Package 'grapherator'

May 8, 2026

Title A Modular Multi-Step Graph Generator

Description Set of functions for step-wise generation of (weighted) graphs. Aimed for research in the field of single- and multi-objective combinatorial optimization. Graphs are generated adding nodes, edges and weights. Each step may be repeated multiple times with different predefined and custom generators resulting in high flexibility regarding the graph topology and structure of edge weights.

Version 1.0.0

Encoding UTF-8

Date 2017-12-20

Maintainer Jakob Bossek <j.bossek@gmail.com>

License BSD_2_clause + file LICENSE

URL https://github.com/jakobbossek/grapherator

BugReports https://github.com/jakobbossek/grapherator/issues

Imports BBmisc (>= 1.6), checkmate (>= 1.1), reshape2 (>= 1.4.1), vegan, ggplot2 (>= 1.0.0), lhs, deldir, grDevices

Suggests testthat (>= 0.9.1), knitr, rmarkdown, gridExtra, magrittr

ByteCompile yes

LazyData yes

RoxygenNote 6.0.1

VignetteBuilder knitr

NeedsCompilation no

Author Jakob Bossek [aut, cre]

Repository CRAN

Date/Publication 2017-12-21 13:19:38 UTC

Contents

`grapherator-package` *grapherator: A modular multi-step graph generator*

Description

Due to lack of real world graphs, e.g., the optimization community often relies on artificial graphs to benchmark algorithms. The **grapherator** package implements a multi-step approach for the generation of weighted graphs. A set of predefined node, edge and weight generators allows for fast and convenient graph generation. Furthermore, the modular structure of the package enables writing user-defined generators and use them within the framework in a plug-and-play style.

Generation philosophy

The graph generation follows a three step procedure. A bare graph (see `graph`), i.e., an empty graph object, the following three serves as a staring point for several iterations of the following steps. Note that once edges have been added, no furhter nodes may be added. Likewise, after weights have been attached to edges, no further edges may be added. 1) Node generation via `addNodes`: nodes are generated by placing node coordinates in the two-dimensional Euclidean plane using different node generators. 2) Edge generation via `addEdges`: links between nodes are established via one or multiple edge generators. 3) Weight generation via `addWeights`: One or more weights are attached to each edge by different weight generators.

`addEdges` *Add edges to graph.*

Description

This method allows to add edges to a `grapherator` graph. The method can be applied multiple times with different parameterizations. E.g., add edges in clusters first and add edges between clusters in a second step.

Usage

```
addEdges(graph, generator, type = "all", k = NULL, cluster.ids = NULL,
  ...)
```

Arguments

graph	[grapherator] Graph.
generator	[function(graph, ...)] Method applied to graph in order to determine which edges to add.
type	[character(1)] Value "all" applies generator to all nodes. Value "intracluster" instead applies the method for each cluster seperately. Value "intercluster" selects each k nodes from each cluster and applies generator to the union. Lastly, value "intercenter" selects the cluster centers exclusively. Default is "all".
k	[integer | NULL] Integer vector specifying the number of nodes selected randomly from each cluster to be selected for edge construction. May be a scalar value or a vector of length graph$n.clusters. NAs are allowed and indicate clusters to be ignored.
cluster.ids	[integer | NULL] Ignored unless type is not set to "intracluster". Integer vector of cluster IDs. If NULL the generator is applied within each cluster.
...	[any] Further arguments passed down to edge generator generator.

Value

[grapherator] Graph.

References

Erdos, P., and A. Renyi. 1959. "On random graphs, I." Publicationes Mathematicae (Debrecen) 6: 290-97.

Waxman, B. M. 1988. "Routing of Multipoint Connections."" IEEE Journal on Selected Areas in Communications 6 (9): 1617-22. doi:10.1109/49.12889.

