has heterogeneous edge costs, FALSE if costs are absent or uniform.

See Also

Other cost_functions: [combineCosts\(\)](#), [dropCosts\(\)](#), [getCosts\(\)](#), [setCosts\(\)](#), [setDistCosts\(\)](#)

Examples

```
hasCosts(rawgraph.10k) # TRUE
hasCosts(worldgraph.10k) # FALSE as not all edges have costs
```

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 [gData](#) object with the following slots:

@coords Coordinates (lon, lat) of each of the 51 population sampling locations.

@nodes.id Node identifiers of the underlying [gGraph](#) ([worldgraph.40k](#)) matched to each population.

@data Metadata associated with the populations, including population name, region, and genetic.div

Details

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

hgdpPlus describes hgdp populations plus 24 native American populations.

References

Cann, H. M. et al. (2002) A human genome diversity cell line panel. *Science* **296**: 261-262.
[doi:10.1126/science.296.5566.261b](https://doi.org/10.1126/science.296.5566.261b)

Examples

```
## check object
hgdp

## plotting the object
plot(hgdp)

## results from Handley et al.
## 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 = NA)
points(hgdp)
points(worldgraph.40k[addis], psize = 3, pch = "x", col = "black")
plot(myPath)

## correlations distance/genetic div.
geo.dist <- gPath2dist(myPath)
gen.div <- getData(hgdp)$Genetic.Div
plot(gen.div ~ geo.dist)
abline(lm(gen.div ~ geo.dist), col = "blue")
summary(lm(gen.div ~ geo.dist))
```

isConnected,gData-method

Test if a set of nodes form a connected set

Description

This function tests if a set of nodes form a connected set on a [gData](#) or [gGraph](#) object.

Usage

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

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

Arguments

object a [gData](#) or [gGraph](#) object
 ... other arguments passed to other methods.

Details

This function is a method for both [gData](#) and [gGraph](#) objects. For a [gData](#) object, it tests if all the nodes in the object form a connected set on the associated [gGraph](#) object. For a [gGraph](#) object, it tests if all nodes in the graph form a connected set. Use [areConnected\(\)](#) if you want to test if a specific subset of nodes form a connected set.

Value

a single logical value, being TRUE if nodes form a connected set.

See Also

Other connectivity_functions: [areConnected\(\)](#), [areNeighbours\(\)](#), [isReachable\(\)](#)

Examples

```
# in the world graph, there are nodes that are not connected
isConnected(worldgraph.10k)
# all the nodes in the hgdp gData object are connected
isConnected(hgdp)
```

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 formats for the output are also available. The area can be defined interactively (current plot window), from the zoom log, or explicitly by providing a bounding box.

Usage

```
isInArea(x, ...)

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

```

)

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

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

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

```

Arguments

<code>x</code>	a matrix, <code>data.frame</code> , valid gGraph , or valid gData object. For matrix and <code>data.frame</code> , 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. Character strings can be "current" (current user window, default) or "zoom" (current zoom). If a list, it must have two components, both being numeric vectors of length two, giving x and y limits of the area, e.g. <code>list(x = c(-10, 30), y = c(35, 70))</code> . A list produced by <code>locator()</code> is also a valid value.
<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.
<code>quiet</code>	logical. If TRUE, suppresses printing of bounding box coordinates and a reproducible <code>reg</code> argument that can be copy-pasted into scripts for exact reproducibility. Defaults to FALSE.

Value

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

- "logical": a vector of logicals having one value for each node.
- "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

See Also

[geo.zoomin](#) to zoom into an area. [gGraph](#) and [gData](#) for the input object classes.

Examples

```
## Zoom into Europe and get the nodes in the current plot
plot(worldgraph.10k, reset = TRUE)
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))

## Different output formats of the current nodes
head(isInArea(worldgraph.10k, quiet = TRUE))
head(isInArea(worldgraph.10k, res.type = "integer", quiet = TRUE))
head(isInArea(worldgraph.10k, res.type = "character", quiet = TRUE))

## subset the gGraph just to visible nodes
x <- worldgraph.10k[isInArea(worldgraph.10k)]
plot(x, reset = TRUE)

## Instead of the current plotted area we can use an explicit bounding box
y <- worldgraph.10k[(isInArea(worldgraph.10k,
  reg = list(x = c(113, 154), y = c(-44, -10)),
  quiet = TRUE
))]
plot(y, reset = TRUE)
```

isReachable	<i>Tests if location reachable from nodes</i>
-------------	---

Description

Tests if one location (actually, the closest node to it) is reachable from the set of nodes of a [gData](#) object.

Usage

