Package 'igraph'

June 16, 2012

Version 0.6

Date Jun 11, 2012

Title Network analysis and visualization

Author Gabor Csardi <csardi.gabor@gmail.com>

Maintainer Gabor Csardi <csardi.gabor@gmail.com>

Description Routines for simple graphs and network analysis. igraph can handle large graphs very well and provides functions for generating random and regular graphs, graph visualization, centrality indices and much more.

Depends stats

Suggests igraphdata, stats4, rgl, tcltk, graph, Matrix, ape

License GPL (>= 2)

URL http://igraph.sourceforge.net

SystemRequirements gmp, libxml2

R topics documented:

igraph-package *The igraph package*

Description

igraph is a library and R package for network analysis.

Introduction

The main goals of the igraph library is to provide a set of data types and functions for 1) pain-free implementation of graph algorithms, 2) fast handling of large graphs, with millions of vertices and edges, 3) allowing rapid prototyping via high level languages like R.

Igraph graphs

Igraph graphs have a class 'igraph'. They are printed to the screen in a special format, here is an example, a ring graph created using graph.ring:

```
IGRAPH U--- 10 10 -- Ring graph
+ attr: name (g/c), mutual (g/x), circular (g/x)
```

The 'IGRAPH' denotes that this is an igraph graph. Then come four bits that denote the kind of the graph: the first is 'U' for undirected and 'D' for directed graphs. The second is 'N' for named graph (i.e. if the graph has the 'name' vertex attribute set). The third is 'W' for weighted graphs (i.e. if the 'weight' edge attribute is set). The fourth is 'B' for bipartite graphs (i.e. if the 'type' vertex attribute is set).

Then comes two numbers, the number of vertices and the number of edges in the graph, and after a double dash, the name of the graph (the 'name' graph attribute) is printed if present. The second line is optional and it contains all the attributes of the graph. This graph has a 'name' graph attribute, of type character, and two other graph attributes called 'mutual' and 'circular', of a complex type. A complex type is simply anything that is not numeric or character. See the documentation of print.igraph for details.

If you want to see the edges of the graph as well, then use the str.igraph function, it is of course enough to type str instead of str.igraph:

```
> str(g)
IGRAPH U--- 10 10 -- Ring graph
+ attr: name (g/c), mutual (g/x), circular (g/x)
+ edges:
 [1] 1-- 2 2-- 3 3-- 4 4-- 5 5-- 6 6-- 7 7-- 8 8-- 9 9--10 1--10
```

Creating graphs

There are many functions in igraph for creating graphs, both deterministic and stochastic; stochastic graph constructors are called 'games'.

To create small graphs with a given structure probably the graph.formula function is easiest. It uses R's formula interface, its manual page contains many examples. Another option is graph, which takes numeric vertex ids directly. graph.atlas creates graph from the Graph Atlas, graph.famous can create some special graphs.

To create graphs from field data, `graph.edgelist`, `graph.data.frame` and `graph.adjacency` are probably the best choices.

The igraph package includes some classic random graphs like the Erdos-Renyi GNP and GNM graphs (`erdos.renyi.game`) and some recent popular models, like preferential attachment (`barabasi.game`) and the small-world model (`watts.strogatz.game`).

Vertex and edge IDs

Vertices and edges have numerical vertex ids in igraph. Vertex ids are always consecutive and they start with one. I.e. for a graph with n vertices the vertex ids are between 1 and n. If some operation changes the number of vertices in the graphs, e.g. a subgraph is created via `induced.subgraph`, then the vertices are renumbered to satisfty this criteria.

The same is true for the edges as well, edge ids are always between one and m, the total number of edges in the graph.

It is often desirable to follow vertices along a number of graph operations, and vertex ids don't allow this because of the renumbering. The solution is to assign attributes to the vertices. These are kept by all operations, if possible. See more about attributes in the next section.

Attributes

In igraph it is possible to assign attributes to the vertices or edges of a graph, or to the graph itself. igraph provides flexible constructs for selecting a set of vertices or edges based on their attribute values, see `get.vertex.attribute` and `iterators` for details.

Some vertex/edge/graph attributes are treated specially. One of them is the 'name' attribute. This is used for printing the graph instead of the numerical ids, if it exists. Vertex names can also be used to specify a vector or set of vertices, in all igraph functions. E.g. `degree` has a `v` argument that gives the vertices for which the degree is calculated. This argument can be given as a character vector of vertex names.

Edges can also have a 'name' attribute, and this is treated specially as well. Just like for vertices, edges can also be selected based on their names, e.g. in the `delete.edges` and other functions.

We note here, that vertex names can also be used to select edges. The form '`from|to`', where '`from`' and '`to`' are vertex names, select a single, possibly directed, edge going from '`from`' to '`to`'. The two forms can also be mixed in the same edge selector.

Other attributes define visualization parameters, see `igraph.plotting` for details.

Attribute values can be set to any R object, but note that storing the graph in some file formats might result the loss of complex attribute values. All attribute values are preserved if you use `save` and `load` to store/retrieve your graphs.

Visualization

igraph provides three different ways for visualization. The first is the `plot.igraph` function. (Actually you don't need to write `plot.igraph`, `plot` is enough. This function uses regular R graphics and can be used with any R device.

The second function is `tkplot`, which uses a Tk GUI for basic interactive graph manipulation. (Tk is quite resource hungry, so don't try this for very large graphs.)

The third way requires the `rgl` package and uses OpenGL. See the `rglplot` function for the details.

Make sure you read `igraph.plotting` before you start plotting your graphs.

File formats

igraph can handle various graph file formats, usually both for reading and writing. We suggest that you use the GraphML file format for your graphs, except if the graphs are too big. For big graphs a simpler format is recommended. See `read.graph` and `write.graph` for details.

Further information

The igraph homepage is at `http://igraph.sourceforge.net`. See especially the documentation section. Join the igraph-help mailing list if you have questions or comments.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

`aging.prefatt.game` *Generate an evolving random graph with preferential attachment and aging*

Description

This function creates a random graph by simulating its evolution. Each time a new vertex is added it creates a number of links to old vertices and the probability that an old vertex is cited depends on its in-degree (preferential attachment) and age.

Usage

```
aging.prefatt.game (n, pa.exp, aging.exp, m = NULL, aging.bin = 300,
     out.dist = NULL, out.seq = NULL, out.pref = FALSE,
     directed = TRUE, zero.deg.appeal = 1, zero.age.appeal = 0,
     deg.coef = 1, age.coef = 1, time.window = NULL)
```

Arguments

`n`	The number of vertices in the graph.
`pa.exp`	The preferantial attachment exponent, see the details below.
`aging.exp`	The exponent of the aging, usually a non-positive number, see details below.
`m`	The number of edges each new vertex creates (except the very first vertex). This argument is used only if both the `out.dist` and `out.seq` arguments are NULL.
`aging.bin`	The number of bins to use for measuring the age of vertices, see details below.
`out.dist`	The discrete distribution to generate the number of edges to add in each time step if `out.seq` is NULL. See details below.
`out.seq`	The number of edges to add in each time step, a vector containing as many elements as the number of vertices. See details below.
`out.pref`	Logical constant, whether to include edges not initiated by the vertex as a basis of preferential attachment. See details below.
`directed`	Logical constant, whether to generate a directed graph. See details below.
`zero.deg.appeal`	The degree-dependent part of the ‘attractiveness’ of the vertices with no adjacent edges. See also details below.

`zero.age.appeal`	The age-dependent part of the ‘attrativeness’ of the vertices with age zero. It is usually zero, see details below.
`deg.coef`	The coefficient of the degree-dependent ‘attractiveness’. See details below.
`age.coef`	The coefficient of the age-dependent part of the ‘attractiveness’. See details below.
`time.window`	Integer constant, if NULL only adjacent added in the last `time.windows` time steps are counted as a basis of the preferential attachment. See also details below.

Details

This is a discrete time step model of a growing graph. We start with a network containing a single vertex (and no edges) in the first time step. Then in each time step (starting with the second) a new vertex is added and it initiates a number of edges to the old vertices in the network. The probability that an old vertex is connected to is proportional to

$$P[i] \sim (c \cdot k_i^\alpha + a)(d \cdot l_i^\beta + b) \cdot$$

Here k_i is the in-degree of vertex i in the current time step and l_i is the age of vertex i. The age is simply defined as the number of time steps passed since the vertex is added, with the extension that vertex age is divided to be in `aging.bin` bins.

c, α, a, d, β and b are parameters and they can be set via the following arguments: `pa.exp` (α, mandatory argument), `aging.exp` (β, mandatory argument), `zero.deg.appeal` (a, optional, the default value is 1), `zero.age.appeal` (b, optional, the default is 0), `deg.coef` (c, optional, the default is 1), and `age.coef` (d, optional, the default is 1).

The number of edges initiated in each time step is governed by the `m`, `out.seq` and `out.pref` parameters. If `out.seq` is given then it is interpreted as a vector giving the number of edges to be added in each time step. It should be of length `n` (the number of vertices), and its first element will be ignored. If `out.seq` is not given (or NULL) and `out.dist` is given then it will be used as a discrete probability distribution to generate the number of edges. Its first element gives the probability that zero edges are added at a time step, the second element is the probability that one edge is added, etc. (`out.seq` should contain non-negative numbers, but if they don’t sum up to 1, they will be normalized to sum up to 1. This behavior is similar to the `prob` argument of the `sample` command.)

By default a directed graph is generated, but it `directed` is set to `FALSE` then an undirected is created. Even if an undirected graph is generaed k_i denotes only the adjacent edges not initiated by the vertex itself except if `out.pref` is set to `TRUE`.

If the `time.window` argument is given (and not NULL) then k_i means only the adjacent edges added in the previous `time.window` time steps.

This function might generate graphs with multiple edges.

Value

A new graph.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

barabasi.game, erdos.renyi.game

Examples

```
# The maximum degree for graph with different aging exponents
g1 <- aging.prefatt.game(10000, pa.exp=1, aging.exp=0, aging.bin=1000)
g2 <- aging.prefatt.game(10000, pa.exp=1, aging.exp=-1,   aging.bin=1000)
g3 <- aging.prefatt.game(10000, pa.exp=1, aging.exp=-3,   aging.bin=1000)
max(degree(g1))
max(degree(g2))
max(degree(g3))
```

alpha.centrality *Find Bonacich alpha centrality scores of network positions*

Description

`alpha.centrality` calculates the alpha centrality of some (or all) vertices in a graph.

Usage

```
alpha.centrality(graph, nodes=V(graph), alpha=1, loops=FALSE,
                   exo=1, weights=NULL, tol=1e-7, sparse=TRUE)
```

Arguments

`graph`	The input graph, can be directed or undirected
`nodes`	Vertex sequence, the vertices for which the alpha centrality values are returned. (For technical reasons they will be calculated for all vertices, anyway.)
`alpha`	Parameter specifying the relative importance of endogenous versus exogenous factors in the determination of centrality. See details below.
`loops`	Whether to eliminate loop edges from the graph before the calculation.
`exo`	The exogenous factors, in most cases this is either a constant – the same factor for every node, or a vector giving the factor for every vertex. Note that too long vectors will be truncated and too short vectors will be replicated to match the number of vertices.
`weights`	A character scalar that gives the name of the edge attribute to use in the adjacency matrix. If it is `NULL`, then the 'weight' edge attribute of the graph is used, if there is one. Otherwise, or if it is `NA`, then the calculation uses the standard adjacency matrix.
`tol`	Tolerance for near-singularities during matrix inversion, see `solve`.
`sparse`	Logical scalar, whether to use sparse matrices for the calculation. The 'Matrix' package is required for sparse matrix support

Details

The alpha centrality measure can be considered as a generalization of eigenvector centerality to directed graphs. It was proposed by Bonacich in 2001 (see reference below).

The alpha centrality of the vertices in a graph is defined as the solution of the following matrix equation:

$$x = \alpha A^T x + e,$$

where A is the (not neccessarily symmetric) adjacency matrix of the graph, e is the vector of exogenous sources of status of the vertices and α is the relative importance of the endogenous versus exogenous factors.

Value

A numeric vector contaning the centrality scores for the selected vertices.

Warning

Singular adjacency matrices cause problems for this algorithm, the routine may fail is certain cases.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Bonacich, P. and Paulette, L. (2001). "Eigenvector-like measures of centrality for asymmetric relations" *Social Networks*, 23, 191-201.

See Also

evcent and bonpow

Examples

```
# The examples from Bonacich's paper
g.1 <- graph( c(1,3,2,3,3,4,4,5) )
g.2 <- graph( c(2,1,3,1,4,1,5,1) )
g.3 <- graph( c(1,2,2,3,3,4,4,1,5,1) )
alpha.centrality(g.1)
alpha.centrality(g.2)
alpha.centrality(g.3,alpha=0.5)
```

---

arpack *ARPACK eigenvector calculation*

Description

Interface to the ARPACK library for calculating eigenvectors of sparse matrices

Usage

```
arpack(func, extra = NULL, sym = FALSE, options = igraph.arpack.default,
    env = parent.frame(), complex=!sym)
arpack.unpack.complex(vectors, values, nev)
```

Arguments

`func`	The function to perform the matrix-vector multiplication. ARPACK requires to perform these by the user. The function gets the vector x as the first argument, and it should return Ax, where A is the “input matrix”. (The input matrix is never given explicitly.) The second argument is `extra`.
`extra`	Extra argument to supply to `func`.
`sym`	Logical scalar, whether the input matrix is symmetric. Always supply `TRUE` here if it is, since it can speed up the computation.
`options`	Options to ARPACK, a named list to overwrite some of the default option values. See details below.
`env`	The environment in which `func` will be evaluated.
`complex`	Whether to convert the eigenvectors returned by ARPACK into R complex vectors. By default this is not done for symmetric problems (these only have real eigenvectors/values), but only non-symmetric ones. If you have a non-symmetric problem, but you're sure that the results will be real, then supply `FALSE` here. The conversion is done by calling `arpack.unpack.complex`.
`vectors`	Eigenvectors, as returned by ARPACK.
`values`	Eigenvalues, as returned by ARPACK
`nev`	The number of eigenvectors/values to extract. This can be less than or equal to the number of eigenvalues requested in the original ARPACK call.

Details

ARPACK is a library for solving large scale eigenvalue problems. The package is designed to compute a few eigenvalues and corresponding eigenvectors of a general n by n matrix A. It is most appropriate for large sparse or structured matrices A where structured means that a matrix-vector product `w <- Av` requires order n rather than the usual order n^2 floating point operations. Please see `http://www.caam.rice.edu/software/ARPACK/` for details.

This function is an interface to ARPACK. igraph does not contain all ARPACK routines, only the ones dealing with symmetric and non-symmetric eigenvalue problems using double precision real numbers.

The eigenvalue calculation in ARPACK (in the simplest case) involves the calculation of the Av product where A is the matrix we work with and v is an arbitrary vector. The function supplied in the `fun` argument is expected to perform this product. If the product can be done efficiently, e.g. if the matrix is sparse, then `arpack` is usually able to calculate the eigenvalues very quickly.

The `options` argument specifies what kind of calculation to perform. It is a list with the following members, they correspond directly to ARPACK parameters. On input it has the following fields:

bmat Character constant, possible values: ‘`I`’, stadard eigenvalue problem, $Ax = \lambda x$; and ‘`G`’, generalized eigenvalue problem, $Ax = \lambda Bx$. Currently only ‘`I`’ is supported.

n Numeric scalar. The dimension of the eigenproblem. You only need to set this if you call `arpack` directly. (I.e. not needed for `evcent`, `page.rank`, etc.)

which Specify which eigenvalues/vectors to compute, character constant with exactly two characters.

Possible values for symmetric input matrices:

- **'LA'** Compute `nev` largest (algebraic) eigenvalues.
- **'SA'** Compute `nev` smallest (algebraic) eigenvalues.
- **'LM'** Compute `nev` largest (in magnitude) eigenvalues.
- **'SM'** Compute `nev` smallest (in magnitude) eigenvalues.
- **'BE'** Compute `nev` eigenvalues, half from each end of the spectrum. When `nev` is odd, compute one more from the high end than from the low end.

Possible values for non-symmetric input matrices:

- **'LM'** Compute `nev` eigenvalues of largest magnitude.
- **'SM'** Compute `nev` eigenvalues of smallest magnitude.
- **'LR'** Compute `nev` eigenvalues of largest real part.
- **'SR'** Compute `nev` eigenvalues of smallest real part.
- **'LI'** Compute `nev` eigenvalues of largest imaginary part.
- **'SI'** Compute `nev` eigenvalues of smallest imaginary part.

This parameter is sometimes overwritten by the various functions, e.g. `page.rank` always sets 'LM'.

nev Numeric scalar. The number of eigenvalues to be computed.

tol Numeric scalar. Stopping criterion: the relative accuracy of the Ritz value is considered acceptable if its error is less than `tol` times its estimated value. If this is set to zero then machine precision is used.

ncv Number of Lanczos vectors to be generated.

ldv Numberic scalar. It should be set to zero in the current implementation.

ishift Either zero or one. If zero then the shifts are provided by the user via reverse communication. If one then exact shifts with respect to the reduced tridiagonal matrix T. Please always set this to one.

maxiter Maximum number of Arnoldi update iterations allowed.

nb Blocksize to be used in the recurrence. Please always leave this on the default value, one.

mode The type of the eigenproblem to be solved. Possible values if the input matrix is symmetric:

- **1** $Ax = \lambda x$, A is symmetric.
- **2** $Ax = \lambda Mx$, A is symmetric, M is symmetric positive definite.
- **3** $Kx = \lambda Mx$, K is symmetric, M is symmetric positive semi-definite.
- **4** $Kx = \lambda KGx$, K is symmetric positive semi-definite, KG is symmetric indefinite.
- **5** $Ax = \lambda Mx$, A is symmetric, M is symmetric positive semi-definite. (Cayley transformed mode.)

Please note that only `mode==1` was tested and other values might not work properly.

Possible values if the input matrix is not symmetric:

- **1** $Ax = \lambda x$.
- **2** $Ax = \lambda Mx$, M is symmetric positive definite.
- **3** $Ax = \lambda Mx$, M is symmetric semi-definite.
- **4** $Ax = \lambda Mx$, M is symmetric semi-definite.

Please note that only `mode==1` was tested and other values might not work properly.

start Not used currently. Later it be used to set a starting vector.

sigma Not used currently.

sigmai Not use currently.

On output the following additional fields are added:

info Error flag of ARPACK. Possible values:

- **0** Normal exit.
- **1** Maximum number of iterations taken.
- **3** No shifts could be applied during a cycle of the Implicitly restarted Arnoldi iteration. One possibility is to increase the size of `ncv` relative to `nev`.

ARPACK can return more error conditions than these, but they are converted to regular igraph errors.

iter Number of Arnoldi iterations taken.

nconv Number of "converged" Ritz values. This represents the number of Ritz values that satisfy the convergence critetion.

numop Total number of matrix-vector multiplications.

numopb Not used currently.

numreo Total number of steps of re-orthogonalization.

Please see the ARPACK documentation for additional details.

`arpack.unpack.complex` is a (semi-)internal function that converts the output of the non-symmetric ARPACK solver to a more readable format. It is called internally by `arpack`.

Value

A named list with the following members:

`values`	Numeric vector, the desired eigenvalues.
`vectors`	Numeric matrix, the desired eigenvectors as columns. If `complex=TRUE` (the default for non-symmetric problems), then the matrix is complex.
`options`	A named list with the supplied `options` and some information about the performed calculation, including an ARPACK exit code. See the details above.

Author(s)

Rich Lehoucq, Kristi Maschhoff, Danny Sorensen, Chao Yang for ARPACK, Gabor Csardi `<csardi.gabor@gmail.com>` for the R interface.

References

D.C. Sorensen, Implicit Application of Polynomial Filters in a k-Step Arnoldi Method. *SIAM J. Matr. Anal. Apps.*, 13 (1992), pp 357-385.

R.B. Lehoucq, Analysis and Implementation of an Implicitly Restarted Arnoldi Iteration. *Rice University Technical Report* TR95-13, Department of Computational and Applied Mathematics.

B.N. Parlett & Y. Saad, Complex Shift and Invert Strategies for Real Matrices. *Linear Algebra and its Applications*, vol 88/89, pp 575-595, (1987).

See Also

`evcent`, `page.rank`, `hub.score`, `leading.eigenvector.community` are some of the functions in igraph which use ARPACK. The ARPACK homepage is at `http://www.caam.rice.edu/software/ARPACK/`.

Examples

```
# Identity matrix
f <- function(x, extra=NULL) x
arpack(f, options=list(n=10, nev=2, ncv=4), sym=TRUE)

# Graph laplacian of a star graph (undirected), n>=2
# Note that this is a linear operation
f <- function(x, extra=NULL) {
  y <- x
  y[1] <- (length(x)-1)*x[1] - sum(x[-1])
  for (i in 2:length(x)) {
    y[i] <- x[i] - x[1]
  }
  y
}

arpack(f, options=list(n=10, nev=1, ncv=3), sym=TRUE)

# double check
eigen(graph.laplacian(graph.star(10, mode="undirected")))
```

`articulation.points` *Articulation points of a graph*

Description

Articuation points or cut vertices are vertices whose removal increases the number of connected components in a graph.

Usage

```
articulation.points(graph)
```

Arguments

`graph`	The input graph. It is treated as an undirected graph, even if it is directed.

Details

Articuation points or cut vertices are vertices whose removal increases the number of connected components in a graph. If the original graph was connected, then the removal of a single articulation point makes it undirected. If a graph contains no articulation points, then its vertex connectivity is at least two.

Value

A numeric vector giving the vertex ids of the articulation points of the input graph.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

biconnected.components, clusters, is.connected, vertex.connectivity

Examples

```
g <- graph.disjoint.union( graph.full(5), graph.full(5) )
clu <- clusters(g)$membership
g <- add.edges(g, c(match(1, clu), match(2, clu)) )
articulation.points(g)
```

---

as.directed *Convert between directed and undirected graphs*

Description

as.directed converts an undirected graph to directed, as.undirected does the opposite, it converts a directed graph to undirected.

Usage

```
as.directed(graph, mode = c("mutual", "arbitrary"))
as.undirected(graph, mode = c("collapse", "each", "mutual"),
       edge.attr.comb = getIgraphOpt("edge.attr.comb"))
```

Arguments

graph	The graph to convert.
mode	Character constant, defines the conversion algorithm. For as.directed it can be mutual or arbitrary. For as.undirected it can be each, collapse or mutual. See details below.
edge.attr.comb	Specifies what to do with edge attributes, if mode="collapse" or mode="mutual". In these cases many edges might be mapped to a single one in the new graph, and their attributes are combined. Please see attribute.combination for details on this.

Details

Conversion algorithms for as.directed:

arbitrary The number of edges in the graph stays the same, an arbitrarily directed edge is created for each undirected edge.

mutual Two directed edges are created for each undirected edge, one in each direction.

Conversion algorithms for as.undirected:

each The number of edges remains constant, an undirected edge is created for each directed one, this version might create graphs with multiple edges.

collapse One undirected edge will be created for each pair of vertices which are connected with at least one directed edge, no multiple edges will be created.

mutual One undirected edge will be created for each pair of mutual edges. Non-mutual edges are ignored. This mode might create multiple edges if there are more than one mutual edge pairs between the same pair of vertices.

Value

A new graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

simplify for removing multiple and/or loop edges from a graph.

Examples

```
g <- graph.ring(10)
as.directed(g, "mutual")
g2 <- graph.star(10)
as.undirected(g)

# Combining edge attributes
g3 <- graph.ring(10, directed=TRUE, mutual=TRUE)
E(g3)$weight <- seq_len(ecount(g3))
ug3 <- as.undirected(g3)
print(ug3, e=TRUE)
## Not run:
  x11(width=10, height=5)
  layout(rbind(1:2))
  plot( g3, layout=layout.circle, edge.label=E(g3)$weight)
  plot(ug3, layout=layout.circle, edge.label=E(ug3)$weight)

## End(Not run)

g4 <- graph(c(1,2, 3,2,3,4,3,4, 5,4,5,4,
              6,7, 7,6,7,8,7,8, 8,7,8,9,8,9,
              9,8,9,8,9,9, 10,10,10,10))
E(g4)$weight <- seq_len(ecount(g4))
ug4 <- as.undirected(g4, mode="mutual",
              edge.attr.comb=list(weight=length))
print(ug4, e=TRUE)
```

---

as.igraph *Conversion to igraph*

Description

These fucntions convert various objects to igraph graphs.

Usage

```
## S3 method for class 'igraphHRG'
as.igraph(x, ...)
```

Arguments

`x`	The object to convert.
`...`	Additional arguments. None currently.

Details

You can use `as.igraph` to convert various objects to igraph graphs. Right now the following objects are supported:

- codeigraphHRG These objects are created by the `hrg.fit` and `hrg.consensus` functions.

Value

All these functions return an igraph graph.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>.

Examples

```
g <- graph.full(5) + graph.full(5)
hrg <- hrg.fit(g)
as.igraph(hrg)
```

`assortativity` *Assortativity coefficient*

Description

The assortativity coefficient is positive is similar vertices (based on some external property) tend to connect to each, and negative otherwise.

Usage

```
assortativity (graph, types1, types2 = NULL, directed = TRUE)
assortativity.nominal (graph, types, directed = TRUE)
assortativity.degree (graph, directed = TRUE)
```

Arguments

`graph`	The input graph, it can be directed or undirected.
`types`	Vector giving the vertex types. They as assumed to be integer numbers, starting with one. Non-integer values are converted to integers with `as.integer`.
`types1`	The vertex values, these can be arbitrary numeric values.
`types2`	A second value vector to be using for the incoming edges when calculating assortativity for a directed graph. Supply `NULL` here if you want to use the same values for outgoing and incoming edges. This argument is ignored (with a warning) if it is not `NULL` and undirected assortativity coefficient is being calculated.
`directed`	Logical scalar, whether to consider edge directions for directed graphs. This argument is ignored for undirected graphs. Supply `TRUE` here to do the natural thing, i.e. use directed version of the measure for directed graphs and the undirected version for undirected graphs.

Details

The assortativity coefficient measures the level of homophyly of the graph, based on some vertex labeling or values assigned to vertices. If the coefficient is high, that means that connected vertices tend to have the same labels or similar assigned values.

M.E.J. Newman defined two kinds of assortativity coefficients, the first one is for categorical labels of vertices. `assortativity.nominal` calculates this measure. It is defines as

$$r = \frac{\sum_i e_{ii} - \sum_i a_i b_i}{1 - \sum_i a_i b_i}$$

where e_{ij} is the fraction of edges connecting vertices of type i and j, $a_i = \sum_j e_{ij}$ and $b_j = \sum_i e_{ij}$.

The second assortativity variant is based on values assigned to the vertices. `assortativity` calculates this measure. It is defined as

$$r = \frac{1}{\sigma_q^2} \sum_{jk} jk(e_{jk} - q_j q_k)$$

for undirected graphs ($q_i = \sum_j e_{ij}$) and as

$$r = \frac{1}{\sigma_o \sigma_i} \sum_{jk} jk(e_{jk} - q_j^o q_k^i)$$

for directed ones. Here $q_i^o = \sum_j e_{ij}$, $q_i^i = \sum_j e_{ji}$, moreover, σ_q, $sigma_o$ and $sigma_i$ are the standard deviations of q, q^o and q^i, respectively.

The reason of the difference is that in directed networks the relationship is not symmetric, so it is possible to assign different values to the outgoing and the incoming end of the edges.

`assortativity.degree` uses vertex degree (minus one) as vertex values and calls `assortativity`.

Value

A single real number.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

M. E. J. Newman: Mixing patterns in networks, *Phys. Rev. E* 67, 026126 (2003) http://arxiv.org/abs/cond-mat/0209450

M. E. J. Newman: Assortative mixing in networks, *Phys. Rev. Lett.* 89, 208701 (2002) http://arxiv.org/abs/cond-mat/0205405/

Examples

```
# random network, close to zero
assortativity.degree(erdos.renyi.game(10000,3/10000))

# BA model, tends to be dissortative
assortativity.degree(ba.game(10000, m=4))
```

`attributes` *Graph, vertex and edge attributes*

Description

Attributes are associated values belonging to a graph, vertices or edges. These can represent some property, like data about how the graph was constructed, the color of the vertices when the graph is plotted, or simply the weights of the edges in a weighted graph.

Usage

```
get.graph.attribute(graph, name)
set.graph.attribute(graph, name, value)
remove.graph.attribute(graph, name)
get.vertex.attribute(graph, name, index=V(graph))
set.vertex.attribute(graph, name, index=V(graph), value)
remove.vertex.attribute(graph, name)
get.edge.attribute(graph, name, index=E(graph))
set.edge.attribute(graph, name, index=E(graph), value)
remove.edge.attribute(graph, name)
```

Arguments

`graph`	The graph object to work on. Note that the original graph is never modified, a new graph object is returned instead; if you don't assign it to a variable your modifications will be lost! See examples below.
`name`	Character constant, the name of the attribute.
`index`	Numeric vector, the ids of the vertices or edges. It is not recycled, even if `value` is longer.
`value`	Numeric vector, the new value(s) of the attributes, it will be recycled if needed.

Details

There are three types of attributes in igraph: graph, vertex and edge attributes. Graph attributes are associated with graph, vertex attributes with vertices and edge attributes with edges.

Examples for graph attributes are the date when the graph data was collected or other types of memos like the type of the data, or whether the graph is a simple graph, ie. one without loops and multiple edges.

Examples of vertex attributes are vertex properties, like the vertex coordinates for the visualization of the graph, or other visualization parameters, or meta-data associated with the vertices, like the gender and the age of the individuals in a friendship network, the type of the neurons in a graph representing neural circuitry or even some pre-computed structual properties, like the betweenness centrality of the vertices.

Examples of edge attributes are data associated with edges: most commonly edge weights, or visualization parameters.

In recent igraph versions, arbitrary R objects can be assigned as graph, vertex or edge attributes.

Some igraph functions use the values or graph, vertex and edge attributes if they are present but this is not done in the current version very extensively. Expect more in the (near) future.

Graph attributes can be created with the `set.graph.attribute` function, and removed with `remove.graph.attribute`. Graph attributes are queried with `get.graph.attribute` and the assigned graph attributes are listed with `list.graph.attributes`.

There is a simpler notation for using graph attributes: the '`$`' operator. It can be used both to query and set graph attributes, see an example below.

The functions for vertex attributes are `set.vertex.attribute`, `get.vertex.attribute`, `remove.vertex.attribute` and `list.vertex.attributes` and for edge attributes they are `set.edge.attribute`, `get.edge.attribute`, `remove.edge.attribute` and `list.edge.attributes`.

There is however a (syntactically) much simpler way to handle vertex and edge attribute by using vertex and edge selectors, it works like this: `V(g)` selects all vertices in a graph, and `V(g)$name` queries the `name` attribute for all vertices. Similarly is `vs` is a vertex set `vs$name` gives the values of the `name` attribute for the vertices in the vertex set.

This form can also be used to set the values of the attributes, like the regular R convention:

```
V(g)$color <- "red"
```

It works for vertex subsets as well:

```
V(g)[1:5]$color <- "green"
```

The notation for edges is similar: `E(g)` means all edges `E(g)$weight` is the `weight` attribute for all edges, etc.

See also the manual page for `iterators` about how to create various vertex and edge sets.

Value

`get.graph.attribute`, `get.vertex.attribute` and `get.edge.attribute` return an R object, or a list of R objects if attributes of more vertices/edges are requested.

`set.graph.attribute`, `set.vertex.attribute`, `set.edge.attribute`, and also `remove.graph.attribute`, `remove.vertex.attribute` and `remove.edge.attribute` return a new graph object with the updates/removes performed.

`list.graph.attributes`, `list.vertex.attributes` and `list.edge.attributes` return a character vector, the names of the attributes present.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`print.igraph` can print attributes. See `attribute.combination` for details on how igraph combines attributes if several vertices or edges are mapped into one.

Examples

```
g <- graph.ring(10)
g <- set.graph.attribute(g, "name", "RING")
# It is the same as
g$name <- "RING"
g$name

g <- set.vertex.attribute(g, "color", value=c("red", "green"))
```

```
get.vertex.attribute(g, "color")

g <- set.edge.attribute(g, "weight", value=runif(ecount(g)))
get.edge.attribute(g, "weight")

# The following notation is more convenient

g <- graph.star(10)

V(g)$color <- c("red", "green")
V(g)$color

E(g)$weight <- runif(ecount(g))
E(g)$weight

print(g, g=TRUE, v=TRUE, e=TRUE)
```

autocurve.edges *Optimal edge curvature when plotting graphs*

Description

If graphs have multiple edges, then drawing them as straight lines does not show them when plotting the graphs; they will be on top of each other. One solution is to bend the edges, with diffenent curvature, so that all of them are visible.

Usage

```
autocurve.edges (graph, start = 0.5)
```

Arguments

`graph`	The input graph.
`start`	The curvature at the two extreme edges. All edges will have a curvature between `-start` and `start`, spaced equally.

Details

`autocurve.edges` calculates the optimal `edge.curved` vector for plotting a graph with multiple edges, so that all edges are visible.

Value

A numeric vector, its length is the number of edges in the graph.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`igraph.plotting` for all plotting parameters, `plot.igraph`, `tkplot` and `rglplot` for plotting functions.

Examples

```
g <- graph( c(0,1,1,0,1,2,1,3,1,3,1,3,
              2,3,2,3,2,3,2,3,0,1)+1 )

autocurve.edges(g)

## Not run:
set.seed(42)
plot(g)

## End(Not run)
```

---

barabasi.game *Generate scale-free graphs according to the Barabasi-Albert model*

Description

The BA-model is a very simple stochastic algorithm for building a graph.

Usage

```
barabasi.game(n, power = 1, m = NULL, out.dist = NULL, out.seq = NULL,
    out.pref = FALSE, zero.appeal = 1, directed = TRUE,
    algorithm = c("psumtree", "psumtree-multiple", "bag"),
    start.graph = NULL)
```

Arguments

`n`	Number of vertices.
`power`	The power of the preferential attachment, the default is one, ie. linear preferential attachment.
`m`	Numeric constant, the number of edges to add in each time step This argument is only used if both `out.dist` and `out.seq` are omitted or NULL.
`out.dist`	Numeric vector, the distribution of the number of edges to add in each time step. This argument is only used if the `out.seq` argument is omitted or NULL.
`out.seq`	Numeric vector giving the number of edges to add in each time step. Its first element is ignored as no edges are added in the first time step.
`out.pref`	Logical, if true the total degree is used for calculating the citation probability, otherwise the in-degree is used.
`zero.appeal`	The 'attractiveness' of the vertices with no adjacent edges. See details below.
`directed`	Whether to create a directed graph.
`algorithm`	The algorithm to use for the graph generation. `psumtree` uses a partial prefix-sum tree to generate the graph, this algorithm can handle any `power` and `zero.appeal` values and never generates multiple edges. `psumtree-multiple` also uses a partial prefix-sum tree, but the generation of multiple edges is allowed. Before the 0.6 version igraph used this algorithm if `power` was not one, or `zero.appeal` was not one. `bag` is the algorithm that was previously (before version 0.6) used if `power` was one and `zero.appeal` was one as well. It works by putting the ids

of the vertices into a bag (mutliset, really), exactly as many times as their (in-)degree, plus once more. Then the required number of cited vertices are drawn from the bag, with replacement. This method might generate multiple edges. It only works if `power` and `zero.appeal` are equal one.

`start.graph` `NULL` or an igraph graph. If a graph, then the supplied graph is used as a starting graph for the preferential attachment algorithm. The graph should have at least one vertex. If a graph is supplied here and the `out.seq` argument is not `NULL`, then it should contain the out degrees of the new vertices only, not the ones in the `start.graph`.

Details

This is a simple stochastic algorithm to generate a graph. It is a discrete time step model and in each time step a single vertex is added.

We start with a single vertex and no edges in the first time step. Then we add one vertex in each time step and the new vertex initiates some edges to old vertices. The probability that an old vertex is chosen is given by

$$P[i] \sim k_i^\alpha + a$$

where k_i is the in-degree of vertex i in the current time step (more precisely the number of adjacent edges of i which were not initiated by i itself) and α and a are parameters given by the `power` and `zero.appeal` arguments.

The number of edges initiated in a time step is given by the `m`, `out.dist` and `out.seq` arguments. If `out.seq` is given and not NULL then it gives the number of edges to add in a vector, the first element is ignored, the second is the number of edges to add in the second time step and so on. If `out.seq` is not given or null and `out.dist` is given and not NULL then it is used as a discrete distribution to generate the number of edges in each time step. Its first element is the probability that no edges will be added, the second is the probability that one edge is added, etc. (`out.dist` does not need to sum up to one, it normalized automatically.) `out.dist` should contain non-negative numbers and at east one element should be positive.

If both `out.seq` and `out.dist` are omitted or NULL then `m` will be used, it should be a positive integer constant and `m` edges will be added in each time step.

`barabasi.game` generates a directed graph by default, set `directed` to `FALSE` to generate an undirected graph. Note that even if an undirected graph is generated k_i denotes the number of adjacent edges not initiated by the vertex itself and not the total (in- + out-) degree of the vertex, unless the `out.pref` argument is set to `TRUE`.

Value

A graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Barabasi, A.-L. and Albert R. 1999. Emergence of scaling in random networks *Science*, 286 509–512.

See Also

`random.graph.game`

Examples

```
g <- barabasi.game(10000)
degree.distribution(g)
```

`betweenness` *Vertex and edge betweenness centrality*

Description

The vertex and edge betweenness are (roughly) defined by the number of geodesics (shortest paths) going through a vertex or an edge.

Usage

```
betweenness(graph, v=V(graph), directed = TRUE, weights = NULL,
     nobigint = TRUE, normalized = FALSE)
edge.betweenness(graph, e=E(graph), directed = TRUE, weights = NULL)
betweenness.estimate(graph, vids = V(graph), directed = TRUE, cutoff,
     weights = NULL, nobigint = TRUE)
edge.betweenness.estimate(graph, e=E(graph),
     directed = TRUE, cutoff, weights = NULL)
```

Arguments

`graph`	The graph to analyze.
`v`	The vertices for which the vertex betweenness will be calculated.
`e`	The edges for which the edge betweenness will be calculated.
`directed`	Logical, whether directed paths should be considered while determining the shortest paths.
`weights`	Optional positive weight vector for calculating weighted betweenness. If the graph has a `weight` edge attribute, then this is used by default.
`nobigint`	Logical scalar, whether to use big integers during the calculation. This is only required for lattice-like graphs that have very many shortest paths between a pair of vertices. If `TRUE` (the default), then big integers are not used.
`normalized`	Logical scalar, whether to normalize the betweenness scores. If `TRUE`, then the results are normalized according to $$B^n = \frac{2B}{n^2 - 3n + 2}$$, where B^n is the normalized, B the raw betweenness, and n is the number of vertices in the graph.
`vids`	The vertices for which the vertex betweenness estimation will be calculated.
`cutoff`	The maximum path length to consider when calculating the betweenness. If zero or negative then there is no such limit.