Knowles, J. D., and D. W. Corne. 2001. "Benchmark Problem Generators and Results for the Multiobjective Degree-Constrained Minimum Spanning Tree Problem." In Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation, 424-31. GECCO'01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

See Also

Other graph generators: addNodes, addWeights, graph

Examples

```
g = graph(0, 1000)
g = addNodes(g, n = 5, generator = addNodesLHS)
g = addNodes(g, n = c(3, 10, 20, 10, 40), by.centers = TRUE, generator = addNodesUniform,
  lower = c(0, 0), upper = c(30, 30))
# user different edge generators for clusters
g = addEdges(g, generator = addEdgesDelauney, type = "intracluster", cluster.ids = 1:3)
g = addEdges(g, generator = addEdgesSpanningTree, type = "intracluster", cluster.ids = 4:5)
# link cluster centers
g = addEdges(g, generator = addEdgesSpanningTree, runs = 2, type = "intercenter")
# additional random links between each 2 nodes from each cluster
g = addEdges(g, generator = addEdgesGilbert, p = 0.4, type = "intercluster", k = 2)
```

---

addEdgesToPlot *Highlight edges in graph.*

Description

Highlights edges in coordinate plot.

Usage

```
addEdgesToPlot(x, g, edge.list, ...)
```

Arguments

x	[ggplot] Coordinate plot generated via plot.grapherator.
g	[grapherator] Graph.
edge.list	[matrix] Matrix of edges (each column is one edge).
...	[any] Arguments passed down to geom_segment.

Value

[ggplot] Modified x.

Examples

```
## Not run:
g = graph(0, 100)
g = addNodes(g, n = 10, generator = addNodesUniform)
g = addEdges(g, generator = addEdgesComplete)
pl = plot(g)$pl.coords
el = matrix(c(1, 2, 1, 3, 4, 5, 3, 4), nrow = 2L)
pl = addEdgesToPlot(pl, g, el)
```

```
print(pl)

## End(Not run)
```

---

addNodes *Add nodes to graph.*

Description

Places node coordinates in the two-dimensional Euclidean plane.

Usage

```
addNodes(graph, n, generator, coordinates = NULL, by.centers = FALSE,
  skip.centers = integer(0L), par.fun = NULL, ...)
```

Arguments

graph	[grapherator] Graph.
n	[integer] Number of nodes to place. If by.centers is FALSE a single integer value is expected. Otherwise, a vector v may be passed. In this case $v[i]$ coordinates are generated for each cluster. However, if a single value is passed and by.center == TRUE, each cluster is assigned the same number of nodes.
generator	[function(graph, ...)] Function used to generate nodes. The functions needs to expect the number of points to generate as the first argument n. Additional control argument are possible.
coordinates	[matrix(n, 2)] Matrix of coordinates (each row is one node/point). Default is NULL. If this is set, setting of generator, by.centers, and par.fun are ignored. This parameter is handy, if one wants to add coordinates by hand. Default is NULL.
by.centers	[logical(1)] Should coordinates be placed for each cluster center seperately? This enables generation of clustered graphs. Default is FALSE.
skip.centers	[integer] Optional IDs of cluster centers not to consider in clustered node generation, i.e., if by.centers = TRUE. Default is not to skip any cluster.
par.fun	[function(cc) | NULL] Optional function which is applied to each cluster center before the generation of coordinates in case by.centers is TRUE. This enables to specifically determine additional parameters for the generator for each cluster.
...	[any] Further arguments passed down to generator.

Value

[grapherator] Graph.

See Also

Other graph generators: addEdges, addWeights, graph

Examples

```
# Clustered graph
g = graph(0, 1000)
g = addNodes(g, n = 5, generator = addNodesLHS)
g = addNodes(g, n = c(3, 10, 20, 10, 40), by.centers = TRUE, generator = addNodesUniform,
 lower = c(0, 0), upper = c(30, 30))
## Not run:
plot(g, show.edges = FALSE)$pl.coords

## End(Not run)

# Mixed graph
g = graph(0, 100)
g = addNodes(g, n = 100, generator = addNodesLHS)
g = addNodes(g, n = 100, generator = addNodesGrid)

## Not run:
plot(g, show.edges = FALSE)$pl.coords

## End(Not run)
```

---

addWeights *Add weights to graph.*

Description

addWeights allows to add edge weights to a graph. This is the last step of the graph generation process. Note that adding edges is not possible once addWeights was called once.