```
isReachable(x, loc)
```

Arguments

x	a gData object.
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.

Value

a named boolean vector, with one element per node of the input [gData](#) object, and names corresponding to node names.

See Also

Other connectivity_functions: [areConnected\(\)](#), [areNeighbours\(\)](#), [isConnected](#), [gData-method](#)

Examples

```
# Select African populations Mandenka, Yoruba, and San
hgdp.sub <- hgdp[getData(hgdp)$Population %in%
  c("Mandenka", "Yoruba", "San")]
# Get a location that is reachable
location <- getCoords(hgdp[getData(hgdp)$Population == "BantuKenya"])
# Check these are reachable
isReachable(x = hgdp.sub, loc = location)
```

keepMaxConnectedSet	<i>Keep only the largest connected set</i>
---------------------	--

Description

This function removes all nodes that are not part of the largest connected set.

Usage

```
keepMaxConnectedSet(x)
```

Arguments

x a [gGraph](#) object

Value

a [gGraph](#) object with only the nodes from the largest set remaining

Examples

```
max_set <- keepMaxConnectedSet(worldgraph.10k)
plot(max_set)
```

makeHexGrid	<i>Make a new gGraph object from a custom hexagonal grid</i>
-------------	--

Description

This function constructs a new [gGraph](#) object based on a hexagonal discrete global grid system (DGGS) with a user-defined spatial resolution. The graph is restricted to a geographic bounding box defined by the user.

Usage

```
makeHexGrid(geo.box, spacing, ...)
```

Arguments

geo.box	A geographic bounding box. With either a named numeric vector with xmin, xmax, ymin, ymax or an object of class bbox or sf. Coordinates must be in longitude/latitude (EPSG:4326).
spacing	A positive numeric value giving the desired spacing (in km) of the discrete global grid. The closest available DGGS resolution is used.
...	Additional arguments passed to other methods (currently not used).

Details

The function constructs a discrete global grid using the `dggridR` package, then identifies neighboring grid cells using the `spdep` package, and finally builds a `gGraph` object with the resulting graph structure and node coordinates.

Value

A `gGraph` object representing the DGGS restricted to the specified geographic region. The `@nodes.attr` slot is empty.

Examples

```
# Define a geographic bounding box (e.g., for a region in Europe)
geo.box <- c(xmin = -10, xmax = 30, ymin = 35, ymax = 60)
# Create a gGraph with a spacing of 300 km
ggraph <- makeHexGrid(geo.box = geo.box, spacing = 300)
plot(ggraph, edge = TRUE)
```

makeSquareGrid

Make a new gGraph object from a custom square grid

Description

The function `makeSquareGrid` builds a `gGraph` using a regular square 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

```
makeSquareGrid(
  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)
<code>n.lat</code>	the number of latitude coordinates of the grid (i.e., height of the grid, in number of cells)
<code>lon.range</code> , <code>lat.range</code>	vectors of length two giving the range covered by the grid, in longitude and latitude, respectively.

Value

A [gGraph](#) object.

See Also

[makeHexGrid](#) for a hexagonal grid that avoids projection distortions over large areas. [findLand](#) to classify nodes as land or sea after creating the grid.

Examples

```
# by defining the range covered by the grid
squareGraph <- makeSquareGrid(
  size      = 10000,
  lon.range = c(-12, 2),
  lat.range = c(49, 61)
)
squareGraph <- findLand(squareGraph)
squareGraph <- setColors(
  squareGraph,
  col.rules = data.frame(
    habitat = c("sea", "land"),
    color   = c("blue", "green")
  )
)
plot(squareGraph, reset = TRUE)

# If no area is specified, currently plotted area is used
geo.zoomin(c(8, 13, 54, 58))
newGraph <- makeSquareGrid(1e3)
newGraph <- findLand(newGraph)
newGraph <- setColors(
  newGraph,
  col.rules = 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

<code>x</code>	a valid gData object. The gData object to which it is linked must exist in the global environment.
<code>type</code>	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.
<code>pch.ori</code>	a numeric or a character indicating the type of point for locations.
<code>pch.nodes</code>	a numeric or a character indicating the type of point for nodes.
<code>col.ori</code>	a character string indicating the color to be used for locations.
<code>col.nodes</code>	a character string indicating the color to be used for nodes.
<code>col.gGraph</code>	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 .
<code>reset</code>	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).
<code>sticky.points</code>	a logical indicating if added points should be kept when replotting (TRUE, default), or not (FALSE). In any case, <code>reset=TRUE</code> will prevent points to be redrawn.
<code>...</code>	further arguments passed to <code>points</code> .

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.

Value

`NULL`.

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](#).

Other plotting_methods: [plot-gGraph](#), [plot.gPath\(\)](#)

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	<i>Plot a gGraph object.</i>
-------------	------------------------------

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`.

Usage