Details

The vertex betweenness of vertex v is defined by

$$\sum_{i \neq j, i \neq v, j \neq v} g_{ivj} / g_{ij}$$

The edge betweenness of edge e is defined by

$$\sum_{i \neq j} giej / g_{ij}.$$

`betweenness` calculates vertex betweenness, `edge.betweenness` calculates edge.betweenness.

`betweenness.estimate` only considers paths of length `cutoff` or smaller, this can be run for larger graphs, as the running time is not quadratic (if `cutoff` is small). If `cutoff` is zero or negative then the function calculates the exact betweenness scores.

`edge.betweenness.estimate` is similar, but for edges.

For calculating the betweenness a similar algorithm to the one proposed by Brandes (see References) is used.

Value

A numeric vector with the betweenness score for each vertex in `v` for `betweenness`.

A numeric vector with the edge betweenness score for each edge in `e` for `edge.betweenness`.

`betweenness.estimate` returns the estimated betweenness scores for vertices in `vids`, `edge.betweenness.estimate` the estimated edge betweenness score for *all* edges; both in a numeric vector.

Note

`edge.betweenness` might give false values for graphs with multiple edges.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Freeman, L.C. (1979). Centrality in Social Networks I: Conceptual Clarification. *Social Networks*, 1, 215-239.

Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology* 25(2):163-177, 2001.

See Also

`closeness`, `degree`

Examples

```
g <- random.graph.game(10, 3/10)
betweenness(g)
edge.betweenness(g)
```

`biconnected.components` *Biconnected components*

Description

Finding the biconnected components of a graph

Usage

```
biconnected.components(graph)
```

Arguments

`graph`	The input graph. It is treated as an undirected graph, even if it is directed.

Details

A graph is biconnected if the removal of any single vertex (and its adjacent edges) does not disconnect it.

A biconnected component of a graph is a maximal biconnected subgraph of it. The biconnected components of a graph can be given by the partition of its edges: every edge is a member of exactly one biconnected component. Note that this is not true for vertices: the same vertex can be part of many biconnected components.

Value

A named list with three components:

`no`	Numeric scalar, an integer giving the number of biconnected components in the graph.
`tree_edges`	The components themselves, a list of numeric vectors. Each vector is a set of edge ids giving the edges in a biconnected component. These edges define a spanning tree of the component.
`component_edges`	A list of numeric vectors. It gives all edges in the components.
`components`	A list of numeric vectors, the vertices of the components.
`articulation_points`	The articulation points of the graph. See `articulation.points`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`articulation.points`, `clusters`, `is.connected`, `vertex.connectivity`

Examples

```
g <- graph.disjoint.union( graph.full(5), graph.full(5) )
clu <- clusters(g)$membership
g <- add.edges(g, c(which(clu==1), which(clu==2)))
bc <- biconnected.components(g)
```

`bipartite.mapping` *Decide whether a graph is bipartite*

Description

This function decides whether the vertices of a network can be mapped to two vertex types in a way that no vertices of the same type are connected.

Usage

```
bipartite.mapping(graph)
```

Arguments

`graph`	The input graph.

Details

A bipartite graph in igraph has a '`type`' vertex attribute giving the two vertex types.

This function simply checks whether a graph *could* be bipartite. It tries to find a mapping that gives a possible division of the vertices into two classes, such that no two vertices of the same class are connected by an edge.

The existence of such a mapping is equivalent of having no circuits of odd length in the graph. A graph with loop edges cannot bipartite.

Note that the mapping is not necessarily unique, e.g. if the graph has at least two components, then the vertices in the separate components can be mapped independently.

Value

A named list with two elements:

`res`	A logical scalar, `TRUE` if the can be bipartite, `FALSE` otherwise.
`type`	A possibly vertex type mapping, a logical vector. If no such mapping exists, then an empty vector.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
## A ring has just one loop, so it is fine
g <- graph.ring(10)
bipartite.mapping(g)

## A star is fine, too
g2 <- graph.star(10)
bipartite.mapping(g2)

## A graph containing a triangle is not fine
g3 <- graph.ring(10)
g3 <- add.edges(g3, c(1,3))
bipartite.mapping(g3)
```

`bipartite.projection` *Project a bipartite graph*

Description

A bipartite graph is projected into two one-mode networks

Usage

```
bipartite.projection.size(graph, types = NULL)
bipartite.projection (graph, types = NULL, multiplicity = TRUE,
       probe1 = NULL)
```

Arguments

`graph`	The input graph. It can be directed, but edge directions are ignored during the computation.
`types`	An optional vertex type vector to use instead of the 'type' vertex attribute. You must supply this argument if the graph has no 'type' vertex attribute.
`multiplicity`	If `TRUE`, then igraph keeps the multiplicity of the edges as an edge attribute. E.g. if there is an A-C-B and also an A-D-B triple in the bipartite graph (but no more X, such that A-X-B is also in the graph), then the multiplicity of the A-B edge in the projection will be 2.
`probe1`	This argument can be used to specify the order of the projections in the resulting list. If given, then it is considered as a vertex id (or a symbolic vertex name); the projection containing this vertex will be the first one in the result list.

Details

Bipartite graphs have a `type` vertex attribute in igraph, this is boolean and `FALSE` for the vertices of the first kind and `TRUE` for vertices of the second kind.

`bipartite.projection.size` calculates the number of vertices and edges in the two projections of the bipartite graphs, without calculating the projections themselves. This is useful to check how much memory the projections would need if you have a large bipartite graph.

`bipartite.projections` calculates the actual projections. You can use the `probe1` argument to specify the order of the projections in the result. By default vertex type `FALSE` is the first and `TRUE` is the second.

`bipartite.projections` keeps vertex attributes.

Value

A list of two undirected graphs. See details above.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
## Projection of a full bipartite graph is a full graph
g <- graph.full.bipartite(10,5)
proj <- bipartite.projection(g)
graph.isomorphic(proj[[1]], graph.full(10))
graph.isomorphic(proj[[2]], graph.full(5))

## The projection keeps the vertex attributes
M <- matrix(0, nr=5, nc=3)
rownames(M) <- c("Alice", "Bob", "Cecil", "Dan", "Ethel")
colnames(M) <- c("Party", "Skiing", "Badminton")
M[] <- sample(0:1, length(M), replace=TRUE)
M
g2 <- graph.incidence(M)
g2$name <- "Event network"
proj2 <- bipartite.projection(g2)
print(proj2[[1]], g=TRUE, e=TRUE)
print(proj2[[2]], g=TRUE, e=TRUE)
```

---

bonpow — *Find Bonacich Power Centrality Scores of Network Positions*

Description

bonpow takes a graph (dat) and returns the Boncich power centralities of positions (selected by nodes). The decay rate for power contributions is specified by exponent (1 by default).

Usage

```
bonpow(graph, nodes=V(graph), loops=FALSE, exponent=1,
    rescale=FALSE, tol=1e-7, sparse=TRUE)
```

Arguments

graph	the input graph.
nodes	vertex sequence indicating which vertices are to be included in the calculation. By default, all vertices are included.
loops	boolean indicating whether or not the diagonal should be treated as valid data. Set this true if and only if the data can contain loops. loops is FALSE by default.
exponent	exponent (decay rate) for the Bonacich power centrality score; can be negative
rescale	if true, centrality scores are rescaled such that they sum to 1.
tol	tolerance for near-singularities during matrix inversion (see solve)
sparse	Logical scalar, whether to use sparse matrices for the calculation. The 'Matrix' package is required for sparse matrix support

Details

Bonacich's power centrality measure is defined by $C_{BP}(\alpha, \beta) = \alpha (\mathbf{I} - \beta \mathbf{A})^{-1} \mathbf{A} \mathbf{1}$, where β is an attenuation parameter (set here by `exponent`) and $\mathbf{A}$ is the graph adjacency matrix. (The coefficient α acts as a scaling parameter, and is set here (following Bonacich (1987)) such that the sum of squared scores is equal to the number of vertices. This allows 1 to be used as a reference value for the "middle" of the centrality range.) When $\beta \to 1/\lambda_{\mathbf{A}1}$ (the reciprocal of the largest eigenvalue of $\mathbf{A}$), this is to within a constant multiple of the familiar eigenvector centrality score; for other values of β, the behavior of the measure is quite different. In particular, β gives positive and negative weight to even and odd walks, respectively, as can be seen from the series expansion $C_{BP}(\alpha, \beta) = \alpha \sum_{k=0}^{\infty} \beta^k \mathbf{A}^{k+1} \mathbf{1}$ which converges so long as $|\beta| < 1/\lambda_{\mathbf{A}1}$. The magnitude of β controls the influence of distant actors on ego's centrality score, with larger magnitudes indicating slower rates of decay. (High rates, hence, imply a greater sensitivity to edge effects.)

Interpretively, the Bonacich power measure corresponds to the notion that the power of a vertex is recursively defined by the sum of the power of its alters. The nature of the recursion involved is then controlled by the power exponent: positive values imply that vertices become more powerful as their alters become more powerful (as occurs in cooperative relations), while negative values imply that vertices become more powerful only as their alters become *weaker* (as occurs in competitive or antagonistic relations). The magnitude of the exponent indicates the tendency of the effect to decay across long walks; higher magnitudes imply slower decay. One interesting feature of this measure is its relative instability to changes in exponent magnitude (particularly in the negative case). If your theory motivates use of this measure, you should be very careful to choose a decay parameter on a non-ad hoc basis.

Value

A vector, containing the centrality scores.

Warning

Singular adjacency matrices cause no end of headaches for this algorithm; thus, the routine may fail in certain cases. This will be fixed when I get a better algorithm. `bonpow` will not symmetrize your data before extracting eigenvectors; don't send this routine asymmetric matrices unless you really mean to do so.

Note

This function was ported (ie. copied) from the SNA package.

Author(s)

Carter T. Butts <`buttsc@uci.edu`>, ported to igraph by Gabor Csardi <`csardi.gabor@gmail.com`>

References

Bonacich, P. (1972). "Factoring and Weighting Approaches to Status Scores and Clique Identification." *Journal of Mathematical Sociology*, 2, 113-120.

Bonacich, P. (1987). "Power and Centrality: A Family of Measures." *American Journal of Sociology*, 92, 1170-1182.

See Also

`evcent` and `alpha.centrality`

Examples

```
# Generate some test data from Bonacich, 1987:
g.c <- graph( c(1,2,1,3,2,4,3,5), dir=FALSE)
g.d <- graph( c(1,2,1,3,1,4,2,5,3,6,4,7), dir=FALSE)
g.e <- graph( c(1,2,1,3,1,4,2,5,2,6,3,7,3,8,4,9,4,10), dir=FALSE)
g.f <- graph( c(1,2,1,3,1,4,2,5,2,6,2,7,3,8,3,9,3,10,4,11,4,12,4,13), dir=FALSE)
# Compute Bonpow scores
for (e in seq(-0.5,.5, by=0.1)) {
  print(round(bonpow(g.c, exp=e)[c(1,2,4)], 2))
}

for (e in seq(-0.4,.4, by=0.1)) {
  print(round(bonpow(g.d, exp=e)[c(1,2,5)], 2))
}

for (e in seq(-0.4,.4, by=0.1)) {
  print(round(bonpow(g.e, exp=e)[c(1,2,5)], 2))
}

for (e in seq(-0.4,.4, by=0.1)) {
  print(round(bonpow(g.f, exp=e)[c(1,2,5)], 2))
}
```

`canonical.permutation` *Canonical permutation of a graph*

Description

The canonical permutation brings every isomorphic graphs into the same (labeled) graph.

Usage

```
canonical.permutation(graph, sh="fm")
```

Arguments

`graph`	The input graph, treated as undirected.
`sh`	Type of the heuristics to use for the BLISS algorithm. See details for possible values.

Details

`canonical.permutation` computes a permutation which brings the graph into canonical form, as defined by the BLISS algorithm. All isomorphic graphs have the same canonical form.

See the paper below for the details about BLISS. This and more information is available at `http://www.tcs.hut.fi/Software/bliss/index.html`.

The possible values for the `sh` argument are:

`f` First non-singleton cell.

`fl` First largest non-singleton cell.

`fs` First smallest non-singleton cell.

`fm` First maximally non-trivially connectec non-singleton cell.

`flm` Largest maximally non-trivially connected non-singleton cell.

`fsm` Smallest maximally non-trivially connected non-singleton cell.

See the paper in references for details about these.

Value

A list with the following members:

`labeling`	The canonical parmutation which takes the input graph into canonical form. A numeric vector, the first element is the new label of vertex 0, the second element for vertex 1, etc.
`info`	Some information about the BLISS computation. A named list with the following members:

`nof_nodes` The number of nodes in the search tree.

`nof_leaf_nodes` The number of leaf nodes in the search tree.

`nof_bad_nodes` Number of bad nodes.

`nof_canupdates` Number of canrep updates.

`max_level` Maximum level.

`group_size` The size of the automorphism group of the input graph, as a string. This number is exact if igraph was compiled with the GMP library, and approximate otherwise.

Author(s)

Tommi Junttila for BLISS, Gabor Csardi <`csardi.gabor@gmail.com`> for the igraph and R interfaces.

References

Tommi Junttila and Petteri Kaski: Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics.* 2007.

See Also

`permute.vertices` to apply a permutation to a graph, `graph.isomorphic` for deciding graph isomorphism, possibly based on canonical labels.

Examples

```
## Calculate the canonical form of a random graph
g1 <- erdos.renyi.game(10, 20, type="gnm")
cp1 <- canonical.permutation(g1)
cf1 <- permute.vertices(g1, cp1$labeling)

## Do the same with a random permutation of it
g2 <- permute.vertices(g1, sample(vcount(g1)))
cp2 <- canonical.permutation(g2)
cf2 <- permute.vertices(g2, cp2$labeling)

## Check that they are the same
```

```
el1 <- get.edgelist(cf1)
el2 <- get.edgelist(cf2)
el1 <- el1[ order(el1[,1], el1[,2]), ]
el2 <- el2[ order(el2[,1], el2[,2]), ]
all(el1 == el2)
```

---

centralization — *Centralization of a graph.tmax*

Description

Centralization is a method for creating a graph level centralization measure from the centrality scores of the vertices.

Usage

```
centralize.scores (scores, theoretical.max, normalized = TRUE)

centralization.degree (graph, mode = c("all", "out", "in", "total"),
    loops = TRUE, normalized = TRUE)
centralization.closeness (graph, mode = c("out", "in", "all", "total"),
    normalized = TRUE)
centralization.betweenness (graph, directed = TRUE, nobigint = TRUE,
    normalized = TRUE)
centralization.evcent (graph, directed = FALSE, scale = TRUE,
    options = igraph.arpack.default, normalized = TRUE)

centralization.degree.tmax (graph = NULL, nodes = 0,
    mode = c("all", "out", "in", "total"), loops = FALSE)
centralization.closeness.tmax (graph = NULL, nodes = 0,
    directed = TRUE)
centralization.betweenness.tmax (graph = NULL, nodes = 0,
    directed = TRUE)
centralization.evcent.tmax (graph = NULL, nodes = 0,
    directed = FALSE, scale = TRUE)
```

Arguments

`scores`	The vertex level centrality scores.
`theoretical.max`	Real scalar. The graph level centrality score of the most centralized graph with the same number of vertices as the graph under study. This is only used if the `normalized` argument is set to `TRUE`.
`normalized`	Logical scalar. Whether to normalize the graph level centrality score by dividing the supplied theoretical maximum.
`graph`	The input graph. For the "tmax" functions it can be `NULL`, see the details below.
`mode`	This is the same as the `mode` argument of `degree` and `closeness`.
`loops`	Logical scalar, whether to consider loops edges when calculating the degree.
`directed`	logical scalar, whether to use directed shortest paths for calculating betweenness.

`nobigint`	Logical scalar, whether to use big integers for the betweenness calculation. This argument is passed to the `betweenness` function.
`scale`	Whether to rescale the eigenvector centrality scores, such that the maximum score is one.
`nodes`	The number of vertices. This is ignored if the graph is given.
`options`	This is passed to `evcent`, the options for the ARPACK eigensolver.

Details

Centralization is a general method for calculating a graph-level centrality score based on node-level centrality measure. The formula for this is

$$C(G) = \sum_v (\max_w c_w - c_v),$$

where c_v is the centrality of vertex v.

The graph-level centrality score can be normalized by dividing by the maximum theoretical score for a graph with the same number of vertices, using the same parameters, e.g. directedness, whether we consider loop edges, etc.

For degree, closeness and betweenness the most centralized structure is some version of the star graph, in-star, out-star or undirected star.

For eigenvector centrality the most centralized structure is the graph with a single edge (and potentially many isolates).

`centralize.scores` using the general centralization formula to calculate a graph-level score from vertex-level scores.

`centralization.degree`, `centralization.closeness`, `centralization.betweenness` calculate both the vertex-level and the graph-level indices.

`centralization.degree.tmax`, `centralization.closeness.tmax`, `centralization.betweenness.tmax` and `centralization.evcent.tmax` return the theoretical maximum scores. They operate in two modes. In the first mode, a graph is given and the maximum score is calculated based on that. E.g. the number of vertices and directedness is taken from this graph.

The other way to call these functions is to omit the `graph` argument, but explicitly specify the rest of the arguments.

Value

For `centralize.scores` a real scalar.

For `centralization.degree`, `centralization.closeness` and `centralization.betweenness` a named list with the following components:

`res`	The node-level centrality scores.
`centralization`	The graph level centrality index.
`theoretical_max`	The maximum theoretical graph level centralization score for a graph with the given number of vertices, using the same parameters. If the `normalized` argument was `TRUE`, then the result was divided by this number.

For `centralization.evcent` a named list with the following components:

`vector`	The node-level centrality scores.

`value`	The corresponding eigenvalue.
`options`	ARPACK options, see the return value of `evcent` for details.
`centralization`	The graph level centrality index.
`theoretical_max`	The same as above, the theoretical maximum centralization score for a graph with the same number of vertices.

For `centralization.degree.tmax`, `centralization.closeness.tmax`, `centralization.betweenness.tmax` and `centralization.evcent.tmax` a real scalar.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Freeman, L.C. (1979). Centrality in Social Networks I: Conceptual Clarification. *Social Networks* 1, 215–239.

Wasserman, S., and Faust, K. (1994). *Social Network Analysis: Methods and Applications.* Cambridge University Press.

Examples

```
# A BA graph is quite centralized
g <- ba.game(1000, m=4)
centralization.degree(g)$centralization
centralization.closeness(g, mode="all")$centralization
centralization.evcent(g, directed=FALSE)$centralization

# The most centralized graph according to eigenvector centrality
g0 <- graph( c(2,1), n=10, dir=FALSE )
g1 <- graph.star(10, mode="undirected")
centralization.evcent(g0)$centralization
centralization.evcent(g1)$centralization
```

`cliques` *The functions find cliques, ie. complete subgraphs in a graph*

Description

These functions find all, the largest or all the maximal cliques in an undirected graph. The size of the largest clique can also be calculated.

Usage

```
cliques(graph, min=NULL, max=NULL)
largest.cliques(graph)
maximal.cliques(graph, min=NULL, max=NULL)
clique.number(graph)
```

Arguments

`graph`	The input graph, directed graphs will be considered as undirected ones, multiple edges and loops are ignored.
`min`	Numeric constant, lower limit on the size of the cliques to find. `NULL` means no limit, ie. it is the same as 0.
`max`	Numeric constant, upper limit on the size of the cliques to find. `NULL` means no limit.

Details

`cliques` find all complete subgraphs in the input graph, obeying the size limitations given in the `min` and `max` arguments.

`largest.cliques` finds all largest cliques in the input graph. A clique is largest if there is no other clique including more vertices.

`maximal.cliques` finds all maximal cliques in the input graph. A clique in maximal if it cannot be extended to a larger clique. The largest cliques are always maximal, but a maximal clique is not neccessarily the largest.

`clique.number` calculates the size of the largest clique(s).

The current implementation of these functions searches for maximal independent vertex sets (see `independent.vertex.sets`) in the complementer graph.

Value

`cliques`, `largest.cliques` and `clique.number` return a list containing numeric vectors of vertex ids. Each list element is a clique.

`clique.number` returns an integer constant.

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com> for the R interface and the manual page.

See Also

`independent.vertex.sets`

Examples

```
# this usually contains cliques of size six
g <- erdos.renyi.game(100, 0.3)
clique.number(g)
cliques(g, min=6)
largest.cliques(g)

# To have a bit less maximal cliques, about 100-200 usually
g <- erdos.renyi.game(100, 0.03)
maximal.cliques(g)
```

`closeness` *Closeness centrality of vertices*

Description

Cloness centrality measures how many steps is required to access every other vertex from a given vertex.

Usage

```
closeness(graph, vids=V(graph), mode = c("out", "in", "all", "total"),
        weights = NULL, normalized = FALSE)
closeness.estimate(graph, vids=V(graph), mode = c("out", "in", "all",
       "total"), cutoff, weights = NULL)
```

Arguments

`graph`	The graph to analyze.
`vids`	The vertices for which closeness will be calculated.
`mode`	Character string, defined the types of the paths used for measuring the distance in directed graphs. "in" measures the paths *to* a vertex, "out" measures paths *from* a vertex, *all* uses undirected paths. This argument is ignored for undirected graphs.
`normalized`	Logical scalar, whether to calculate the normalized closeness. Normalization is performed by multiplying the raw closeness by $n-1$, where n is the number of vertices in the graph.
`cutoff`	The maximum path length to consider when calculating the betweenness. If zero or negative then there is no such limit.
`weights`	Optional positive weight vector for calculating weighted closeness. If the graph has a `weight` edge attribute, then this is used by default.

Details

The closeness centrality of a vertex is defined by the inverse of the average length of the shortest paths to/from all the other vertices in the graph:

$$\frac{1}{\sum_{i \neq v} d_v i}$$

If there is no (directed) path between vertex v and i then the total number of vertices is used in the formula instead of the path length.

`closeness.estimate` only considers paths of length `cutoff` or smaller, this can be run for larger graphs, as the running time is not quadratic (if `cutoff` is small). If `cutoff` is zero or negative then the function calculates the exact closeness scores.

Value

Numeric vector with the closeness values of all the vertices in `v`.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Freeman, L.C. (1979). Centrality in Social Networks I: Conceptual Clarification. *Social Networks*, 1, 215-239.

See Also

betweenness, degree

Examples

```
g <- graph.ring(10)
g2 <- graph.star(10)
closeness(g)
closeness(g2, mode="in")
closeness(g2, mode="out")
closeness(g2, mode="all")
```

---

clusters *Connected components of a graph*

Description

Calculate the maximal (weakly or strongly) connected components of a graph

Usage

```
is.connected(graph, mode=c("weak", "strong"))
clusters(graph, mode=c("weak", "strong"))
no.clusters(graph, mode=c("weak", "strong"))
cluster.distribution(graph, cumulative = FALSE, mul.size = FALSE, ...)
```

Arguments

`graph`	The graph to analyze.
`mode`	Character string, either “weak” or “strong”. For directed graphs “weak” implies weakly, “strong” strongly connected components to search. It is ignored for undirected graphs.
`cumulative`	Logical, if TRUE the cumulative distirubution (relative frequency) is calculated.
`mul.size`	Logical. If TRUE the relative frequencies will be multiplied by the cluster sizes.
`...`	Additional attributes to pass to `cluster`, right now only `mode` makes sense.

Details

`is.connected` decides whether the graph is weakly or strongly connected.

`clusters` finds the maximal (weakly or strongly) connected components of a graph.

`no.clusters` does almost the same as `clusters` but returns only the number of clusters found instead of returning the actual clusters.

`cluster.distribution` creates a histogram for the maximal connected component sizes.

The weakly connected components are found by a simple breadth-first search. The strongly connected components are implemented by two consecutive depth-first searches.

Value

For `is.connected` a logical constant.

For `clusters` a named list with three components:

`membership`	numeric vector giving the cluster id to which each vertex belongs.
`csize`	numeric vector giving the sizes of the clusters.
`no`	numeric constant, the number of clusters.

For `no.clusters` an integer constant is returned.

For `cluster.distribution` a numeric vector with the relative frequencies. The length of the vector is the size of the largest component plus one. Note that (for currently unknown reasons) the first element of the vector is the number of clusters of size zero, so this is always zero.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`subcomponent`

Examples

```
g <- erdos.renyi.game(20, 1/20)
clusters(g)
```

`cocitation` *Cocitation coupling*

Description

Two vertices are cocited if there is another vertex citing both of them. `cocitation` siply counts how many types two vertices are cocited. The bibliographic coupling of two vertices is the number of other vertices they both cite, `bibcoupling` calculates this.

Usage

```
cocitation(graph, v=V(graph))
bibcoupling(graph, v=V(graph))
```

Arguments

`graph`	The graph object to analyze
`v`	Vertex sequence or numeric vector, the vertex ids for which the cocitation or bibliographic coupling values we want to calculate. The default is all vertices.

Details

`cocitation` calculates the cocitation counts for the vertices in the `v` argument and all vertices in the graph.

`bibcoupling` calculates the bibliographic coupling for vertices in `v` and all vertices in the graph.

Calculating the cocitation or bibliographic coupling for only one vertex costs the same amount of computation as for all vertices. This might change in the future.

Value

A numeric matrix with `length(v)` lines and `vcount(graph)` columns. Element `(i,j)` contains the cocitation or bibliographic coupling for vertices `v[i]` and `j`.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

Examples

```
g <- graph.ring(10)
cocitation(g)
bibcoupling(g)
```

`cohesive.blocks` *Calculate Cohesive Blocks*

Description

Calculates cohesive blocks for objects of class `igraph`.

Usage

```
cohesive.blocks(graph, labels = TRUE)

blocks(blocks)
blockGraphs(blocks, graph)
cohesion(blocks)
hierarchy(blocks)
parent(blocks)
plotHierarchy(blocks,
   layout=layout.reingold.tilford(hierarchy(blocks), root=1), ...)
exportPajek(blocks, graph, file, project.file = TRUE)
maxcohesion(blocks)

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

```
## S3 method for class 'cohesiveBlocks'
summary(object, ...)
## S3 method for class 'cohesiveBlocks'
length(x)
## S3 method for class 'cohesiveBlocks'
plot(x, y, colbar = rainbow(max(cohesion(x))+1),
     col = colbar[maxcohesion(x)+1], mark.groups = blocks(x)[-1], ...)
```

Arguments

`graph`	For `cohesive.blocks` a graph object of class `igraph`. It must be undirected and simple. (See `is.simple`.) For `blockGraphs` and `exportPajek` the same graph must be supplied whose cohesive block structure is given in the `blocks` argument.
`labels`	Logical scalar, whether to add the vertex labels to the result object. These labels can be then used when reporting and plotting the cohesive blocks.
`blocks,x,object`	A `cohesiveBlocks` object, created with the `cohesive.blocks` function.
`file`	Defines the file (or connection) the Pajek file is written to. If the `project.file` argument is `TRUE`, then it can be a filename (with extension), a file object, or in general any king of connection object. The file/connection will be opened if it wasn't already. If the `project.file` argument is `FALSE`, then several files are created and `file` must be a character scalar containing the base name of the files, without extension. (But it can contain the path to the files.) See also details below.
`project.file`	Logical scalar, whether to create a single Pajek project file containing all the data, or to create separated files for each item. See details below.
`y`	The graph whose cohesive blocks are supplied in the `x` argument.
`colbar`	Color bar for the vertex colors. Its length should be at least $m + 1$, where m is the maximum cohesion in the graph. Alternatively, the vertex colors can also be directly specified via the `col` argument.
`col`	A vector of vertex colors, in any of the usual formats. (Symbolic color names (e.g. 'red', 'blue', etc.) , RGB colors (e.g. '#FF9900FF'), integer numbers referring to the current palette. By default the given `colbar` is used and vertices with the same maximal cohesion will have the same color.
`mark.groups`	A list of vertex sets to mark on the plot by circling them. By default all cohesive blocks are marked, except the one corresponding to the all vertices.
`layout`	The layout of a plot, it is simply passed on to `plot.igraph`, see the possible formats there. By default the Reingold-Tilford layout generator is used.
`...`	Additional arguments. `plotHierarchy` and `plot` pass them to `plot.igraph`. `print` and `summary` ignore them.

Details

Cohesive blocking is a method of determining hierarchical subsets of graph vertices based on their structural cohesion (or vertex connectivity). For a given graph G, a subset of its vertices $S \subset V(G)$ is said to be maximally k-cohesive if there is no superset of S with vertex connectivity greater than or equal to k. Cohesive blocking is a process through which, given a k-cohesive set of vertices, maximally l-cohesive subsets are recursively identified with $l > k$. Thus a hiearchy of vertex subsets is found, whith the entire graph G at its root.

The function `cohesive.blocks` implements cohesive blocking. It returns a `cohesiveBlocks` object. `cohesiveBlocks` should be handled as an opaque class, i.e. its internal structure should not be accessed directly, but through the functions listed here.

The function `length` can be used on `cohesiveBlocks` objects and it gives the number of blocks.

The function `blocks` returns the actual blocks stored in the `cohesiveBlocks` object. They are returned in a list of numeric vectors, each containing vertex ids.

The function `blockGraphs` is similar, but returns the blocks as (induced) subgraphs of the input graph. The various (graph, vertex and edge) attributes are kept in the subgraph.

The function `cohesion` returns a numeric vector, the cohesion of the different blocks. The order of the blocks is the same as for the `blocks` and `blockGraphs` functions.

The block hierarchy can be queried using the `hierarchy` function. It returns an igraph graph, its vertex ids are ordered according the order of the blocks in the `blocks` and `blockGraphs`, `cohesion`, etc. functions.

`parent` gives the parent vertex of each block, in the block hierarchy, for the root vertex it gives 0.

`plotHierarchy` plots the hierarchy tree of the cohesive blocks on the active graphics device, by calling `igraph.plot`.

The `exportPajek` function can be used to export the graph and its cohesive blocks in Pajek format. It can either export a single Pajek project file with all the information, or a set of files, depending on its `project.file` argument. If `project.file` is `TRUE`, then the following information is written to the file (or connection) given in the `file` argument: (1) the input graph, together with its attributes, see `write.graph` for details; (2) the hierarchy graph; and (3) one binary partition for each cohesive block. If `project.file` is `FALSE`, then the `file` argument must be a character scalar and it is used as the base name for the generated files. If `file` is ‘basename’, then the following files are created: (1) ‘basename.net’ for the original graph; (2) ‘basename_hierarchy.net’ for the hierarchy graph; (3) ‘basename_block_x.net’ for each cohesive block, where ‘x’ is the number of the block, starting with one.

`maxcohesion` returns the maximal cohesion of each vertex, i.e. the cohesion of the most cohesive block of the vertex.

The generic function `summary` works on `cohesiveBlocks` objects and it prints a one line summary to the terminal.

The generic function `print` is also defined on `cohesiveBlocks` objects and it is invoked automatically if the name of the `cohesiveBlocks` object is typed in. It produces an output like this:

```
Cohesive block structure:
B-1         c 1, n 23
'- B-2      c 2, n 14   oooooooo.. .o......oo ooo
   '- B-4   c 5, n  7   ooooooo... .......... ...
'- B-3      c 2, n 10   ......o.oo o.ooooooo.. ...
   '- B-5   c 3, n  4   ......o.oo o......... ...
```

The left part shows the block structure, in this case for five blocks. The first block always corresponds to the whole graph, even if its cohesion is zero. Then cohesion of the block and the number of vertices in the block are shown. The last part is only printed if the display is wide enough and shows the vertices in the blocks, ordered by vertex ids. ‘o’ means that the vertex is included, a dot means that it is not, and the vertices are shown in groups of ten.

The generic function `plot` plots the graph, showing one or more cohesive blocks in it.

Value

cohesive.blocks returns a cohesiveBlocks object.

blocks returns a list of numeric vectors, containing vertex ids.

blockGraphs returns a list of igraph graphs, corresponding to the cohesive blocks.

cohesion returns a numeric vector, the cohesion of each block.

hierarchy returns an igraph graph, the representation of the cohesive block hierarchy.

parent returns a numeric vector giving the parent block of each cohesive block, in the block hierarchy. The block at the root of the hierarchy has no parent and 0 is returned for it.

plotHierarchy, plot and exportPajek return NULL, invisibly.

maxcohesion returns a numeric vector with one entry for each vertex, giving the cohesion of its most cohesive block.

print and summary return the cohesiveBlocks object itself, invisibly.

length returns a numeric scalar, the number of blocks.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com> for the current implementation, Peter McMahan <peter.mcmahan@gmail.c
wrote the first version in R.

References

J. Moody and D. R. White. Structural cohesion and embeddedness: A hierarchical concept of social groups. *American Sociological Review*, 68(1):103–127, Feb 2003.

See Also

graph.cohesion

Examples

```
## The graph from the Moody-White paper
mw <- graph.formula(1-2:3:4:5:6, 2-3:4:5:7, 3-4:6:7, 4-5:6:7,
                    5-6:7:21, 6-7, 7-8:11:14:19, 8-9:11:14, 9-10,
                    10-12:13, 11-12:14, 12-16, 13-16, 14-15, 15-16,
                    17-18:19:20, 18-20:21, 19-20:22:23, 20-21,
                    21-22:23, 22-23)

mwBlocks <- cohesive.blocks(mw)

# Inspect block membership and cohesion
mwBlocks
blocks(mwBlocks)
cohesion(mwBlocks)

# Save results in a Pajek file
## Not run:
exportPajek(mwBlocks, mw, file="/tmp/mwBlocks.paj")

## End(Not run)

# Plot the results
if (interactive()) {
  plot(mwBlocks, mw)
}

## The science camp network
camp <- graph.formula(Harry:Steve:Don:Bert - Harry:Steve:Don:Bert,
                      Pam:Brazey:Carol:Pat - Pam:Brazey:Carol:Pat,
                      Holly   - Carol:Pat:Pam:Jennie:Bill,
                      Bill    - Pauline:Michael:Lee:Holly,
                      Pauline - Bill:Jennie:Ann,
                      Jennie  - Holly:Michael:Lee:Ann:Pauline,
                      Michael - Bill:Jennie:Ann:Lee:John,
                      Ann     - Michael:Jennie:Pauline,
                      Lee     - Michael:Bill:Jennie,
                      Gery    - Pat:Steve:Russ:John,
                      Russ    - Steve:Bert:Gery:John,
                      John    - Gery:Russ:Michael)
campBlocks <- cohesive.blocks(camp)
campBlocks

if (interactive()) {
  plot(campBlocks, camp, vertex.label=V(camp)$name, margin=-0.2,
       vertex.shape="rectangle", vertex.size=24, vertex.size2=8,
       mark.border=1, colbar=c(NA, NA,"cyan","orange") )
}
```

`Combining attributes` *How igraph functions handle attributes when the graph changes*

Description

Many times, when the structure of a graph is modified, vertices/edges map of the original graph map to vertices/edges in the newly created (modified) graph. For example `simplify` maps multiple edges to single edges. igraph provides a flexible mechanism to specify what to do with the vertex/edge attributes in these cases.

Details

The functions that support the combination of attributes have one or two extra arguments called `vertex.attr.comb` and/or `edge.attr.comb` that specify how to perform the mapping of the attributes. E.g. `contract.vertices` contracts many vertices into a single one, the attributes of the vertices can be combined and stores as the vertex attributes of the new graph.

The specification of the combination of (vertex or edge) attributes can be given as

1. a character scalar,
2. a function object or
3. a list of character scalars and/or function objects.

If it is a character scalar, then it refers to one of the predefined combinations, see their list below.

If it is a function, then the given function is expected to perform the combination. It will be called once for each new vertex/edge in the graph, with a single argument: the attribute values of the vertices that map to that single vertex.

The third option, a list can be used to specify different combination methods for different attributes. A named entry of the list corresponds to the attribute with the same name. An unnamed entry (i.e. if the name is the empty string) of the list specifies the default combination method. I.e.