Usage

```
addWeights(graph, generator = NULL, weights = NULL, symmetric = TRUE,
  to.int = FALSE, ...)
```

Arguments

graph	[grapherator] Graph.
generator	[function(graph, ...)] Function used to generate weights. The functions needs to expect the graph as the first argument graph. Additional control argument are possible.

weights [matrix]
Square matrix of weights. If some weights are already assigned, pay attention to the correct dimensions. If this is passed all other arguments are ignored. Default is NULL.

symmetric [logical(1)]
Should the weights be symmetric, i.e., $w(i,j) = w(j,i)$ for each pair i, j of nodes? Default is TRUE.

to.int [logical(1)]
Should weights be rounded to integer? Default is FALSE.

... [any]
Additional arguments passed down to generator.

Value

[grapherator] Graph.

See Also

Other graph generators: addEdges, addNodes, graph

Examples

```
# first we define a simple graph
g = graph(0, 100)
g = addNodes(g, n = 5, generator = addNodesLHS)
g = addNodes(g, n = c(3, 10, 20, 10, 40), by.centers = TRUE, generator = addNodesUniform,
  lower = c(0, 0), upper = c(15, 15))
g = addEdges(g, generator = addEdgesDelauney)

# first graph contains two integer random weights per edge
g1 = addWeights(g, generator = addWeightsRandom, method = runif, min = 10, max = 20, to.int = TRUE)
g1 = addWeights(g, generator = addWeightsRandom, method = runif, min = 10, max = 30, to.int = TRUE)
## Not run:
plot(g1)$pl.weights

## End(Not run)

# next one contains correlated weights. The first weight corresponds to the
# Euclidean distance of the points, the second is generated in a way, that
# a given correlation rho is achieved.
g2 = addWeights(g, generator = addWeightsCorrelated, rho = -0.7)
## Not run:
plot(g2)$pl.weights

## End(Not run)

# Last example contains two weights per edge: the first one is the Manhattan
# block distance between the nodes in the plane. The second one is the Euclidean
# distance plus a normally distributed jitter. Here we write a custom weight
# generator which returns two weight matrizes.
myWeightGenerator = function(graph, ...) {
```

```
  n = getNumberOfNodes(graph)
  adj.mat = getAdjacencyMatrix(graph)
  coords = getNodeCoordinates(graph)

  man.dist = as.matrix(dist(coords), method = "manhattan")
  euc.dist = as.matrix(dist(coords)) + abs(rnorm(n * n, ...))

  # keep in mind non-existent edges
  euc.dist[!adj.mat] = man.dist[!adj.mat] = Inf

  # return the necessary format
  return(list(weights = list(man.dist, euc.dist), generator = "MyWG"))
}

g3 = addWeights(g, generator = myWeightGenerator, mean = 30, sd = 5)
## Not run:
plot(g3)$pl.weights

## End(Not run)
```

---

addWeightsConcave *@title Weight generators.*

Description

Function for adding weight(s) to edges. The following functions are implemented and may be passed as argument `generator` to `addWeights`:

`addWeightsRandom` Add purely random weights. Calls the passed `method`, e.g., `method = runif` to generate weights.

`addWeightsDistance` Weights correspond to a distance metric based on the node coordinates in the Euclidean plane. Internally function `dist` is called.

`addWeightsCorrelated` This method generates two weight matrices with correlated weights. The correlation may be adjusted by the `rho` argument. Here, the first weight of an edge is the Euclidean distance between the nodes in the plane and the second one is generated in a way, that the correlation is close to `rho`.

`addWeightsCocave` This method is interesting for generating bi-objective graphs to benchmark algorithms for the multi-criteria spanning tree problem. Graphs generated this way expose a concave Pareto-front.