```
## S4 method for signature 'gGraph,missing'
plot(
  x,
  shape = "world",
  psize = NULL,
  pch = 19,
  col = NULL,
  edges = FALSE,
  reset = FALSE,
  bg.col = "gray",
  border.col = "dark gray",
  lwd = 1,
  useCosts = NULL,
  maxLwd = 3,
  col.rules = NULL,
  ...
)

## S4 method for signature 'gGraph'
points(
  x,
  psize = NULL,
  pch = NULL,
  col = NULL,
  edges = FALSE,
  lwd = 1,
  useCosts = NULL,
  maxLwd = 3,
  col.rules = NULL,
  sticky.points = FALSE,
  ...
)

plotEdges(
  x,
```

```

    useCosts = NULL,
    col = "black",
    lwd = 1,
    lty = 1,
    pch = NULL,
    psize = NULL,
    pcol = NULL,
    maxLwd = 3,
    col.rules = NULL,
    sticky.edges = FALSE,
    ...
)

```

Arguments

<code>x</code>	a gGraph object.
<code>shape</code>	a shapefile of the class <code>sf</code> (see <code>sf::st_read()</code> to import a GIS shapefile). Alternatively, a character string indicating one shapefile released with <code>geoGraph</code> ; currently, only 'world' is available. If <code>NULL</code> , the graph is plotted without any background layer.
<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 (<code>TRUE</code>) or not (<code>FALSE</code>).
<code>reset</code>	a logical indicating if plotting parameters should be reset (<code>TRUE</code>) or not (<code>FALSE</code>).
<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 proportional to edge cost (<code>TRUE</code>) or not (<code>FALSE</code>).
<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 sought from the <code>@meta\$color</code> slot of the object.
<code>...</code>	further arguments passed to the generic methods (<code>plot</code> , <code>points</code> , and <code>segments</code> , respectively).
<code>sticky.points</code>	a logical indicating if added points should be kept when replotting (<code>TRUE</code>), or not (<code>FALSE</code>). 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 (<code>TRUE</code>), or not (<code>FALSE</code> , default). In any case, <code>reset=TRUE</code> will prevent points to be redrawn.

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

Value

NULL.

See Also

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

Other plotting_methods: [plot-gData](#), [plot.gPath\(\)](#)

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)
```

plot.gPath

Plot a gPath object

Description

This method plots a [gPath](#) object, which is the output of the [dijkstraBetween](#), [polygonBetween](#) and [dijkstraFrom](#) functions.

Usage

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

## S3 method for class 'gPath'
print(x, ...)
```

Arguments

x	a gPath object
col	a character string indicating a color or a palette of colors to be used for plotting edges.
lwd	a numeric value indicating the width of edges.
seed	an optional integer value to set the seed for random color generation when col = "rainbow".
...	further arguments passed to geo.segments .

Value

NULL.

Functions

- `print(gPath)`: Print a summary of a `gPath` object

See Also

Other plotting_methods: [plot-gData](#), [plot-gGraph](#)

Examples

```

hgdp.sub <- hgdp[getData(hgdp)$Population %in%
  c("French", "Balochi", "BantuKenya", "Papuan", "Pima")]
hgdp.path <- dijkstraBetween(hgdp.sub) # compute shortest path

## plotting
plot(worldgraph.40k, reset = TRUE, pch = "")
points(hgdp.sub, lwd = 1) # plot populations
plot(hgdp.path) # plot the path

## printing
print(hgdp.path)
#####
# plot method
#####

```

polygonBetween	<i>Least-cost paths between two polygons</i>
----------------	--

Description

This function computes least-cost path distances between all nodes belonging to two polygons in a [gData](#) or [gGraph](#) using `dijkstraBetween`. The polygons are specified via a node attribute layer. Optionally, only outline nodes (nodes with at least one neighbor outside the polygon) are used to reduce computation time.

Usage