```
list(weight="sum", "ignore")
```

specifies that the weight of the new edge should be sum of the weights of the corresponding edges in the old graph; and that the rest of the attributes should be ignored (=dropped).

Predefined combination functions

The following combination behaviors are predefined:

'ignore' The attribute is ignored and dropped.

'sum' The sum of the attributes is calculated. This does not work for character attributes and works for complex attributes only if they have a `sum` generic defined. (E.g. it works for sparse matrices from the `Matrix` package, because they have a `sum` method.)

'prod' The product of the attributes is calculated. This does not work for character attributes and works for complex attributes only if they have a `prod` function defined.

'min' The minimum of the attributes is calculated and returned. For character and complex attributes the standard R `min` function is used.

'max' The maximum of the attributes is calculated and returned. For character and complex attributes the standard R `max` function is used.

'random' Chooses one of the supplied attribute values, uniformly randomly. For character and complex attributes this is implemented by calling `sample`.

'first' Always chooses the first attribute value. It is implemented by calling the `head` function.

'last' Always chooses the last attribute value. It is implemented by calling the `tail` function.

'mean' The mean of the attributes is calculated and returned. For character and complex attributes this simply calls the `mean` function.

'median' The median of the attributes is selected. Calls the R `median` function for all attribute types.

'concat' Concatenate the attributes, using the `c` function. This results almost always a complex attribute.

Specifying combination methods for all graphs

The are two standard igraph parameters that define the default behavior when combining vertices and edges: `vertex.attr.comb` specifies how to combine vertices by default, `edge.attr.comb` does the same for edges.

E.g. if you want to drop all vertex attributes when combining vertices, you can specify

```
igraph.options(vertex.attr.comb="ignore")
```

As another example, if – when combining edges – you want to keep the mean weight of the edges, concatenate their names into a single character scalar, and drop everything else, then use

```
igraph.options(edge.attr.comb=list(weight="mean",
    name=toString, "ignore")
```

Simple and complex attributes

An attribute is simple if (for all vertices/edges) it can be specified as an atomic vector. Character and numeric attributes are always simple. E.g. a vertex attribute that is a numeric vector of arbitrary length for each vertex, is a complex attribute.

Combination of attributes might turn a complex attribute into a single one, and the opposite is possible, too. E.g. when contatenating attribute values to form the new attribute value, the result will be typically a complex attribute.

See also examples below.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

attributes on how to use graph/vertex/edges attributes in general. igraph.options on igraph parameters.

Examples

```
g <- graph( c(1,2, 1,2, 1,2, 2,3, 3,4) )
E(g)$weight <- 1:5

## print attribute values with the graph
igraph.options(print.graph.attributes=TRUE)
igraph.options(print.vertex.attributes=TRUE)
igraph.options(print.edge.attributes=TRUE)

## new attribute is the sum of the old ones
simplify(g, edge.attr.comb="sum")

## collect attributes into a string
simplify(g, edge.attr.comb=toString)

## concatenate them into a vector, this creates a complex
## attribute
simplify(g, edge.attr.comb="concat")

E(g)$name <- letters[seq_len(ecount(g))]

## both attributes are collected into strings
simplify(g, edge.attr.comb=toString)

## harmonic average of weights, names are dropped
simplify(g, edge.attr.comb=list(weight=function(x) length(x)/sum(1/x),
                                name="ignore"))
```

communities *Functions to deal with the result of network community detection*

Description

igraph community detection functions return their results as an object from the `communities` class. This manual page describes the operations of this class.

Usage

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

## S3 method for class 'communities'
length(x)
sizes(communities)
membership(communities)
## S3 method for class 'communities'
modularity(x, ...)
algorithm(communities)
crossing(communities, graph)

is.hierarchical(communities, full = FALSE)
merges(communities)
cutat(communities, no, steps)
## S3 method for class 'communities'
as.dendrogram(object, hang=-1,
   use.modularity=FALSE, ...)
## S3 method for class 'communities'
as.hclust(x, hang = -1,
   use.modularity = FALSE, ...)
## S3 method for class 'communities'
asPhylo(x, use.modularity=FALSE, ...)
showtrace(communities)

code.length(communities)

## S3 method for class 'communities'
plot(x, y,
   colbar=rainbow(length(x)),
   col=colbar[membership(x)],
   mark.groups=communities(x),
   edge.color=c("black", "red")[crossing(x,y)+1],
   ...)
```

Arguments

`communities,x,object`	A `communities` object, the result of an igraph community detection function.
`graph`	An igraph graph object, corresponding to `communities`.

`full`	Logical scalar, if `TRUE`, then `is.hierarchical` only returns `TRUE` for fully hierarchical algorithms. The 'leading eigenvector' algorithm is hierarchical, it gives a hierarchy of groups, but not a full dendrogram with all vertices, so it is not fully hierarchical.
`y`	An igraph graph object, corresponding to the communities in `x`.
`no`	Integer scalar, the desired number of communities. If too low or two high, then an error message is given. Exactly one of `no` and `steps` must be supplied.
`steps`	The number of merge operations to perform to produce the communities. Exactly one of `no` and `steps` must be supplied.
`colbar`	A vector of colors, in any format that is accepted by the regular R plotting methods. E.g. it may be an integer vector, a character vector of color names, a character vector of RGB colors. This vector gives the color bar for the vertices. The length of the vector should be the same as the number of communities.
`col`	A vector of colors, in any format that is accepted by the regular R plotting methods. This vector gives the colors of the vertices explicitly.
`mark.groups`	A list of numeric vectors. The communities can be highlighted using colored polygons. The groups for which the polygons are drawn are given here. The default is to use the groups given by the communities. Supply `NULL` here if you do not want to highlight any groups.
`edge.color`	The colors of the edges. By default the edges within communities are colored green and other edges are red.
`hang`	Numeric scalar indicating how the height of leaves should be computed from the heights of their parents; see `plot.hclust`.
`use.modularity`	Logical scalar, whether to use the modularity values to define the height of the branches.
`...`	Additional arguments. `plot.communities` passes these to `plot.igraph`. The other functions silently ignore them.

Details

Community structure detection algorithms try to find dense subgraphs in directed or undirected graphs, by optimizing some criteria, and usually using heuristics.

igraph implements a number of commmunity detection methods (see them below), all of which return an object of the class `communities`. Because the community structure detection algorithms are different, `communities` objects do not always have the same structure. Nevertheless, they have some common operations, these are documented here.

The `print` generic function is defined for `communities`, it prints a short summary.

The `length` generic function call be called on `communities` and returns the number of communities.

The `sizes` function returns the community sizes, in the order of their ids.

`membership` gives the division of the vertices, into communities. It returns a numeric vector, one value for each vertex, the id of its community. Community ids start from one. Note that some algorithms calculate the complete (or incomplete) hierarchical structure of the communities, and not just a single partitioning. For these algorithms typically the membership for the highest modularity value is returned, but see also the manual pages of the individual algorithms.

`modularity` gives the modularity score of the partitioning. (See `modularity.igraph` for details. For algorithms that do not result a single partitioning, the highest modularity value is returned.

`algorithm` gives the name of the algorithm that was used to calculate the community structure.

`crossing` returns a logical vector, with one value for each edge, ordered according to the edge ids. The value is `TRUE` iff the edge connects two different communities, according to the (best) membership vector, as returned by `membership()`.

`is.hierarchical` checks whether a hierarchical algorithm was used to find the community structure. Some functions only make sense for hierarchical methods (e.g. `merges`, `cutat` and `as.dendrogram`).

`merges` returns the merge matrix for hierarchical methods. An error message is given, if a non-hierarchical method was used to find the community structure. You can check this by calling `is.hierarchical` on the `communities` object.

`cutat` cuts the merge tree of a hierarchical community finding method, at the desired place and returns a membership vector. The desired place can be expressed as the desired number of communities or as the number of merge steps to make. The function gives an error message, if called with a non-hierarchical method.

`as.dendrogram` converts a hierarchical community structure to a `dendrogram` object. It only works for hierarchical methods, and gives an error message to others. See `dendrogram` for details.

`as.hclust` is similar to `as.dendrogram`, but converts a hierarchical community structure to a `hclust` object.

`asPhylo` converts a hierarchical community structure to a `phylo` object, you will need the `ape` package for this.

`showtrace` works (currently) only for communities found by the leading eigenvector method (`leading.eigenvector.co` and returns a character vector that gives the steps performed by the algorithm while finding the communities.

`code.length` is defined for the InfoMAP method (`infomap.community` and returns the code length of the partition.

It is possibly to call the `plot` function on `communities` objects. This will plot the graph (and uses `plot.igraph` internally), with the communities shown. By default it colores the vertices according to their communities, and also marks the vertex groups corresponding to the communities. It passes additional arguments to `plot.igraph`, please see that and also `igraph.plotting` on how to change the plot.

Value

`print` returns the `communities` object itself, invisibly.

`length` returns an integer scalar.

`sizes` returns a numeric vector.

`membership` returns a numeric vector, one number for each vertex in the graph that was the input of the community detection.

`modularity` returns a numeric scalar.

`algorithm` returns a character scalar.

`crossing` returns a logical vector.

`is.hierarchical` returns a logical scalar.

`merges` returns a two-column numeric matrix.

`cutat` returns a numeric vector, the membership vector of the vertices.

`as.dendrogram` returns a `dendrogram` object.

`showtrace` returns a character vector.

`code.length` returns a numeric scalar for communities found with the InfoMAP method and `NULL` for other methods.

`plot` for `communities` objects returns `NULL`, invisibly.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

See dendPlot for plotting community structure dendrograms.

See compare.communities for comparing two community structures on the same graph.

The different methods for finding communities, they all return a communities object: edge.betweenness.community, fastgreedy.community, label.propagation.community, leading.eigenvector.community, multilevel.community, optimal.community, spinglass.community, walktrap.community.

Examples

```
karate <- graph.famous("Zachary")
wc <- walktrap.community(karate)
modularity(wc)
membership(wc)
plot(wc, karate)
```

---

community.to.membership — *Common functions supporting community detection algorithms*

Description

community.to.membership takes a merge matrix, a typical result of community structure detection algorithms and creates a membership vector by performing a given number of merges in the merge matrix.

Usage

```
community.to.membership(graph, merges, steps, membership=TRUE, csize=TRUE)
```

Arguments

graph	The graph to which the merge matrix belongs.
merges	The merge matrix, see e.g. walktrap.community for the exact format.
steps	The number of steps, ie. merges to be performed.
membership	Logical scalar, whether to include the membership vector in the result.
csize	Logical scalar, whether to include the sizes of the communities in the result.

Value

A named list with two members:

membership	The membership vector.
csize	A numeric vector giving the sizes of the communities.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

walktrap.community, edge.betweenness.community, fastgreedy.community, spinglass.community for various community detection methods.

Examples

```
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(1,6, 1,11, 6, 11))
wtc <- walktrap.community(g)
community.to.membership(g, wtc$merges, steps=12)
```

---

compare.communities *Compares community structures using various metrics*

Description

This function assesses the distance between two community structures.

Usage

```
## S3 method for class 'communities'
compare(comm1, comm2, method = c("vi", "nmi",
                                  "split.join", "rand",
                                  "adjusted.rand"))
## S3 method for class 'numeric'
compare(comm1, comm2, method = c("vi", "nmi",
                                  "split.join", "rand",
                                  "adjusted.rand"))
```

Arguments

comm1	A communities object containing a community structure; or a numeric vector, the membership vector of the first community structure. The membership vector should contain the community id of each vertex, the numbering of the communities starts with one.
comm2	A communities object containing a community structure; or a numeric vector, the membership vector of the second community structure, in the same format as for the previous argument.
method	Character scalar, the comparison method to use. Possible values: 'vi' is the variation of information (VI) metric of Meila (2003), 'nmi' is the normalized mutual information measure proposed by Danon et al. (2005), 'split.join' is the split-join distance of can Dongen (2000), 'rand' is the Rand index of Rand (1971), 'adjusted.rand' is the adjusted Rand index by Hubert and Arabie (1985).

Value

A real number.

Author(s)

Tamas Nepusz <ntamas@gmail.com>

References

Meila M: Comparing clusterings by the variation of information. In: Scholkopf B, Warmuth MK (eds.). *Learning Theory and Kernel Machines: 16th Annual Conference on Computational Learning Theory and 7th Kernel Workshop*, COLT/Kernel 2003, Washington, DC, USA. Lecture Notes in Computer Science, vol. 2777, Springer, 2003. ISBN: 978-3-540-40720-1.

Danon L, Diaz-Guilera A, Duch J, Arenas A: Comparing community structure identification. *J Stat Mech* P09008, 2005.

van Dongen S: Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.

Rand WM: Objective criteria for the evaluation of clustering methods. *J Am Stat Assoc* 66(336):846-850, 1971.

Hubert L and Arabie P: Comparing partitions. *Journal of Classification* 2:193-218, 1985.

See Also

`walktrap.community`, `edge.betweenness.community`, `fastgreedy.community`, `spinglass.community` for various community detection methods.

Examples

```
g <- graph.famous("Zachary")
sg <- spinglass.community(g)
le <- leading.eigenvector.community(g)
compare(sg, le, method="rand")
compare(membership(sg), membership(le))
```

`components` *In- or out- component of a vertex*

Description

Finds all vertices reachable from a given vertex, or the opposite: all vertices from which a given vertex is reachable via a directed path.

Usage

```
subcomponent(graph, v, mode = c("all", "out", "in"))
```

Arguments

`graph`	The graph to analyze.
`v`	The vertex to start the search from.
`mode`	Character string, either “in”, “out” or “all”. If “in” all vertices from which `v` is reachable are listed. If “out” all vertices reachable from `v` are returned. If “all” returns the union of these. It is ignored for undirected graphs.

Details

A breadh-first search is conducted starting from vertex `v`.

Value

Numeric vector, the ids of the vertices in the same component as `v`.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

See Also

`clusters`

Examples

```
g <- erdos.renyi.game(100, 1/200)
subcomponent(g, 1, "in")
subcomponent(g, 1, "out")
subcomponent(g, 1, "all")
```

`constraint` *Burt's constraint*

Description

Given a graph, `constraint` calculates Burt's constraint for each vertex.

Usage

```
constraint(graph, nodes=V(graph), weights=NULL)
```

Arguments

`graph`	A graph object, the input graph.
`nodes`	The vertices for which the constraint will be calculated. Defaults to all vertices.
`weights`	The weights of the edges. If this is `NULL` and there is a `weight` edge attribute this is used. If there is no such edge attribute all edges will have the same weight.

Details

Burt's constraint is higher if ego has less, or mutually stronger related (i.e. more redundant) contacts. Burt's measure of constraint, C_i, of vertex i's ego network V_i, is defined for directed and valued graphs,

$$C_i = \sum_{j \in V_i \setminus \{i\}} (p_{ij} + \sum_{q \in V_i \setminus \{i,j\}} p_{iq} p_{qj})^2$$

for a graph of order (ie. number of vertices) N, where proportional tie strengths are defined as

$$p_{ij} = \frac{a_{ij} + a_{ji}}{\sum_{k \in V_i \setminus \{i\}} (a_{ik} + a_{ki})},$$

a_{ij} are elements of A and the latter being the graph adjacency matrix. For isolated vertices, constraint is undefined.

Value

A numeric vector of constraint scores

Author(s)

Jeroen Bruggeman <`j.p.bruggeman@uva.nl`> and Gabor Csardi <`csardi.gabor@gmail.com`>

References

Burt, R.S. (2004). Structural holes and good ideas. *American Journal of Sociology* 110, 349-399.

Examples

```
g <- erdos.renyi.game(20, 5/20)
constraint(g)
```

`contract.vertices` *Contract several vertices into a single one*

Description

This function creates a new graph, by merging several vertices into one. The vertices in the new graph correspond to sets of vertices in the input graph.

Usage

```
contract.vertices(graph, mapping,
                  vertex.attr.comb=getIgraphOpt("vertex.attr.comb"))
```

Arguments

`graph`	The input graph, it can be directed or undirected.
`mapping`	A numeric vector that specifies the mapping. Its elements correspond to the vertices, and for each element the id in the new graph is given.
`vertex.attr.comb`	Specifies how to combine the vertex attributes in the new graph. Please see `attribute.combination` for details.

Details

The attributes of the graph are kept. Graph and edge attributes are unchanged, vertex attributes are combined, according to the `vertex.attr.comb` parameter.

Value

A new graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
g <- graph.ring(10)
g$name <- "Ring"
V(g)$name <- letters[1:vcount(g)]
E(g)$weight <- runif(ecount(g))

g2 <- contract.vertices(g, rep(1:5, each=2),
                        vertex.attr.comb=toString)

## graph and edge attributes are kept, vertex attributes are
## combined using the 'toString' function.
print(g2, g=TRUE, v=TRUE, e=TRUE)
```

`conversion` *Convert a graph to an adjacency matrix or an edge list*

Description

Sometimes it is useful to have a standard representation of a graph, like an adjacency matrix or an edge list.

Usage

```
get.adjacency(graph, type=c("both", "upper", "lower"),
       attr=NULL, edges=FALSE, names=TRUE,
       sparse=getIgraphOpt("sparsematrices"))
get.edgelist(graph, names=TRUE)
```

Arguments

`graph`	The graph to convert.
`type`	Gives how to create the adjacency matrix for undirected graphs. It is ignored for directed graphs. Possible values: `upper`: the upper right triangle of the matrix is used, `lower`: the lower left triangle of the matrix is used. `both`: the whole matrix is used, a symmetric matrix is returned.
`attr`	Either `NULL` or a character string giving an edge attribute name. If `NULL` a traditional adjacency matrix is returned. If not `NULL` then the values of the given edge attribute are included in the adjacency matrix. If the graph has multiple edges, the edge attribute of an arbitrarily chosen edge (for the multiple edges) is included. This argument is ignored if `edges` is `TRUE`.
`edges`	Logical scalar, whether to return the edge ids in the matrix. For non-existant edges zero is returned.
`names`	Logical constant. For `graph.adjacenct` it gives whether to assign row and column names to the matrix. These are only assigned if the `name` vertex attribute is present in the graph. for `get.edgelist` it gives whether to return a character matrix containing vertex names (ie. the `name` vertex attribute) if they exist or numeric vertex ids.
`sparse`	Logical scalar, whether to create a sparse matrix. The '`Matrix`' package must be installed for creating sparse matrices.

Details

get.adjacency returns the adjacency matrix of a graph, a regular R matrix if sparse is FALSE, or a sparse matrix, as defined in the 'Matrix' package, if sparse if TRUE.

get.edgelist returns the list of edges in a graph.

Value

A vcount(graph) by vcount(graph) (usually) numeric matrix for get.adjacency. (This can be huge!) Note that a non-numeric matrix might be returned if attr is a non-numeric edge attribute.

A ecount(graph) by 2 numeric matrix for get.edgelist.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

graph.adjacency, read.graph

Examples

```
g <- erdos.renyi.game(10, 2/10)
get.edgelist(g)
get.adjacency(g)
V(g)$name <- letters[1:vcount(g)]
get.adjacency(g)
E(g)$weight <- runif(ecount(g))
get.adjacency(g, attr="weight")
```

---

conversion between igraph and graphNEL graphs

Convert igraph graphs to graphNEL objects or back

Description

The graphNEL class is defined in the graph package, it is another way to represent graphs. These functions are provided to convert between the igraph and the graphNEL objects.

Usage

```
igraph.from.graphNEL(graphNEL, name = TRUE, weight = TRUE,
               unlist.attrs = TRUE)
igraph.to.graphNEL(graph)
```

Arguments

graphNEL	The graphNEL graph.
name	Logical scalar, whether to add graphNEL vertex names as an igraph vertex attribute called 'name'.
weight	Logical scalar, whether to add graphNEL edge weights as an igraph edge attribute called 'weight'. (graphNEL graphs are always weighted.)

`unlist.attrs`	Logical scalar. graphNEL attribute query functions return the values of the attributes in R lists, if this argument is `TRUE` (the default) these will be converted to atomic vectors, whenever possible, before adding them to the igraph graph.
`graph`	An igraph graph object.

Details

`igraph.from.graphNEL` takes a graphNEL graph and converts it to an igraph graph. It handles all graph/vertex/edge attributes. If the graphNEL graph has a vertex attribute called '`name`' it will be used as igraph vertex attribute '`name`' and the graphNEL vertex names will be ignored.

Because graphNEL graphs poorly support multiple edges, the edge attributes of the multiple edges are lost: they are all replaced by the attributes of the first of the multiple edges.

`igraph.to.graphNEL` converts and igraph graph to a graphNEL graph. It converts all graph/vertex/edge attributes. If the igraph graph has a vertex attribute '`name`', then it will be used to assign vertex names in the graphNEL graph. Otherwise igraph vertex ids will be used for this purpose.

Value

`igraph.from.graphNEL` returns an igraph graph object.

`igraph.to.graphNEL` returns a graphNEL graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`get.adjacency`, `graph.adjacency`, `get.adjlist` and `graph.adjlist`.

Examples

```
## Undirected
g <- graph.ring(10)
V(g)$name <- letters[1:10]
GNEL <- igraph.to.graphNEL(g)
g2 <- igraph.from.graphNEL(GNEL)
g2

## Directed
g3 <- graph.star(10, mode="in")
V(g3)$name <- letters[1:10]
GNEL2 <- igraph.to.graphNEL(g3)
g4 <- igraph.from.graphNEL(GNEL2)
g4
```

convex.hull *Convex hull of a set of vertices*

Description

Calculate the convex hull of a set of points, i.e. the covering polygon that has the smallest area.

Usage

```
convex.hull(data)
```

Arguments

data	The data points, a numeric matrix with two columns.

Value

A named list with components:

resverts	The indices of the input vertices that constritute the convex hull.
rescoords	The coordinates of the corners of the convex hull.

Author(s)

Tamas Nepusz <ntamas@gmail.com>

References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0262032937. Pages 949-955 of section 33.3: Finding the convex hull.

Examples

```
M <- cbind( runif(100), runif(100) )
convex.hull(M)
```

decompose.graph *Decompose a graph into components*

Description

Creates a separate graph for each component of a graph.

Usage

```
decompose.graph(graph, mode = c("weak", "strong"),
       max.comps = NA, min.vertices = 0)
```

Arguments

`graph`	The original graph.
`mode`	Character constant giving the type of the components, wither `weak` for weakly connected components or `strong` for strongly connected components.
`max.comps`	The maximum number of components to return. The first `max.comps` components will be returned (which hold at least `min.vertices` vertices, see the next parameter), the others will be ignored. Supply `NA` here if you don't want to limit the number of components.
`min.vertices`	The minimum number of vertices a component should contain in order to place it in the result list. Eg. supply 2 here to ignore isolate vertices.

Value

A list of graph objects.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`is.connected` to decide whether a graph is connected, `clusters` to calculate the connected components of a graph.

Examples

```
# the diameter of each component in a random graph
g <- erdos.renyi.game(1000, 1/1000)
comps <- decompose.graph(g, min.vertices=2)
sapply(comps, diameter)
```

`degree` *Degree and degree distribution of the vertices*

Description

The degree of a vertex is its most basic structural property, the number of its adjacent edges.

Usage

```
degree(graph, v=V(graph), mode = c("all", "out", "in", "total"),
       loops = TRUE, normalized = FALSE)
degree.distribution(graph, cumulative = FALSE, ...)
```

Arguments

`graph`	The graph to analyze.
`v`	The ids of vertices of which the degree will be calculated.
`mode`	Character string, "out" for out-degree, "in" for in-degree or "total" for the sum of the two. For undirected graphs this argument is ignored. "all" is a synonym of "total".
`loops`	Logical; whether the loop edges are also counted.
`normalized`	Logical scalar, whether to normalize the degree. If `TRUE` then the result is divided by $n - 1$, where n is the number of vertices in the graph.
`cumulative`	Logical; whether the cumulative degree distribution is to be calculated.
`...`	Additional arguments to pass to `degree`, eg. `mode` is useful but also `v` and `loops` make sense.

Value

For `degree` a numeric vector of the same length as argument `v`.

For `degree.distribution` a numeric vector of the same length as the maximum degree plus one. The first element is the relative frequency zero degree vertices, the second vertices with degree one, etc.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
g <- graph.ring(10)
degree(g)
g2 <- erdos.renyi.game(1000, 10/1000)
degree.distribution(g2)
```

`degree.sequence.game` *Generate random graphs with a given degree sequence*

Description

It is often useful to create a graph with given vertex degrees. This is exactly what `degree.sequence.game` does.

Usage

```
degree.sequence.game(out.deg, in.deg = numeric(0),
     method = c("simple", "vl"), ...)
```

Arguments

`out.deg`	Numeric vector, the sequence of degrees (for undirected graphs) or out-degrees (for directed graphs). For undirected graphs its sum should be even. For directed graphs its sum should be the same as the sum of `in.deg`.
`in.deg`	For directed graph, the in-degree sequence.
`method`	Character, the method for generating the graph. Right now the "simple" and "vl" methods are implemented.
`...`	Additional arguments, these are used as graph attributes.

Details

The "simple" method connects the out-stubs of the edges (undirected graphs) or the out-stubs and in-stubs (directed graphs) together. This way loop edges and also multiple edges may be generated.

This method is not adequate if one needs to generate simple graphs with a given degree sequence. The multiple and loop edges can be deleted, but then the degree sequence is distorted and there is nothing to ensure that the graphs are sampled uniformly.

THe "vl" method is a more sophisticated generator. The algorithm and the implementation was done by Fabien Viger and Matthieu Latapy. This generator always generates undirected, connected simple graphs, it is an error to pass the `in.deg` argument to it. The algorithm relies on first creating an initial (possibly unconnected) simple undirected graph with the given degree sequence (if this is possible at all). Then some rewiring is done to make the graph connected. Finally a Monte-Carlo algorithm is used to randomize the graph. The "vl" samples from the undirected, connected simple graphs unformly. See `http://www-rp.lip6.fr/~latapy/FV/generation.html` for details.

Value

The new graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`erdos.renyi.game`, `barabasi.game`, `simplify` to get rid of the multiple and/or loops edges.

Examples

```
## The simple generator
g <- degree.sequence.game(rep(2,100))
degree(g)
is.simple(g)   # sometimes TRUE, but can be FALSE
g2 <- degree.sequence.game(1:10, 10:1)
degree(g2, mode="out")
degree(g2, mode="in")

## The vl generator
g3 <- degree.sequence.game(rep(2,100), method="vl")
degree(g3)
is.simple(g3)  # always TRUE

## Exponential degree distribution
## Note, that we correct the degree sequence if its sum is odd
```

```
degs <- sample(1:100, 100, replace=TRUE, prob=exp(-0.5*(1:100)))
if (sum(degs) %% 2 != 0) { degs[1] <- degs[1] + 1 }
g4 <- degree.sequence.game(degs, method="vl")
all(degree(g4) == degs)

## Power-law degree distribution
## Note, that we correct the degree sequence if its sum is odd
degs <- sample(1:100, 100, replace=TRUE, prob=(1:100)^-2)
if (sum(degs) %% 2 != 0) { degs[1] <- degs[1] + 1 }
g5 <- degree.sequence.game(degs, method="vl")
all(degree(g5) == degs)
```

`dendPlot` *Plot dendrograms*

Description

This is generic function that can plot various objects as dendrograms.

Details

Currently the function is defined for `communities` (see `dendPlot.communities`) and `igraphHRG` (see `dendPlot.igraphHRG`) objects.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
karate <- graph.famous("Zachary")
fc <- fastgreedy.community(karate)
dendPlot(fc)
```

`dendPlot.communities` *Plot fully hierarchical community structures as dendrograms*

Description

Plot a dendrogram that corresponds to a fully hierarchical community structure.

Usage

```
## S3 method for class 'communities'
dendPlot(x, mode = getIgraphOpt("dend.plot.type"),
         ..., use.modularity = FALSE)
```

Arguments

`x`	A `communities` object, the community structure, as returned by some igraph community detection function.
`mode`	Which dendrogram plotting function to use. See details below.
`...`	Additional arguments to supply to the dendrogram plotting function.
`use.modularity`	Logical scalar, whether to use the modularity scores calculated by the community detection algorithm, to set the heights of the dendrogram branches. Right now this must be `FALSE` for undirected graphs, for technical readons.

Details

`dendPlot` supports three different plotting functions, selected via the `mode` argument. By default the plotting function is taken from the `dend.plot.type` igraph option, and it has for possible values:

- `auto` Choose automatically between the plotting functions. As `plot.phylo` is the most sophisticated, that is choosen, whenever the `ape` package is available. Otherwise `plot.hclust` is used.
- `phylo` Use `plot.phylo` from the `ape` package.
- `hclust` Use `plot.hclust` from the `stats` package.
- `dendrogram` Use `plot.dendrogram` from the `stats` package.

The different plotting functions take different sets of arguments. When using `plot.phylo` (`mode="phylo"`), we have the following syntax:

```
dendPlot(communities, mode="phylo",
     colbar = rainbow(length(communities)),
     col = colbar[membership(communities)],
     mark.groups = communities(communities),
     edge.color = "#AAAAAAFF", edge.lty = c(1,2), ...,
     use.modularity = FALSE)
```

The extra arguments not documented above:

- `colbar` This is a vector of colors, in any format supported by R. These colors will be used to mark the communities on the dendrogram, if `mark.groups` are given.
- `col` The colors of the vertex labels, by default the community membership is color-coded here, using the colors from the `colbar` argument.
- `mark.groups` A list of groups to mark on the tree. Groups must correspond to disjunct subtrees of the dendrogram, otherwise you might get strange results. This argument is interpreted as a list of vertex numeric ids.
- `edge.color` The default branch color. This gives the color of the dendrogram branches that are not marked according to the `mark.groups` argument.
- `edge.lty` The line types of the branches, a vector of two line types, the first value is for branches marked via `mark.groups`, the second value is for unmarked branches. See the documentation of `lty` at the manual page of `par` for possible values. By default unmarked branches use dashed, marked branches use solid lines.
- `...` Additional arguments to pass to `plot.phylo`.

The syntax for `plot.hclust` (`mode="hclust"`):

```
    dendPlot(communities, mode = "hclust", rect = length(communities),
         colbar = rainbow(rect), hang = -1, ann = FALSE, main = "",
 sub = "", xlab = "", ylab = "", ..., use.modularity = FALSE)
```

The extra arguments not documented above:

- `rect` A numeric constant, the number of groups to mark on the dendrogram. The dendrogram is cut into exactly `rect` groups and they are marked via the `rect.hclust` command. Set this to zero if you don't want to mark any groups.
- `colbar` The colors of the rectanges that mark the communities via the `rect` argument.
- `hang` Where to put the leaf nodes, this corresponds to the `hang` argument of `plot.hclust`.
- `ann` Whether to annotate the plot, the `ann` argument of `plot.hclust`.
- `main` The main title of the plot, the `main` argument of `plot.hclust`.
- `sub` The sub-title of the plot, the `sub` argument of `plot.hclust`.
- `xlab` The label on the horizontal axis, passed to `plot.hclust`.
- `ylab` The label on the vertical axis, passed to `plot.hclust`.
- `dots` Attitional arguments to pass to `plot.hclust`.

The syntax for `plot.dendrogram` (`mode="dendrogram"`):

```
    dendPlot(communities, mode="dendrogram", hang = -1, ...,
         use.modularity = FALSE)
```

The extra argument not documented above:

- `hang` Where to place the leaf nodes on the plot, this is passed to `as.dendrogram`.

Value

Returns whatever the return value was from the plotting function, `plot.phylo`, `plot.dendrogram` or `plot.hclust`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`communities` for other operations on `communities` obejcts.

Examples

```
karate <- graph.famous("Zachary")
fc <- fastgreedy.community(karate)
dendPlot(fc)
```

dendPlot.igraphHRG *HRG dendrogram plot*

Description

Plot a hierarchical random graph as a dendrogram.

Usage

```
## S3 method for class 'igraphHRG'
dendPlot(x, mode = getIgraphOpt("dend.plot.type"),
        ...)
```

Arguments

x	An igraphHRG, a hierarchical random graph, as returned by the hrg.fit function.
mode	Which dendrogram plotting function to use. See details below.
...	Additional arguments to supply to the dendrogram plotting function.

Details

dendPlot supports three different plotting functions, selected via the mode argument. By default the plotting function is taken from the dend.plot.type igraph option, and it has for possible values:

- auto Choose automatically between the plotting functions. As plot.phylo is the most sophisticated, that is choosen, whenever the ape package is available. Otherwise plot.hclust is used.
- phylo Use plot.phylo from the ape package.
- hclust Use plot.hclust from the stats package.
- dendrogram Use plot.dendrogram from the stats package.

The different plotting functions take different sets of arguments. When using plot.phylo (mode="phylo"), we have the following syntax:

```
    dendPlot(x, mode="phylo",
         colbar = rainbow(11, start=0.7, end=0.1),
 edge.color = NULL, use.edge.length = FALSE, ...)
```

The extra arguments not documented above:

- colbar Color bar for the edges.
- edge.color Edge colors. If NULL, then the colbar argument is used.
- use.edge.length Passed to plot.phylo.
- dots Attitional arguments to pass to plot.phylo.

The syntax for plot.hclust (mode="hclust"):

```
    dendPlot(x, mode="hclust", rect = 0, colbar = rainbow(rect),
         hang = 0.01, ann = FALSE, main = "", sub = "", xlab = "",
 ylab = "", ...)
```

The extra arguments not documented above:

- `rect` A numeric scalar, the number of groups to mark on the dendrogram. The dendrogram is cut into exactly `rect` groups and they are marked via the `rect.hclust` command. Set this to zero if you don't want to mark any groups.
- `colbar` The colors of the rectanges that mark the vertex groups via the `rect` argument.
- `hang` Where to put the leaf nodes, this corresponds to the `hang` argument of `plot.hclust`.
- `ann` Whether to annotate the plot, the `ann` argument of `plot.hclust`.
- `main` The main title of the plot, the `main` argument of `plot.hclust`.
- `sub` The sub-title of the plot, the `sub` argument of `plot.hclust`.
- `xlab` The label on the horizontal axis, passed to `plot.hclust`.
- `ylab` The label on the vertical axis, passed to `plot.hclust`.
- `dots` Attitional arguments to pass to `plot.hclust`.

The syntax for `plot.dendrogram` (`mode="dendrogram"`):

```
    dendPlot(x, ...)
```

The extra arguments are simply passed to `as.dendrogram`.

Value

Returns whatever the return value was from the plotting function, `plot.phylo`, `plot.dendrogram` or `plot.hclust`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
g <- graph.full(5) + graph.full(5)
hrg <- hrg.fit(g)
dendPlot(hrg)
```

`diameter` *Diameter of a graph*

Description

The diameter of a graph is the length of the longest geodesic.

Usage

```
diameter(graph, directed = TRUE, unconnected = TRUE, weights = NULL)
get.diameter (graph, directed = TRUE, unconnected = TRUE, weights = NULL)
farthest.nodes (graph, directed = TRUE, unconnected = TRUE, weights = NULL)
```

Arguments

`graph`	The graph to analyze.
`directed`	Logical, whether directed or undirected paths are to be considered. This is ignored for undirected graphs.
`unconnected`	Logical, what to do if the graph is unconnected. If FALSE, the function will return a number that is one larger the largest possible diameter, which is always the number of vertices. If TRUE, the diameters of the connected components will be calculated and the largest one will be returned.
`weights`	Optional positive weight vector for calculating weighted distances. If the graph has a `weight` edge attribute, then this is used by default.

Details

The diameter is calculated by using a breadth-first search like method.

`get.diameter` returns a path with the actual diameter. If there are many shortest paths of the length of the diameter, then it returns the first one found.

`farthest.points` returns two vertex ids, the vertices which are connected by the diameter path.

Value

A numeric constant for `diameter`, a numeric vector for `get.diameter` and a numeric vector of length two for `farthest.nodes`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`shortest.paths`

Examples

```
g <- graph.ring(10)
g2 <- delete.edges(g, c(1,2,1,10))
diameter(g2, unconnected=TRUE)
diameter(g2, unconnected=FALSE)

## Weighted diameter
set.seed(1)
g <- graph.ring(10)
E(g)$weight <- sample(seq_len(ecount(g)))
diameter(g)
get.diameter(g)
diameter(g, weights=NA)
get.diameter(g, weights=NA)
```

`dominator.tree` *Dominator tree*

Description

Dominator tree of a directed graph.

Usage

```
dominator.tree (graph, root, mode = c("out", "in"))
```

Arguments

`graph`	A directed graph. If it is not a flowgraph, and it contains some vertices not reachable from the root vertex, then these vertices will be collected and returned as part of the result.
`root`	The id of the root (or source) vertex, this will be the root of the tree.
`mode`	Constant, must be '`in`' or '`out`'. If it is '`in`', then all directions are considered as opposite to the original one in the input graph.

Details

A flowgraph is a directed graph with a distinguished start (or root) vertex r, such that for any vertex v, there is a path from r to v. A vertex v dominates another vertex w (not equal to v), if every path from r to w contains v. Vertex v is the immediate dominator or w, $v = \mathrm{idom}(w)$, if v dominates w and every other dominator of w dominates v. The edges $(\mathrm{idom}(w), w)|w \neq r$ form a directed tree, rooted at r, called the dominator tree of the graph. Vertex v dominates vertex w if and only if v is an ancestor of w in the dominator tree.

This function implements the Lengauer-Tarjan algorithm to construct the dominator tree of a directed graph. For details see the reference below.

Value

A list with components:

`dom`	A numeric vector giving the immediate dominators for each vertex. For vertices that are unreachable from the root, it contains `NaN`. For the root vertex itself it contains minus one.
`domtree`	A graph object, the dominator tree. Its vertex ids are the as the vertex ids of the input graph. Isolate vertices are the ones that are unreachable from the root.
`leftout`	A numeric vector containing the vertex ids that are unreachable from the root.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Thomas Lengauer, Robert Endre Tarjan: A fast algorithm for finding dominators in a flowgraph, *ACM Transactions on Programming Languages and Systems (TOPLAS)* I/1, 121–141, 1979.

Examples

```
## The example from the paper
g <- graph.formula(R-+A:B:C, A-+D, B-+A:D:E, C-+F:G, D-+L,
                   E-+H, F-+I, G-+I:J, H-+E:K, I-+K, J-+I,
                   K-+I:R, L-+H)
dtree <- dominator.tree(g, root="R")
layout <- layout.reingold.tilford(dtree$domtree, root="R")
layout[,2] <- -layout[,2]
if (interactive()) {
  plot(dtree$domtree, layout=layout, vertex.label=V(dtree$domtree)$name)
}
```

`Drawing graphs` *Drawing graphs*

Description

The common bits of the three plotting functions `plot.igraph`, `tkplot` and `rglplot` are discussed in this manual page

Details

There are currently three different functions in the igraph package which can draw graph in various ways:

`plot.igraph` does simple non-interactive 2D plotting to R devices. Actually it is an implementation of the `plot` generic function, so you can write `plot(graph)` instead of `plot.igraph(graph)`. As it used the standard R devices it supports every output format for which R has an output device. The list is quite impressing: PostScript, PDF files, XFig files, SVG files, JPG, PNG and of course you can plot to the screen as well using the default devices, or the good-looking anti-aliased Cairo device. See `plot.igraph` for some more information.

tkplot does interactive 2D plotting using the tcltk package. It can only handle graphs of moderate size, a thousend vertices is probably already too many. Some parameters of the plotted graph can be changed interactively after issuing the tkplot command: the position, color and size of the vertices and the color and width of the edges. See tkplot for details.

rglplot is an experimental function to draw graphs in 3D using OpenGL. See rglplot for some more information.

Please also check the examples below.

How to specify graphical parameters

There are three ways to give values to the parameters described below, in section 'Parameters'. We give these three ways here in the order of their precedence.

The first method is to supply named arguments to the plotting commands: plot.igraph, tkplot or rglplot. Parameters for vertices start with prefix 'vertex.', parameters for edges have prefix 'edge.', and global parameters have now prefix. Eg. the color of the vertices can be given via argument vertex.color, whereas edge.color sets the color of the edges. layout gives the layout of the graphs.

The second way is to assign vertex, edge and graph attributes to the graph. These attributes have now prefix, ie. the color of the vertices is taken from the color vertex attribute and the color of the edges from the color edge attribute. The layout of the graph is given by the layout graph attribute. (Always assuming that the corresponding command argument is not present.) Setting vertex and edge attributes are handy if you want to assign a given 'look' to a graph, attributes are saved with the graph is you save it with save or in GraphML format with write.graph, so the graph will have the same look after loading it again.