Usage

```
addWeightsConcave(graph, xhi = 10, nu = 20, M = 100, ...)

addWeightsCorrelated(graph, rho, ...)

addWeightsDistance(graph, method, ...)

addWeightsRandom(graph, method, ...)
```

Arguments

graph	[grapherator] Graph.
xhi	[integer(1)] Positive integer for addWeightsConcave. Default is 10.
nu	[integer(1)] Positive integer for addWeightsConcave. Default is 20.
M	[integer(1)] Maximum weight for weights generated via addWeightsConcave. Note that M minus xhi needs to be much bigger than nu. Default is 100.
...	[any] Further arguments. Not used at the moment. This may be useful for user-written weight generators.
rho	[numeric(1)] Desired correlation, i.e., value between -1 and 1, of edge weights for addWeightsCorrelated.
method	[character(1) | function(n, ...)] String representing the distance measure to use for addWeightsDistance (see method argument of dist) or function(n, ...) used to generate random weights in case of addWeightsRandom.

Value

[list] A list with components

weights [list] List of weight matrices. Even in the case of one weight matrix it is wrapped in a list of length one.

generator [character(1)] String description of the generator used.

Note

These functions are not meant to be called directly. Instead, they need to be assigned to the generator argument of addWeights.

---

as.character.grapherator

Graph string representation.

Description

Given a grapherator object the function returns a string representation. Basically this is a concatenation of meta data, node, edge and weight generator types of the following format: N<n.nodes>-E<n.edges>-C<n.clusters>-W<n.weights>—<node-types>—<edge-types>—<weight-types> where n.x is the number of x of the graph.

Usage

```
## S3 method for class 'grapherator'
as.character(x, ...)
```

Arguments

x	[grapherator] Graph.
...	[any] Not used at the moment.

Value

[character(1)]

Examples

```
g = graph(lower = c(0, 0), upper = c(100, 100))
g = addNodes(g, n = 3, generator = addNodesUniform)
g = addNodes(g, n = 14, by.centers = TRUE, generator = addNodesUniform,
  lower = c(0, 0), upper = c(10, 10))
g = addEdges(g, generator = addEdgesWaxman, alpha = 0.2,
  beta = 0.2, type = "intracluster")
g = addEdges(g, generator = addEdgesDelauney, type = "intercenter")
g = addWeights(g, generator = addWeightsCorrelated, rho = -0.9)
g = addWeights(g, generator = addWeightsDistance, method = "euclidean")
as.character(g)
```

---

edgeGenerators *Edge generators.*

Description

Function to add edges into a graph. The following methods are implemented so far:

addEdgesComplete Generates a simple complete graph. I.e., an edge exists between each two nodes. However, no self-loops or multi-edges are included.

addEdgesGrid Only usefull if nodes are generated via addNodesGrid. This method generates a Manhattan-like street network.

addEdgesOnion This method determines the nodes on the convex hull of the node cloud in the euclidean plane and adds edges between neighbour nodes. Ignoring all nodes on the hull, this process is repeated iteratively resulting in an onion like peeling topololgy. Note that the graph is not connected! In order to ensure connectivity, another edge generator must be applied in addition, e.g., addEdgesSpanningTree.

addEdgesDelauney Edges are determined by means of a Delauney triangulation of the node coordinates in the Euclidean plane.

`addEdgesWaxman` Edges are generated using the Waxman-model, i.e., the probability p_{ij} for the edge (i, j) is given by

$$p_{ij} = \alpha e^{-\beta d_{ij}}$$

, where $\alpha, \beta \geq 0$ are control parameters and d_{ij} is the Euclidean distance of the nodes i and j.

`addEdgesSpanningTree` A minimum spanning tree is computed based on a complete random weight matrix. All edges of the spanning tree are added. If `runs` is greater 1, the process is repeated for `runs`. However, already added edges are ignored in subsequent runs. This method is particularly useful to assist probablistic methods, e.g., Waxman model, in order to generate connected graphs.

`addEdgesGilbert` Use Gilbert-model to generate edges. I.e., each edge is added with probability $p \in [0, 1]$.