```

polygonBetween(g, layer, from, to, outline = TRUE)

```

Arguments

<code>g</code>	gData or gGraph object with edge costs already defined.
<code>layer</code>	Character. Name of the node attribute layer containing polygon membership.
<code>from</code>	Character. Name of the first polygon.
<code>to</code>	Character. Name of the second polygon.
<code>outline</code>	Logical. If TRUE, only outline nodes of each polygon (nodes with at least one neighbor outside the polygon) are used. Defaults to TRUE.

Details

The function extracts all nodes belonging to the two specified polygons and computes least-cost paths between them using `dijkstraBetween`. If `outline = TRUE`, the computation is restricted to outline nodes of each polygon, which can substantially reduce runtime for large polygons.

Value

A numeric vector containing the least-cost distances between nodes of the two polygons.

Examples

```
world.countries <- rnaturalearth::ne_countries(
  scale = "medium",
  returnclass = "sf"
)
newGraph <- assignByPolygon(rawgraph.10k,
  layer = world.countries,
  attr = c("continent", "name")
)
test <- polygonBetween(newGraph, layer = "name", "Spain", "Germany", outline = TRUE)
plot(newGraph, col = NA, reset = TRUE)
plot(test, col = "red")
```

setColors

Set color rules for a gGraph object

Description

The function `setColors` sets the color rules stored in the `@meta$colors` slot of a [gGraph](#) object. Color rules control how node attribute values are mapped to colors when plotting.

Usage

```
setColors(x, col.rules)
```

Arguments

<code>x</code>	a valid gGraph object.
<code>col.rules</code>	a two-column data.frame mapping attribute values to colors.

Details

Color rules must be provided as a two-column data.frame. The first column must be named after the node attribute and contain its possible values; the second column must be named "color" and contain valid R color strings.

Value

A [gGraph](#) object with the updated color rules.

See Also

[getColor](#) to retrieve colors or color rules.

Other accessor_methods: [getColor](#)(), [getCoords](#)(), [getCosts](#)(), [getData](#)(), [getEdges](#)(), [getGraph](#)(), [getNodes](#)(), [getNodesAttr](#)(), [setGraph](#)()

Examples

```
## get current rules
col.rules <- getColors(worldgraph.10k, res.type = "rules")
col.rules

## modify a color
col.rules$color[col.rules$habitat == "sea"] <- "lightblue"

## set back
x <- setColors(worldgraph.10k, col.rules)
getColors(x, res.type = "rules")
```

setCosts

Set friction in a gGraph object

Description

The function `setCosts` defines costs for the edges of a [gGraph](#) object according to a node attribute and cost rules defined in `@meta$costs`. Each node has a value for the chosen attribute which is associated to a cost. The cost of an edge is computed as a function of the costs of its two nodes.

Usage

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

Arguments

<code>x</code>	a gGraph object with at least one node attribute and a <code>@meta\$costs</code> component (see <code>worldgraph.10k</code> for an example).
<code>attr.name</code>	the name of the node attribute used to compute costs.
<code>node.values</code>	a numeric vector of costs for the nodes. If provided, overrides <code>attr.name</code> .
<code>cost.rules</code>	a two-column <code>data.frame</code> to update <code>x@meta\$costs</code> before computing edge costs. If <code>NULL</code> , existing <code>x@meta\$costs</code> is used.
<code>method</code>	how edge costs are computed from node costs: "mean", "product", or "function" (requires <code>FUN</code>).
<code>FUN</code>	a function to compute edge cost from two node costs. Required when <code>method = "function"</code> .
<code>...</code>	additional arguments passed to <code>FUN</code> .

Details

Costs are inversely proportional to connectivity: the larger the cost of an edge, the lower the connectivity between its two nodes. Costs in `geoGraph` are equivalent to weights in the `graph` and `RBGL` packages.

Value

A `gGraph` object with the newly defined edge costs.

See Also

`dropDeadEdges`, `getCosts`, `hasCosts`

Other `cost_functions`: `combineCosts()`, `dropCosts()`, `getCosts()`, `hasCosts()`, `setDistCosts()`

Examples

```
## get and modify cost rules then set costs in one call
cost.rules <- getCosts(worldgraph.10k, res.type = "rules")
cost.rules