If a parameter is not given in the command line, and the corresponding vertex/edge/graph attribute is also missing then the general igraph parameters handled by igraph.options are also checked. Vertex parameters have prefix 'vertex.', edge parameters are prefixed with 'edge.', general parameters like layout are prefixed with 'plot'. These parameters are useful if you want all or most of your graphs to have the same look, vertex size, vertex color, etc. Then you don't need to set these at every plotting, and you also don't need to assign vertex/edge attributes to every graph.

If the value of a parameter is not specified by any of the three ways described here, its default valued is used, as given in the source code.

Different parameters can have different type, eg. vertex colors can be given as a character vector with color names, or as an integer vector with the color numbers from the current palette. Different types are valid for different parameters, this is discussed in detail in the next section. It is however always true that the parameter can always be a function object in which it will be called with the graph as its single argument to get the "proper" value of the parameter. (If the function returns another function object that will *not* be called again. . .)

The list of parameters

Vertex parameters first, note that the 'vertex.' prefix needs to be added if they are used as an argument or when setting via igraph.options. The value of the parameter may be scalar valid for every vertex or a vector with a separate value for each vertex. (Shorter vectors are recycled.)

size The size of the vertex, a numeric scalar or vector, in the latter case each vertex sizes may differ. This vertex sizes are scaled in order have about the same size of vertices for a given value for all three plotting commands. It does not need to be an integer number.

The default value is 15. This is big enough to place short labels on vertices.

size2 The “other” size of the vertex, for some vertex shapes. For the various rectangle shapes this gives the height of the vertices, whereas `size` gives the width. It is ignored by shapes for which the size can be specified with a single number.

The default is 15.

color The fill color of the vertex. If it is numeric then the current palette is used, see `palette`. If it is a character vector then it may either contain named colors or RGB specified colors with three or four bytes. All strings starting with ‘#’ are assumed to be RGB color specifications. It is possible to mix named color and RGB colors. Note that `tkplot` ignores the fourth byte (alpha channel) in the RGB color specification.

If you don’t want (some) vertices to have any color, supply `NA` as the color name.

The default value is “`SkyBlue2`”.

frame.color The color of the frame of the vertices, the same formats are allowed as for the fill color.

If you don’t want vertices to have a frame, supply `NA` as the color name.

By default it is “black”.

shape The shape of the vertex, currently “`circle`”, “`square`”, “`csquare`”, “`rectangle`”, “`crectangle`”, “`vrectangle`”, “`pie`” and “`none`” are supported, and only by the `plot.igraph` command. “`none`” does not draw the vertices at all, although vertex label are plotted (if given). See `igraph.vertex.shapes` for details about vertex shapes and `vertex.shape.pie` for using pie charts as vertices.

By default vertices are drawn as circles.

label The vertex labels. They will be converted to character. Specify `NA` to omit vertex labels.

The default vertex labels are the vertex ids.

label.family The font family to be used for vertex labels. As different plotting commands can used different fonts, they interpret this parameter different ways. The basic notation is, however, understood by both `plot.igraph` and `tkplot`. `rglplot` does not support fonts at all right now, it ignores this parameter completely.

For `plot.igraph` this parameter is simply passed to `text` as argument `family`.

For `tkplot` some conversion is performed. If this parameter is the name of an exixting Tk font, then that font is used and the `label.font` and `label.cex` parameters are ignored complerely. If it is one of the base families (serif, sans, mono) then Times, Helvetica or Courier fonts are used, there are guaranteed to exist on all systems. For the ‘symbol’ base family we used the symbol font is available, otherwise the first font which has ‘symbol’ in its name. If the parameter is not a name of the base families and it is also not a named Tk font then we pass it to `tkfont.create` and hope the user knows what she is doing. The `label.font` and `label.cex` parameters are also passed to `tkfont.create` in this case.

The default value is ‘serif’.

label.font The font within the font family to use for the vertex labels. It is interpreted the same way as the the `font` graphical parameter: 1 is plain text, 2 is bold face, 3 is italic, 4 is bold and italic and 5 specifies the symbol font.

For `plot.igraph` this parameter is simply passed to `text`.

For `tkplot`, if the `label.family` parameter is not the name of a Tk font then this parameter is used to set whether the newly created font should be italic and/or boldface. Otherwise it is ignored.

For `rglplot` it is ignored.

The default value is 1.

label.cex The font size for vertex labels. It is interpreted as a multiplication factor of some device-dependent base font size.

For `plot.igraph` it is simply passed to `text` as argument `cex`.

For `tkplot` it is multiplied by 12 and then used as the `size` argument for `tkfont.create`. The base font is thus 12 for tkplot.

For `rglplot` it is ignored.

The default value is 1.

label.dist The distance of the label from the center of the vertex. If it is 0 then the label is centered on the vertex. If it is 1 then the label is displayed beside the vertex.

The default value is 0.

label.degree It defines the position of the vertex labels, relative to the center of the vertices. It is interpreted as an angle in radian, zero means 'to the right', and '`pi`' means to the left, up is `-pi/2` and down is `pi/2`.

The default value is `-pi/4`.

label.color The color of the labels, see the `color` vertex parameter discussed earlier for the possible values.

The default value is `black`.

Edge parameters require to add the '`edge.`' prefix when used as arguments or set by `igraph.options`. The edge parameters:

color The color of the edges, see the `color` vertex parameter for the possible values.

By default this parameter is `darkgrey`.

width The width of the edges.

The default value is 1.

arrow.size The size of the arrows. Currently this is a constant, so it is the same for every edge. If a vector is submitted then only the first element is used, ie. if this is taken from an edge attribute then only the attribute of the first edge is used for all arrows. This will likely change in the future.

The default value is 1.

arrow.width The width of the arrows. Currently this is a constant, so it is the same for every edge. If a vector is submitted then only the first element is used, ie. if this is taken from an edge attribute then only the attribute of the first edge is used for all arrows. This will likely change in the future.

This argument is currently only used by `plot.igraph`.

The default value is 1, which gives the same width as before this option appeared in igraph.

lty The line type for the edges. Almost the same format is accepted as for the standard graphics `par`, 0 and "blank" mean no edges, 1 and "solid" are for solid lines, the other possible values are: 2 ("dashed"), 3 ("dotted"), 4 ("dotdash"), 5 ("longdash"), 6 ("twodash").

`tkplot` also accepts standard Tk line type strings, it does not however support "blank" lines, instead of type '0' type '1', ie. solid lines will be drawn.

This argument is ignored for `rglplot`.

The default value is type 1, a solid line.

label The edge labels. They will be converted to character. Specify `NA` to omit edge labels.

Edge labels are omitted by default.

label.family Font family of the edge labels. See the vertex parameter with the same name for the details.

label.font The font for the edge labels. See the corresponding vertex parameter discussed earlier for details.

label.cex The font size for the edge labels, see the corresponding vertex parameter for details.

label.color The color of the edge labels, see the `color` vertex parameters on how to specify colors.

curved Specifies whether to draw curved edges, or not. This can be a logical or a numeric vector or scalar.

First the vector is replicated to have the same length as the number of edges in the graph. Then it is interpreted for each edge separately. A numeric value specifies the curvature of the edge; zero curvature means straight edges, negative values means the edge bends clockwise, positive values the opposite. `TRUE` means curvature 0.5, `FALSE` means curvature zero.

By default the vector specifying the curvatire is calculated via a call to the `autocurve.edges` function. This function makes sure that multiple edges are curved and are all visible. This parameter is ignored for loop edges.

The default value is `FALSE`.

This parameter is currently ignored by `rglplot`.

arrow.mode This parameter can be used to specify for which edges should arrows be drawn. If this parameter is given by the user (in either of the three ways) then it specifies which edges will have forward, backward arrows, or both, or no arrows at all. As usual, this parameter can be a vector or a scalar value. It can be an integer or character type. If it is integer then 0 means no arrows, 1 means backward arrows, 2 is for forward arrows and 3 for both. If it is a character vector then “<” and “<-” specify backward, “>” and “->” forward arrows and “<>” and “<->” stands for both arrows. All other values mean no arrows, perhaps you should use “-” or “–” to specify no arrows.

Hint: this parameter can be used as a ‘cheap’ solution for drawing “mixed” graphs: graphs in which some edges are directed some are not. If you want do this, then please create a *directed* graph, because as of version 0.4 the vertex pairs in the edge lists can be swapped in undirected graphs.

By default, no arrows will be drawn for undirected graphs, and for directed graphs, an arrow will be drawn for each edge, according to its direction. This is not very surprising, it is the expected behavior.

loop.angle Gives the angle in radian for plotting loop edges. See the `label.dist` vertex parameter to see how this is interpreted.

The default value is 0.

loop.angle2 Gives the second angle in radian for plotting loop edges. This is only used in 3D, `loop.angle` is enough in 2D.

The default value is 0.

Other parameters:

layout Either a function or a numeric matrix. It specifies how the vertices will be placed on the plot.

If it is a numeric matrix, then the matrix has to have one line for each vertex, specifying its coordinates. The matrix should have at least two columns, for the `x` and `y` coordinates, and it can also have third column, this will be the `z` coordinate for 3D plots and it is ignored for 2D plots.

If a two column matrix is given for the 3D plotting function `rglplot` then the third column is assumed to be 1 for each vertex.

If `layout` is a function, this function will be called with the `graph` as the single parameter to determine the actual coordinates. The function should return a matrix with two or three columns. For the 2D plots the third column is ignored.

The default value is `layout.random`, ie. a function returning with 2D random placement.

margin The amount of empty space below, over, at the left and right of the plot, it is a numeric vector of length four. Usually values between 0 and 0.5 are meaningful, but negative values are also possible, that will make the plot zoom in to a part of the graph. If it is shorter than four then it is recycled.

rglplot does not support this parameter, as it can zoom in and out the graph in a more flexible way.

Its default value is 0.

rescale Logical constant, whether to rescale the coordinates to the [-1,1]x[-1,1](x[-1,1]) interval. This parameter is not implemented for `tkplot`.

Defaults to `TRUE`, the layout will be rescaled.

asp A numeric constant, it gives the `asp` parameter for plot, the aspect ratio. Supply 0 here if you don't want to give an aspect ratio. It is ignored by `tkplot` and `rglplot`.

Defaults to 1.

frame Boolean, whether to plot a frame around the graph. It is ignored by `tkplot` and `rglplot`.

Defaults to `FALSE`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

plot.igraph, tkplot, rglplot, igraph.options

Examples

```
## Not run:

# plotting a simple ring graph, all default parameters, except the layout
g <- graph.ring(10)
g$layout <- layout.circle
plot(g)
tkplot(g)
rglplot(g)

# plotting a random graph, set the parameters in the command arguments
g <- barabasi.game(100)
plot(g, layout=layout.fruchterman.reingold, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)

# plot a random graph, different color for each component
g <- erdos.renyi.game(100, 1/100)
comps <- clusters(g)$membership
colbar <- rainbow(max(comps)+1)
V(g)$color <- colbar[comps+1]
plot(g, layout=layout.fruchterman.reingold, vertex.size=5, vertex.label=NA)

# plot communities in a graph
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(0,5, 0,10, 5,10))
com <- spinglass.community(g, spins=5)
V(g)$color <- com$membership+1
g <- set.graph.attribute(g, "layout", layout.kamada.kawai(g))
plot(g, vertex.label.dist=1.5)

# draw a bunch of trees, fix layout
```

```
igraph.options(plot.layout=layout.reingold.tilford)
plot(graph.tree(20, 2))
plot(graph.tree(50, 3), vertex.size=3, vertex.label=NA)
tkplot(graph.tree(50, 2, mode="undirected"), vertex.size=10, vertex.color="green")

## End(Not run)
```

`dyad.census` *Dyad census of a graph*

Description

Classify dyads in a directed graphs. The relationship between each pair of vertices is measured. It can be in three states: mutual, asymmetric or non-existent.

Usage

```
dyad.census(graph)
```

Arguments

`graph`	The input graph. A warning is given if it is not directed.

Value

A named numeric vector with three elements:

`mut`	The number of pairs with mutual connections.
`asym`	The number of pairs with non-mutual connections.
`null`	The number of pairs with no connection between them.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

References

Holland, P.W. and Leinhardt, S. A Method for Detecting Structure in Sociometric Data. *American Journal of Sociology*, 76, 492–513. 1970.

Wasserman, S., and Faust, K. *Social Network Analysis: Methods and Applications.* Cambridge: Cambridge University Press. 1994.

See Also

`triad.census` for the same classification, but with triples.

Examples

```
g <- ba.game(100)
dyad.census(g)
```

`eccentricity` *Eccentricity and radius*

Description

The eccentricity of a vertex is its shortest path distance from the farthest other node in the graph. The smallest eccentricity in a graph is called its radius

Usage

```
eccentricity(graph, vids=V(graph), mode=c("all", "out", "in", "total"))
radius(graph, mode=c("all", "out", "in", "total"))
```

Arguments

`graph`	The input graph, it can be directed or undirected.
`vids`	The vertices for which the eccentricity is calculated.
`mode`	Character constant, gives whether the shortest paths to or from the given vertices should be calculated for directed graphs. If `out` then the shortest paths *from* the vertex, if `in` then *to* it will be considered. If `all`, the default, then the corresponding undirected graph will be used, edge directions will be ignored. This argument is ignored for undirected graphs.

Details

The eccentricity of a vertex is calculated by measuring the shortest distance from (or to) the vertex, to (or from) all vertices in the graph, and taking the maximum.

This implementation ignores vertex pairs that are in different components. Isolate vertices have eccentricity zero.

Value

`eccentricity` returns a numeric vector, containing the eccentricity score of each given vertex.

`radius` returns a numeric scalar.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Harary, F. Graph Theory. Reading, MA: Addison-Wesley, p. 35, 1994.

See Also

`shortest.paths` for general shortest path calculations.

Examples

```
g <- graph.star(10, mode="undirected")
eccentricity(g)
radius(g)
```

`edge.betweenness.community` *Community structure detection based on edge betweenness*

Description

Many networks consist of modules which are densely connected themselves but sparsely connected to other modules.

Usage

```
edge.betweenness.community (graph, weights = E(graph)$weight,
    directed = TRUE, edge.betweenness = TRUE, merges = TRUE,
    bridges = TRUE, modularity = TRUE, membership = TRUE)
```

Arguments

`graph`	The graph to analyze.
`weights`	The edge weights. Supply `NULL` to omit edge weights. By default the '`weight`' edge attribute is used, if it is present.
`directed`	Logical constant, whether to calculate directed edge betweenness for directed graphs. It is ignored for undirected graphs.
`edge.betweenness`	Logical constant, whether to return the edge betweenness of the edges at the time of their removal.
`merges`	Logical constant, whether to return the merge matrix representing the hierarchical community structure of the network. This argument is called `merges`, even if the community structure algorithm itself is divisive and not agglomerative: it builds the tree from top to bottom. There is one line for each merge (i.e. split) in matrix, the first line is the first merge (last split). The communities are identified by integer number starting from one. Community ids smaller than or equal to N, the number of vertices in the graph, belong to singleton communities, ie. individual vertices. Before the first merge we have N communities numbered from one to N. The first merge, the first line of the matrix creates community $N+1$, the second merge creates community $N+2$, etc.
`bridges`	Logical constant, whether to return a list the edge removals which actually splitted a component of the graph.
`modularity`	Logical constant, whether to calculate the maximum modularity score, considering all possibly community structures along the edge-betweenness based edge removals.
`membership`	Logical constant, whether to calculate the membership vector corresponding to the highest possible modularity score.

Details

The edge betweenness score of an edge measures the number of shortest paths through it, see `edge.betweenness` for details. The idea of the edge betweenness based community structure detection is that it is likely that edges connecting separate modules have high edge betweenness as all the shortest paths from one module to another must traverse through them. So if we gradually remove the edge with the highest edge betweenness score we will get a hierarchical map, a rooted tree, called a dendrogram of the graph. The leafs of the tree are the individual vertices and the root of the tree represents the whole graph.

`edge.betweenness.community` performs this algorithm by calculating the edge betweenness of the graph, removing the edge with the highest edge betweenness score, then recalculating edge betweenness of the edges and again removing the one with the highest score, etc.

`edge.betweeness.community` returns various information collected throught the run of the algorithm. See the return value down here.

Value

`edge.betweenness.community` returns a `communities` object, please see the `communities` manual page for details.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

M Newman and M Girvan: Finding and evaluating community structure in networks, *Physical Review E* 69, 026113 (2004)

See Also

`edge.betweenness` for the definition and calculation of the edge betweenness, `walktrap.community`, `fastgreedy.community`, `leading.eigenvector.community` for other community detection methods.

See `communities` for extracting the results of the community detection.

Examples

```
g <- barabasi.game(100,m=2)
eb <- edge.betweenness.community(g)

g <- graph.full(10) %du% graph.full(10)
g <- add.edges(g, c(1,11))
eb <- edge.betweenness.community(g)
eb
```

`edge.connectivity` *Edge connectivity.*

Description

The edge connectivity of a graph or two vertices, this is recently also called group adhesion.

Usage

```
edge.connectivity(graph, source=NULL, target=NULL, checks=TRUE)
edge.disjoint.paths(graph, source, target)
graph.adhesion(graph, checks=TRUE)
```

Arguments

`graph`	The input graph.
`source`	The id of the source vertex, for `edge.connectivity` it can be `NULL`, see details below.
`target`	The id of the target vertex, for `edge.connectivity` it can be `NULL`, see details below.
`checks`	Logical constant. Whether to check that the graph is connected and also the degree of the vertices. If the graph is not (strongly) connected then the connectivity is obviously zero. Otherwise if the minimum degree is one then the edge connectivity is also one. It is a good idea to perform these checks, as they can be done quickly compared to the connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter.

Details

The edge connectivity of a pair of vertices (`source` and `target`) is the minimum number of edges needed to remove to eliminate all (directed) paths from `source` to `target`. `edge.connectivity` calculates this quantity if both the `source` and `target` arguments are given (and not `NULL`).

The edge connectivity of a graph is the minimum of the edge connectivity of every (ordered) pair of vertices in the graph. `edge.connectivity` calculates this quantity if neither the `source` nor the `target` arguments are given (ie. they are both `NULL`).

A set of edge disjoint paths between two vertices is a set of paths between them containing no common edges. The maximum number of edge disjoint paths between two vertices is the same as their edge connectivity.

The adhesion of a graph is the minimum number of edges needed to remove to obtain a graph which is not strongly connected. This is the same as the edge connectivity of the graph.

The three functions documented on this page calculate similar properties, more precisely the most general is `edge.connectivity`, the others are included only for having more descriptive function names.

Value

A scalar real value.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Douglas R. White and Frank Harary: The cohesiveness of blocks in social networks: node connectivity and conditional density, TODO: citation

See Also

`graph.maxflow`, `vertex.connectivity`, `vertex.disjoint.paths`, `graph.cohesion`

Examples

```
g <- barabasi.game(100, m=1)
g2 <- barabasi.game(100, m=5)
edge.connectivity(g, 100, 1)
edge.connectivity(g2, 100, 1)
edge.disjoint.paths(g2, 100, 1)

g <- erdos.renyi.game(50, 5/50)
g <- as.directed(g)
g <- induced.subgraph(g, subcomponent(g, 1))
graph.adhesion(g)
```

---

erdos.renyi.game *Generate random graphs according to the Erdos-Renyi model*

Description

This model is very simple, every possible edge is created with the same constant probability.

Usage

```
erdos.renyi.game(n, p.or.m, type=c("gnp", "gnm"),
                 directed = FALSE, loops = FALSE, ...)
```

Arguments

n	The number of vertices in the graph.
p.or.m	Either the probability for drawing an edge between two arbitrary vertices (G(n,p) graph), or the number of edges in the graph (for G(n,m) graphs).
type	The type of the random graph to create, either gnp (G(n,p) graph) or gnm (G(n,m) graph).
directed	Logical, whether the graph will be directed, defaults to FALSE.
loops	Logical, whether to add loop edges, defaults to FALSE.
...	Additional arguments, ignored.

Details

In G(n,p) graphs, the graph has ‘n’ vertices and for each edge the probability that it is present in the graph is ‘p’.

In G(n,m) graphs, the graph has ‘n’ vertices and ‘m’ edges, and the ‘m’ edges are chosen uniformly randomly from the set of all possible edges. This set includes loop edges as well if the `loops` parameter is TRUE.

`random.graph.game` is an alias to this function.

Value

A graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Erdos, P. and Renyi, A., On random graphs, *Publicationes Mathematicae* 6, 290–297 (1959).

See Also

barabasi.game

Examples

```
g <- erdos.renyi.game(1000, 1/1000)
degree.distribution(g)
```

---

evcent *Find Eigenvector Centrality Scores of Network Positions*

Description

evcent takes a graph (graph) and returns the eigenvector centralities of positions v within it

Usage

```
evcent (graph, directed = FALSE, scale = TRUE, weights = NULL,
     options = igraph.arpack.default)
```

Arguments

graph	Graph to be analyzed.
directed	Logical scalar, whether to consider direction of the edges in directed graphs. It is ignored for undirected graphs.
scale	Logical scalar, whether to scale the result to have a maximum score of one. If no scaling is used then the result vector has unit length in the Euclidean norm.
weights	A numerical vector or NULL. This argument can be used to give edge weights for calculating the weighted eigenvector centrality of vertices. If this is NULL and the graph has a weight edge attribute then that is used. If weights is a numerical vector then it used, even if the graph has a weights edge attribute. If this is NA, then no edge weights are used (even if the graph has a weight edge attribute. Note that if there are negative edge weights and the direction of the edges is considered, then the eigenvector might be complex. In this case only the real part is reported.
options	A named list, to override some ARPACK options. See arpack for details.

Details

Eigenvector centrality scores correspond to the values of the first eigenvector of the graph adjacency matrix; these scores may, in turn, be interpreted as arising from a reciprocal process in which the centrality of each actor is proportional to the sum of the centralities of those actors to whom he or she is connected. In general, vertices with high eigenvector centralities are those which are connected to many other vertices which are, in turn, connected to many others (and so on). (The perceptive may realize that this implies that the largest values will be obtained by individuals in large cliques (or high-density substructures). This is also intelligible from an algebraic point of view, with the first eigenvector being closely related to the best rank-1 approximation of the adjacency matrix (a relationship which is easy to see in the special case of a diagonalizable symmetric real matrix via the SLS^-1 decomposition).)

From igraph version 0.5 this function uses ARPACK for the underlying computation, see `arpack` for more about ARPACK in igraph.

Value

A named list with components:

`vector`	A vector containing the centrality scores.
`value`	The eigenvalue corresponding to the calculated eigenvector, i.e. the centrality scores.
`options`	A named list, information about the underlying ARPACK computation. See `arpack` for the details.

WARNING

`evcent` will not symmetrize your data before extracting eigenvectors; don't send this routine asymmetric matrices unless you really mean to do so.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`> and Carter T. Butts <`buttsc@uci.edu`> for the manual page

References

Bonacich, P. (1987). Power and Centrality: A Family of Measures. *American Journal of Sociology*, 92, 1170-1182.

Katz, L. (1953). A New Status Index Derived from Sociometric Analysis. *Psychometrika*, 18, 39-43.

Examples

```
#Generate some test data
g <- graph.ring(10, directed=FALSE)
#Compute eigenvector centrality scores
evcent(g)
```

`fastgreedy.community` *Community structure via greedy optimization of modularity*

Description

This function tries to find dense subgraph, also called communities in graphs via directly optimizing a modularity score.

Usage

```
fastgreedy.community(graph, merges=TRUE, modularity=TRUE,
      membership=TRUE, weights=E(graph)$weight)
```

Arguments

`graph`	The input graph
`merges`	Logical scalar, whether to return the merge matrix.
`modularity`	Logical scalar, whether to return a vector containing the modularity after each merge.
`membership`	Logical scalar, whether to calculate the membership vector corresponding to the maximum modularity score, considering all possible community structures along the merges.
`weights`	If not `NULL`, then a numeric vector of edge weights. The length must match the number of edges in the graph. By default the '`weight`' edge attribute is used as weights. If it is not present, then all edges are considered to have the same weight.

Details

This function implements the fast greedy modularity optimization algorithm for finding community structure, see A Clauset, MEJ Newman, C Moore: Finding community structure in very large networks, http://www.arxiv.org/abs/cond-mat/0408187 for the details.

Value

`fastgreedy.community` returns a `communities` object, please see the `communities` manual page for details.

Author(s)

Tamas Nepusz `<ntamas@gmail.com>` and Gabor Csardi `<csardi.gabor@gmail.com>` for the R interface.

References

A Clauset, MEJ Newman, C Moore: Finding community structure in very large networks, http://www.arxiv.org/abs/cond-mat/0408187

See Also

communities for extracting the results.

See also walktrap.community, spinglass.community, leading.eigenvector.community and edge.betweenness.community for other methods.

Examples

```
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(1,6, 1,11, 6, 11))
fc <- fastgreedy.community(g)
membership(fc)
sizes(fc)
```

`forest.fire.game` *Forest Fire Network Model*

Description

This is a growing network model, which resembles of how the forest fire spreads by igniting trees close by.

Usage

```
forest.fire.game (nodes, fw.prob, bw.factor = 1, ambs = 1, directed = TRUE)
```

Arguments

`nodes`	The number of vertices in the graph.
`fw.prob`	The forward burning probability, see details below.
`bw.factor`	The backward burning ratio. The backward burning probability is calculated as `bw.factor*fw.prob`.
`ambs`	The number of ambassador vertices.
`directed`	Logical scalar, whether to create a directed graph.

Details

The forest fire model intends to reproduce the following network characteristics, observed in real networks:

- Heavy-tailed in-degree distribution.
- Heavy-tailed out-degree distribution.
- Communities.
- Densification power-law. The network is densifying in time, according to a power-law rule.
- Shrinking diameter. The diameter of the network decreases in time.

The network is generated in the following way. One vertex is added at a time. This vertex connects to (cites) `ambs` vertices already present in the network, chosen uniformly random. Now, for each cited vertex v we do the following procedure:

1. We generate two random number, x and y, that are geometrically distributed with means $p/(1-p)$ and $rp(1-rp)$. (p is `fw.prob`, r is `bw.factor`.) The new vertex cites x outgoing neighbors and y incoming neighbors of v, from those which are not yet cited by the new vertex. If there are less than x or y such vertices available then we cite all of them.
2. The same procedure is applied to all the newly cited vertices.

Value

A simple graph, possibly directed if the `directed` argument is `TRUE`.

Note

The version of the model in the published paper is incorrect in the sense that it cannot generate the kind of graphs the authors claim. A corrected version is available from `http://www.cs.cmu.edu/~jure/pubs/powergrowth-tkdd.pdf`, our implementation is based on this.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Jure Leskovec, Jon Kleinberg and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 177–187, 2005.

See Also

`barabasi.game` for the basic preferential attachment model.

Examples

```
g <- forest.fire.game(10000, fw.prob=0.37, bw.factor=0.32/0.37)
dd1 <- degree.distribution(g, mode="in")
dd2 <- degree.distribution(g, mode="out")
if (interactive()) {
  plot(seq(along=dd1)-1, dd1, log="xy")
  points(seq(along=dd2)-1, dd2, col=2, pch=2)
}
```

`get.adjlist` *Adjacency lists*

Description

Create adjacency lists from a graph, either for adjacent edges or for neighboring vertices

Usage

```
get.adjlist(graph, mode = c("all", "out", "in", "total"))
get.adjedgelist(graph, mode = c("all", "out", "in", "total"))
```

Arguments

`graph`	The input graph.
`mode`	Character scalar, it gives what kind of adjacent edges/vertices to include in the lists. '`out`' is for outgoing edges/vertices, '`in`' is for incoming edges/vertices, '`all`' is for both. This argument is ignored for undirected graphs.

Details

`get.adjlist` returns a list of numeric vectors, which include the ids of neighbor vertices (according to the `mode` argument) of all vertices.

`get.adjedgelist` returns a list of numeric vectors, which include the ids of adjacent edgs (according to the `mode` argument) of all vertices.

Value

A list of numeric vectors.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`get.edgelist`, `get.adjacency`

Examples

```
g <- graph.ring(10)
get.adjlist(g)
get.adjedgelist(g)
```

`get.edge.ids` *Find the edge ids based on the incident vertices of the edges*

Description

Find the edges in an igraph graph that have the specified end points. This function handles multigraph (graphs with multiple edges) and can consider or ignore the edge directions in directed graphs.

Usage

```
get.edge.ids(graph, vp, directed = TRUE, error = FALSE, multi = FALSE)
```

Arguments

`graph`	The input graph.
`vp`	The indicent vertices, given as vertex ids or symbolic vertex names. They are interpreted pairwise, i.e. the first and second are used for the first edge, the third and fourth for the second, etc.
`directed`	Logical scalar, whether to consider edge directions in directed graphs. This argument is ignored for undirected graphs.

`error`	Logical scalar, whether to report an error if an edge is not found in the graph. If `FALSE`, then no error is reported, and zero is returned for the non-existant edge(s).
`multi`	Logical scalar, whether to handle multiple edges properly. If `FALSE`, and a pair of vertices are given twice (or more), then always the same edge id is reported back for them. If `TRUE`, then the edge ids of multiple edges are correctly reported.

Details

igraph vertex ids are natural numbers, starting from one, up to the number of vertices in the graph. Similarly, edges are also numbered from one, up to the number of edges.

This function allows finding the edges of the graph, via their incident vertices.

Value

A numeric vector of edge ids, one for each pair of input vertices. If there is no edge in the input graph for a given pair of vertices, then zero is reported. (If the `error` argument is `FALSE`.)

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
g <- graph.ring(10)
ei <- get.edge.ids(g, c(1,2, 4,5))
E(g)[ei]

## non-existant edge
get.edge.ids(g, c(2,1, 1,4, 5,4))
```

`get.incidence` *Incidence matrix of a bipartite graph*

Description

This function can return a sparse or dense incidence matrix of a bipartite network. The incidence matrix is an n times m matrix, n and m are the number of vertices of the two kinds.

Usage

```
get.incidence(graph, types=NULL, attr=NULL, names=TRUE, sparse=FALSE)
```

Arguments

`graph`	The input graph. The direction of the edges is ignored in directed graphs.
`types`	An optional vertex type vector to use instead of the `type` vertex attribute. You must supply this argument if the graph has no `type` vertex attribute.

`attr`	Either `NULL` or a character string giving an edge attribute name. If `NULL`, then a traditional incidence matrix is returned. If not `NULL` then the values of the given edge attribute are included in the incidence matrix. If the graph has multiple edges, the edge attribute of an arbitrarily chosen edge (for the multiple edges) is included.
`names`	Logical scalar, if `TRUE` and the vertices in the graph are named (i.e. the graph has a vertex attribute called `name`), then vertex names will be added to the result as row and column names. Otherwise the ids of the vertices are used as row and column names.
`sparse`	Logical scalar, if it is `TRUE` then a sparse matrix is created, you will need the `Matrix` package for this.

Details

Bipartite graphs have a `type` vertex attribute in igraph, this is boolean and `FALSE` for the vertices of the first kind and `TRUE` for vertices of the second kind.

Value

A sparse or dense matrix.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`graph.incidence` for the opposite operation.

Examples

```
g <- graph.bipartite( c(0,1,0,1,0,0), c(1,2,2,3,3,4) )
get.incidence(g)
```

---

get.stochastic *Stochastic matrix of a graph*

Description

Retrieves the stochastic matrix of a graph of class `igraph`.

Usage

```
get.stochastic(graph, column.wise = FALSE, sparse =
      getIgraphOpt("sparsematrices"))
```

Arguments

`graph`	The input graph. Must be of class `igraph`.
`column.wise`	If `FALSE`, then the rows of the stochastic matrix sum up to one; otherwise it is the columns.
`sparse`	Logical scalar, whether to return a sparse matrix. The `Matrix` package is needed for sparse matrices.

Details

Let M be an $n \times n$ adjacency matrix with real non-negative entries. Let us define $D = \mathrm{diag}(\sum_i M_{1i}, \dots, \sum_i M_{ni})$

The (row) stochastic matrix is defined as

$$W = D^{-1}M,$$

where it is assumed that D is non-singular. Column stochastic matrices are defined in a symmetric way.

Value

A regular R matrix or a matrix of class `Matrix` if a `sparse` argument was `TRUE`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`get.adjacency`

Examples

```
## g is a large sparse graph
g <- barabasi.game(n = 10^5, power = 2, directed = FALSE)
W <- get.stochastic(g, sparse=TRUE)

## a dense matrix here would probably not fit in the memory
class(W)

## may not be exactly 1, due to numerical errors
max(abs(rowSums(W))-1)
```

`girth` *Girth of a graph*

Description

The girth of a graph is the length of the shortest circle in it.

Usage

```
girth(graph, circle=TRUE)
```

Arguments

`graph`	The input graph. It may be directed, but the algorithm searches for undirected circles anyway.
`circle`	Logical scalar, whether to return the shortest circle itself.

Details

The current implementation works for undirected graphs only, directed graphs are treated as undirected graphs. Loop edges and multiple edges are ignored. If the graph is a forest (ie. acyclic), then zero is returned.

This implementation is based on Alon Itai and Michael Rodeh: Finding a minimum circuit in a graph *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1-10, 1977. The first implementation of this function was done by Keith Briggs, thanks Keith.

Value

A named list with two components:

`girth`	Integer constant, the girth of the graph, or 0 if the graph is acyclic.
`circle`	Numeric vector with the vertex ids in the shortest circle.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Alon Itai and Michael Rodeh: Finding a minimum circuit in a graph *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1-10, 1977

Examples

```
# No circle in a tree
g <- graph.tree(1000, 3)
girth(g)

# The worst case running time is for a ring
g <- graph.ring(100)
girth(g)

# What about a random graph?
g <- erdos.renyi.game(1000, 1/1000)
girth(g)
```

`graph-isomorphism` *Graph Isomorphism*

Description

These functions deal with graph isomorphism.

Usage

```
graph.isomorphic(graph1, graph2)
graph.isomorphic.34(graph1, graph2)
graph.isomorphic.bliss(graph1, graph2, sh1="fm", sh2="fm")
graph.isomorphic.vf2(graph1, graph2, vertex.color1, vertex.color2,
     edge.color1, edge.color2)

graph.count.isomorphisms.vf2(graph1, graph2,
     vertex.color1, vertex.color2,
     edge.color1, edge.color2)
graph.get.isomorphisms.vf2(graph1, graph2,
     vertex.color1, vertex.color2,
     edge.color1, edge.color2)

graph.subisomorphic.vf2(graph1, graph2,
     vertex.color1, vertex.color2,
     edge.color1, edge.color2)
graph.count.subisomorphisms.vf2(graph1, graph2,
     vertex.color1, vertex.color2,
     edge.color1, edge.color2)
graph.get.subisomorphisms.vf2(graph1, graph2,
     vertex.color1, vertex.color2,
     edge.color1, edge.color2)

graph.isoclass(graph)
graph.isoclass.subgraph(graph, vids)
graph.isocreate(size, number, directed=TRUE)
```

Arguments

`graph`	A graph object.
`graph1,graph2`	Graph objects
`vertex.color1,vertex.color2`	Optional integer vectors giving the colors of the vertices for colored (sub)graph isomorphism. If they are not given, but the graph has a “color” vertex attribute, then it will be used. If you want to ignore these attributes, then supply `NULL` for both of these arguments. See also examples below.
`edge.color1,edge.color2`	Optional integer vectors giving the colors of the edges for edge-colored (sub)graph isomorphism. If they are not given, but the graph has a “color” edge attribute, then it will be used. If you want to ignore these attributes, then supply `NULL` for both of these arguments.
`size`	A numeric integer giving the number of vertices in the graph to create. Only three or four are suppported right now.
`number`	The number of the isomorphism class of the graph to be created.
`directed`	Whether to create a directed graph.
`sh1`	Character constant, the heuristics to use in the BLISS algorithm, for `graph1`. See the `sh` argument of `canonical.permutation` for possible values.
`sh2`	Character constant, the heuristics to use in the BLISS algorithm, for `graph2`. See the `sh` argument of `canonical.permutation` for possible values.

`vids`	Numeric vector, the vertex ids of vertices to form the induced subgraph for determining the isomorphism class.

Details

`graph.isomorphic` decides whether two graphs are isomorphic. The input graphs must be both directed or both undirected. This function is a higher level interface to the other graph isomorphism decision functions. Currently it does the following:

1. If the two graphs do not agree in the number of vertices and the number of edges then `FALSE` is returned.
2. Otherwise if the graphs have 3 or 4 vertices, then `igraph.isomorphic.34` is called.
3. Otherwise if the graphs are directed, then `igraph.isomorphic.vf2` is called.
4. Otherwise `igraph.isomorphic.bliss` is called.

`igraph.isomorphic.34` decides whether two graphs, both of which contains only 3 or 4 vertices, are isomorphic. It works based on a precalculated and stored table.

`igraph.isomorphic.bliss` uses the BLISS algorithm by Junttila and Kaski, and it works for undirected graphs. For both graphs the `canonical.permutation` and then the `permute.vertices` function is called to transfer them into canonical form; finally the canonical forms are compared.

`graph.isomorphic.vf2` decides whethe two graphs are isomorphic, it implements the VF2 algorithm, by Cordella, Foggia et al., see references.

`graph.count.isomorphisms.vf2` counts the different isomorphic mappings between `graph1` and `graph2`. (To count automorphisms you can supply the same graph twice, but it is better to call `graph.automorphisms`.) It uses the VF2 algorithm.

`graph.get.isomorphisms.vf2` calculates all isomorphic mappings between `graph1` and `graph2`. It uses the VF2 algorithm.

`graph.subisomorphic.vf2` decides whether `graph2` is isomorphic to some subgraph of `graph1`. It uses the VF2 algorithm.

`graph.count.subisomorphisms.vf2` counts the different isomorphic mappings between `graph2` and the subgraphs of `graph1`. It uses the VF2 algorithm.

`graph.get.subisomorphisms.vf2` calculates all isomorphic mappings between `graph2` and the subgraphs of `graph1`. It uses the VF2 algorithm.

`graph.isoclass` returns the isomorphism class of a graph, a non-negative integer number. Graphs (with the same number of vertices) having the same isomorphism class are isomorphic and isomorphic graphs always have the same isomorphism class. Currently it can handle only graphs with 3 or 4 vertices.

`graph.isoclass.subgraph` calculates the isomorphism class of a subgraph of the input graph. Currently it only works for subgraphs with 3 or 4 vertices.

`graph.isocreate` create a graph from the given isomorphic class. Currently it can handle only graphs with 3 or 4 vertices.

Value

`graph.isomorphic` and `graph.isomorphic.34` return a logical scalar, `TRUE` if the input graphs are isomorphic, `FALSE` otherwise.

`graph.isomorphic.bliss` returns a named list with elements:

`iso`	A logical scalar, whether the two graphs are isomorphic.

`map12`	A numeric vector, an mapping from `graph1` to `graph2` if `iso` is `TRUE`, an empty numeric vector otherwise.
`map21`	A numeric vector, an mapping from `graph2` to `graph1` if `iso` is `TRUE`, an empty numeric vector otherwise.
`info1`	Some information about the canonical form calculation for `graph1`. A named list, see the return value of `canonical.permutation` for details.
`info2`	Some information about the canonical form calculation for `graph2`. A named list, see the return value of `canonical.permutation` for details.

`graph.isomorphic.vf2` returns a names list with three elements:

`iso`	A logical scalar, whether the two graphs are isomorphic.
`map12`	A numeric vector, an mapping from `graph1` to `graph2` if `iso` is `TRUE`, an empty numeric vector otherwise.
`map21`	A numeric vector, an mapping from `graph2` to `graph1` if `iso` is `TRUE`, an empty numeric vector otherwise.

`graph.count.isomorphisms.vf2` returns a numeric scalar, an integer, the number of isomorphic mappings between the two input graphs.

`graph.get.isomorphisms.vf2` returns a list of numeric vectors. Every numeric vector is a permutation which takes `graph2` into `graph1`.

`graph.subisomorphic.vf2` returns a named list with three elements:

`iso`	Logical scalar, `TRUE` if a subgraph of `graph1` is isomorphic to `graph2`.
`map12`	Numeric vector, empty if `iso` is `FALSE`. Otherwise a mapping from a subgraph of `graph1` to `graph2`. -1 denotes the vertices which are not part of the mapping.
`map21`	Numeric vector, empty if `iso` is `FALSE`. Otherwise a mapping from `graph2` into `graph1`.

`graph.count.subisomorphisms.vf2` returns a numeric scalar, an integer.

`graph.get.subisomorphisms.vf2` returns a list of numeric vectors, each numeric vector is an isomorphic mapping from `graph2` to a subgraph of `graph1`.

`graph.isoclass` and `graph.isoclass.subgraph` return a non-negative integer number.

`graph.isocreate` returns a graph object.

Note

Functions `graph.isoclass`, `graph.isoclass.subgraph` and `graph.isocreate` are considered experimental and might be reorganized/rewritten later.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Tommi Junttila and Petteri Kaski: Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics.* 2007.

LP Cordella, P Foggia, C Sansone, and M Vento: An improved algorithm for matching large graphs, *Proc. of the 3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition*, 149–159, 2001.

See Also

graph.motifs

Examples