`addEdgesErdosRenyi` In total $m \leq n(n-1)/2$ edges are added at random.

Usage

```
addEdgesComplete(graph, ...)

addEdgesGrid(graph, ...)

addEdgesOnion(graph, ...)

addEdgesDelauney(graph, ...)

addEdgesWaxman(graph, alpha = 0.5, beta = 0.5, ...)

addEdgesGilbert(graph, p, ...)

addEdgesErdosRenyi(graph, m, ...)

addEdgesSpanningTree(graph, runs = 1L, ...)
```

Arguments

`graph`	[grapherator] Graph.
`...`	[any] Not used at the moment.
`alpha`	[numeric(1)] Positive number indicating the average degree of nodes in the Waxman model. Default is 0.5.
`beta`	[numeric(1)] Positive number indicating the scale between short and long edges in the Waxman model. Default is 0.5.
`p`	[numeric(1)] Probability for each edge $(v_i, v_j), i, j = 1, \ldots, n$ to be added for Gilbert graphs.

m	[integer(1)] Number of edges to sample for Erdos-Renyi graphs. Must be at most $n(n-1)/2$ where n is the number of nodes of graph.
runs	[integer(1)] Number of runs to perform by addEdgesSpanningTree. Default is 1.

Details

Currently all edge generators create symmetric edges only.

Value

[list] List with components:

adj.mat matrix Adjacency matrix.

generator [character(1)] String description of the generator used.

Note

These functions are not meant to be called directly. Instead, they need to be assigned to the generator argument of addEdges.

---

getter *Getter functions.*

Description

Functions to extract meta information of grapherator object.

Usage

```
getNumberOfNodes(graph)

getNumberOfEdges(graph)

getNumberOfClusters(graph)

getNumberOfWeights(graph)

getNodeCoordinates(graph, cluster.centers = FALSE)

getWeightMatrix(graph, objective)

getAdjacencyMatrix(graph)

getNodeDegrees(graph)
```

Arguments

graph	[grapherator] Graph.
cluster.centers	[logical(1)] Return coordinates of cluster centers only? Default is FALSE.
objective	[integer(1)] Number of weight matrix to return.

Examples

```
g = graph(0, 100)
g = addNodes(g, n = 25, generator = addNodesGrid)
g = addEdges(g, generator = addEdgesGrid)
g = addWeights(g, generator = addWeightsRandom, method = runif, min = 5, max = 100, to.int = TRUE)
g = addWeights(g, generator = addWeightsDistance, method = "euclidean")

getNumberOfNodes(g)
getNumberOfEdges(g)
getNumberOfClusters(g)
getNumberOfWeights(g)
getNodeCoordinates(g)
getWeightMatrix(g, 2)
getAdjacencyMatrix(g)
getNodeDegrees(g)
```

---

graph *Generate a bare graph.*

Description

This function generates a bare graph object of type grapherator. The generated object does not contain nodes, edges or edge weights. It serves as a starting point for a three step approach of grapherator graph construction: 1) Add nodes respectively coordinates via addNodes, 2) add edges via addEdges and finally 3) add edge weights with the function addWeights.

Usage

```
graph(lower, upper)
```

Arguments

lower	[integer(1)] Lower bounds for node coordinates in the Euclidean plane.
upper	[integer(1)] Upper bounds for node coordinates in the Euclidean plane.

Value

[grapherator] Graph.