## make sea travel cheaper
cost.rules$cost[cost.rules$habitat == "sea"] <- 50

## update rules and set costs in one call
x <- setCosts(worldgraph.10k, attr.name = "habitat", cost.rules = cost.rules, method = "mean")
```

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 the library [graph](#) (and thus [graph::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.

Other `cost_functions`: [combineCosts\(\)](#), [dropCosts\(\)](#), [getCosts\(\)](#), [hasCosts\(\)](#), [setCosts\(\)](#)

Examples

```
plot(rawgraph.10k, reset = TRUE)
geo.zoomin(list(x = c(110, 150), y = c(-10, -40)))
plotEdges(rawgraph.10k)

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

plotEdges(x)
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 dataframe containing node names of edges to be added. The first element of the list (or column of the data.frame) 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 data.frame.
<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

[getEdges](#) to retrieve edges. [geo.add.edges](#) and [geo.remove.edges](#) for interactive versions.

Examples

```
# check which nodes are neighbours of node "1"
head(getEdges(worldgraph.10k, res.type = "matNames"))

# remove an edge between two neighbouring nodes
node.from <- "1"
node.to <- getEdges(worldgraph.10k, res.type = "matNames")[1, 2]

x <- setEdges(worldgraph.10k,
  remove = data.frame(from = node.from, to = node.to)
)

# verify the edge is gone
areNeighbours(node.from, node.to, getGraph(x))

# add it back
x <- setEdges(x, add = data.frame(from = node.from, to = node.to))
areNeighbours(node.from, node.to, getGraph(x))
```

setGraph*Set the linked gGraph for a gData object*

Description

The function `setGraph` sets the name of the `gGraph` object linked to a `gData` object. It validates that the named object exists in the global environment and is a valid `gGraph`.

Usage

```
setGraph(x, graph)
```

```
## S4 method for signature 'gData'  
setGraph(x, graph)
```

Arguments

<code>x</code>	a valid <code>gData</code> object.
<code>graph</code>	either a character string giving the name of a <code>gGraph</code> object in the global environment, or a <code>gGraph</code> object itself.

Value

A `gData` object with the updated linked `gGraph`.

Functions

- `setGraph(gData)`: Method for `gData` objects

See Also

`getGraph` to retrieve the linked graph. `gData` for the class definition.

Other accessor_methods: `getColors()`, `getCoords()`, `getCosts()`, `getData()`, `getEdges()`, `getGraph()`, `getNodes()`, `getNodesAttr()`, `setColors()`

Examples

```
myGraph <- dropCosts(rawgraph.40k)  
hgdp2 <- setGraph(hgdp, "myGraph")  
getGraph(hgdp2)
```

setNodesAttr	<i>Set node attributes in a gGraph object</i>
--------------	---

Description

The function `setNodesAttr` adds or replaces a node attribute in a [gGraph](#) object.

Usage

```
setNodesAttr(x, attr.name, values, ...)
```

Arguments

<code>x</code>	a valid gGraph object.
<code>attr.name</code>	a character string giving the name of the attribute to set.
<code>values</code>	a vector of values, one per node.
<code>...</code>	additional arguments passed to other methods (currently unused).

Value

A [gGraph](#) object with the new node attribute added or replaced.

See Also

[getNodeAttr](#) to retrieve node attributes, [setCosts](#) to set edge costs.

Examples

```
### for gGraphs
node.attr <- getNodeAttr(rawgraph.10k, attr.name = "habitat")
neigh.list <- rawgraph.10k@graph$edgeL

# reclassify sea nodes as "coast" if they have any land neighbors
levels(node.attr$habitat) <- c(levels(node.attr$habitat), "coast")
for (i.node in seq_len(nrow(node.attr))) {
  if (node.attr$habitat[i.node] == "sea") {
    neighbour.names <- neigh.list[[i.node]]$edges
    neighbour.land.values <- node.attr[neighbour.names, "habitat"]
    if (any(neighbour.land.values %in% c("land"))) {
      node.attr$habitat[i.node] <- "coast"
    }
  }
}

# create coast graph
coastGraph <- setNodesAttr(rawgraph.10k, attr.name = "habitat", values = node.attr$habitat)

colors <- data.frame(
```

```

    habitat = c("sea", "land", "coast"),
    color = c("blue", "green", "lightblue")
  )

  coastGraph <- setColors(coastGraph, col.rules = colors)

  plot(coastGraph, reset = TRUE)

```

subset-gData

*Subset a gData object***Description**

Select a subset of locations from a [gData](#) object by index, logical vector, or node name. The @coords, @nodes.id, and @data slots are all subsetted consistently.