```
# create some non-isomorphic graphs
g1 <- graph.isocreate(3, 10)
g2 <- graph.isocreate(3, 11)
graph.isoclass(g1)
graph.isoclass(g2)
graph.isomorphic(g1, g2)

# create two isomorphic graphs, by
# permuting the vertices of the first
g1 <- barabasi.game(30, m=2, directed=FALSE)
g2 <- permute.vertices(g1, sample(vcount(g1)))
# should be TRUE
graph.isomorphic(g1, g2)
graph.isomorphic.bliss(g1, g2)
graph.isomorphic.vf2(g1, g2)

# colored graph isomorphism
g1 <- graph.ring(10)
g2 <- graph.ring(10)
graph.isomorphic.vf2(g1, g2)

V(g1)$color <- rep(1:2, length=vcount(g1))
V(g2)$color <- rep(1:2, length=vcount(g2))
graph.count.isomorphisms.vf2(g1, g2)
graph.count.isomorphisms.vf2(g1, g2, vertex.color1=NULL,
    vertex.color2=NULL)

V(g1)$color <- 1
V(g2)$color <- 2
graph.isomorphic.vf2(g1, g2)
graph.isomorphic.vf2(g2, g2, vertex.color1=NULL,
    vertex.color2=NULL)
```

---

graph-motifs *Graph motifs*

Description

Graph motifs are small subgraphs with a well-defined strucure. These functions search a graph for various motifs.

Usage

```
graph.motifs(graph, size = 3, cut.prob = rep(0, size))
graph.motifs.no(graph, size = 3, cut.prob = rep(0, size))
graph.motifs.est(graph, size = 3, cut.prob = rep(0, size), sample.size =
    vcount(graph)/10, sample = NULL)
```

Arguments

`graph`	Graph object, the input graph.
`size`	The size of the motif, currently 3 and 4 are supported only.
`cut.prob`	Numeric vector giving the probabilities that the search graph is cut at a certain level. Its length should be the same as the size of the motif (the `size` argument). By default no cuts are made.
`sample.size`	The number of vertices to use as a starting point for finding motifs. Only used if the `sample` argument is `NULL`.
`sample`	If not `NULL` then it specifies the vertices to use as a starting point for finding motifs.

Details

`graph.motifs` searches a graph for motifs of a given size and returns a numeric vector containing the number of different motifs. The order of the motifs is defined by their isomorphism class, see `graph.isoclass`.

`graph.motifs.no` calculates the total number of motifs of a given size in graph.

`graph.motifs.est` estimates the total number of motifs of a given size in a graph based on a sample.

Value

`graph.motifs` returns a numeric vector.

`graph.motifs.no` and `graph.motifs.est` return a numeric constant.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`graph.isoclass`

Examples

```
g <- barabasi.game(100)
graph.motifs(g, 3)
graph.motifs.no(g, 3)
graph.motifs.est(g, 3)
```

`graph-operators` *Graph operators*

Description

Graph operators handle graphs in terms of set theory.

Usage

```
graph.union(...)
graph.disjoint.union(...)
graph.intersection(...)
graph.compose(g1, g2)
graph.difference(big, small)
graph.complementer(graph, loops=FALSE)
x %c% y
x %du% y
x %m% y
x %s% y
x %u% y
```

Arguments

`...`	Graph objects or lists of graph objects.
`g1,g2,big,small,graph,x,y`	Graph objects.
`loops`	Logical constant, whether to generate loop edges.

Details

A graph is homogenous binary relation over the set 0, . . . , |V|-1, |V| is the number of vertices in the graph. A homogenous binary relation is a set of ordered (directed graphs) or unordered (undirected graphs) pairs taken from 0, . . . , |V|-1. The functions documented here handle graphs as relations.

`graph.union` creates the union of two or more graphs. Ie. only edges which are included in at least one graph will be part of the new graph. This function can be also used via the %u% operator.

`graph.disjoint.union` creates a union of two or more disjoint graphs. Thus first the vertices in the second, third, etc. graphs are relabeled to have completely disjoint graphs. Then a simple union is created. This function can also be used via the %du% operator.

`graph.intersection` creates the intersection of two or more graphs: only edges present in all graphs will be included. The corresponding operator is %s%.

`graph.difference` creates the difference of two graphs. Only edges present in the first graph but not in the second will be be included in the new graph. The corresponding operator is %m%.

`graph.complementer` creates the complementer of a graph. Only edges which are *not* present in the original graph will be included in the new graph.

`graph.compose` creates the composition of two graphs. The new graph will contain an (a,b) edge only if there is a vertex c, such that edge (a,c) is included in the first graph and (c,b) is included in the second graph. The corresponding operator is %c%.

`graph.complementer` keeps graph and vertex attriubutes, edge attributes are lost. `graph.difference` keeps all attributes (graph, vertex and edge) of the first graph. The other functions do not handle vertex and edge attributes, the new graph will have no attributes at all.

Value

A new graph object.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

Examples

```
g1 <- graph.ring(10)
g2 <- graph.star(10, mode="undirected")
graph.union(g1, g2)
graph.disjoint.union(g1, g2)
graph.intersection(g1, g2)
graph.difference(g1, g2)
graph.complementer(g2)
graph.compose(g1, g2)

## graph complementer keeps graph and vertex attributes
g2$layout <- layout.circle
V(g2)$name <- letters[1:vcount(g2)]
cg2 <- graph.complementer(g2)
cg2$layout
V(cg2)$name

## graph difference keeps the attributes of the first graph
g1$name <- "Ring"
V(g1)$name <- LETTERS[1:vcount(g1)]
E(g1)$weight <- round(runif(ecount(g1))*10)
dg <- graph.difference(g1, g2)
print(dg, g=TRUE, v=TRUE, e=TRUE)
```

`graph-operators-by-name` *Graph operators based on symbolic vertex names*

Description

Normally igraph operators are based on internal igraph vertex ids. This function works with vertex names instead.

Usage

```
graph.intersection.by.name(g1, g2, keep.x.vertices = FALSE,
      keep.x.vertex.attributes = FALSE,
      keep.x.edge.attributes = FALSE)
graph.union.by.name(g1, g2)
graph.difference.by.name(g1, g2, keep.x.vertex.attributes = FALSE,
      keep.x.edge.attributes = FALSE)
```

Arguments

`g1,g2`	The input graphs, they should be both directed or both undirected.
`keep.x.vertices`	Logical scalar, whether to keep all the vertices of `g1`, even if they do not appear in `g2`.
`keep.x.vertex.attributes`	Logical scalar, whether to keep all vertex attributes of the `g1`, even if they do not appear in `g2`.

`keep.x.edge.attributes`
: Logical scalar, whether to keep edge attributes of `g1`, even if they do not appear in `g2`.

Details

These functions treat the graphs as sets of ordered (if they are directed), or unordered (if they are undirected) pairs of symbolic vertex names and perform set operations on them.

Value

A new graph object.

Author(s)

Magnus Torfason

See Also

`graph.union`, `link{graph.intersection}` and `link{graph.difference}` for the functions that are based on vertex ids.

Examples

```
# Print attributes as well
igraph.options(print.vertex.attributes=TRUE,
               print.edge.attributes=TRUE)

# Creating a wheel graph
g1 <- graph.ring(10)
V(g1)$name <- letters[1:vcount(g1)]
g2 <- graph.star(11, mode="undirected")
V(g2)$name <- c("z", letters[1:vcount(g1)])
g <- graph.union.by.name(g1, g2)
if (interactive()) {
  plot(g, layout=layout.kamada.kawai, vertex.label=V(g)$name)
}
g

# Some more examples
g1 <- graph.formula(a-b-c)
V(g1)$v.attr=c(1,2,3)
E(g1)$e.attr=c(5,7)
g2 <- graph.formula(b-c-d)

# Test the functions
graph.intersection.by.name(g1,g2) # Vertices are intersected as well
graph.union.by.name(g1,g2)        # Vertices are unioned as well
graph.difference.by.name(g1,g2)   # Vertices from x (g1) are used

# graph.intersection.by.name() has some extra parameters
graph.intersection.by.name(g1,g2,
    keep.x.vertices          = TRUE) # Keep all x vertices (only intersect edges)

graph.intersection.by.name(g1,g2,
    keep.x.vertices          = FALSE, # Keep all x vertices (only intersect edges)
    keep.x.vertex.attributes = TRUE, # Don't throw away V(g1) attributes
    keep.x.edge.attributes   = TRUE) # Don't throw away E(g1) attributes

# graph.difference.by.name() has some extra parameters
graph.difference.by.name(g1,g2,
    keep.x.vertex.attributes = TRUE, # Don't throw away V(g1) attributes
    keep.x.edge.attributes   = TRUE) # Don't throw away E(g1) attributes
```

`graph.adjacency` *Create graphs from adjacency matrices*

Description

`graph.adjacency` is a flexible function for creating `igraph` graphs from adjacency matrices.

Usage

```
graph.adjacency(adjmatrix, mode=c("directed", "undirected", "max",
         "min", "upper", "lower", "plus"), weighted=NULL, diag=TRUE,
         add.colnames=NULL, add.rownames=NA)
```

Arguments

`adjmatrix`	A square adjacency matrix. From igraph version 0.5.1 this can be a sparse matrix created with the `Matrix` package.
`mode`	Character scalar, specifies how igraph should interpret the supplied matrix. See also the `weighted` argument, the interpretation depends on that too. Possible values are: `directed`, `undirected`, `upper`, `lower`, `max`, `min`, `plus`. See details below.
`weighted`	This argument specifies whether to create a weighted graph from an adjacency matrix. If it is `NULL` then an unweighted graph is created and the elements of the adjacency matrix gives the number of edges between the vertices. If it is a character constant then for every non-zero matrix entry an edge is created and the value of the entry is added as an edge attribute named by the `weighted` argument. If it is `TRUE` then a weighted graph is created and the name of the edge attribute will be `weight`. See also details below.
`diag`	Logical scalar, whether to include the diagonal of the matrix in the calculation. If this is `FALSE` then the diagonal is zerod out first.
`add.colnames`	Character scalar, whether to add the column names as vertex attributes. If it is '`NULL`' (the default) then, if present, column names are added as vertex attribute 'name'. If '`NA`' then they will not be added. If a character constant, then it gives the name of the vertex attribute to add.
`add.rownames`	Character scalar, whether to add the row names as vertex attributes. Possible values the same as the previous argument. By default row names are not added. If '`add.rownames`' and '`add.colnames`' specify the same vertex attribute, then the former is ignored.

Details

graph.adjacency creates a graph from an adjacency matrix.

The order of the vertices are preserved, i.e. the vertex corresponding to the first row will be vertex 0 in the graph, etc.

graph.adjacency operates in two main modes, depending on the weighted argument.

If this argument is NULL then an unweighted graph is created and an element of the adjacency matrix gives the number of edges to create between the two corresponding vertices. The details depend on the value of the mode argument:

directed The graph will be directed and a matrix element gives the number of edges between two vertices.

undirected This is exactly the same as max, for convenience. Note that it is *not* checked whether the matrix is symmetric.

max An undirected graph will be created and max(A(i,j), A(j,i)) gives the number of edges.

upper An undirected graph will be created, only the upper right triangle (including the diagonal) is used for the number of edges.

lower An undirected graph will be created, only the lower left triangle (including the diagonal) is used for creating the edges.

min undirected graph will be created with min(A(i,j), A(j,i)) edges between vertex i and j.

plus undirected graph will be created with A(i,j)+A(j,i) edges between vertex i and j.

If the weighted argument is not NULL then the elements of the matrix give the weights of the edges (if they are not zero). The details depend on the value of the mode argument:

directed The graph will be directed and a matrix element gives the edge weights.

undirected First we check that the matrix is symmetric. It is an error if not. Then only the upper triangle is used to create a weighted undirected graph.

max An undirected graph will be created and max(A(i,j), A(j,i)) gives the edge weights.

upper An undirected graph will be created, only the upper right triangle (including the diagonal) is used (for the edge weights).

lower An undirected graph will be created, only the lower left triangle (including the diagonal) is used for creating the edges.

min An undirected graph will be created, min(A(i,j), A(j,i)) gives the edge weights.

plus An undirected graph will be created, A(i,j)+A(j,i) gives the edge weights.

Value

An igraph graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

graph and graph.formula for other ways to create graphs.

Examples

```
adjm <- matrix(sample(0:1, 100, replace=TRUE, prob=c(0.9,0.1)), nc=10)
g1 <- graph.adjacency( adjm )
adjm <- matrix(sample(0:5, 100, replace=TRUE,
                      prob=c(0.9,0.02,0.02,0.02,0.02,0.02)), nc=10)
g2 <- graph.adjacency(adjm, weighted=TRUE)
E(g2)$weight

## various modes for weighted graphs, with some tests
nzs <- function(x) sort(x [x!=0])
adjm <- matrix(runif(100), 10)
adjm[ adjm<0.5 ] <- 0
g3 <- graph.adjacency((adjm + t(adjm))/2, weighted=TRUE,
                      mode="undirected")

g4 <- graph.adjacency(adjm, weighted=TRUE, mode="max")
all(nzs(pmax(adjm, t(adjm))[upper.tri(adjm)]) == sort(E(g4)$weight))

g5 <- graph.adjacency(adjm, weighted=TRUE, mode="min")
all(nzs(pmin(adjm, t(adjm))[upper.tri(adjm)]) == sort(E(g5)$weight))

g6 <- graph.adjacency(adjm, weighted=TRUE, mode="upper")
all(nzs(adjm[upper.tri(adjm)]) == sort(E(g6)$weight))

g7 <- graph.adjacency(adjm, weighted=TRUE, mode="lower")
all(nzs(adjm[lower.tri(adjm)]) == sort(E(g7)$weight))

g8 <- graph.adjacency(adjm, weighted=TRUE, mode="plus")
d2 <- function(x) { diag(x) <- diag(x)/2; x }
all(nzs((d2(adjm+t(adjm)))[lower.tri(adjm)]) == sort(E(g8)$weight))

g9 <- graph.adjacency(adjm, weighted=TRUE, mode="plus", diag=FALSE)
d0 <- function(x) { diag(x) <- 0 }
all(nzs((d0(adjm+t(adjm)))[lower.tri(adjm)]) == sort(E(g9)$weight))

## row/column names
rownames(adjm) <- sample(letters, nrow(adjm))
colnames(adjm) <- seq(ncol(adjm))
g10 <- graph.adjacency(adjm, weighted=TRUE, add.rownames="code")
summary(g10)
```

`graph.automorphisms` *Number of automorphisms*

Description

Calculate the number of automorphisms of a graph, i.e. the number of isomorphisms to itself.

Usage

```
graph.automorphisms(graph, sh="fm")
```

Arguments

`graph`	The input graph, it is treated as undirected.
`sh`	The splitting heuristics for the BLISS algorithm. Possible values are: ‘`f`’: first non-singleton cell, ‘`fl`’: first largest non-singleton cell, ‘`fs`’: first smallest non-singleton cell, ‘`fm`’: first maximally non-trivially connected non-singleton cell, ‘`flm`’: first largest maximally non-trivially connected non-singleton cell, ‘`fsm`’: first smallest maximally non-trivially connected non-singleton cell.

Details

An automorphism of a graph is a permutation of its vertices which brings the graph into itself.

This function calculates the number of automorphism of a graph using the BLISS algorithm. See also the BLISS homepage at `http://www.tcs.hut.fi/Software/bliss/index.html`.

Value

A named list with the following members:

`group_size`	The size of the automorphism group of the input graph, as a string. This number is exact if igraph was compiled with the GMP library, and approximate otherwise.
`nof_nodes`	The number of nodes in the search tree.
`nof_leaf_nodes`	The number of leaf nodes in the search tree.
`nof_bad_nodes`	Number of bad nodes.
`nof_canupdates`	Number of canrep updates.
`max_level`	Maximum level.

Author(s)

Tommi Juntilla <`google@for.it`> for BLISS and Gabor Csardi <`csardi.gabor@gmail.com`> for the igraph glue code and this manual page.

References

Tommi Junttila and Petteri Kaski: Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics.* 2007.

See Also

`canonical.permutation`, `permute.vertices`

Examples

```
## A ring has n*2 automorphisms, you can "turn" it by 0-9 vertices
## and each of these graphs can be "flipped"
g <- graph.ring(10)
graph.automorphisms(g)
```

graph.bfs *Breadth-first search*

Description

Breadth-first search is an algorithm to traverse a graph. We start from a root vertex and spread along every edge “simultaneously”.

Usage

```
graph.bfs (graph, root, neimode = c("out", "in", "all", "total"),
    unreachable = TRUE, restricted = NULL, order = TRUE,
    rank = FALSE, father = FALSE, pred = FALSE, succ = FALSE,
    dist = FALSE, callback = NULL, extra = NULL,
    rho = parent.frame())
```

Arguments

`graph`	The input graph.
`root`	Numeric vector, usually of length one. The root vertex, or root vertices to start the search from.
`neimode`	For directed graphs specifies the type of edges to follow. ‘out’ follows outgoing, ‘in’ incoming edges. ‘all’ ignores edge directions completely. ‘total’ is a synonym for ‘all’. This argument is ignored for undirected graphs.
`unreachable`	Logical scalar, whether the search should visit the vertices that are unreachable from the given root vertex (or vertices). If `TRUE`, then additional searches are performed until all vertices are visited.
`restricted`	`NULL` (=no restriction), or a vector of vertices (ids or symbolic names). In the latter case, the search is restricted to the given vertices.
`order`	Logical scalar, whether to return the ordering of the vertices.
`rank`	Logical scalar, whether to return the rank of the vertices.
`father`	Logical scalar, whether to return the father of the vertices.
`pred`	Logical scalar, whether to return the predecessors of the vertices.
`succ`	Logical scalar, whether to return the successors of the vertices.
`dist`	Logical scalar, whether to return the distance from the root of the search tree.
`callback`	If not `NULL`, then it must be callback function. This is called whenever a vertex is visited. See details below.
`extra`	Additional argument to supply to the callback function.
`rho`	The environment in which the callback function is evaluated.

Details

The callback function must have the following arguments:

graph The input graph is passed to the callback function here.

data A named numeric vector, with the following entries: ‘vid’, the vertex that was just visited, ‘pred’, its predecessor, ‘succ’, its successor, ‘rank’, the rank of the current vertex, ‘dist’, its distance from the root of the search tree.

extra The extra argument.

See examples below on how to use the callback function.

Value

A named list with the following entries:

`root`	Numeric scalar. The root vertex that was used as the starting point of the search.
`neimode`	Character scalar. The `neimode` argument of the function call. Note that for undirected graphs this is always ‘all’, irrespectively of the supplied value.
`order`	Numeric vector. The vertex ids, in the order in which they were visited by the search.
`rank`	Numeric vector. The rank for each vertex.
`father`	Numeric vector. The father of each vertex, i.e. the vertex it was discovered from.
`pred`	Numeric vector. The previously visited vertex for each vertex, or 0 if there was no such vertex.
`succ`	Numeric vector. The next vertex that was visited after the current one, or 0 if there was no such vertex.
`dist`	Numeric vector, for each vertex its distance from the root of the search tree.

Note that `order`, `rank`, `father`, `pred`, `succ` and `dist` might be `NULL` if their corresponding argument is `FALSE`, i.e. if their calculation is not requested.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`graph.dfs` for depth-first search.

Examples

```
## Two rings
graph.bfs(graph.ring(10) %du% graph.ring(10), root=1, "out",
          order=TRUE, rank=TRUE, father=TRUE, pred=TRUE,
          succ=TRUE, dist=TRUE)

## How to use a callback
f <- function(graph, data, extra) {
  print(data)
  FALSE
}
tmp <- graph.bfs(graph.ring(10) %du% graph.ring(10), root=1, "out",
                 callback=f)

## How to use a callback to stop the search
## We stop after visiting all vertices in the initial component
f <- function(graph, data, extra) {
 data['succ'] == -1
}
graph.bfs(graph.ring(10) %du% graph.ring(10), root=1, callback=f)
```

graph.bipartite *Create a bipartite graph*

Description

A bipartite graph has two kinds of vertices and connections are only allowed between different kinds.

Usage

```
graph.bipartite(types, edges, directed=FALSE)
is.bipartite(graph)
```

Arguments

`types`	A vector giving the vertex types. It will be coerced into boolean. The length of the vector gives the number of vertices in the graph.
`edges`	A vector giving the edges of the graph, the same way as for the regular `graph` function. It is checked that the edges indeed connect vertices of different kind, accoding to the supplied `types` vector.
`directed`	Whether to create a directed graph, boolean constant. Note that by default undirected graphs are created, as this is more common for bipartite graphs.
`graph`	The input graph.

Details

Bipartite graphs have a `type` vertex attribute in igraph, this is boolean and `FALSE` for the vertices of the first kind and `TRUE` for vertices of the second kind.

`graph.bipartite` basically does three things. First it checks tha `edges` vector against the vertex `types`. Then it creates a graph using the `edges` vector and finally it adds the `types` vector as a vertex attribute called `type`.

`is.bipartite` checks whether the graph is bipartite or not. It just checks whether the graph has a vertex attribute called `type`.

Value

`graph.bipartite` returns a bipartite igraph graph. In other words, an igraph graph that has a vertex attribute named `type`.

`is.bipartite` returns a logical scalar.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`graph` to create one-mode networks

Examples

```
g <- graph.bipartite( rep(0:1,length=10), c(1:10))
print(g, v=TRUE)
```

---

graph.constructors *Various methods for creating graphs*

Description

These method can create various (mostly regular) graphs: empty graphs, graphs with the given edges, graphs from adjacency matrices, star graphs, lattices, rings, trees.

Usage

```
graph.lattice(dimvector = NULL, length = NULL, dim = NULL, nei = 1,
              directed = FALSE, mutual = FALSE, circular = FALSE, ...)
```

Usage

```
graph.empty(n=0, directed=TRUE)
graph(edges, n=max(edges)+1, directed=TRUE)
graph.star(n, mode = c("in", "out", "mutual", "undirected"), center = 0)
graph.lattice(dimvector, nei = 1, directed = FALSE, mutual = FALSE,
              circular = FALSE)
graph.lattice(length, dim, nei = 1, directed = FALSE, mutual = FALSE,
              circular = FALSE)
graph.ring(n, directed = FALSE, mutual = FALSE, circular=TRUE)
graph.tree(n, children = 2, mode="out")
graph.full(n, directed = FALSE, loops = FALSE)
graph.full.citation(n, directed = TRUE)
graph.atlas(n)
graph.edgelist(el, directed=TRUE)
graph.extended.chordal.ring(n, w)
```

Arguments

`edges`	Numeric vector defining the edges, the first edge points from the first element to the second, the second edge from the third to the fourth, etc.
`directed`	Logical, if TRUE a directed graph will be created. Note that for while most constructors the default is TRUE, for `graph.lattice` and `graph.ring` it is FALSE. For `graph.star` the `mode` argument should be used for creating an undirected graph.
`n`	The number of vertices in the graph for most functions. For `graph` this parameter is ignored if there is a bigger vertex id in `edges`. This means that for this function it is safe to supply zero here if the vertex with the largest id is not an isolate. For `graph.atlas` this is the number (id) of the graph to create.

`mode`	For `graph.star` it defines the direction of the edges, `in`: the edges point *to* the center, `out`: the edges point *from* the center, `mutual`: a directed star is created with mutual edges, `undirected`: the edges are undirected. For `igraph.tree` this parameter defines the direction of the edges. `out` indicates that the edges point from the parent to the children, `in` indicates that they point from the children to their parents, while `undirected` creates an undirected graph.
`center`	For `graph.star` the center vertex of the graph, by default the first vertex.
`dimvector`	A vector giving the size of the lattice in each dimension, for `graph.lattice`.
`nei`	The distance within which (inclusive) the neighbors on the lattice will be connected. This parameter is not used right now.
`mutual`	Logical, if TRUE directed lattices will be mutually connected.
`circular`	Logical, if TRUE the lattice or ring will be circular.
`length`	Integer constant, for regular lattices, the size of the lattice in each dimension.
`dim`	Integer constant, the dimension of the lattice.
`children`	Integer constant, the number of children of a vertex (except for leafs) for `graph.tree`.
`loops`	If TRUE also loops edges (self edges) are added.
`graph`	An object.
`el`	An edge list, a two column matrix, character or numeric. See details below.
`w`	A matrix which specifies the extended chordal ring. See details below.

Details

All these functions create graphs in a deterministic way.

`graph.empty` is the simplest one, this creates an empty graph.

`graph` creates a graph with the given edges.

`graph.star` creates a star graph, in this every single vertex is connected to the center vertex and nobody else.

`graph.lattice` is a flexible function, it can create lattices of arbitrary dimensions, periodic or unperiodic ones.

`graph.ring` is actually a special case of `graph.lattice`, it creates a one dimensional circular lattice.

`graph.tree` creates regular trees.

`graph.full` simply creates full graphs.

`graph.full.citation` creates a full citation graph. This is a directed graph, where every i->j edge is present if and only if j<i. If `directed=FALSE` then the graph is just a full graph.

`graph.atlas` creates graphs from the book An Atlas of Graphs by Roland C. Read and Robin J. Wilson. The atlas contains all undirected graphs with up to seven vertices, numbered from 0 up to 1252. The graphs are listed:

1. in increasing order of number of nodes;
2. for a fixed number of nodes, in increasing order of the number of edges;
3. for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example 111223 < 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

graph.edgelist creates a graph from an edge list. Its argument is a two-column matrix, each row defines one edge. If it is a numeric matrix then its elements are interpreted as vertex ids. If it is a character matrix then it is interpreted as symbolic vertex names and a vertex id will be assigned to each name, and also a name vertex attribute will be added.

graph.extended.chordal.ring creates an extended chordal ring. An extended chordal ring is regular graph, each node has the same degree. It can be obtained from a simple ring by adding some extra edges specified by a matrix. Let p denote the number of columns in the 'W' matrix. The extra edges of vertex i are added according to column i mod p in 'W'. The number of extra edges is the number of rows in 'W': for each row j an edge i->i+w[ij] is added if i+w[ij] is less than the number of total nodes. See also Kotsis, G: Interconnection Topologies for Parallel Processing Systems, PARS Mitteilungen 11, 1-6, 1993.

Value

Every function documented here returns a graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

graph.adjacency to create graphs from adjacency matrices, graph.formula for a handy way to create small graphs, graph.data.frame for an easy way to create graphs with many edge/vertex attributes.

Examples

```
g1 <- graph.empty()
g2 <- graph( c(1,2,2,3,3,4,5,6), directed=FALSE )
g5 <- graph.star(10, mode="out")
g6 <- graph.lattice(c(5,5,5))
g7 <- graph.lattice(length=5, dim=3)
g8 <- graph.ring(10)
g9 <- graph.tree(10, 2)
g10 <- graph.full(5, loops=TRUE)
g11 <- graph.full.citation(10)
g12 <- graph.atlas(sample(0:1252, 1))
el <- matrix( c("foo", "bar", "bar", "foobar"), nc=2, byrow=TRUE)
g13 <- graph.edgelist(el)
g15 <- graph.extended.chordal.ring(15, matrix(c(3,12,4,7,8,11), nr=2))
```

---

graph.coreness *K-core decomposition of graphs*

Description

The k-core of graph is a maximal subgraph in which each vertex has at least degree k. The coreness of a vertex is k if it belongs to the k-core but not to the (k+1)-core.

Usage

```
graph.coreness(graph, mode=c("all", "out", "in"))
```

Arguments

`graph`	The input graph, it can be directed or undirected
`mode`	The type of the core in directed graphs. Character constant, possible values: `in`: in-cores are computed, `out`: out-cores are computed, `all`: the corresponding undirected graph is considered. This argument is ignored for undirected graphs.

Details

The k-core of a graph is the maximal subgraph in which every vertex has at least degree k. The cores of a graph form layers: the (k+1)-core is always a subgraph of the k-core.

This function calculates the coreness for each vertex.

Value

Numeric vector of integer numbers giving the coreness of each vertex.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

References

Vladimir Batagelj, Matjaz Zaversnik: An O(m) Algorithm for Cores Decomposition of Networks, 2002

Seidman S. B. (1983) Network structure and minimum degree, *Social Networks*, 5, 269–287.

See Also

`degree`

Examples

```
g <- graph.ring(10)
g <- add.edges(g, c(1,2, 2,3, 1,3))
graph.coreness(g)
```

`graph.data.frame` *Creating igraph graphs from data frames*

Description

This function creates an igraph graph from one or two data frames containing the (symbolic) edge list and edge/vertex attributes.

Usage

```
graph.data.frame(d, directed=TRUE, vertices=NULL)
```

Arguments

`d`	A data frame containing a symbolic edge list in the first two columns. Additional columns are considered as edge attributes.
`directed`	Logical scalar, whether or not to create a directed graph.
`vertices`	A data frame with vertex metadata, or `NULL`. See details below.

Details

`graph.data.frame` creates igraph graphs from one or two data frames. It has two modes of operatation, depending whether the `vertices` argument is `NULL` or not.

If `vertices` is `NULL`, then the first two columns of `d` are used as a symbolic edge list and additional columns as edge attributes. The names of the attributes are taken from the names of the columns.

If `vertices` is not `NULL`, then it must be a data frame giving vertex metadata. The first column of `vertices` is assumed to contain symbolic vertex names, this will be added to the graphs as the '`name`' vertex attribute. Other columns will be added as additional vertex attributes. If `vertices` is not `NULL` then the symbolic edge list given in `d` is checked to contain only vertex names listed in `vertices`.

Typically, the data frames are exported from some speadsheat software like Excel and are imported into R via `read.table`, `read.delim` or `read.csv`.

Value

An igraph graph object.

Note

`NA` elements in the first two columns 'd' are replaced by the string "NA" before creating the graph. This means that all `NA`s will correspond to a single vertex.

`NA` elements in the first column of 'vertices' are also replaced by the string "NA", but the rest of 'vertices' is not touched. In other words, vertex names (=the first column) cannot be `NA`, but other vertex attributes can.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`graph.constructors` and `graph.formula` for other ways to create graphs, `read.table` to read in tables from files.

Examples

```
## A simple example with a couple of actors
## The typical case is that these tables are read in from files....
actors <- data.frame(name=c("Alice", "Bob", "Cecil", "David",
                            "Esmeralda"),
                     age=c(48,33,45,34,21),
                     gender=c("F","M","F","M","F"))
relations <- data.frame(from=c("Bob", "Cecil", "Cecil", "David",
                               "David", "Esmeralda"),
                        to=c("Alice", "Bob", "Alice", "Alice", "Bob", "Alice"),
                        same.dept=c(FALSE,FALSE,TRUE,FALSE,FALSE,TRUE),
                        friendship=c(4,5,5,2,1,1), advice=c(4,5,5,4,2,3))
g <- graph.data.frame(relations, directed=TRUE, vertices=actors)
print(g, e=TRUE, v=TRUE)
```

---

graph.de.bruijn *De Bruijn graphs.*

Description

De Bruijn graphs are labeled graphs representing the overlap of strings.

Usage

```
graph.de.bruijn(m,n)
```

Arguments

m	Integer scalar, the size of the alphabet. See details below.
n	Integer scalar, the length of the labels. See details below.

Details

A de Bruijn graph represents relationships between strings. An alphabet of `m` letters are used and strings of length `n` are considered. A vertex corresponds to every possible string and there is a directed edge from vertex `v` to vertex `w` if the string of `v` can be transformed into the string of `w` by removing its first letter and appending a letter to it.

Please note that the graph will have `m` to the power `n` vertices and even more edges, so probably you don't want to supply too big numbers for `m` and `n`.

De Bruijn graphs have some interesting properties, please see another source, eg. Wikipedia for details.

Value

A graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`graph.kautz`, `line.graph`

Examples

```
# de Bruijn graphs can be created recursively by line graphs as well
g <- graph.de.bruijn(2,1)
graph.de.bruijn(2,2)
line.graph(g)
```

graph.density *Graph density*

Description

The density of a graph is the ratio of the number of edges and the number of possible edges.

Usage

```
graph.density(graph, loops=FALSE)
```

Arguments

`graph`	The input graph.
`loops`	Logical constant, whether to allow loop edges in the graph. If this is TRUE then self loops are considered to be possible. If this is FALSE then we assume that the graph does not contain any loop edges and that loop edges are not meaningful.

Details

Note that this function may return strange results for graph with multiple edges, density is ill-defined for graphs with multiple edges.

Value

A real constant. This function returns `NaN` (=0.0/0.0) for an empty graph with zero vertices.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Wasserman, S., and Faust, K. (1994). Social Network Analysis: Methods and Applications. Cambridge: Cambridge University Press.

See Also

`vcount`, `ecount`, `simplify` to get rid of the multiple and/or loop edges.

Examples

```
g1 <- graph.empty(n=10)
g2 <- graph.full(n=10)
g3 <- erdos.renyi.game(n=10, 0.4)

# loop edges
g <- graph( c(1,2, 2,2, 2,3) )
graph.density(g, loops=FALSE)              # this is wrong!!!
graph.density(g, loops=TRUE)               # this is right!!!
graph.density(simplify(g), loops=FALSE)    # this is also right, but different
```

graph.dfs *Depth-first search*

Description

Depth-first search is an algorithm to traverse a graph. It starts from a root vertex and tries to go quickly as far from as possible.

Usage

```
graph.dfs (graph, root, neimode = c("out", "in", "all", "total"),
    unreachable = TRUE, order = TRUE, order.out = FALSE,
    father = FALSE, dist = FALSE, in.callback = NULL,
    out.callback = NULL, extra = NULL, rho = parent.frame())
```

Arguments

`graph`	The input graph.
`root`	The single root vertex to start the search from.
`neimode`	For directed graphs specifies the type of edges to follow. 'out' follows outgoing, 'in' incoming edges. 'all' ignores edge directions completely. 'total' is a synonym for 'all'. This argument is ignored for undirected graphs.
`unreachable`	Logical scalar, whether the search should visit the vertices that are unreachable from the given root vertex (or vertices). If `TRUE`, then additional searches are performed until all vertices are visited.
`order`	Logical scalar, whether to return the DFS ordering of the vertices.
`order.out`	Logical scalar, whether to return the ordering based on leaving the subtree of the vertex.
`father`	Logical scalar, whether to return the father of the vertices.
`dist`	Logical scalar, whether to return the distance from the root of the search tree.
`in.callback`	If not `NULL`, then it must be callback function. This is called whenever a vertex is visited. See details below.
`out.callback`	If not `NULL`, then it must be callback function. This is called whenever the subtree of a vertex is completed by the algorithm. See details below.
`extra`	Additional argument to supply to the callback function.
`rho`	The environment in which the callback function is evaluated.

Details

The callback functions must have the following arguments:

graph The input graph is passed to the callback function here.

data A named numeric vector, with the following entries: 'vid', the vertex that was just visited and 'dist', its distance from the root of the search tree.

extra The extra argument.

See examples below on how to use the callback functions.

Value

A named list with the following entries:

`root`	Numeric scalar. The root vertex that was used as the starting point of the search.
`neimode`	Character scalar. The `neimode` argument of the function call. Note that for undirected graphs this is always 'all', irrespectively of the supplied value.
`order`	Numeric vector. The vertex ids, in the order in which they were visited by the search.
`order.out`	Numeric vector, the vertex ids, in the order of the completion of their subtree.
`father`	Numeric vector. The father of each vertex, i.e. the vertex it was discovered from.
`dist`	Numeric vector, for each vertex its distance from the root of the search tree.

Note that `order`, `order.out`, `father`, and `dist` might be `NULL` if their corresponding argument is `FALSE`, i.e. if their calculation is not requested.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`graph.bfs` for breadth-first search.

Examples