See Also

Other graph generators: addEdges, addNodes, addWeights

Examples

```
# complete graph with one U(10, 20) sampled weight per edge
g = graph(0, 10)
g = addNodes(g, n = 10, generator = addNodesUniform)
g = addEdges(g, generator = addEdgesComplete)
g = addWeights(g, generator = addWeightsRandom, method = runif, min = 10, max = 20)
## Not run:
do.call(gridExtra::grid.arrange, plot(g, show.edges = FALSE))

## End(Not run)

# we extend the graph by adding another weight which is based
# on the Euclidean distance between the node coordinates
g = addWeights(g, generator = addWeightsDistance, method = "euclidean")
## Not run:
do.call(gridExtra::grid.arrange, plot(g, show.edges = FALSE))

## End(Not run)

# next we generate a graph with each two weights per edge which resembles
# a street network. The edge weights have a positive correlation.
g = graph(0, 100)
g = addNodes(g, n = 5, generator = addNodesLHS)
g = addNodes(g, n = c(10, 10, 15, 20, 50), by.centers = TRUE,
  generator = addNodesUniform, lower = c(0, 0), upper = c(10, 10))
g = addEdges(g, generator = addEdgesDelauney, type = "intracluster")
g = addEdges(g, generator = addEdgesDelauney, type = "intercluster", k = 4L)
g = addWeights(g, generator = addWeightsCorrelated, rho = 0.6)
## Not run:
print(g)
do.call(gridExtra::grid.arrange, plot(g, show.edges = FALSE))

## End(Not run)
```

---

grapherator *Graph object.*

Description

S3 object describing a graph with the following fields:

lower [numeric(2)] Lower bounds for node coordinates in the Euclidean plane.

upper [`numeric(2)`] Upper bounds for node coordinates in the Euclidean plane.

n.clusters [`integer(1)`] Number of clusters.

n.nodes [`integer(1)`] Number of nodes.

n.edges [`integer(1)`] Number of edges.

n.weights [`integer(1)`] Number of weights associated with each edge.

node.types [`character`] Character vector describing the node generators used to create nodes.

edge.types [`character`] Character vector describing the node generators used to create edges.

weight.types [`character`] Character vector describing the node generators used to create weights.

weights [`list` **of** `matrix`] List of weight/distance/cost matrizes.

degree [`integer`] Integer vector of node degrees.

membership [`integer` | `NULL`] Integer vector which stores the cluster membership of each node. Not `NULL` only if graph is clustered.

coordinates [`matrix(n.nodes, 2)`] Matrix of node coordinates. Each row contains the node coordinates of one node.

`nodeGenerators` *Node generators.*

Description

Functions for the placement of nodes / node coordinates in the Euclidean plane. Function `addNodesLHS` generates a space-filling Latin-Hypercube-Sample (LHS), function `addNodesUniform` samples points from a bivariate uniform distribution, `addNodesGrid` generates a regular grid/lattice of points, `addNodesTriangular` generates a regular triangular grid/lattice and `addNodesNormal` generates nodes on basis of a normal distribution.

Usage

```
addNodesLHS(n, lower = 0, upper = 1, method = NULL)

addNodesUniform(n, lower, upper)

addNodesTriangular(n, lower, upper)

addNodesGrid(n, lower, upper)

addNodesNormal(n, lower, upper, x.mean, x.sd, y.mean, y.sd)
```

Arguments

n	[integer(1)] Number of nodes to generate.
lower	[numeric(2)] Minimal values for the first and second node coordinates respectively. Default is 0 for both dimensions.
upper	[numeric(2)] Maximal values for the first and second node coordinates respectively. Default is 1 for both dimensions.
method	[function] Function from package **lhs**. Default is maximinLHS. Only relevant for addNodesLHS.
x.mean	[numeric] Mean value of normal distribution for x-value generation. Only relevant for addNodesNormal.
x.sd	[numeric] Standard deviation of normal distribution for x-value generation. Only relevant for addNodesNormal.
y.mean	[numeric] Mean value of normal distribution for y-value generation. Only relevant for addNodesNormal.
y.sd	[numeric] Standard deviation of normal distribution for y-value generation. Only relevant for addNodesNormal.

Value

[list] List with components:

coords [matrix(n, 2)] Matrix of node coordinates.

generator [character(1)] String description of the generator used.

Note

These functions are not meant to be called directly. Instead, they need to be assigned to the generator argument of addNodes.

---

plot.grapherator *Visualize graph.*

Description

plot.grapherator generates a scatterplot of the nodes in the Euclidean plane. Additionally, the edge weights are visualized. In case of one weight per edge either a histogram or an empirical distribution function is drawn. For graphs with two weights per edge a scatterplot is used.