Arguments

x	a valid gData object.
i	indices for subsetting locations — a logical vector, integer indices, or character node names. If missing, all locations are kept.
j	indices for subsetting data columns. If missing, all columns are kept.
...	additional arguments passed to the <code>[]</code> method of @data.
drop	logical passed to the <code>[]</code> method of @data. Defaults to FALSE.

Value

A [gData](#) object with @coords, @nodes.id, and @data all subsetted consistently.

See Also

[getCoords](#), [getNodes](#), [getData](#)

Other basic_methods: [subset-gGraph](#)

Examples

```

## subset to northern hemisphere populations
north <- hgdp[hgdp@data$Latitude > 40]
plot(worldgraph.40k, reset = TRUE)
points(north)

```

subset-gGraph	<i>Subset a gGraph object</i>
---------------	-------------------------------

Description

Select a subset of nodes from a [gGraph](#) object by index, logical vector, or node name. The @coords, @nodes.attr, and @graph slots are all subsetted consistently.

Arguments

x	a valid gGraph object.
i	indices for subsetting nodes — a logical vector, integer indices, or character node names. If missing, all nodes are kept.
j	indices for subsetting node attributes (columns of @nodes.attr). If missing, all attributes are kept.
...	additional arguments (currently unused).
drop	logical, currently unused.

Value

A [gGraph](#) object containing only the selected nodes.

See Also

[getNodeNames](#) to retrieve node names. [isInArea](#) to select nodes within a geographic area.

Other basic_methods: [subset-gData](#)

Examples

```
## subset to nodes in a geographic area
plot(worldgraph.10k, reset = TRUE)
geo.zoomin(list(x = c(-6, 38), y = c(35, 73)))
x <- worldgraph.10k[isInArea(worldgraph.10k, quiet = TRUE)]

## subset by node name
x <- worldgraph.10k[c("1", "2", "3")]
```

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 vertex; 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.

Source

Graph constructed 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, quiet = TRUE)]
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

- * **accessor_methods**
 - getColors, [31](#)
 - getCoords, [32](#)
 - getCosts, [33](#)
 - getData, [35](#)
 - getEdges, [36](#)
 - getGraph, [37](#)
 - getNodes, [38](#)
 - getNodesAttr, [39](#)
 - setColors, [60](#)
 - setGraph, [65](#)
- * **basic_methods**
 - subset-gData, [67](#)
 - subset-gGraph, [68](#)
- * **classes**
 - gGraph-class, [40](#)
- * **connectivity_functions**
 - areConnected, [3](#)
 - areNeighbours, [4](#)
 - isConnected, gData-method, [45](#)
 - isReachable, [49](#)
- * **cost_functions**
 - combineCosts, [13](#)
 - dropCosts, [19](#)
 - getCosts, [33](#)
 - hasCosts, [43](#)
 - setCosts, [61](#)
 - setDistCosts, [62](#)
- * **datasets**
 - fst_hgdp, [22](#)
 - hgdp, [44](#)
 - worldgraph, [69](#)
- * **dijkstra_methods**
 - dijkstraBetween, [16](#)
 - dijkstraBuffer, [17](#)
 - dijkstraFrom, [18](#)
 - gPath2dist, [42](#)
- * **ggplot_methods**
 - autoplot.gData, [8](#)
 - autoplot.gGraph, [9](#)
 - geom_gdata, [28](#)
 - geom_ggraph, [29](#)
 - geom_gpath, [30](#)
- * **graphs**
 - gGraph-class, [40](#)
- * **hplot**
 - plot-gData, [52](#)
 - plot-gGraph, [55](#)
 - zoom, [70](#)
- * **methods**
 - buffer, [10](#)
 - dropDeadEdges, [20](#)
 - findLand, [21](#)
 - geo.segments, [27](#)
 - getEdges, [36](#)
 - getNodesAttr, [39](#)
 - hasCosts, [43](#)
 - makeSquareGrid, [51](#)
 - plot-gData, [52](#)
 - plot-gGraph, [55](#)
 - setDistCosts, [62](#)
 - setEdges, [63](#)
- * **plotting_methods**
 - plot-gData, [52](#)
 - plot-gGraph, [55](#)
 - plot.gPath, [58](#)
- * **spatial**
 - gGraph-class, [40](#)
 - plot-gData, [52](#)
 - plot-gGraph, [55](#)
- * **utilities**
 - buffer, [10](#)
 - dropDeadEdges, [20](#)
 - findLand, [21](#)
 - geo.add.edges, [24](#)
 - geo.change.attr, [25](#)
 - geo.segments, [27](#)
 - getEdges, [36](#)