```
## A graph with two separate trees
graph.dfs(graph.tree(10) %du% graph.tree(10), root=1, "out",
          TRUE, TRUE, TRUE, TRUE)

## How to use a callback
f.in <- function(graph, data, extra) {
  cat("in:", paste(collapse=", ", data), "\n")
  FALSE
}
f.out <- function(graph, data, extra) {
  cat("out:", paste(collapse=", ", data), "\n")
  FALSE
}
tmp <- graph.dfs(graph.tree(10), root=1, "out",
                 in.callback=f.in, out.callback=f.out)

## Terminate after the first component, using a callback
f.out <- function(graph, data, extra) {
 data['vid'] == 1
}
tmp <- graph.dfs(graph.tree(10) %du% graph.tree(10), root=1,
                 out.callback=f.out)
```

`graph.diversity` *Graph diversity*

Description

Calculates a measure of diversity for all vertices.

Usage

```
graph.diversity(graph, weights = NULL, vids = V(graph))
```

Arguments

`graph`	The input graph. Edge directions are ignored.
`weights`	`NULL`, or the vector of edge weights to use for the computation. If `NULL`, then the 'weight' attibute is used. Note that this measure is not defined for unweighted graphs.
`vids`	The vertex ids for which to calculate the measure.

Details

The diversity of a vertex is defined as the (scaled) Shannon entropy of the weights of its incident edges:

$$D(i) = \frac{H(i)}{\log k_i}$$

and

$$H(i) = -\sum_{j=1}^{k_i} p_{ij} \log p_{ij},$$

where

$$p_{ij} = \frac{w_{ij}}{\sum_{l=1}^{k_i}} V_{il},$$

and k_i is the (total) degree of vertex i, w_{ij} is the weight of the edge(s) between vertices i and j.

For vertices with degree less than two the function returns `NaN`.

Value

A numeric vector, its length is the number of vertices.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Nathan Eagle, Michael Macy and Rob Claxton: Network Diversity and Economic Development, *Science* **328**, 1029–1031, 2010.

Examples

```
g1 <- erdos.renyi.game(20, 2/20)
g2 <- erdos.renyi.game(20, 2/20)
g3 <- erdos.renyi.game(20, 5/20)
E(g1)$weight <- 1
E(g2)$weight <- runif(ecount(g2))
E(g3)$weight <- runif(ecount(g3))
graph.diversity(g1)
graph.diversity(g2)
graph.diversity(g3)
```

---

graph.famous *Creating named graphs*

Description

There are some famous, named graphs, sometimes counterexamples to some conjecture or unique graphs with given features. These can be created with this function

Usage

```
graph.famous(name)
```

Arguments

`name`	Character constant giving the name of the graph. It is case insensitive.

Details

`graph.famous` knows the following graphs:

Bull The bull graph, 5 vertices, 5 edges, resembles to the head of a bull if drawn properly.

Chvatal This is the smallest triangle-free graph that is both 4-chromatic and 4-regular. According to the Grunbaum conjecture there exists an m-regular, m-chromatic graph with n vertices for every m>1 and n>2. The Chvatal graph is an example for m=4 and n=12. It has 24 edges.

Coxeter A non-Hamiltonian cubic symmetric graph with 28 vertices and 42 edges.

Cubical The Platonic graph of the cube. A convex regular polyhedron with 8 vertices and 12 edges.

Diamond A graph with 4 vertices and 5 edges, resembles to a schematic diamond if drawn properly.

Dodecahedral, Dodecahedron Another Platonic solid with 20 vertices and 30 edges.

Folkman The semisymmetric graph with minimum number of vertices, 20 and 40 edges. A semisymmetric graph is regular, edge transitive and not vertex transitive.

Franklin This is a graph whose embedding to the Klein bottle can be colored with six colors, it is a counterexample to the neccessity of the Heawood conjecture on a Klein bottle. It has 12 vertices and 18 edges.

Frucht The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element. It has 12 vertices and 18 edges.

Grotzsch The Grötzsch graph is a triangle-free graph with 11 vertices, 20 edges, and chromatic number 4. It is named after German mathematician Herbert Grötzsch, and its existence demonstrates that the assumption of planarity is necessary in Grötzsch's theorem that every triangle-free planar graph is 3-colorable.

Heawood The Heawood graph is an undirected graph with 14 vertices and 21 edges. The graph is cubic, and all cycles in the graph have six or more edges. Every smaller cubic graph has shorter cycles, so this graph is the 6-cage, the smallest cubic graph of girth 6.

Herschel The Herschel graph is the smallest nonhamiltonian polyhedral graph. It is the unique such graph on 11 nodes, and has 18 edges.

House The house graph is a 5-vertex, 6-edge graph, the schematic draw of a house if drawn properly, basicly a triangle of the top of a square.

HouseX The same as the house graph with an X in the square. 5 vertices and 8 edges.

Icosahedral, Icosahedron A Platonic solid with 12 vertices and 30 edges.

Krackhardt_Kite A social network with 10 vertices and 18 edges. Krackhardt, D. Assessing the Political Landscape: Structure, Cognition, and Power in Organizations. Admin. Sci. Quart. 35, 342-369, 1990.

Levi The graph is a 4-arc transitive cubic graph, it has 30 vertices and 45 edges.

McGee The McGee graph is the unique 3-regular 7-cage graph, it has 24 vertices and 36 edges.

Meredith The Meredith graph is a quartic graph on 70 nodes and 140 edges that is a counterexample to the conjecture that every 4-regular 4-connected graph is Hamiltonian.

Noperfectmatching A connected graph with 16 vertices and 27 edges containing no perfect matching. A matching in a graph is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex. A perfect matching is a matching which covers all vertices of the graph.

Nonline A graph whose connected components are the 9 graphs whose presence as a vertex-induced subgraph in a graph makes a nonline graph. It has 50 vertices and 72 edges.

Octahedral, Octahedron Platonic solid with 6 vertices and 12 edges.

Petersen A 3-regular graph with 10 vertices and 15 edges. It is the smallest hypohamiltonian graph, ie. it is non-hamiltonian but removing any single vertex from it makes it Hamiltonian.

Robertson The unique (4,5)-cage graph, ie. a 4-regular graph of girth 5. It has 19 vertices and 38 edges.

Smallestcyclicgroup A smallest nontrivial graph whose automorphism group is cyclic. It has 9 vertices and 15 edges.

Tetrahedral, Tetrahedron Platonic solid with 4 vertices and 6 edges.

Thomassen The smallest hypotraceable graph, on 34 vertices and 52 edges. A hypotracable graph does not contain a Hamiltonian path but after removing any single vertex from it the remainder always contains a Hamiltonian path. A graph containing a Hamiltonian path is called tracable.

Tutte Tait's Hamiltonian graph conjecture states that every 3-connected 3-regular planar graph is Hamiltonian. This graph is a counterexample. It has 46 vertices and 69 edges.

Uniquely3colorable Returns a 12-vertex, triangle-free graph with chromatic number 3 that is uniquely 3-colorable.

Walther An identity graph with 25 vertices and 31 edges. An identity graph has a single graph automorphism, the trivial one.

Zachary Social network of friendships between 34 members of a karate club at a US university in the 1970s. See W. W. Zachary, An information flow model for conflict and fission in small groups, Journal of Anthropological Research 33, 452-473 (1977).

Value

A graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

graph can create arbitrary graphs, see also the other functions on the its manual page for creating special graphs.

Examples

```
solids <- list(graph.famous("Tetrahedron"),
               graph.famous("Cubical"),
               graph.famous("Octahedron"),
               graph.famous("Dodecahedron"),
               graph.famous("Icosahedron"))
```

---

graph.formula *Creating (small) graphs via a simple interface*

Description

This function is useful if you want to create a small (named) graph quickly, it works for both directed and undirected graphs.

Usage

```
graph.formula(..., simplify = TRUE)
```

Arguments

...	The formulae giving the structure of the graph, see details below.
simplify	Logical scalar, whether to call simplify on the created graph. By default the graph is simplified, loop and multiple edges are removed.

Details

graph.formula is very handy for creating small graphs quickly. You need to supply one or more R expressions giving the structure of the graph. The expressions consist of vertex names and edge operators. An edge operator is a sequence of '-' and '+' characters, the former is for the edges and the latter is used for arrow heads. The edges can be arbitrarily long, ie. you may use as many '-' characters to “draw” them as you like.

If all edge operators consist of only '-' characters then the graph will be undirected, whereas a single '+' character implies a directed graph.

Let us see some simple examples. Without arguments the function creates an empty graph:

```
graph.formula()
```

A simple undirected graph with two vertices called ‘A’ and ‘B’ and one edge only:

```
graph.formula(A-B)
```

Remember that the length of the edges does not matter, so we could have written the following, this creates the same graph:

```
graph.formula( A-----B )
```

If you have many disconnected components in the graph, separate them with commas. You can also give isolate vertices.

```
graph.formula( A--B, C--D, E--F, G--H, I, J, K )
```

The ‘:’ operator can be used to define vertex sets. If an edge operator connects two vertex sets then every edge from the first set will be connected to every edge in the second set. The following form creates a full graph, including loop edges:

```
graph.formula( A:B:C:D -- A:B:C:D )
```

In directed graphs, edges will be created only if the edge operator includes a arrow head (‘+’) *at the end* of the edge:

```
graph.formula( A -+ B -+ C )
graph.formula( A +- B -+ C )
graph.formula( A +- B -- C )
```

Thus in the third example no edge is created between vertices `B` and `C`.

Mutual edges can be also created with a simple edge operator:

```
graph.formula( A +-+ B +---+ C ++ D + E)
```

Note again that the length of the edge operators is arbitrary, ‘+’, ‘++’ and ‘+-----+’ have exactly the same meaning.

If the vertex names include spaces or other special characters then you need to quote them:

```
graph.formula( "this is" +- "a silly" -+ "graph here" )
```

You can include any character in the vertex names this way, even ‘+’ and ‘-’ characters.

See more examples below.

Value

A new graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

graph for more general graph creation methods.

Examples

```
# A simple undirected graph
g <- graph.formula( Alice-Bob-Cecil-Alice, Daniel-Cecil-Eugene, Cecil-Gordon )
g

# Another undirected graph, ":" notation
g2 <- graph.formula( Alice-Bob:Cecil:Daniel, Cecil:Daniel-Eugene:Gordon )
g2

# A directed graph
g3 <- graph.formula( Alice +-+ Bob --+ Cecil +-- Daniel,
                     Eugene --+ Gordon:Helen )
g3

# A graph with isolate vertices
g4 <- graph.formula( Alice -- Bob -- Daniel, Cecil:Gordon, Helen )
g4
V(g4)$name

# "Arrows" can be arbitrarily long
g5 <- graph.formula( Alice +---------+ Bob )
g5

# Special vertex names
g6 <- graph.formula( "+" -- "-", "*" -- "/", "%%" -- "%/%" )
g6
```

---

graph.full.bipartite *Create a full bipartite graph*

Description

Bipartite graphs are also called two-mode by some. This function creates a bipartite graph in which every possible edge is present.

Usage

```
graph.full.bipartite (n1, n2, directed = FALSE, mode = c("all", "out", "in"))
```

Arguments

n1	The number of vertices of the first kind.
n2	The number of vertices of the second kind.
directed	Logical scalar, whether the graphs is directed.

`mode`	Scalar giving the kind of edges to create for directed graphs. If this is ‘`out`’ then all vertices of the first kind are connected to the others; ‘`in`’ specifies the opposite direction; ‘`all`’ creates mutual edges. This argument is ignored for undirected graphs.x

Details

Bipartite graphs have a ‘`type`’ vertex attribute in igraph, this is boolean and `FALSE` for the vertices of the first kind and `TRUE` for vertices of the second kind.

Value

An igraph graph, with the ‘`type`’ vertex attribute set.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`graph.full` for creating one-mode full graphs

Examples

```
g <- graph.full.bipartite(2, 3)
g2 <- graph.full.bipartite(2, 3, dir=TRUE)
g3 <- graph.full.bipartite(2, 3, dir=TRUE, mode="in")
g4 <- graph.full.bipartite(2, 3, dir=TRUE, mode="all")
```

`graph.graphdb` *Load a graph from the graph database for testing graph isomorphism.*

Description

This function downloads a graph from a database created for the evaluation of graph isomorphism testing algothitms.

Usage

```
graph.graphdb (url = NULL, prefix = "iso", type = "r001", nodes = NULL,
    pair = "A", which = 0, base = "http://cneurocvs.rmki.kfki.hu/graphdb/gzip",
    compressed = TRUE, directed = TRUE)
```

Arguments

`url`	If not `NULL` it is a complete URL with the file to import.
`prefix`	Gives the prefix. See details below. Possible values: `iso`, `i2`, `si4`, `si6`, `mcs10`, `mcs30`, `mcs50`, `mcs70`, `mcs90`.
`type`	Gives the graph type identifier. See details below. Possible values: `r001`, `r005`, `r01`, `r02`, `m2D`, `m2Dr2`, `m2Dr4`, `m2Dr6` `m3D`, `m3Dr2`, `m3Dr4`, `m3Dr6`, `m4D`, `m4Dr2`, `m4Dr4`, `m4Dr6`, `b03`, `b03m`, `b06`, `b06m`, `b09`, `b09m`.
`nodes`	The number of vertices in the graph.

pair	Specifies which graph of the pair to read. Possible values: A and B.
which	Gives the number of the graph to read. For every graph type there are a number of actual graphs in the database. This argument specifies which one to read.
base	The base address of the database. See details below.
compressed	Logical constant, if TRUE than the file is expected to be compressed by gzip. If url is NULL then a '.gz' suffix is added to the filename.
directed	Logical constant, whether to create a directed graph.

Details

graph.graphdb reads a graph from the graph database from an FTP or HTTP server or from a local copy. It has two modes of operation:

If the url argument is specified then it should the complete path to a local or remote graph database file. In this case we simply call read.graph with the proper arguments to read the file.

If url is NULL, and this is the default, then the filename is assembled from the base, prefix, type, nodes, pair and which arguments.

See the documentation for the graph database at http://amalfi.dis.unina.it/graph/db/doc/graphdbat.html for the actual format of a graph database file and other information.

Value

A new graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

M. De Santo, P. Foggia, C. Sansone, M. Vento: A large database of graphs and its use for benchmarking graph isomorphism algorithms, *Pattern Recognition Letters*, Volume 24, Issue 8 (May 2003)

See Also

read.graph, graph.isomorphic.vf2

Examples

```
## Not run:
g <- graph.graphdb(prefix="iso", type="r001", nodes=20, pair="A",
  which=10, compressed=TRUE)
g2 <- graph.graphdb(prefix="iso", type="r001", nodes=20, pair="B",
  which=10, compressed=TRUE)
graph.isomorphic.vf2(g, g2)
g3 <- graph.graphdb(url="http://cneurocvs.rmki.kfki.hu/graphdb/gzip/iso/bvg/b06m/iso_b06m_m200.A09.gz")

## End(Not run)
```

graph.incidence *Create graphs from an incidence matrix*

Description

graph.incidence creates a bipartite igraph graph from an incidence matrix.

Usage

```
graph.incidence(incidence, directed = FALSE, mode = c("all", "out",
    "in", "total"), multiple = FALSE, weighted = NULL, add.names = NULL)
```

Arguments

incidence	The input incidence matrix. It can also be a sparse matrix from the Matrix package.
directed	Logical scalar, whether to create a directed graph.
mode	A character constant, defines the direction of the edges in directed graphs, ignored for undirected graphs. If 'out', then edges go from vertices of the first kind (corresponding to rows in the incidence matrix) to vertices of the second kind (columns in the incidence matrix). If 'in', then the opposite direction is used. If 'all' or 'total', then mutual edges are created.
multiple	Logical scalar, specifies how to interpret the matrix elements. See details below.
weighted	This argument specifies whether to create a weighted graph from the incidence matrix. If it is NULL then an unweighted graph is created and the multiple argument is used to determine the edges of the graph. If it is a character constant then for every non-zero matrix entry an edge is created and the value of the entry is added as an edge attribute named by the weighted argument. If it is TRUE then a weighted graph is created and the name of the edge attribute will be 'weight'.
add.names	A character constant, NA or NULL. graph.incidence can add the row and column names of the incidence matrix as vertex attributes. If this argument is NULL (the default) and the incidence matrix has both row and column names, then these are added as the 'name' vertex attribute. If you want a different vertex attribute for this, then give the name of the attributes as a character string. If this argument is NA, then no vertex attributes (other than type) will be added.

Details

Bipartite graphs have a 'type' vertex attribute in igraph, this is boolean and FALSE for the vertices of the first kind and TRUE for vertices of the second kind.

graph.incidence can operate in two modes, depending on the multiple argument. If it is FALSE then a single edge is created for every non-zero element in the incidence matrix. If multiple is TRUE, then the matrix elements are rounded up to the closest non-negative integer to get the number of edges to create between a pair of vertices.

Value

A bipartite igraph graph. In other words, an igraph graph that has a vertex attribute type.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

graph.bipartite for another way to create bipartite graphs

Examples

```
inc <- matrix(sample(0:1, 15, repl=TRUE), 3, 5)
colnames(inc) <- letters[1:5]
rownames(inc) <- LETTERS[1:3]
graph.incidence(inc)
```

---

graph.kautz *Kautz graphs*

Description

Kautz graphs are labeled graphs representing the overlap of strings.

Usage

```
graph.kautz(m,n)
```

Arguments

m	Integer scalar, the size of the alphabet. See details below.
n	Integer scalar, the length of the labels. See details below.

Details

A Kautz graph is a labeled graph, vertices are labeled by strings of length n+1 above an alphabet with m+1 letters, with the restriction that every two consecutive letters in the string must be different. There is a directed edge from a vertex v to another vertex w if it is possible to transform the string of v into the string of w by removing the first letter and appending a letter to it.

Kautz graphs have some interesting properties, see eg. Wikipedia for details.

Value

A graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>, the first version in R was written by Vincent Matossian.

See Also

graph.de.bruijn, line.graph

Examples

```
line.graph(graph.kautz(2,1))
graph.kautz(2,2)
```

`graph.knn` *Average nearest neighbor degree*

Description

Calculate the average nearest neighbor degree of the given vertices and the same quantity in the function of vertex degree

Usage

```
graph.knn(graph, vids=V(graph), weights=NULL)
```

Arguments

`graph`	The input graph. It can be directed, but it will be treated as undirected, i.e. the direction of the edges is ignored.
`vids`	The vertices for which the calculation is performed. Normally it includes all vertices. Note, that if not all vertices are given here, then both '`knn`' and '`knnk`' will be calculated based on the given vertices only.
`weights`	Weight vector. If the graph has a `weight` edge attribute, then this is used by default. If this argument is given, then vertex strength (see `graph.strength`) is used instead of vertex degree. But note that `knnk` is still given in the function of the normal vertex degree.

Details

Note that for zero degree vertices the answer in '`knn`' is `NaN` (zero divided by zero), the same is true for '`knnk`' if a given degree never appears in the network.

Value

A list with two members:

`knn`	A numeric vector giving the average nearest neighbor degree for all vertices in `vids`.
`knnk`	A numeric vector, its length is the maximum (total) vertex degree in the graph. The first element is the average nearest neighbor degree of vertices with degree one, etc.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, Alessandro Vespignani: The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004)

Examples

```
# Some trivial ones
g <- graph.ring(10)
graph.knn(g)
g2 <- graph.star(10)
graph.knn(g2)

# A scale-free one, try to plot 'knnk'
g3 <- ba.game(1000, m=5)
graph.knn(g3)

# A random graph
g4 <- random.graph.game(1000, p=5/1000)
graph.knn(g4)

# A weighted graph
g5 <- graph.star(10)
E(g5)$weight <- seq(ecount(g5))
graph.knn(g5)
```

`graph.laplacian` *Graph Laplacian*

Description

The Laplacian of a graph.

Usage

```
graph.laplacian(graph, normalized=FALSE, weights=NULL,
   sparse=getIgraphOpt("sparsematrices"))
```

Arguments

`graph`	The input graph.
`normalized`	Whether to calculate the normalized Laplacian. See definitions below.
`weights`	An optional vector giving edge weights for weighted Laplacian matrix. If this is `NULL` and the graph has an edge attribute called `weight`, then it will be used automatically. Set this to `NA` if you want the unweighted Laplacian on a graph that has a `weight` edge attribute.
`sparse`	Logical scalar, whether to return the result as a sparse matrix. The `Matrix` package is required for sparse matrices.

Details

The Laplacian Matrix of a graph is a symmetric matrix having the same number of rows and columns as the number of vertices in the graph and element (i,j) is d[i], the degree of vertex i if if i==j, -1 if i!=j and there is an edge between vertices i and j and 0 otherwise.

A normalized version of the Laplacian Matrix is similar: element (i,j) is 1 if i==j, -1/sqrt(d[i] d[j]) if i!=j and there is an edge between vertices i and j and 0 otherwise.

The weighted version of the Laplacian simply works with the weighted degree instead of the plain degree. I.e. (i,j) is d[i], the weighted degree of vertex i if if i==j, -w if i!=j and there is an edge between vertices i and j with weight w, and 0 otherwise. The weighted degree of a vertex is the sum of the weights of its adjacent edges.

Value

A numeric matrix.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

Examples

```
g <- graph.ring(10)
graph.laplacian(g)
graph.laplacian(g, norm=TRUE)
graph.laplacian(g, norm=TRUE, sparse=FALSE)
```

`graph.lcf` *Creating a graph from LCF notation*

Description

LCF is short for Lederberg-Coxeter-Frucht, it is a concise notation for 3-regular Hamiltonian graphs. It constists of three parameters, the number of vertices in the graph, a list of shifts giving additional edges to a cycle backbone and another integer giving how many times the shifts should be performed. See `http://mathworld.wolfram.com/LCFNotation.html` for details.

Usage

```
graph.lcf(n, shifts, repeats)
```

Arguments

`n`	Integer, the number of vertices in the graph.
`shifts`	Integer vector, the shifts.
`repeats`	Integer constant, how many times to repeat the shifts.

Value

A graph object.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

See Also

`graph` can create arbitrary graphs, see also the other functions on the its manual page for creating special graphs.

Examples

```
# This is the Franklin graph:
g1 <- graph.lcf(12, c(5,-5), 6)
g2 <- graph.famous("Franklin")
graph.isomorphic.vf2(g1, g2)
```

---

graph.matching *Graph matching*

Description

A matching in a graph means the selection of a set of edges that are pairwise non-adjacenct, i.e. they have no common incident vertices. A matching is maximal if it is not a proper subset of any other matching.

Usage

```
is.matching(graph, matching, types = NULL)
is.maximal.matching(graph, matching, types = NULL)
maximum.bipartite.matching(graph, types = NULL, weights = NULL,
       eps = .Machine$double.eps)
```

Arguments

`graph`	The input graph. It might be directed, but edge directions will be ignored.
`types`	Vertex types, if the graph is bipartite. By default they are taken from the '`type`' vertex attribute, if present.
`matching`	A potential matching. An integer vector that gives the pair in the matching for each vertex. For vertices without a pair, supply `NA` here.
`weights`	Potential edge weights. If the graph has an edge attribute called '`weight`', and this argument is `NULL`, then the edge attribute is used automatically.
`eps`	A small real number used in equality tests in the weighted bipartite matching algorithm. Two real numbers are considered equal in the algorithm if their difference is smaller than `eps`. This is required to avoid the accumulation of numerical errors. By default it is set to the smallest x, such that $1 + x \neq 1$ holds. If you are running the algorithm with no weights, this argument is ignored.

Details

`is.matching` checks a matching vector and verifies whether its length matches the number of vertices in the given graph, its values are between zero (inclusive) and the number of vertices (inclusive), and whether there exists a corresponding edge in the graph for every matched vertex pair. For bipartite graphs, it also verifies whether the matched vertices are in different parts of the graph.

`is.maximal.matching` checks whether a matching is maximal. A matching is maximal if and only if there exists no unmatched vertex in a graph such that one of its neighbors is also unmatched.

`maximum.bipartite.matching` calculates a maximum matching in a bipartite graph. A matching in a bipartite graph is a partial assignment of vertices of the first kind to vertices of the second kind such that each vertex of the first kind is matched to at most one vertex of the second kind and vice versa, and matched vertices must be connected by an edge in the graph. The size (or cardinality) of a matching is the number of edges. A matching is a maximum matching if there exists no other matching with larger cardinality. For weighted graphs, a maximum matching is a matching whose edges have the largest possible total weight among all possible matchings.

Maximum matchings in bipartite graphs are found by the push-relabel algorithm with greedy initialization and a global relabeling after every $n/2$ steps where n is the number of vertices in the graph.

Value

`is.matching` and `is.maximal.matching` return a logical scalar.

`maximum.bipartite.matching` returns a list with components:

`matching_size`	The size of the matching, i.e. the number of edges connecting the matched vertices.
`matching_weight`	The weights of the matching, if the graph was weighted. For unweighted graphs this is the same as the size of the matching.
`matching`	The matching itself. Numeric vertex id, or vertex names if the graph was named. Non-matched vertices are denoted by `NA`.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`>

Examples

```
g <- graph.formula( a-b-c-d-e-f )
m1 <- c("b", "a", "d", "c", "f", "e")   # maximal matching
m2 <- c("b", "a", "d", "c", NA, NA)     # non-maximal matching
m3 <- c("b", "c", "d", "c", NA, NA)     # not a matching
is.matching(g, m1)
is.matching(g, m2)
is.matching(g, m3)
is.maximal.matching(g, m1)
is.maximal.matching(g, m2)
is.maximal.matching(g, m3)

V(g)$type <- c(FALSE,TRUE)
str(g, v=TRUE)
maximum.bipartite.matching(g)

g2 <- graph.formula( a-b-c-d-e-f-g )
V(g2)$type <- rep(c(FALSE,TRUE), length=vcount(g2))
str(g2, v=TRUE)
maximum.bipartite.matching(g2)
```

`graph.maxflow` *Maximum flow in a network*

Description

In a graph where each edge has a given flow capacity the maximal flow between two vertices is calculated.

Usage

```
graph.maxflow(graph, source, target, capacity=NULL)
graph.mincut(graph, source=NULL, target=NULL, capacity=NULL,
       value.only = TRUE)
```

Arguments

graph	The input graph.
source	The id of the source vertex.
target	The id of the target vertex (sometimes also called sink).
capacity	Vector giving the capacity of the edges. If this is NULL (the default) then the capacity edge attribute is used.
value.only	Logical scalar, if TRUE only the minumum cut value is returned, if FALSE the edges in the cut and a the two (or more) partitions are also returned.

Details

graph.maxflow calculates the maximum flow between two vertices in a weighted (ie. valued) graph. A flow from source to target is an assignment of non-negative real numbers to the edges of the graph, satisfying two properties: (1) for each edge the flow (ie. the assigned number) is not more than the capacity of the edge (the capacity parameter or edge attribute), (2) for every vertex, except the source and the target the incoming flow is the same as the outgoing flow. The value of the flow is the incoming flow of the target vertex. The maximum flow is the flow of maximum value.

graph.mincut calculates the minimum st-cut between two vertices in a graph (if the source and target arguments are given) or the minimum cut of the graph (if both source and target are NULL).

The minimum st-cut between source and target is the minimum total weight of edges needed to remove to eliminate all paths from source to target.

The minimum cut of a graph is the minimum total weight of the edges needed to remove to separate the graph into (at least) two components. (Which is to make the graph *not* strongly connected in the directed case.)

The maximum flow between two vertices in a graph is the same as the minimum st-cut, so graph.maxflow and graph.mincut essentially calculate the same quantity, the only difference is that graph.mincut can be invoked without giving the source and target arguments and then minimum of all possible minimum cuts is calculated.

For undirected graphs the Stoer-Wagner algorithm (see reference below) is used to calculate the minimum cut.

Value

For graph.maxflow a named list with components:

value	A numeric scalar, the value of the maximum flow.
flow	A numeric vector, the flow itself, one entry for each edge. For undirected graphs this entry is bit trickier, since for these the flow direction is not predetermined by the edge direction. For these graphs the elements of the this vector can be negative, this means that the flow goes from the bigger vertex id to the smaller one. Positive values mean that the flow goes from the smaller vertex id to the bigger one.

`cut`	A numeric vector of edge ids, the minimum cut corresponding to the maximum flow.
`partition1`	A numeric vector of vertex ids, the vertices in the first partition of the minimum cut corresponding to the maximum flow.
`partition2`	A numeric vector of vertex ids, the vertices in the second partition of the minimum cut corresponding to the maximum flow.

For `graph.mincut` a numeric constant, the value of the minimum cut, except if `value.only=FALSE`. In this case a named list with components:

`value`	Numeric scalar, the cut value.
`cut`	Numeric vector, the edges in the cut.
`partition1`	The vertices in the first partition after the cut edges are removed. Note that these vertices might be actually in different components (after the cut edges are removed), as the graph may fall apart into more than two components.
`partition2`	The vertices in the second partition after the cut edges are removed. Note that these vertices might be actually in different components (after the cut edges are removed), as the graph may fall apart into more than two components.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

A. V. Goldberg and R. E. Tarjan: A New Approach to the Maximum Flow Problem *Journal of the ACM* 35:921-940, 1988.

M. Stoer and F. Wagner: A simple min-cut algorithm, *Journal of the ACM*, 44 585-591, 1997.

See Also

`shortest.paths`, `edge.connectivity`, `vertex.connectivity`

Examples

```
E <- rbind( c(1,3,3), c(3,4,1), c(4,2,2), c(1,5,1), c(5,6,2), c(6,2,10))
colnames(E) <- c("from", "to", "capacity")
g1 <- graph.data.frame(as.data.frame(E))
graph.maxflow(g1, source=V(g1)["1"], target=V(g1)["2"])

g <- graph.ring(100)
graph.mincut(g, capacity=rep(1,vcount(g)))
graph.mincut(g, value.only=FALSE, capacity=rep(1,vcount(g)))

g2 <- graph( c(1,2,2,3,3,4, 1,6,6,5,5,4, 4,1) )
E(g2)$capacity <- c(3,1,2, 10,1,3, 2)
graph.mincut(g2, value.only=FALSE)
```

graph.strength *Strength or weighted vertex degree*

Description

Summing up the edge weights of the adjacent edges for each vertex.

Usage

```
graph.strength (graph, vids = V(graph), mode = c("all", "out", "in", "total"),
      loops = TRUE, weights = NULL)
```

Arguments

graph	The input graph.
vids	The vertices for which the strength will be calculated.
mode	Character string, "out" for out-degree, "in" for in-degree or "all" for the sum of the two. For undirected graphs this argument is ignored.
loops	Logical; whether the loop edges are also counted.
weights	Weight vector. If the graph has a `weight` edge attribute, then this is used by default. If the graph does not have a `weight` edge attribute and this argument is `NULL`, then a warning is given and `degree` is called.

Value

A numeric vector giving the strength of the vertices.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, Alessandro Vespignani: The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004)

See Also

`degree` for the unweighted version.

Examples

```
g <- graph.star(10)
E(g)$weight <- seq(ecount(g))
graph.strength(g)
graph.strength(g, mode="out")
graph.strength(g, mode="in")

# No weights, a warning is given
g <- graph.ring(10)
graph.strength(g)
```

`graph.structure` *Method for structural manipulation of graphs*

Description

These are the methods for simple manipulation of graphs: adding and deleting edges and vertices.

Usage

```
## S3 method for class 'igraph'
x[i, j, ..., from, to,
                   sparse=getIgraphOpt("sparsematrices"),
                   edges=FALSE, drop=TRUE,
                   attr=if (is.weighted(x)) "weight" else NULL]
## S3 method for class 'igraph'
x[[i, j, ..., directed=TRUE, edges=FALSE, exact=TRUE]]
## S3 replacement method for class 'igraph'
x[i, j, ..., from, to,
                   attr=if (is.weighted(x)) "weight" else NULL] <- value

## S3 method for class 'igraph'
e1 + e2
## S3 method for class 'igraph'
e1 - e2
vertex(...)
vertices(...)
edge(...)
edges(...)
path(...)

add.edges(graph, edges, ..., attr=list())
add.vertices(graph, nv, ..., attr=list())
delete.edges(graph, edges)
delete.vertices(graph, v)
```

Arguments

`x,graph,e1`	The graph to work on.
`i,j`	Vertex ids or names or logical vectors. See details below.
`...`	These are currently ignored for the indexing operators. For `vertex`, `vertices`, `edge`, `edges` and `path` see details below. For `add.edges` and `add.vertices` these additional parameters will be added as edge/vertex attributes. Note that these arguments have to be named.
`from`	A numeric or character vector giving vertex ids or names. Together with the `to` argument, it can be used to query/set a sequence of edges. See details below. This argument cannot be present together with any of the `i` and `j` arguments and if it is present, then the `to` argument must be present as well.
`to`	A numeric or character vector giving vertex ids or names. Together with the `from` argument, it can be used to query/set a sequence of edges. See details below.

	This argument cannot be present together with any of the `i` and `j` arguments and if it is present, then the `from` argument must be present as well.
`sparse`	Logical scalar, whether to use sparse matrix.
`directed`	Logical scalar, whether to consider edge directions in directed graphs. It is ignored for undirected graphs.
`edges`	Logical scalar, whether to return edge ids.
`drop,exact`	These arguments are ignored.
`value`	A logical or numeric scalar or `NULL`. If `FALSE`, `NULL` or zero, then the specified edges will be deleted. If `TRUE` or a non-zero numeric value, then the specified edges will be added. (Only if they don't yet exist.)
`e2`	See details below.
`attr`	For the indexing operators: if not `NULL`, then it should be the name of an edge attribute. This attribute is queried, or updated to the given value. For `add.edges` and `add.vertices`: additional edge/vertex attributes to add. This will be concatenated to the other supplied attributes.
`nv`	Numeric constant, the number of vertices to add.
`v`	Vector sequence, the vertices to remove.

Details

There are, by and large, three ways to manipulate the structure of a graph in igraph. The first way is using the '`[`' and '`[[`' indexing operators on the graph object, very much like the graph was an adjacency matrix (`[`) or an adjacency list (`[`). The single bracket indexes the (possibly weighted) adjacency matrix of the graph. The double bracket operator is similar, but queries the adjacencly list of the graph. The details on how to use the indexing operators are discussed below.

The addition ('`+`') and division ('`-`') operators can also be used to add and remove vertices and edges. This form is sometimes more readable, and is usually the best if the user also wants to add attributes, together with the new vertices/edges. Please see the details below.

In addition, the four functions, `add.vertices`, `add.edges`, `delete.vertices` and `delete.edges` can also be used to manipulate the structure.

Value

For the indexing operators see the description above. The other functions return a new graph.

The indexing operators

The one-bracket ('`[`') and two-brackets ('`[[`') indexing operators allow relatively straightforward query and update operations on graphs. The one bracket operator works on the (imaginary) adjacency matrix of the graph. Here is what you can do with it:

1. Check whether there is an edge between two vertices (v and w) in the graph:

   ```
   graph[v, w]
   ```

   A numeric scalar is returned, one if the edge exists, zero otherwise.

2. Extract the (sparse) adjacency matrix of the graph, or part of it:

   ```
   graph[]
   graph[1:3,5:6]
   graph[c(1,3,5),]
   ```

The first variants returns the full adjacency matrix, the other two return part of it.

3. The `from` and `to` arguments can be used to check the existence of many edges. In this case, both `from` and `to` must be present and they must have the same length. They must contain vertex ids or names. A numeric vector is returned, of the same length as `from` and `to`, it contains ones for existing edges edges and zeros for non-existing ones. Example:

   ```
   graph[from=1:3, to=c(2,3,5)]
   ```

   .
4. For weighted graphs, the `[` operator returns the edge weights. For non-esistent edges zero weights are returned. Other edge attributes can be queried as well, by giving the `attr` argument.
5. Querying edge ids instead of the existance of edges or edge attributes. E.g.

   ```
   graph[1, 2, edges=TRUE]
   ```

   returns the id of the edge between vertices 1 and 2, or zero if there is no such edge.
6. Adding one or more edges to a graph. For this the element(s) of the imaginary adjacency matrix must be set to a non-zero numeric value (or `TRUE`):

   ```
   graph[1, 2] <- 1
   graph[1:3,1] <- 1
   graph[from=1:3, to=c(2,3,5)] <- TRUE
   ```

   This does not affect edges that are already present in the graph, i.e. no multiple edges are created.
7. Adding weighted edges to a graph. The `attr` argument contains the name of the edge attribute to set, so it does not have to be 'weight':

   ```
   graph[1, 2, attr="weight"]<- 5
   graph[from=1:3, to=c(2,3,5)] <- c(1,-1,4)
   ```

   If an edge is already present in the network, then only its weigths or other attribute are updated. If the graph is already weighted, then the `attr="weight"` setting is implicit, and one does not need to give it explicitly.
8. Deleting edges. The replacement syntax allow the deletion of edges, by specifying `FALSE` or `NULL` as the replacement value:

   ```
   graph[v, w] <- FALSE
   ```

   removes the edge from vertex v to vertex w. As this can be used to delete edges between two sets of vertices, either pairwise:

   ```
   graph[from=v, to=w] <- FALSE
   ```

   or not:

   ```
   graph[v, w] <- FALSE
   ```

   if v and w are vectors of edge ids or names.

The double bracket operator indexes the (imaginary) adjacency list of the graph. This can used for the following operations:

1. Querying the adjacent vertices for one or more vertices:

   ```
   graph[[1:3,]]
   graph[[,1:3]]
   ```

The first form gives the successors, the second the predessors or the 1:3 vertices. (For undirected graphs they are equivalent.)

2. Querying the incident edges for one or more vertices, if the `edges` argument is set to `TRUE`:

   ```
   graph[[1:3, , edges=TRUE]]
   graph[[, 1:3, edges=TRUE]]
   ```

3. Querying the edge ids between two sets or vertices, if both indices are used. E.g.

   ```
   graph[[v, w, edges=TRUE]]
   ```

   gives the edge ids of all the edges that exist from vertices v to vertices w.

Both the '`[`' and '`[[`' operators allow logical indices and negative indices as well, with the usual R semantics. E.g.