Usage

```
## S3 method for class 'grapherator'
plot(x, y = NULL, show.cluster.centers = TRUE,
  highlight.clusters = FALSE, show.edges = TRUE,
  weight.plot.type = "histogram", ...)
```

Arguments

x	[grapherator] Graph.
y	Not used at the moment.
show.cluster.centers	[logical(1)] Display cluster centers? Default is TRUE. This option is ignored silently if the instance is not clustered.
highlight.clusters	[logical(1)] Shall nodes be coloured by cluster membership? Default is FALSE.
show.edges	[logical(1)] Display edges? Keep in mind, that the number of edges is $O(n^2)$ where n is the number of nodes. Default is TRUE.
weight.plot.type	[character(1)] Type of visualization which should be used for weights in case x has only as single weight attached to each edge. Either "histogram" or "ecdf" (empirical distribution function) are possible choices. Default is histogram.
...	[any] Not used at the moment.

Value

[list] A list of ggplot objects with components pl.weights (scatterplot of edge weights) and eventually pl.coords (scatterplot of nodes). The latter is NULL, if graph has no associated coordinates.

Examples

```
g = graph(0, 100)
g = addNodes(g, n = 25, generator = addNodesGrid)
g = addEdges(g, generator = addEdgesDelauney)
g = addWeights(g, generator = addWeightsDistance, method = "manhattan")
## Not run:
pls = plot(g, weight.plot.type = "ecdf")

## End(Not run)

g = addWeights(g, generator = addWeightsRandom,
  method = rpois, lambda = 0.1)
```

```
## Not run:
pls = plot(g, show.edges = FALSE)

## End(Not run)

g = graph(0, 100)
g = addNodes(g, n = 25, generator = addNodesGrid)
g = addNodes(g, n = 9, by.centers = TRUE, generator = addNodesGrid,
  lower = c(0, 0), upper = c(7, 7))
g = addEdges(g, generator = addEdgesDelauney)
g = addWeights(g, generator = addWeightsCorrelated, rho = -0.8)
## Not run:
do.call(gridExtra::grid.arrange, plot(g, show.edges = FALSE))
do.call(gridExtra::grid.arrange, plot(g, show.edges = TRUE,
  show.cluster.centers = FALSE))

## End(Not run)
```

`writeGP` *Export/import graph.*

Description

Given a `grapherator` graph function `writeGP` saves the graph to a file. Function `readGP` imports a graph given a filename.

Usage

```
writeGP(graph, file)

readGP(file)
```

Arguments

`graph`	[`grapherator`] Graph.
`file`	[`character(1)`] Path to file where the graph shall be stored (for `writeGP`) or which contains the graph to be imported (for `link{readGP}`).

Details

Instances are stored in a format similar to the one used by Cardoso et al. in their MOST project. Note that all values in each line are separated by comma. First line contains four integer values: number of nodes n, number of edges m, number of clusters cl and number of weights p per edge. The second line contains the weight types. The third line contains the node types. The next n lines contain the node coordinates. In case of a clustered instance the next line contains the node to cluster membership mapping. The last m lines contain the following information each: i,j,w1(i,j),...,wp(i,j) I.e., each two node numbers i and j followed by the p weights of the edge (i, j).

Value

Function writeGP silently returns the passed filename file whereas writeGP returns a grapherator object.

Examples

```
g = graph(0, 100)
g = addNodes(g, n = 25, generator = addNodesGrid)
g = addEdges(g, generator = addEdgesGrid)
g = addWeights(g, generator = addWeightsRandom, method = runif, min = 5, max = 100, to.int = TRUE)
g = addWeights(g, generator = addWeightsRandom, method = runif, min = 10, max = 100, to.int = TRUE)
## Not run:
filename = tempfile()
writeGP(g, file = filename)
g2 = readGP(file = filename)
unlink(filename)
do.call(gridExtra::grid.arrange, c(plot(g), plot(g2), list(nrow = 2)))

## End(Not run)
```

Index