- getNodeAttr, 39
 - hasCosts, 43
 - makeSquareGrid, 51
 - setDistCosts, 62
 - setEdges, 63
 - zoom, 70
- .zoomlog.up (zoom), 70
- [, gData, ANY, ANY, ANY-method (subset-gData), 67
- [, gData, ANY, ANY-method (subset-gData), 67
- [, gData-method (subset-gData), 67
- [, gGraph, ANY, ANY, ANY-method (subset-gGraph), 68
- [, gGraph, ANY, ANY-method (subset-gGraph), 68
- [, gGraph-method (subset-gGraph), 68
- areConnected, 3
- areConnected(), 4, 5, 46, 49
- areNeighbours, 4
- areNeighbours(), 4, 46, 49
- as.dist(), 43
- assignByPolygon, 5, 8, 22
- assignByPolygon, data.frame-method (assignByPolygon), 5
- assignByPolygon, gData-method (assignByPolygon), 5
- assignByPolygon, gGraph-method (assignByPolygon), 5
- assignByPolygon, list-method (assignByPolygon), 5
- assignByPolygon, matrix-method (assignByPolygon), 5
- assignByPolygon, numeric-method (assignByPolygon), 5
- assignByRaster, 6, 7
- autoplot.gData, 8
- autoplot.gData(), 9, 28–30
- autoplot.gGraph, 9
- autoplot.gGraph(), 9, 28–30
- buffer, 10, 17
- buffer, gData-method (buffer), 10
- buffer, gGraph-method (buffer), 10
- buffer-methods (buffer), 10
- closestNode, 11
- closestNode, gData-method (closestNode), 11
- closestNode, gGraph-method (closestNode), 11
- combineCosts, 13
- combineCosts(), 20, 34, 44, 62, 63
- connectivityPlot, 15
- connectivityPlot, gData-method (connectivityPlot), 15
- connectivityPlot, gGraph-method (connectivityPlot), 15
- dijkstraBetween, 16, 41, 42, 58
- dijkstraBetween(), 17, 19, 30, 42, 43
- dijkstraBetween, gData-method (dijkstraBetween), 16
- dijkstraBetween, gGraph-method (dijkstraBetween), 16
- dijkstraBuffer, 17
- dijkstraBuffer(), 16, 19, 43
- dijkstraFrom, 17, 18, 41, 42, 58
- dijkstraFrom(), 16, 17, 30, 42, 43
- dijkstraFrom, gData-method (dijkstraFrom), 18
- dijkstraFrom, gGraph-method (dijkstraFrom), 18
- dist, 42, 43
- dropCosts, 19, 34
- dropCosts(), 14, 34, 44, 62, 63
- dropCosts, gGraph-method (dropCosts), 19
- dropDeadEdges, 20, 62
- dropDeadNodes (dropDeadEdges), 20
- findLand, 6, 21, 52
- findLand, data.frame-method (findLand), 21
- findLand, gGraph-method (findLand), 21
- findLand, matrix-method (findLand), 21
- fst_hgdp, 22
- gData, 4–6, 8, 10–12, 15, 16, 18, 28, 32, 35, 37–39, 41, 42, 44–49, 52, 53, 59, 65, 67
- gData (gData-class), 23
- gData-class, 23
- geo.add.edges, 24, 37, 64
- geo.back, 54, 57
- geo.back (zoom), 70
- geo.bookmark, 54