```
graph[degree(graph)==0, 1] <- 1
```

adds an edge from every isolate vertex to vertex one, and

```
G <- graph.empty(10)
G[-1,1] <- TRUE
```

creates a star graph.

Of course, the indexing operators support vertex names, so instead of a numeric vertex id a vertex can also be given to '`[`' and '`[[`'.

The plus operator for adding vertices and edges

The plus operator can be used to add vertices or edges to graph. The actual operation that is performed depends on the type of the right hand side argument.

- If it is another igraph graph object, then the disjoint union of the two graphs is calculated, see `graph.disjoint.union`.
- If it is a numeric scalar, then the specified number of vertices are added to the graph.
- If it is a character scalar or vector, then it is interpreted as the names of the vertices to add to the graph.
- If it is an object created with the `vertex` or `vertices` function, then new vertices are added to the graph. This form is appropriate when one wants to add some vertex attributes as well. The operands of the `vertices` function specifies the number of vertices to add and their attributes as well.

  The unnamed arguments of `vertices` are concatenated and used as the '`name`' vertex attribute (i.e. vertex names), the named arguments will be added as additional vertex attributes. Examples:

  ```
  g <- g + vertex(shape="circle", color="red")
  g <- g + vertex("foo", color="blue")
  g <- g + vertex("bar", "foobar")
  g <- g + vertices("bar2", "foobar2", color=1:2, shape="rectangle")
  ```

  See more examples below.

  `vertex` is just an alias to `vertices`, and it is provided for readability. The user should use it if a single vertex is added to the graph.

- If it is an object created with the edge or edges function, then new edges will be added to the graph. The new edges and possibly their attributes can be specified as the arguments of the edges function.

  The unnamed arguments of edges are concatenated and used as vertex ids of the end points of the new edges. The named arguments will be added as edge attributes.

  Examples:

  ```
  g <- graph.empty() + vertices(letters[1:10]) +
  vertices("foo", "bar", "bar2", "foobar2")
  g <- g + edge("a", "b")
  g <- g + edges("foo", "bar", "bar2", "foobar2")
  g <- g + edges(c("bar", "foo", "foobar2", "bar2"), color="red", weight=1:2)
  ```

  See more examples below.

  edge is just an alias to edges and it is provided for readability. The user should use it if a single edge is added to the graph.

- If it is an object created with the path function, then new edges that form a path are added. The edges and possibly their attributes are specified as the arguments to the path function. The non-named arguments are concatenated and interpreted as the vertex ids along the path. The remaining arguments are added as edge attributes.

  Examples:

  ```
  g <- graph.empty() + vertices(letters[1:10])
  g <- g + path("a", "b", "c", "d")
  g <- g + path("e", "f", "g", weight=1:2, color="red")
  g <- g + path(c("f", "c", "j", "d"), width=1:3, color="green")
  ```

It is important to note that, although the plus operator is commutative, i.e. is possible to write

```
graph <- "foo" + graph.empty()
```

it is not associative, e.g.

```
graph <- "foo" + "bar" + graph.empty()
```

results a syntax error, unless parentheses are used:

```
graph <- "foo" + ( "bar" + graph.empty() )
```

For clarity, we suggest to always put the graph object on the left hand side of the operator:

```
graph <- graph.empty() + "foo" + "bar"
```

The minus operator for deleting vertices and edges

The minus operator (‘-’) can be used to remove vertices or edges from the graph. The operation performed is selected based on the type of the right hand side argument:

- If it is an igraph graph object, then the difference of the two graphs is calculated, see graph.difference.
- If it is a numeric or character vector, then it is interpreted as a vector of vertex ids and the specified vertices will be deleted from the graph. Example:

  ```
  g <- graph.ring(10)
  V(g)$name <- letters[1:10]
  g <- g - c("a", "b")
  ```

- If `e2` is a vertex sequence (e.g. created by the `V` function), then these vertices will be deleted from the graph.
- If it is an edge sequence (e.g. created by the `E` function), then these edges will be deleted from the graph.
- If it is an object created with the `vertex` (or the `vertices`) function, then all arguments of `vertices` are concatenated and the result is interpreted as a vector of vertex ids. These vertices will be removed from the graph.
- If it is an object created with the `edge` (or the `edges`) function, then all arguments of `edges` are concatenated and then interpreted as edges to be removed from the graph. Example:

  ```
  g <- graph.ring(10)
  V(g)$name <- letters[1:10]
  E(g)$name <- LETTERS[1:10]
  g <- g - edge("e|f")
  g <- g - edge("H")
  ```
- If it is an object created with the `path` function, then all `path` arguments are concatenated and then interpreted as a path along which edges will be removed from the graph. Example:

  ```
  g <- graph.ring(10)
  V(g)$name <- letters[1:10]
  g <- g - path("a", "b", "c", "d")
  ```

More functions to manipulate graph structure

`add.edges` adds the specified edges to the graph. The ids of the vertices are preserved. The additionally supplied named arguments will be added as edge attributes for the new edges. If an attribute was not present in the original graph, its value for the original edges will be `NA`.

`add.vertices` adds the specified number of isolate vertices to the graph. The ids of the old vertices are preserved. The additionally supplied named arguments will be added as vertex attributes for the new vertices. If an attribute was not present in the original graph, its value is set to `NA` for the original vertices.

`delete.edges` removes the specified edges from the graph. If a specified edge is not present, the function gives an error message, and the original graph remains unchanged. The ids of the vertices are preserved.

`delete.vertices` removes the specified vertices from the graph together with their adjacent edges. The ids of the vertices are *not* preserved.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
# 10 vertices named a,b,c,... and no edges
g <- graph.empty() + vertices(letters[1:10])

# Add edges to make it a ring
g <- g + path(letters[1:10], letters[1], color="grey")

# Add some extra random edges
g <- g + edges(sample(V(g), 10, replace=TRUE), color="red")
g$layout <- layout.circle
if (interactive()) {
```

```
  plot(g)
}

# The old-style operations
g <- graph.ring(10)
add.edges(g, c(2,6,3,7) )
delete.edges(g, E(g, P=c(1,10, 2,3)) )
delete.vertices(g, c(2,7,8) )
```

`Graphs from adjacency lists` *Create graphs from adjacency lists*

Description

An adjacency list is a list of numeric vectors, containing the neighbor vertices for each vertex. This function creates an igraph graph object from such a list.

Usage

```
graph.adjlist(adjlist, mode = c("out", "in", "all", "total"),
               duplicate = TRUE)
```

Arguments

`adjlist`	The adjacency list. It should be consistent, i.e. the maximum throughout all vectors in the list must be less than the number of vectors (=the number of vertices in the graph). Note that the list is expected to be 0-indexed.
`mode`	Character scalar, it specifies whether the graph to create is undirected ('all' or 'total') or directed; and in the latter case, whether it contains the outgoing ('out') or the incoming ('in') neighbors of the vertices.
`duplicate`	Logical scalar. For undirected graphs it gives whether edges are included in the list twice. E.g. if it is `TRUE` then for an undirected `{A,B}` edge `graph.adjlist` expects `A` included in the neighbors of `B` and `B` to be included in the neighbors of `A`. This argument is ignored if `mode` is `out` or `in`.

Details

Adjacency lists are handy if you intend to do many (small) modifications to a graph. In this case adjacency lists are more efficient than igraph graphs.

The idea is that you convert your graph to an adjacency list by `get.adjlist`, do your modifications to the graphs and finally create again an igraph graph by calling `graph.adjlist`.

Value

An igraph graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`get.edgelist`

Examples

```
## Directed
g <- graph.ring(10, dir=TRUE)
al <- get.adjlist(g, mode="out")
g2 <- graph.adjlist(al)
graph.isomorphic(g, g2)

## Undirected
g <- graph.ring(10)
al <- get.adjlist(g)
g2 <- graph.adjlist(al, mode="all")
graph.isomorphic(g, g2)
ecount(g2)
g3 <- graph.adjlist(al, mode="all", duplicate=FALSE)
ecount(g3)
is.multiple(g3)
```

`grg.game` *Geometric random graphs*

Description

Generate a random graph based on the distance of random point on a unit square

Usage

```
grg.game(nodes, radius, torus = FALSE, coords = FALSE)
```

Arguments

`nodes`	The number of vertices in the graph.
`radius`	The radius within which the vertices will be connected by an edge.
`torus`	Logical constant, whether to use a torus instead of a square.
`coords`	Logical scalar, whether to add the positions of the vertices as vertex attributes called ‘`x`’ and ‘`y`’.

Details

First a number of points are dropped on a unit square, these points correspond to the vertices of the graph to create. Two points will be connected with an undirected edge if they are closer to each other in Euclidean norm than a given radius. If the `torus` argument is `TRUE` then a unit area torus is used instead of a square.

Value

A graph object. If `coords` is `TRUE` then with vertex attributes ‘`x`’ and ‘`y`’.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>, first version was written by Keith Briggs <keith.briggs@bt.com>

See Also

random.graph.game

Examples

```
g <- grg.game(1000, 0.05, torus=FALSE)
g2 <- grg.game(1000, 0.05, torus=TRUE)
```

---

growing.random.game *Growing random graph generation*

Description

This function creates a random graph by simulating its stochastic evolution.

Usage

```
growing.random.game(n, m = 1, directed = TRUE, citation = FALSE)
```

Arguments

n	Numeric constant, number of vertices in the graph.
m	Numeric constant, number of edges added in each time step.
directed	Logical, whether to create a directed graph.
citation	Logical. If TRUE a citation graph is created, ie. in each time step the added edges are originating from the new vertex.

Details

This is discrete time step model, in each time step a new vertex is added to the graph and m new edges are created. If citation is FALSE these edges are connecting two uniformly randomly chosen vertices, otherwise the edges are connecting new vertex to uniformly randomly chosen old vertices.

Value

A new graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

barabasi.game, erdos.renyi.game

Examples

```
g <- growing.random.game(500, citation=FALSE)
g2 <- growing.random.game(500, citation=TRUE)
```

`Hierarchical random graphs` *Hierarchical random graphs*

Description

Fitting and sampling hierarchical random graph models.

Usage

```
hrg.fit (graph, hrg = NULL, start = FALSE, steps = 0)
hrg.consensus (graph, hrg = NULL, start = FALSE, num.samples = 10000)

hrg.create (graph, prob)
hrg.dendrogram (hrg)

hrg.game (hrg)

hrg.predict (graph, hrg = NULL, start = FALSE, num.samples = 10000,
             num.bins = 25)

## S3 method for class 'igraphHRG'
print(x, type=c("auto", "tree", "plain"),
           level = 3, ...)
## S3 method for class 'igraphHRGConsensus'
print(x, ...)
```

Arguments

`graph`	The graph to fit the model to. Edge directions are ignored in directed graphs.
`hrg`	A hierarchical random graph model, in the form of an `igraphHRG` object. `hrg.fit` allows this to be `NULL`, in which case a random starting point is used for the fitting. `hrg.consensus` and `hrg.predict` allow this to be `NULL` as well, then a HRG is fitted to the graph first, from a random starting point.
`start`	Logical, whether to start the fitting/sampling from the supplied `igraphHRG` object, or from a random starting point.
`steps`	The number of MCMC steps to make. If this is zero, then the MCMC procedure is performed until convergence.
`num.samples`	Number of samples to use for consensus generation or missing edge prediction.
`prob`	A vector of probabilities, one for each vertex, in the order of vertex ids.
`num.bins`	Number of bins for the edge probabilities. Give a higher number for a more accurate prediction.
`x`	`igraphHRG` or `igraphHRGConsensus` object to print.
`type`	How to print the dendrogram, see details below.
`level`	The number of top levels to print from the dendrogram.
`...`	Additional arguments, not used currently.

Details

A hierarchical random graph is an ensemble of undirected graphs with n vertices. It is defined via a binary tree with n leaf and $n - 1$ internal vertices, where the internal vertices are labeled with probabilities. The probability that two vertices are connected in the random graph is given by the probability label at their closest common ancestor.

Please see references below for more about hierarchical random graphs.

igraph contains functions for fitting HRG models to a given network (`hrg.fit`, for generating networks from a given HRG ensemble (`hrg.game`), converting an igraph graph to a HRG and back (`hrg.create`, `hrg.dendrogram`), for calculating a consensus tree from a set of sampled HRGs (`hrg.consensus`) and for predicting missing edges in a network based on its HRG models (`hrg.predict`).

The igraph HRG implementation is heavily based on the code published by Aaron Clauset, at his website, `http://tuvalu.santafe.edu/~aaronc/hierarchy/`.

`hrg.fit` fits a HRG to a given graph. It takes the specified `steps` number of MCMC steps to perform the fitting, or a convergence criteria if the specified number of steps is zero. `hrg.fit` can start from a given HRG, if this is given in the `hrg` argument and the `start` argument is `TRUE`.

`hrg.consensus` creates a consensus tree from several fitted hierarchical random graph models, using phylogeny methods. If the `hrg` argument is given and `start` is set to `TRUE`, then it starts sampling from the given HRG. Otherwise it optimizes the HRG log-likelihood first, and then samples starting from the optimum.

`hrg.create` creates a HRG from an igraph graph. The igraph graph must be a directed binary tree, with $n - 1$ internal and n leaf vertices. The `prob` argument contains the HRG probability labels for each vertex; these are ignored for leaf vertices.

`hrg.dendrogram` creates the corresponsing igraph tree of a hierarchical random graph model.

`hrg.game` samples a graph from a given hierarchical random graph model.

`hrg.predict` uses a hierarchical random graph model to predict missing edges from a network. This is done by sampling hierarchical models around the optimum model, proportionally to their likelihood. The MCMC sampling is stated from `hrg`, if it is given and the `start` argument is set to `TRUE`. Otherwise a HRG is fitted to the graph first.

Value

`hrg.fit` returns an `igraphHRG` object. This is a list with the following members:

`left`	Vector that contains the left children of the internal tree vertices. The first vertex is always the root vertex, so the first element of the vector is the left child of the root vertex. Internal vertices are denoted with negative numbers, starting from -1 and going down, i.e. the root vertex is -1. Leaf vertices are denoted by non-negative number, starting from zero and up.
`right`	Vector that contains the right children of the vertices, with the same encoding as the `left` vector.
`prob`	The connection probabilities attached to the internal vertices, the first number belongs to the root vertex (i.e. internal vertex -1), the second to internal vertex -2, etc.
`edges`	The number of edges in the subtree below the given internal vertex.
`vertices`	The number of vertices in the subtree below the given internal vertex, including itself.

`hrg.consensus` returns a list of two objects. The first is an `igraphHRGConsensus` object, the second is an `igraphHRG` object. The `igraphHRGConsensus` object has the following members:

`parents`	For each vertex, the id of its parent vertex is stored, or zero, if the vertex is the root vertex in the tree. The first n vertex ids (from 0) refer to the original vertices of the graph, the other ids refer to vertex groups.
`weights`	Numeric vector, counts the number of times a given tree split occured in the generated network samples, for each internal vertices. The order is the same as in the `parents` vector.

`hrg.create` returns an `igraphHRG` object.

`hrg.dendrogram` returns an igraph graph.

`hrg.game` returns an igraph graph.

Printing HRGs to the screen

`igraphHRG` objects can be printed to the screen in two forms: as a tree or as a list, depending on the `type` argument of the print function. By default the `auto` type is used, which selects `tree` for small graphs and `simple` (=list) for bigger ones. The `tree` format looks like this:

```
Hierarchical random graph, at level 3:
g1        p=   0
'- g15    p=0.33  1
   '- g13 p=0.88  6  3  9  4  2  10 7  5  8
'- g8     p= 0.5
   '- g16 p= 0.2  20 14 17 19 11 15 16 13
   '- g5  p=   0  12 18
```

This is a graph with 20 vertices, and the top three levels of the fitted hierarchical random graph are printed. The root node of the HRG is always vertex group #1 ('`g1`' in the the printout). Vertex pairs in the left subtree of `g1` connect to vertices in the right subtree with probability zero, according to the fitted model. `g1` has two subgroups, `g15` and `g8`. `g15` has a subgroup of a single vertex (vertex 1), and another larger subgroup that contains vertices 6, 3, etc. on lower levels, etc.

The `plain` printing is simpler and faster to produce, but less visual:

```
Hierarchical random graph:
g1  p=0.0 -> g12 g10   g2  p=1.0 -> 7 10      g3  p=1.0 -> g18 14
g4  p=1.0 -> g17 15    g5  p=0.4 -> g15 17    g6  p=0.0 -> 1 4
g7  p=1.0 -> 11 16     g8  p=0.1 -> g9 3      g9  p=0.3 -> g11 g16
g10 p=0.2 -> g4 g5     g11 p=1.0 -> g6 5      g12 p=0.8 -> g8 8
g13 p=0.0 -> g14 9     g14 p=1.0 -> 2 6       g15 p=0.2 -> g19 18
g16 p=1.0 -> g13 g2    g17 p=0.5 -> g7 13     g18 p=1.0 -> 12 19
g19 p=0.7 -> g3 20
```

It lists the two subgroups of each internal node, in as many columns as the screen width allows.

Consensus dendrograms (`igraphHRGConsensus` objects) are printed simply by listing the children of each internal node of the dendrogram:

```
HRG consensus tree:
g1 -> 11 12 13 14 15 16 17 18 19 20
g2 -> 1  2  3  4  5  6  7  8  9  10
g3 -> g1 g2
```

The root of the dendrogram is `g3` (because it has no incoming edges), and it has two subgroups, `g1` and `g2`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>, based on code from Aaron Clauset, thanks Aaron!

References

A. Clauset, C. Moore, and M.E.J. Newman. Hierarchical structure and the prediction of missing links in networks. *Nature* 453, 98–101 (2008);

A. Clauset, C. Moore, and M.E.J. Newman. Structural Inference of Hierarchies in Networks. In E. M. Airoldi et al. (Eds.): ICML 2006 Ws, *Lecture Notes in Computer Science* 4503, 1–13. Springer-Verlag, Berlin Heidelberg (2007).

Examples

```
## A graph with two dense groups
g <- erdos.renyi.game(10, p=1/2) + erdos.renyi.game(10, p=1/2)
hrg <- hrg.fit(g)
hrg

## The consensus tree for it
hrg.consensus(g, hrg=hrg, start=TRUE)

## Prediction of missing edges
g2 <- graph.full(4) + (graph.full(4) - path(1,2))
hrg.predict(g2)
```

`igraph console` *The igraph console*

Description

The igraph console is a GUI windows that shows what the currently running igraph function is doing.

Usage

```
igraph.console()
```

Details

The console can be started by calling the `igraph.console` function. Then it stays open, until the user closes it.

Another way to start it to set the `verbose` igraph option to “tkconsole” via `igraph.options`. Then the console (re)opens each time an igraph function supporting it starts; to close it, set the `verbose` option to another value.

The console is written in Tcl/Tk and required the `tcltk` package.

Value

`NULL`, invisibly.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

igraph.options and the verbose option.

---

igraph options *Parameters for the igraph package*

Description

igraph has some parameters which (usually) affect the behavior of many functions. These can be set for the whole session via igraph.options.

Usage

```
igraph.options(...)
getIgraphOpt(x, default = NULL)
igraph.par(parid, parvalue = NULL)
```

Arguments

...	A list may be given as the only argument, or any number of arguments may be in the name=value form, or no argument at all may be given. See the Value and Details sections for explanation.
x	A character string holding an option name.
default	If the specified option is not set in the options list, this value is returned. This facilitates retrieving an option and checking whether it is set and setting it separately if not.
parid	The name of the parameter. See the currently used parameters below.
parvalue	The new value of the parameter. If NULL then the current value of the parameter is listed.

Details

From igraph version 0.6, igraph.par is deprecated. Please use the more flexible igraph.options and getIgraphOpt functions instead.

The parameter values set via a call to the igraph.options function will remain in effect for the rest of the session, affecting the subsequent behaviour of the other functions of the igraph package for which the given parameters are relevant.

This offers the possibility of customizing the functioning of the igraph package, for instance by insertions of appropriate calls to igraph.options in a load hook for package **igraph**.

The currently used parameters in alphabetical order:

add.params Logical scalar, whether to add model parameter to the graphs that are created by the various graph constructors. By default it is TRUE.

add.vertex.names Logical scalar, whether to add vertex names to node level indices, like degree, betweenness scores, etc. By default it is TRUE.

dend.plot.type The plotting function to use when plotting community structure dendrograms via `dendPlot`. Possible values are 'auto' (the default), 'phylo', 'hclust' and 'dendrogram'. See `dendPlot` for details.

edge.attr.comb Specifies what to do with the edge attributes if the graph is modified. The default value is `list(weight="sum", name="concat", "ignore")`. See `attribute.combination` for details on this.

nexus.url The base URL of the default Nexus server. See `nexus` for details.

print.edge.attributes Logical constant, whether to print edge attributes when printing graphs. Defaults to `FALSE`.

print.full Logical scalar, whether `print.igraph` should show the graph structure as well, or only a summary of the graph.

print.graph.attributes Logical constant, whether to print graph attributes when printing graphs. Defaults to `FALSE`.

print.vertex.attributes Logical constant, whether to print vertex attributes when printing graphs. Defaults to `FALSE`.

sparsematrices Whether to use the `Matrix` package for (sparse) matrices. It is recommended, if the user works with larger graphs.

verbose Logical constant, whether igraph functions should talk more than minimal. Eg. if `TRUE` thne some functions will use progress bars while computing. Defaults to `FALSE`.

vertex.attr.comb Specifies what to do with the vertex attributes if the graph is modified. The default value is `list(name="concat", "ignore")` See `attribute.combination` for details on this.

Value

`igraph.options` returns a list with the updated values of the parameters. If the argument list is not empty, the returned list is invisible.

For `getIgraphOpt`, the current value set for option `x`, or `NULL` if the option is unset.

If `parvalue` is `NULL` then `igraph.par` returns the current value of the parameter. Otherwise the new value of the parameter is returned invisibly.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

See Also

`igraph.options` is similar to `options` and `getIgraphOpt` is similar to `getOption`.

Examples

```
oldval <- getIgraphOpt("verbose")
igraph.options(verbose=TRUE)
layout.kamada.kawai(graph.ring(10))
igraph.options(verbose=oldval)
```

igraph.sample *Sampling a random integer sequence*

Description

This function provides a very efficient way to pull an integer random sample sequence from an integer interval.

Usage

```
igraph.sample(low, high, length)
```

Arguments

low	The lower limit of the interval (inclusive).
high	The higher limit of the interval (inclusive).
length	The length of the sample.

Details

The algorithm runs in `O(length)` expected time, even if `high-low` is big. It is much faster (but of course less general) than the builtin `sample` function of R.

Value

An increasing numeric vector containing integers, the sample.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Jeffrey Scott Vitter: An Efficient Algorithm for Sequential Random Sampling, *ACM Transactions on Mathematical Software*, 13/1, 58–67.

Examples

```
rs <- igraph.sample(1, 100000000, 10)
rs
```

`igraph.undocumented` *Undocumented and unsupportted igraph functions*

Description

These functions are still in the alpha stage or their arguments are expected to change, so they're not documented yet. They are also not very useful for the general audience.

Usage

```
lastcit.game(n, edges=1, agebins=n/7100, pref=(1:(agebins+1))^-3,
     directed=TRUE)
cited.type.game(n, edges=1, types=rep(0, n),
     pref=rep(1, length(types)),
     directed=TRUE, attr=TRUE)
citing.cited.type.game(n, edges=1, types=rep(0, n),
    pref=matrix(1, nrow=length(types), ncol=length(types)),
    directed=TRUE, attr=TRUE)
```

Arguments

`n`	Number of vertices.
`edges`	Number of edges per step.
`agebins`	Number of aging bins.
`pref`	Vector (`lastcit.game` and `cited.type.game` or matrix (`citing.cited.type.game`) giving the (unnormalized) citation probabilities for the different vertex types.
`directed`	Logical scalar, whether to generate directed networks.
`types`	Vector of length '`n`', the types of the vertices. Types are numbered from zero.
`attr`	Logical scalar, whether to add the vertex types to the generated graph as a vertex attribute called '`type`'.

Value

A new graph.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

igraphdemo — *Run igraph demos, step by step*

Description

Run one of the accompanying igraph demos, somewhat interactively, using a Tk window.

Usage

```
igraphdemo(which)
```

Arguments

`which`	If not given, then the names of the available demos are listed. Otherwise it should be either a filename or the name of an igraph demo.

Details

This function provides a somewhat nicer interface to igraph demos that come with the package, than the standard `demo` function. Igraph demos are divided into chunks and `igraphdemo` runs them chunk by chunk, with the possibility of inspecting the workspace between two chunks.

The `tcltk` package is needed for `igraphdemo`.

Value

Returns `NULL`, invisibly.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`demo`

Examples

```
igraphdemo()
if (interactive()) {
  igraphdemo("centrality")
}
```

independent.vertex.sets *Independent vertex sets*

Description

A vertex set is called independent if there no edges between any two vertices in it. These functions find independent vertex sets in undirected graphs

Usage

```
independent.vertex.sets(graph, min=NULL, max=NULL)
largest.independent.vertex.sets(graph)
maximal.independent.vertex.sets(graph)
independence.number(graph)
```

Arguments

`graph`	The input graph, directed graphs are considered as undirected, loop edges and multiple edges are ignored.
`min`	Numeric constant, limit for the minimum size of the independent vertex sets to find. `NULL` means no limit.
`max`	Numeric constant, limit for the maximum size of the independent vertex sets to find. `NULL` means no limit.

Details

`independent.vertex.sets` finds all independent vertex sets in the network, obeying the size limitations given in the `min` and `max` arguments.

`largest.independent.vertex.sets` finds the largest independent vertex sets in the graph. An independent vertex set is largest if there is no independent vertex set with more vertices.

`maximal.independent.vertex.sets` finds the maximal independent vertex sets in the graph. An independent vertex set is maximal if it cannot be extended to a larger independent vertex set. The largest independent vertex sets are maximal, but the opposite is not always true.

`independece.number` calculate the size of the largest independent vertex set(s).

These functions use the algorithm described by Tsukiyama et al., see reference below.

Value

`independent.vertex.sets`, `largest.independent.vertex.sets` and `maximal.independent.vertex.sets` return a list containing numeric vertex ids, each list element is an independent vertex set.

`independence.number` returns an integer constant.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`> ported it from the Very Nauty Graph Library by Keith Briggs <`google@for.it`> and Gabor Csardi <`csardi.gabor@gmail.com`> wrote the R interface and this manual page.

References

S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirawaka. A new algorithm for generating all the maximal independent sets. *SIAM J Computing*, 6:505–517, 1977.

See Also

cliques

Examples

```
# A quite dense graph
g <- erdos.renyi.game(100, 0.8)
independence.number(g)
independent.vertex.sets(g, min=independence.number(g))
largest.independent.vertex.sets(g)
# Empty graph
induced.subgraph(g, largest.independent.vertex.sets(g)[[1]])

length(maximal.independent.vertex.sets(g))
```

`infomap.community` *Infomap community finding*

Description

Find community structure that minimizes the expected description length of a random walker trajectory

Usage

```
infomap.community (graph, e.weights = NULL, v.weights = NULL,
                     nb.trials = 10, modularity = TRUE)
```

Arguments

`graph`	The input graph.
`e.weights`	If not `NULL`, then a numeric vector of edge weights. The length must match the number of edges in the graph. By default the '`weight`' edge attribute is used as weights. If it is not present, then all edges are considered to have the same weight.
`v.weights`	If not `NULL`, then a numeric vector of vertex weights. The length must match the number of vertices in the graph. By default the '`weight`' vertex attribute is used as weights. If it is not present, then all vertices are considered to have the same weight.
`nb.trials`	The number of attempts to partition the network (can be any integer value equal or larger than 1).
`modularity`	Logical scalar, whether to calculate the modularity score of the detected community structure.

Details

Please see the details of this method in the references given below.

Value

`infomap.community` returns a `communities` object, please see the `communities` manual page for details.

Author(s)

Martin Rosvall `<martin.rosvall at physics dot umu dot se>` wrote the original C++ code. This was ported to be more igraph-like by Emmanuel Navarro `<navarro at irit dot fr>`. The R interface and some cosmetics was done by Gabor Csardi `<csardi.gabor@gmail.com>`.

References

The original paper: M. Rosvall and C. T. Bergstrom, Maps of information flow reveal community structure in complex networks, *PNAS* 105, 1118 (2008) `http://dx.doi.org/10.1073/pnas.0706851105`, `http://arxiv.org/abs/0707.0609`

A more detailed paper: M. Rosvall, D. Axelsson, and C. T. Bergstrom, The map equation, *Eur. Phys. J. Special Topics* 178, 13 (2009). `http://dx.doi.org/10.1140/epjst/e2010-01179-1`, `http://arxiv.org/abs/0906.1405`.

See Also

Other community finding methods and `communities`.

Examples

```
## Zachary's karate club
g <- graph.famous("Zachary")

imc <- infomap.community(g)
membership(imc)
communities(imc)
```

`interconnected.islands` *A graph with subgraphs that are each a random graph.*

Description

Create a number of Erdos-Renyi random graphs with identical parameters, and connect them with the specified number of edges.

Usage

```
interconnected.islands.game (islands.n, islands.size, islands.pin,
                             n.inter)
```

Arguments

`islands.n`	The number of islands in the graph.
`islands.size`	The size of islands in the graph.
`islands.pin`	The probability to create each possible edge into each island.
`n.inter`	The number of edges to create between two islands.

Value

An igraph graph.

Author(s)

Samuel Thiriot `<samuel.thiriot at res-ear dot ch>`

See Also

`erdos.renyi.game`

Examples

```
g <- interconnected.islands.game(3, 10, 5/10, 1)
oc <- optimal.community(g)
oc
```

`is.chordal` *Chordality of a graph*

Description

A graph is chordal (or triangulated) if each of its cycles of four or more nodes has a chord, which is an edge joining two nodes that are not adjacent in the cycle. An equivalent definition is that any chordless cycles have at most three nodes.

Usage

```
is.chordal(graph, alpha = NULL, alpham1 = NULL, fillin = FALSE,
           newgraph = FALSE)
```

Arguments

`graph`	The input graph. It may be directed, but edge directions are ignored, as the algorithm is defined for undirected graphs.
`alpha`	Numeric vector, the maximal chardinality ordering of the vertices. If it is `NULL`, then it is automatically calculated by calling `maximum.cardinality.search`, or from `alpham1` if that is given..
`alpham1`	Numeric vector, the inverse of `alpha`. If it is `NULL`, then it is automatically calculated by calling `maximum.cardinality.search`, or from `alpha`.
`fillin`	Logical scalar, whether to calculate the fill-in edges.
`newgraph`	Logical scalar, whether to calculate the triangulated graph.

Details

The chordality of the graph is decided by first performing maximum cardinality search on it (if the `alpha` and `alpham1` arguments are `NULL`), and then calculating the set of fill-in edges.

The set of fill-in edges is empty if and only if the graph is chordal.

It is also true that adding the fill-in edges to the graph makes it chordal.

Value

A list with three members:

`chordal`	Logical scalar, it is `TRUE` iff the input graph is chordal.
`fillin`	If requested, then a numeric vector giving the fill-in edges. `NULL` otherwise.
`newgraph`	If requested, then the triangulated graph, an `igraph` object. `NULL` otherwise.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Robert E Tarjan and Mihalis Yannakakis. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computation* 13, 566–579.

See Also

`maximum.cardinality.search`

Examples

```
## The examples from the Tarjan-Yannakakis paper
g1 <- graph.formula(A-B:C:I, B-A:C:D, C-A:B:E:H, D-B:E:F,
                    E-C:D:F:H, F-D:E:G, G-F:H, H-C:E:G:I,
                    I-A:H)
maximum.cardinality.search(g1)
is.chordal(g1, fillin=TRUE)

g2 <- graph.formula(A-B:E, B-A:E:F:D, C-E:D:G, D-B:F:E:C:G,
                    E-A:B:C:D:F, F-B:D:E, G-C:D:H:I, H-G:I:J,
                    I-G:H:J, J-H:I)
maximum.cardinality.search(g2)
is.chordal(g2, fillin=TRUE)
```

`is.dag` *Directed acyclic graphs*

Description

This function tests whether the given graph is a DAG, a directed acyclic graph.

Usage

```
is.dag(graph)
```

Arguments

`graph`	The input graph. It may be undirected, in which case `FALSE` is reported.

Details

`is.dag` checks whether there is a directed cycle in the graph. If not, the graph is a DAG.

Value

A logical vector of length one.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`> for the C code, Gabor Csardi <`csardi.gabor@gmail.com`> for the R interface.

Examples

```
g <- graph.tree(10)
is.dag(g)
g2 <- g + edge(5,1)
is.dag(g2)
```

`is.igraph` *Is this object a graph?*

Description

`is.graph` makes its decision based on the class attribute of the object.

Usage

```
is.igraph(graph)
```

Arguments

`graph`	An R object.

Value

A logical constant, `TRUE` if argument `graph` is a graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
g <- graph.ring(10)
is.igraph(g)
is.igraph(numeric(10))
```

`is.multiple` *Find the multiple or loop edges in a graph*

Description

A loop edge is an edge from a vertex to itself. An edge is a multiple edge if it has exactly the same head and tail vertices as another edge. A graph without multiple and loop edges is called a simple graph.

Usage

```
is.loop(graph, eids=E(graph))
has.multiple(graph)
is.multiple(graph, eids=E(graph))
count.multiple(graph, eids=E(graph))
```

Arguments

`graph`	The input graph.
`eids`	The edges to which the query is restricted. By default this is all edges in the graph.

Details

`is.loop` decides whether the edges of the graph are loop edges.

`has.multiple` decides whether the graph has any multiple edges.

`is.multiple` decides whether the edges of the graph are multiple edges.

`count.multiple` counts the multiplicity of each edge of a graph.

Note that the semantics for `is.multiple` and `count.multiple` is different. `is.multiple` gives `TRUE` for all occurences of a multiple edge except for one. Ie. if there are three `i-j` edges in the graph then `is.multiple` returns `TRUE` for only two of them while `count.multiple` returns '3' for all three.

See the examples for getting rid of multiple edges while keeping their original multiplicity as an edge attribute.

Value

has.multiple returns a logical scalar. is.loop and is.multiple return a logical vector. count.multiple returns a numeric vector.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

simplify to eliminate loop and multiple edges.

Examples

```
# Loops
g <- graph( c(1,1,2,2,3,3,4,5) )
is.loop(g)

# Multiple edges
g <- barabasi.game(10, m=3, algorithm="bag")
has.multiple(g)
is.multiple(g)
count.multiple(g)
is.multiple(simplify(g))
all(count.multiple(simplify(g)) == 1)

# Direction of the edge is important
is.multiple(graph( c(1,2, 2,1) ))
is.multiple(graph( c(1,2, 2,1), dir=FALSE ))

# Remove multiple edges but keep multiplicity
g <- barabasi.game(10, m=3, algorithm="bag")
E(g)$weight <- count.multiple(g)
g <- simplify(g)
any(is.multiple(g))
E(g)$weight
```

---

is.mutual *Find mutual edges in a directed graph*

Description

This function checks the reciproc pair of the supplied edges.

Usage

```
is.mutual(graph, es = E(graph))
```

Arguments

graph	The input graph.
es	Edge sequence, the edges that will be probed. By default is includes all edges in the order of their ids.

Details

In a directed graph an (A,B) edge is mutual if the graph also includes a (B,A) directed edge.

Note that multi-graphs are not handled properly, i.e. if the graph contains two copies of (A,B) and one copy of (B,A), then these three edges are considered to be mutual.

Undirected graphs contain only mutual edges by definition.

Value

A logical vector of the same length as the number of edges supplied.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

reciprocity, dyad.census if you just want some statistics about mutual edges.

Examples

```
g <- erdos.renyi.game(10,50,type="gnm",directed=TRUE)
reciprocity(g)
dyad.census(g)
is.mutual(g)
sum(is.mutual(g))/2 == dyad.census(g)$mut
```

---

is.named *Named graphs*

Description

An igraph graph is named, if there is a symbolic name associated with its vertices.

Usage

```
is.named(graph)
```

Arguments

graph	The input graph.

Details

In igraph vertices can always be identified and specified via their numeric vertex ids. This is, however, not always convenient, and in many cases there exist symbolic ids that correspond to the vertices. To allow this more flexible identification of vertices, one can assign a vertex attribute called 'name' to an igraph graph. After doing this, the symbolic vertex names can be used in all igraph functions, instead of the numeric ids.

Note that the uniqueness of vertex names are currently not enforced in igraph, you have to check that for yourself, when assigning the vertex names.

Value

A logical scalar.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- graph.ring(10)
is.named(g)
V(g)$name <- letters[1:10]
is.named(g)
neighbors(g, "a")
```

---

is.separator *Vertex separators*

Description

These functions check whether a given set of vertices is a vertex separator, or a minimal vertex separator.

Usage

```
is.separator(graph, candidate)
is.minimal.separator(graph, candidate)
```

Arguments

graph	The input graph. It may be directed, but edge directions are ignored.
candidate	A numeric vector giving the vertex ids of the candidate separator.

Details

`is.separator` decides whether the supplied vertex set is a vertex separator. A vertex set is a vertex separator if its removal results a disconnected graph.

`is.minimal.separator` decides whether the supplied vertex set is a minimal vertex separator. A minimal vertex separator is a vertex separator, such that none of its subsets is a vertex separator.

In the special case of a fully connected graph with n vertices, each set of $n-1$ vertices is considered to be a vertex separator.

Value

A logical scalar, whether the supplied vertex set is a (minimal) vertex separator or not.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

`minimum.size.separators` lists all vertex separator of minimum size.

Examples

```
# The graph from the Moody-White paper
mw <- graph.formula(1-2:3:4:5:6, 2-3:4:5:7, 3-4:6:7, 4-5:6:7,
                    5-6:7:21, 6-7, 7-8:11:14:19, 8-9:11:14, 9-10,
                    10-12:13, 11-12:14, 12-16, 13-16, 14-15, 15-16,
                    17-18:19:20, 18-20:21, 19-20:22:23, 20-21,
                    21-22:23, 22-23)

# Cohesive subgraphs
mw1 <- induced.subgraph(mw, as.character(c(1:7, 17:23)))
mw2 <- induced.subgraph(mw, as.character(7:16))
mw3 <- induced.subgraph(mw, as.character(17:23))
mw4 <- induced.subgraph(mw, as.character(c(7,8,11,14)))
mw5 <- induced.subgraph(mw, as.character(1:7))

check.sep <- function(G) {
  sep <- minimum.size.separators(G)
  sapply(sep, is.minimal.separator, graph=G)
}

check.sep(mw)
check.sep(mw1)
check.sep(mw2)
check.sep(mw3)
check.sep(mw4)
check.sep(mw5)
```

`is.weighted` *Weighted graphs*

Description

In weighted graphs, a real number is assigned to each (directed or undirected) edge.