- geo.bookmark (zoom), 70
- geo.change.attr, 25
- geo.goto, 54
- geo.goto (zoom), 70
- geo.remove.edges, 37, 64
- geo.remove.edges (geo.add.edges), 24
- geo.segments, 27, 58
- geo.slide, 54, 57
- geo.slide (zoom), 70
- geo.zoomin, 48, 54, 57
- geo.zoomin (zoom), 70
- geo.zoomout, 54, 57
- geo.zoomout (zoom), 70
- geoGraph, 27, 44
- geom_gdata, 28
- geom_gdata(), 9, 29, 30
- geom_ggraph, 29
- geom_ggraph(), 9, 28, 30
- geom_gpath, 30
- geom_gpath(), 9, 28, 29
- getColor, 31, 60
- getColor(), 33–35, 37–39, 60, 65
- getColor, gGraph-method (getColor), 31
- getCoords, 32, 35, 37–39, 67
- getCoords(), 32, 34, 35, 37–39, 60, 65
- getCoords, gData-method (getCoords), 32
- getCoords, gGraph-method (getCoords), 32
- getCosts, 20, 33, 62, 63
- getCosts(), 14, 20, 32, 33, 35, 37–39, 44, 60, 62, 63, 65
- getCosts, gGraph-method (getCosts), 33
- getData, 35, 67
- getData(), 32–34, 37–39, 60, 65
- getData, gData-method (getData), 35
- getEdges, 36, 37, 39, 64
- getEdges(), 32–35, 37–39, 60, 65
- getEdges, gGraph-method (getEdges), 36
- getGraph, 33, 37, 38, 65
- getGraph(), 32–35, 37–39, 60, 65
- getGraph, gData-method (getGraph), 37
- getGraph, gGraph-method (getGraph), 37
- getNodeCosts (getCosts), 33
- getNodeCosts, gGraph-method (getCosts), 33
- getNodes, 33, 35, 37, 38, 39, 67, 68
- getNodes(), 32–35, 37, 39, 60, 65
- getNodes, gData-method (getNodes), 38
- getNodes, gGraph-method (getNodes), 38
- getNodeAttr, 32, 35, 38, 39, 66
- getNodeAttr(), 32–35, 37, 38, 60, 65
- getNodeAttr, gData-method (getNodeAttr), 39
- getNodeAttr, gGraph-method (getNodeAttr), 39
- ggplot2::geom_sf(), 28–30
- gGraph, 3–7, 9–26, 29, 31–39, 41–48, 50–53, 55, 56, 59–66, 68–71
- gGraph (gGraph-class), 40
- gGraph-class, 40
- gPath, 16, 19, 30, 58
- gPath (gPath-class), 41
- gPath-class, 41
- gPath2dist, 17, 42, 42
- gPath2dist(), 16, 17, 19
- graph::graphNEL, 4, 36, 37, 40, 63
- graph::isConnected(), 4
- hasCosts, 20, 34, 43, 62
- hasCosts(), 14, 20, 34, 62, 63
- hgdp, 23, 44
- hgdpPlus (hgdp), 44
- isConnected (isConnected, gData-method), 45
- isConnected(), 4
- isConnected, gData-method, 45
- isConnected, gGraph-method (isConnected, gData-method), 45
- isInArea, 46, 57, 68
- isInArea, data.frame-method (isInArea), 46
- isInArea, gData-method (isInArea), 46
- isInArea, gGraph-method (isInArea), 46
- isInArea, matrix-method (isInArea), 46
- isReachable, 49
- isReachable(), 4, 5, 46
- keepMaxConnectedSet, 50
- locator(), 12
- makeHexGrid, 50, 52
- makeSquareGrid, 51, 51
- plot, gData, missing-method (plot-gData), 52
- plot, gGraph, missing-method (plot-gGraph), 55

plot, gGraph-method (plot-gGraph), 55
plot-gData, 52
plot-gGraph, 55
plot.gData (plot-gData), 52
plot.gGraph, 71
plot.gGraph (plot-gGraph), 55
plot.gPath, 42, 58
plot.gPath(), 54, 57
plot_gData (plot-gData), 52
plot_gGraph (plot-gGraph), 55
plotEdges (plot-gGraph), 55
points, gData-method (plot-gData), 52
points, gGraph-method (plot-gGraph), 55
polygonBetween, 41, 42, 58, 59
print.gPath (plot.gPath), 58

rawgraph.10k (worldgraph), 69
rawgraph.40k (worldgraph), 69
rdist.earth, 62

setColors, 32, 60
setColors(), 32–35, 37–39, 65
setCosts, 14, 17, 20, 26, 34, 61, 66
setCosts(), 14, 20, 34, 44, 63
setDistCosts, 62
setDistCosts(), 14, 20, 34, 44, 62
setDistCosts, gGraph-method
 (setDistCosts), 62
setDistCosts-methods (setDistCosts), 62
setEdges, 25, 37, 63
setEdges, gGraph-method (setEdges), 63
setGraph, 65
setGraph(), 32–35, 37–39, 60
setGraph, gData-method (setGraph), 65
setNodesAttr, 7, 8, 66
sf::st_read, 6
sf::st_read(), 21, 56
subset-gData, 67
subset-gGraph, 68

worldgraph, 69
worldgraph.40k, 44

zoom, 70