Usage

```
is.weighted(graph)
```

Arguments

`graph`	The input graph.

Details

In igraph edge weights are represented via an edge attribute, called 'weight'. The `is.weighted` function only checks that such an attribute exists. (It does not even checks that it is a numeric edge attribute.)

Edge weights are used for different purposes by the different functions. E.g. shortest path functions use it as the cost of the path; community finding methods use it as the strength of the relationship between two vertices, etc. Check the manual pages of the functions working with weighted graphs for details.

Value

A logical scalar.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- graph.ring(10)
get.shortest.paths(g, 8, 2)
E(g)$weight <- seq_len(ecount(g))
get.shortest.paths(g, 8, 2)
```

---

iterators *Vertex and edge sequences and iterators*

Description

Vertex and edge sequences are central concepts of igraph.

Usage

```
V(graph)
E(graph, P=NULL, path=NULL, directed=TRUE)
```

Arguments

`graph`	A graph object.
`P`	Numeric vector for selecting edges by giving their end points. See details below.
`path`	Numeric vector, this is for selecting all edges along a path. See also details below.
`directed`	Logcal constant, can be supplied only if either `P` or `path` is also present and gives whether the pairs or the path are directed or not.

Details

One often needs to perform an operation on a subset of vertices of edges in a graph.

A vertex sequence is simply a vector containing vertex ids, but it has a special class attribute which makes it possible to perform graph specific operations on it, like selecting a subset of the vertices based on some vertex attributes.

A vertex sequence is created by `V(g)` this selects are vertices in increasing vertex id order. A vertex sequence can be indexed by a numeric vector, and a subset of all vertices can be selected.

Vertex sequences provide powerful operations for dealing with vertex attributes. A vertex sequence can be indexed with the '`$`' operator to select (or modify) the attributes of a subset of vertices. A vertex sequence can be indexed by a logical expression, and this expression may contain the names of the vertex attributes and ordinary variables as well. The return value of such a construct (ie. a vertex sequence indexed by a logical expression) is another vertex sequence containing only vertices from the original sequence for which the expression evaluates to TRUE.

Let us see an example to make everything clear. We assign random numbers between 1 and 100 to the vertices, and select those vertices for which the number is less than 50. We set the color of these vertices to red.

```
g <- graph.ring(10)
V(g)$number <- sample(1:100, vcount(g), replace=TRUE)
V(g)$color <- "grey"
V(g)[ number < 50 ]$color <- "red"
plot(g, layout=layout.circle, vertex.color=V(g)$color,
     vertex.label=V(g)$number)
```

There is a similar notation for edges. `E(g)` selects all edges from the 'g' graph. Edge sequences can be also indexed with logical expressions containing edge attributes:

```
g <- graph.ring(10)
E(g)$weight <- runif(ecount(g))
E(g)$width <- 1
E(g)[ weight >= 0.5 ]$width <- 3
plot(g, layout=layout.circle, edge.width=E(g)$width, edge.color="black")
```

It is important to note that, whenever we use iterators to assign new attribute values, the new values are recycled. So in the following example half of the vertices will be black, the other half red, in an alternated way.

```
g <- graph.ring(10)
V(g)$color <- c("black", "red")
plot(g, layout=layout.circle)
```

For the recycling, the standard R rules apply and a warning is given if the number of items to replace is not a multiple of the replacement length. E.g. the following code gives a warning, because we set the attribute for three vertices, but supply only two values:

```
g <- graph.tree(10)
V(g)$color <- "grey"
V(g)[1:3]$color <- c("green", "blue")
```

If a new vertex/edge attribute is created with an assignment, but only a subset of of vertices are specified, then the rest is set to `NA` if the new values are in a vector and to `NULL` if they are a list. Try the following:

```
V(g)[5]$foo <- "foo"
V(g)$foo
V(g)[5]$bar <- list(bar="bar")
V(g)$bar
```

There are some special functions which are only defined in the indexing expressions of vertex and edge sequences. For vertex sequences these are: `nei`, `inc`, `from` and `to`, `innei` and `outnei`. (The `adj` special function is an alias for `inc`, for compatibility reasons.)

`nei` has a mandatory and an optional argument, the first is another vertex sequence, the second is a mode argument similar to that of the `neighbors` function. `nei` returns a logical vector of the same length as the indexed vertex sequence and evaluates to `TRUE` for those vertices only which have a neighbor vertex in the vertex sequence supplied as a parameter. Thus for selecting all neighbors of vertices 1 and 2 one can write:

```
V(g) [ nei( 1:2 ) ]
```

The mode argument (just like for `neighbors`) gives the type of the neighbors to be included, it is interpreted only in directed graphs, and defaults to all types of neighbors. See the example below. `innei(v)` is a shorthand for the 'incoming' neighbors (`nei(v, mode="in")`), and `outnei(v)` is a shorthand for the 'outgoing' neighbors (`nei(v,mode="out")`).

`inc` takes an edge sequence as an argument and returns `TRUE` for vertices which have at least one incident edge in it.

`from` and `to` are similar to `inc` but only edges originated at (`from`) or pointing to (`to`) are taken into account.

For edge sequences the special functions are: `inc`, `from`, `to`, `%--%`, `%->%` and `%<-%`.

`inc` takes a vertex sequence as an argument and returns `TRUE` for edges which have an incident vertex in it.

`from` and `to` are similar to `inc`, but only vertices at the source (`from`) or target (`to`) of the edge.

The `%--%` operator selects edges connecting two vertex sequences, the direction of the edges is ignored. The `%->%` is different only for directed graphs and only edges pointing from the left hand side argument to the right hand side argument are selected. `%<-%` is exactly the opposite, it selects edges pointing from the right hand side to the left hand side.

`E` has two optional arguments: `P` and `path`. If given `P` can be used to select edges based on their end points, eg. `E(g, P=c(1,2))` selects edge `1->2`.

`path` can be used to select all edges along a path. The path should be given with the visited vertex ids in the appropriate order.

See also the examples below.

Note

A note about the performance of the `V` and `E` functions, and the selection of edges and vertices. Since all selectors are evaluated as logical vectors on all vertices/edges, their performance is bad on large graphs. (Time complexity is proportional to the total number of vertices/edges.) We suggest using the `neighbors`, `incident` functions and simple R vector operations for manipulating vertex/edge sequences in large graphs.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

Examples

```
# mean degree of vertices in the largest cluster in a random graph
g <- erdos.renyi.game(100, 2/100)
c <- clusters(g)
vsl <- which(which.max(c$csize)==c$membership)
mean(degree(g, vsl))

# set the color of these vertices to red, others greens
V(g)$color <- "green"
V(g)[vsl]$color <- "red"
## Not run: plot(g, vertex.size=3, labels=NA, vertex.color="a:color",
              layout=layout.fruchterman.reingold)
## End(Not run)

# the longest geodesic within the largest cluster
long <- numeric()
for (v in vsl) {
  paths <- get.shortest.paths(g, from=v, to=vsl)
  fl <- paths[[ which.max(sapply(paths, length)) ]]
  if (length(fl) > length(long)) {
    long <- fl
  }
}

# the mode argument of the nei() function
g <- graph( c(1,2, 2,3, 2,4, 4,2) )
V(g)[ nei( c(2,4) ) ]
V(g)[ nei( c(2,4), "in") ]
V(g)[ nei( c(2,4), "out") ]

# operators for edge sequences
g <- barabasi.game(100, power=0.3)
E(g) [ 1:3 %--% 2:6 ]
E(g) [ 1:5 %->% 1:6 ]
E(g) [ 1:3 %<-% 2:6 ]

# the edges along the diameter
g <- barabasi.game(100, directed=FALSE)
d <- get.diameter(g)
E(g, path=d)

# performance for large graphs is bad
largeg <- graph.lattice(c(1000, 100))
system.time(E(largeg)[inc(1)])
system.time(incident(largeg, 1))
```

`kleinberg` *Kleinberg's centrality scores.*

Description

Kleinberg's hub and authority scores.

Usage

```
authority.score (graph, scale = TRUE, weights=NULL, options = igraph.arpack.default)
hub.score (graph, scale = TRUE, weights=NULL, options = igraph.arpack.default)
```

Arguments

`graph`	The input graph.
`scale`	Logical scalar, whether to scale the result to have a maximum score of one. If no scaling is used then the result vector has unit length in the Euclidean norm.
`weights`	Optional positive weight vector for calculating weighted scores. If the graph has a `weight` edge attribute, then this is used by default.
`options`	A named list, to override some ARPACK options. See `arpack` for details.

Details

The authority scores of the vertices are defined as the principal eigenvector of $A^T A$, where A is the adjacency matrix of the graph.

The hub scores of the vertices are defined as the principal eigenvector of AA^T, where A is the adjacency matrix of the graph.

Obviously, for undirected matrices the adjacency matrix is symmetric and the two scores are the same.

Value

A named list with members:

`vector`	The authority/hub scores of the vertices.
`value`	The corresponding eigenvalue of the calculated principal eigenvector.
`options`	Some information about the ARPACK computation, it has the same members as the `options` member returned by `arpack`, see that for documentation.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

J. Kleinberg. Authoritative sources in a hyperlinked environment. *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998. Extended version in *Journal of the ACM* 46(1999). Also appears as IBM Research Report RJ 10076, May 1997.

See Also

`evcent` for eigenvector centrality, `page.rank` for the Page Rank scores. `arpack` for the underlining machinery of the computation.

Examples

```
## An in-star
g <- graph.star(10)
hub.score(g)$vector
authority.score(g)$vector

## A ring
g2 <- graph.ring(10)
hub.score(g2)$vector
authority.score(g2)$vector
```

`label.propagation.community` *Finding communities based on propagating labels*

Description

This is a fast, nearly linear time algorithm for detecting community structure in networks. In works by labeling the vertices with unique labels and then updating the labels by majority voting in the neighborhood of the vertex.

Usage

```
label.propagation.community (graph, weights = NULL,
             initial = NULL, fixed = NULL)
```

Arguments

`graph`	The input graph, should be undirected to make sense.
`weights`	An optional weight vector. It should contain a positive weight for all the edges. The 'weight' edge attribute is used if present. Supply '`NA`' here if you want to ignore the 'weight' edge attribute.
`initial`	The initial state. If `NULL`, every vertex will have a different label at the beginning. Otherwise it must be a vector with an entry for each vertex. Non-negative values denote different labels, negative entries denote vertices without labels.
`fixed`	Logical vector denoting which labels are fixed. Of course this makes sense only if you provided an initial state, otherwise this element will be ignored. Also note that vertices without labels cannot be fixed.

Details

This function implements the community detection method described in: Raghavan, U.N. and Albert, R. and Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Phys Rev E 76, 036106. (2007). This version extends the original method by the ability to take edge weights into consideration and also by allowing some labels to be fixed.

From the abstract of the paper: "In our algorithm every node is initialized with a unique label and at every step each node adopts the label that most of its neighbors currently have. In this iterative process densely connected groups of nodes form a consensus on a unique label to form communities."

Value

`label.propagation.community` returns a `communities` object, please see the `communities` manual page for details.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`> for the C implementation, Gabor Csardi <`csardi.gabor@gmail.com`> for this manual page.

References

Raghavan, U.N. and Albert, R. and Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. *Phys Rev E* 76, 036106. (2007)

See Also

`communities` for extracting the actual results.

`fastgreedy.community`, `walktrap.community` and `spinglass.community` for other community detection methods.

Examples

```
g <- erdos.renyi.game(10, 5/10) %du% erdos.renyi.game(9, 5/9)
g <- add.edges(g, c(1, 12))
label.propagation.community(g)
```

---

layout *Generate coordinates for plotting graphs*

Description

Some simple and not so simple functions determining the placement of the vertices for drawing a graph.

Usage

```
layout.auto(graph, dim=2, ...)
layout.random(graph, params, dim=2)
layout.circle(graph, params)
layout.sphere(graph, params)
layout.fruchterman.reingold(graph, ..., dim=2, params)
layout.kamada.kawai(graph, ..., dim=2, params)
layout.spring(graph, ..., params)
layout.reingold.tilford(graph, ..., params)
layout.fruchterman.reingold.grid(graph, ..., params)
layout.lgl(graph, ..., params)
layout.graphopt(graph, ..., params=list())
layout.mds(graph, dist=NULL, dim=2, options=igraph.arpack.default)
layout.svd(graph, d=shortest.paths(graph), ...)
layout.norm(layout, xmin = NULL, xmax = NULL, ymin = NULL, ymax = NULL,
      zmin = NULL, zmax = NULL)
```

Arguments

`graph`	The graph to place.
`params`	The list of function dependent parameters.
`dim`	Numeric constant, either 2 or 3. Some functions are able to generate 2d and 3d layouts as well, supply this argument to change the default behavior. `layout.mds` supports dimensions up to the number of nodes minus one, but only if the graph is connected; for undirected graphs, the only possible values is 2. This is because `layout.merge` only works in 2D.
`...`	Function dependent parameters, this is an alternative notation to the `params` argument. For `layout.auto` these extra parameters are simply passed to the real layout function, if one is called.
`dist`	The distance matrix for the multidimensional scaling. If `NULL` (the default), then the unweighted shortest path matrix is used.
`options`	A named list, to override some ARPACK options. See `arpack` for details.
`d`	The matrix used for singulat value decomposition. By default it is the distance matrix of the graph.
`layout`	A matrix with two or three columns, the layout to normalize.
`xmin,xmax`	The limits for the first coordinate, if one of them or both are `NULL` then no normalization is performed along this direction.
`ymin,ymax`	The limits for the second coordinate, if one of them or both are `NULL` then no normalization is performed along this direction.
`zmin,zmax`	The limits for the third coordinate, if one of them or both are `NULL` then no normalization is performed along this direction.

Details

These functions calculate the coordinates of the vertices for a graph usually based on some optimality criterion.

`layout.auto` tries to choose an appropriate layout function for the supplied graph, and uses that to generate the layout. The current implementations works like this:

1. If the graph has a graph attribute called 'layout', then this is used. If this attribute is an R function, then it is called, with the graph and any other extra arguments.
2. Otherwise, if the graph has vertex attributes called 'x' and 'y', then these are used as coordinates. If the graph has an additional 'z' vertex attribute, that is also used.
3. Otherwise, if the graph is connected and has less than 100 vertices, the Kamada-Kawai layout is used, by calling `layout.kamada.kawai`.
4. Otherwise, if the graph has less than 1000 vertices, then the Fruchterman-Reingold layout is used, by calling `layout.fruchterman.reingold`.
5. Otherwise the DrL layout is used, `layout.drl` is called.

`layout.random` simply places the vertices randomly on a square. It has no parameters.

`layout.circle` places the vertices on a unit circle equidistantly. It has no paramaters.

`layout.sphere` places the vertices (approximately) uniformly on the surface of a sphere, this is thus a 3d layout. It is not clear however what "uniformly on a sphere" means.

`layout.fruchterman.reingold` uses a force-based algorithm proposed by Fruchterman and Reingold, see references. Parameters and their default values:

niter Numeric, the number of iterations to perform (500).

coolexp Numeric, the cooling exponent for the simulated annealing (3).

maxdelta Maximum change (`vcount(graph)`).

area Area parameter (`vcount(graph)^2`).

repulserad Cancellation radius (`area`*vcount(graph)).

weights A vector giving edge weights or `NULL`. If not `NULL` then the attraction along the edges will be multiplied by the given edge weights (`NULL`).

minx If not `NULL`, then it must be a numeric vector that gives lower boundaries for the 'x' coordinates of the vertices. The length of the vector must match the number of vertices in the graph.

maxx Similar to `minx`, but gives the upper boundaries.

miny Similar to `minx`, but gives the lower boundaries of the 'y' coordinates.

maxy Similar to `minx`, but gives the upper boundaries of the 'y' coordinates.

minz Similar to `minx`, but gives the lower boundaries of the 'z' coordinates, if the `dim` argument is 3. Otherwise it is ignored.

maxz Similar to `minx`, but gives the upper boundaries of the 'z' coordinates, if the `dim` argument is 3. Otherwise it is ignored.

start If given, then it should be a matrix with two columns and one line for each vertex. This matrix will be used as starting positions for the algorithm. If not given, then a random starting matrix is used.

This function was ported from the SNA package.

`layout.kamada.kawai` is another force based algorithm. Parameters and default values:

niter Number of iterations to perform (1000).

sigma Sets the base standard deviation of position change proposals (vcount(graph)/4).

initemp The initial temperature (10).

coolexp The cooling exponent (0.99).

kkconst Sets the Kamada-Kawai vertex attraction constant (vcount(graph)**2).

minx If not `NULL`, then it must be a numeric vector that gives lower boundaries for the 'x' coordinates of the vertices. The length of the vector must match the number of vertices in the graph.

maxx Similar to `minx`, but gives the upper boundaries.

miny Similar to `minx`, but gives the lower boundaries of the 'y' coordinates.

maxy Similar to `minx`, but gives the upper boundaries of the 'y' coordinates.

minz Similar to `minx`, but gives the lower boundaries of the 'z' coordinates, if the `dim` argument is 3. Otherwise it is ignored.

maxz Similar to `minx`, but gives the upper boundaries of the 'z' coordinates, if the `dim` argument is 3. Otherwise it is ignored.

start If given, then it should be a matrix with two columns and one line for each vertex. This matrix will be used as starting positions for the algorithm. If not given, then a random starting matrix is used.

This function performs very well for connected graphs, but it gives poor results for unconnected ones. This function was ported from the SNA package.

`layout.spring` is a spring embedder algorithm. Parameters and default values:

mass The vertex mass (in ‘quasi-kilograms’). (Defaults to 0.1.)

equil The equilibrium spring extension (in ‘quasi-meters’). (Defaults to 1.)

k The spring coefficient (in ‘quasi-Newtons per quasi-meter’). (Defaults to 0.001.)

repeqdis The point at which repulsion (if employed) balances out the spring extension force (in ‘quasi-meters’). (Defaults to 0.1.)

kfr The base coefficient of kinetic friction (in ‘quasi-Newton quasi-kilograms’). (Defaults to 0.01.)

repulse Should repulsion be used? (Defaults to FALSE.)

This function was ported from the SNA package.

`layout.reingold.tilford` generates a tree-like layout, so it is mainly for trees. Parameters and default values:

root The id of the root vertex, defaults to 1.

circular Logical scalar, whether to plot the tree in a circular fashion, defaults to `FALSE`.

flip.y Logical scalar, whether to flip the ‘y’ coordinates. The default is flipping because that puts the root vertex on the top.

`layout.fruchterman.reingold.grid` is similar to `layout.fruchterman.reingold` but repelling force is calculated only between vertices that are closer to each other than a limit, so it is faster. Patameters and default values:

niter Numeric, the number of iterations to perform (500).

maxdelta Maximum change for one vertex in one iteration. (The number of vertices in the graph.)

area The area of the surface on which the vertices are placed. (The square of the number of vertices.)

coolexp The cooling exponent of the simulated annealing (1.5).

repulserad Cancellation radius for the repulsion (the `area` times the number of vertices).

cellsize The size of the cells for the grid. When calculating the repulsion forces between vertices only vertices in the same or neighboring grid cells are taken into account (the fourth root of the number of `area`.

start If given, then it should be a matrix with two columns and one line for each vertex. This matrix will be used as starting positions for the algorithm. If not given, then a random starting matrix is used.

`layout.lgl` is for large connected graphs, it is similar to the layout generator of the Large Graph Layout software (`http://bioinformatics.icmb.utexas.edu/lgl`). Parameters and default values:

maxiter The maximum number of iterations to perform (150).

maxdelta The maximum change for a vertex during an iteration (the number of vertices).

area The area of the surface on which the vertices are placed (square of the number of vertices).

coolexp The cooling exponent of the simulated annealing (1.5).

repulserad Cancellation radius for the repulsion (the `area` times the number of vertices).

cellsize The size of the cells for the grid. When calculating the repulsion forces between vertices only vertices in the same or neighboring grid cells are taken into account (the fourth root of the number of `area`.

root The id of the vertex to place at the middle of the layout. The default value is -1 which means that a random vertex is selected.

`layout.graphopt` is a port of the graphopt layout algorithm by Michael Schmuhl. graphopt version 0.4.1 was rewritten in C and the support for layers was removed (might be added later) and a code was a bit reorganized to avoid some unneccessary steps is the node charge (see below) is zero.

graphopt uses physical analogies for defining attracting and repelling forces among the vertices and then the physical system is simulated until it reaches an equilibrium. (There is no simulated annealing or anything like that, so a stable fixed point is not guaranteed.)

See also `http://www.schmuhl.org/graphopt/` for the original graphopt.

Parameters and default values:

niter Integer scalar, the number of iterations to perform. Should be a couple of hundred in general. If you have a large graph then you might want to only do a few iterations and then check the result. If it is not good enough you can feed it in again in the `start` argument. The default value is 500.

charge The charge of the vertices, used to calculate electric repulsion. The default is 0.001.

mass The mass of the vertices, used for the spring forces. The default is 30.

spring.length The length of the springs, an integer number. The default value is zero.

spring.constant The spring constant, the default value is one.

max.sa.movement Real constant, it gives the maximum amount of movement allowed in a single step along a single axis. The default value is 5.

start If given, then it should be a matrix with two columns and one line for each vertex. This matrix will be used as starting positions for the algorithm. If not given, then a random starting matrix is used.

`layout.mds` uses metric multidimensional scaling for generating the coordinates. This function does not have the usual `params` argument. This function generates the layout separately for each graph component and then merges them via `layout.merge`. `layout.mds` is an *experimental* function currently.

`layout.svd` is a currently *experimental* layout function based on singular value decomposition. It does not have the usual `params` argument, but take a single argument, the distance matrix of the graph. This function generates the layout separately for each graph component and then merges them via `layout.merge`.

`layout.norm` normalizes a layout, it linearly transforms each coordinate separately to fit into the given limits.

`layout.drl` is another force-driven layout generator, it is suitable for quite large graphs. See `layout.drl` for details.

Value

All these functions return a numeric matrix with at least two columns and the same number of lines as the number of vertices.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Fruchterman, T.M.J. and Reingold, E.M. (1991). Graph Drawing by Force-directed Placement. *Software - Practice and Experience*, 21(11):1129-1164.

Kamada, T. and Kawai, S. (1989). An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31(1):7-15.

Reingold, E and Tilford, J (1981). Tidier drawing of trees. *IEEE Trans. on Softw. Eng.*, SE-7(2):223–228.

See Also

`layout.drl`, `plot.igraph`, `tkplot`

Examples

```
g <- graph.ring(10)
layout.random(g)
layout.kamada.kawai(g)

# Fixing ego
g <- ba.game(20, m=2)
minC <- rep(-Inf, vcount(g))
maxC <- rep(Inf, vcount(g))
minC[1] <- maxC[1] <- 0
co <- layout.fruchterman.reingold(g, minx=minC, maxx=maxC,
                                  miny=minC, maxy=maxC)
co[1,]
## Not run: plot(g, layout=co, vertex.size=30, edge.arrow.size=0.2,
     vertex.label=c("ego", rep("", vcount(g)-1)), rescale=FALSE,
     xlim=range(co[,1]), ylim=range(co[,2]), vertex.label.dist=1,
     vertex.label.color="red")
axis(1)
axis(2)

## End(Not run)
```

`layout.drl` *The DrL graph layout generator*

Description

DrL is a force-directed graph layout toolbox focused on real-world large-scale graphs, developed by Shawn Martin and colleagues at Sandia National Laboratories.

Usage

```
layout.drl (graph, use.seed = FALSE, seed = matrix(runif(vcount(graph) *
    2), ncol = 2), options = igraph.drl.default, weights = E(graph)$weight,
    fixed = NULL, dim = 2)
```

Arguments

`graph`	The input graph, in can be directed or undirected.
`use.seed`	Logical scalar, whether to use the coordinates given in the seed argument as a starting point.
`seed`	A matrix with two columns, the starting coordinates for the vertices is `use.seed` is `TRUE`. It is ignored otherwise.

`options`	Options for the layout generator, a named list. See details below.
`weights`	Optional edge weights. Supply `NULL` here if you want to weight edges equally. By default the `weight` edge attribute is used if the graph has one.
`fixed`	Logical vector, it can be used to fix some vertices. All vertices for which it is `TRUE` are kept at the coordinates supplied in the `seed` matrix. It is ignored it `NULL` or if `use.seed` is `FALSE`.
`dim`	Either '2' or '3', it specifies whether we want a two dimensional or a three dimensional layout. Note that because of the nature of the DrL algorithm, the three dimensional layout takes significantly longer to compute.

Details

This function implements the force-directed DrL layout generator.

The generator has the following parameters:

edge.cut Edge cutting is done in the late stages of the algorithm in order to achieve less dense layouts. Edges are cut if there is a lot of stress on them (a large value in the objective function sum). The edge cutting parameter is a value between 0 and 1 with 0 representing no edge cutting and 1 representing maximal edge cutting.

init.iterations Number of iterations in the first phase.

init.temperature Start temperature, first phase.

init.attraction Attraction, first phase.

init.damping.mult Damping, first phase.

liquid.iterations Number of iterations, liquid phase.

liquid.temperature Start temperature, liquid phase.

liquid.attraction Attraction, liquid phase.

liquid.damping.mult Damping, liquid phase.

expansion.iterations Number of iterations, expansion phase.

expansion.temperature Start temperature, expansion phase.

expansion.attraction Attraction, expansion phase.

expansion.damping.mult Damping, expansion phase.

cooldown.iterations Number of iterations, cooldown phase.

cooldown.temperature Start temperature, cooldown phase.

cooldown.attraction Attraction, cooldown phase.

cooldown.damping.mult Damping, cooldown phase.

crunch.iterations Number of iterations, crunch phase.

crunch.temperature Start temperature, crunch phase.

crunch.attraction Attraction, crunch phase.

crunch.damping.mult Damping, crunch phase.

simmer.iterations Number of iterations, simmer phase.

simmer.temperature Start temperature, simmer phase.

simmer.attraction Attraction, simmer phase.

simmer.damping.mult Damping, simmer phase.

There are five pre-defined parameter settings as well, these are called `igraph.drl.default`, `igraph.drl.coarsen`, `igraph.drl.coarsest`, `igraph.drl.refine` and `igraph.drl.final`.

Value

A numeric matrix with two columns.

Author(s)

Shawn Martin <`google@for.it`> and Gabor Csardi <`csardi.gabor@gmail.com`> for the R/igraph interface and the three dimensional version.

References

`http://www.cs.sandia.gov/~smartin/software.html`

See Also

`layout` for other layout generators.

Examples

```
g <- as.undirected(ba.game(100, m=1))
l <- layout.drl(g, options=list(simmer.attraction=0))
## Not run:
plot(g, layout=l, vertex.size=3, vertex.label=NA)

## End(Not run)
```

`layout.grid` *Simple grid layout*

Description

This layout places vertices on a rectangulat grid, in two or three dimensions.

Usage

```
layout.grid (graph, width = 0)
layout.grid.3d (graph, width = 0, height = 0)
```

Arguments

`graph`	The input graph.
`width`	The number of vertices in a single row of the grid. If this is zero or negative for `layout.grid`, then the width of the grid will be the square root of the number of vertices in the graph, rounded up to the next integer. Similarly, it will be the cube root for `layout.grid.3d`.
`height`	The number of vertices in a single column of the grid, for three dimensional layouts. If this is zero or negative, then it is determinted automatically.

Details

These functions place the vertices on a simple rectangular grid, one after the other. If you want to change the order of the vertices, then see the `permute.vertices` function.

Value

A two-column matrix for `layout.grid`, a three-column matrix for `layout.grid.3d`.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`>

See Also

`layout` for other layout generators

Examples

```
g <- graph.lattice( c(3,3) )
layout.grid(g)

g2 <- graph.lattice( c(3,3,3) )
layout.grid.3d(g2)

## Not run:
plot(g, layout=layout.grid)
rglplot(g, layout=layout.grid.3d)

## End(Not run)
```

`layout.merge` *Merging graph layouts*

Description

Place several graphs on the same layout

Usage

```
layout.merge(graphs, layouts, method = "dla")
piecewise.layout(graph, layout=layout.kamada.kawai, ...)
```

Arguments

`graphs`	A list of graph objects.
`layouts`	A list of two-column matrices.
`method`	Character constant giving the method to use. Right now only `dla` is implemented.
`graph`	The input graph.
`layout`	A function object, the layout function to use.
`...`	Additional arguments to pass to the `layout` layout function.

Details

layout.merge takes a list of graphs and a list of coordinates and places the graphs in a common layout. The method to use is chosen via the method parameter, although right now only the dla method is implemented.

The dla method covers the graph with circles. Then it sorts the graphs based on the number of vertices first and places the largest graph at the center of the layout. Then the other graphs are placed in decreasing order via a DLA (diffision limited aggregation) algorithm: the graph is placed randomly on a circle far away from the center and a random walk is conducted until the graph walks into the larger graphs already placed or walks too far from the center of the layout.

The piecewise.layout function disassembles the graph first into maximal connected components and calls the supplied layout function for each component separately. Finally it merges the layouts via calling layout.merge.

Value

A matrix with two columns and as many lines as the total number of vertices in the graphs.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

plot.igraph, tkplot, layout, graph.disjoint.union

Examples

```
# create 20 scale-free graphs and place them in a common layout
graphs <- lapply(sample(5:20, 20, replace=TRUE),
          barabasi.game, directed=FALSE)
layouts <- lapply(graphs, layout.kamada.kawai)
lay <- layout.merge(graphs, layouts)
g <- graph.disjoint.union(graphs)
## Not run: plot(g, layout=lay, vertex.size=3, labels=NA, edge.color="black")
```

layout.star *Generate coordinates to place the vertices of a graph in a star-shape*

Description

A simple layout generator, that places one vertex in the center of a circle and the rest of the vertices equidistantly on the perimeter.

Usage

```
layout.star(graph, center = V(graph)[1], order = NULL)
```

Arguments

`graph`	The graph to layout.
`center`	The id of the vertex to put in the center. By default it is the first vertex.
`order`	Numeric vector, the order of the vertices along the perimeter. The default ordering is given by the vertex ids.

Details

It is possible to choose the vertex that will be in the center, and the order of the vertices can be also given.

Value

A matrix with two columns and as many rows as the number of vertices in the input graph.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`layout` and `layout.drl` for other layout algorithms, `plot.igraph` and `tkplot` on how to plot graphs and `graph.star` on how to create ring graphs.

Examples

```
g <- graph.star(10)
layout.star(g)
```

`layout.sugiyama` *The Sugiyama graph layout generator*

Description

Sugiyama layout algorithm for layered directed acyclic graphs. The algorithm minimized edge crossings.

Usage

```
layout.sugiyama (graph, layers = NULL, hgap = 1, vgap = 1, maxiter = 100,
    weights = NULL, attributes = c("default", "all", "none"))
```

Arguments

`graph`	The input graph.
`layers`	A numeric vector or `NULL`. If not `NULL`, then it should specify the layer index of the vertices. Layers are numbered from one. If `NULL`, then igraph calculates the layers automatically.
`hgap`	Real scalar, the minimum horizontal gap between vertices in the same layer.
`vgap`	Real scalar, the distance between layers.

`maxiter`	Integer scalar, the maximum number of iterations in the crossing minimization stage. 100 is a reasonable default; if you feel that you have too many edge crossings, increase this.
`weights`	Optional edge weight vector. If `NULL`, then the 'weight' edge attribute is used, if there is one. Supply `NA` here and igraph ignores the edge weights.
`attributes`	Which graph/vertex/edge attributes to keep in the extended graph. 'default' keeps the 'size', 'size2', 'shape', 'label' and 'color' vertex attributes and the 'arrow.mode' and 'arrow.size' edge attributes. 'all' keep all graph, vertex and edge attributes, 'none' keeps none of them.

Details

This layout algorithm is designed for directed acyclic graphs where each vertex is assigned to a layer. Layers are indexed from zero, and vertices of the same layer will be placed on the same horizontal line. The X coordinates of vertices within each layer are decided by the heuristic proposed by Sugiyama et al. to minimize edge crossings.

You can also try to lay out undirected graphs, graphs containing cycles, or graphs without an a priori layered assignment with this algorithm. igraph will try to eliminate cycles and assign vertices to layers, but there is no guarantee on the quality of the layout in such cases.

The Sugiyama layout may introduce "bends" on the edges in order to obtain a visually more pleasing layout. This is achieved by adding dummy nodes to edges spanning more than one layer. The resulting layout assigns coordinates not only to the nodes of the original graph but also to the dummy nodes. The layout algorithm will also return the extended graph with the dummy nodes.

For more details, see the reference below.

Value

A list with the components:

`layout`	The layout, a two-column matrix, for the original graph vertices.
`layout.dummy`	The layout for the dummy vertices, a two column matrix.
`extd_graph`	The original graph, extended with dummy vertices. The 'dummy' vertex attribute is set on this graph, it is a logical attributes, and it tells you whether the vertex is a dummy vertex. The 'layout' graph attribute is also set, and it is the layout matrix for all (original and dummy) vertices.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`>

References

K. Sugiyama, S. Tagawa and M. Toda, "Methods for Visual Understanding of Hierarchical Systems". IEEE Transactions on Systems, Man and Cybernetics 11(2):109-125, 1981.

Examples

```
## Data taken from http://tehnick-8.narod.ru/dc_clients/
DC <- graph.formula("DC++" -+
                "LinuxDC++":"BCDC++":"EiskaltDC++":"StrongDC++":"DiCe!++",
                "LinuxDC++" -+ "FreeDC++", "BCDC++" -+ "StrongDC++",
                "FreeDC++" -+ "BMDC++":"EiskaltDC++",
                "StrongDC++" -+ "AirDC++":"zK++":"ApexDC++":"TkDC++",
                "StrongDC++" -+ "StrongDC++ SQLite":"RSX++",
                "ApexDC++" -+ "FlylinkDC++ ver <= 4xx",
                "ApexDC++" -+ "ApexDC++ Speed-Mod":"DiCe!++",
                "StrongDC++ SQLite" -+ "FlylinkDC++ ver >= 5xx",
                "ApexDC++ Speed-Mod" -+ "FlylinkDC++ ver <= 4xx",
                "ApexDC++ Speed-Mod" -+ "GreylinkDC++",
                "FlylinkDC++ ver <= 4xx" -+ "FlylinkDC++ ver >= 5xx",
                "FlylinkDC++ ver <= 4xx" -+ AvaLink,
                "GreylinkDC++" -+ AvaLink:"RayLinkDC++":"SparkDC++":PeLink)

## Use edge types
E(DC)$lty <- 1
E(DC)["BCDC++" %->% "StrongDC++"]$lty <- 2
E(DC)["FreeDC++" %->% "EiskaltDC++"]$lty <- 2
E(DC)["ApexDC++" %->% "FlylinkDC++ ver <= 4xx"]$lty <- 2
E(DC)["ApexDC++" %->% "DiCe!++"]$lty <- 2
E(DC)["StrongDC++ SQLite" %->% "FlylinkDC++ ver >= 5xx"]$lty <- 2
E(DC)["GreylinkDC++" %->% "AvaLink"]$lty <- 2

## Layers, as on the plot
layers <- list(c("DC++"),
               c("LinuxDC++", "BCDC++"),
               c("FreeDC++", "StrongDC++"),
               c("BMDC++", "EiskaltDC++", "AirDC++", "zK++", "ApexDC++",
                 "TkDC++", "RSX++"),
               c("StrongDC++ SQLite", "ApexDC++ Speed-Mod", "DiCe!++"),
               c("FlylinkDC++ ver <= 4xx", "GreylinkDC++"),
               c("FlylinkDC++ ver >= 5xx", "AvaLink", "RayLinkDC++",
                 "SparkDC++", "PeLink"))

## Check that we have all nodes
all(sort(unlist(layers)) == sort(V(DC)$name))

## Add some graphical parameters
V(DC)$color <- "white"
V(DC)$shape <- "rectangle"
V(DC)$size <- 20
V(DC)$size2 <- 10
V(DC)$label <- lapply(V(DC)$name, function(x)
                      paste(strwrap(x, 12), collapse="\n"))
E(DC)$arrow.size <- 0.5

## Create a similar layout using the predefined layers
lay1 <- layout.sugiyama(DC, layers=apply(sapply(layers,
                        function(x) V(DC)$name %in% x), 1, which))

## Simple plot, not very nice
par(mar=rep(.1, 4))
plot(DC, layout=lay1$layout, vertex.label.cex=0.5)

## Sugiyama plot
plot(lay1$extd_graph, vertex.label.cex=0.5)

## The same with automatic layer calculation
## Keep vertex/edge attributes in the extended graph
lay2 <- layout.sugiyama(DC, attributes="all")
```

```
plot(lay2$extd_graph, vertex.label.cex=0.5)

## Another example, from the following paper:
## Markus Eiglsperger, Martin Siebenhaller, Michael Kaufmann:
## An Efficient Implementation of Sugiyama's Algorithm for
## Layered Graph Drawing, Journal of Graph Algorithms and
## Applications 9, 305--325 (2005).

ex <- graph.formula( 0 -+ 29: 6: 5:20: 4,
                     1 -+ 12,
                     2 -+ 23: 8,
                     3 -+  4,
                     4,
                     5 -+  2:10:14:26: 4: 3,
                     6 -+  9:29:25:21:13,
                     7,
                     8 -+ 20:16,
                     9 -+ 28: 4,
                    10 -+ 27,
                    11 -+  9:16,
                    12 -+  9:19,
                    13 -+ 20,
                    14 -+ 10,
                    15 -+ 16:27,
                    16 -+ 27,
                    17 -+  3,
                    18 -+ 13,
                    19 -+  9,
                    20 -+  4,
                    21 -+ 22,
                    22 -+  8: 9,
                    23 -+  9:24,
                    24 -+ 12:15:28,
                    25 -+ 11,
                    26 -+ 18,
                    27 -+ 13:19,
                    28 -+  7,
                    29 -+ 25                    )

layers <- list( 0, c(5, 17), c(2, 14, 26, 3), c(23, 10, 18), c(1, 24),
                12, 6, c(29,21), c(25,22), c(11,8,15), 16, 27, c(13,19),
                c(9, 20), c(4, 28), 7 )

layex <- layout.sugiyama(ex, layers=apply(sapply(layers,
                        function(x) V(ex)$name %in% as.character(x)),
                        1, which))

origvert <- c(rep(TRUE, vcount(ex)), rep(FALSE, nrow(layex$layout.dummy)))
realedge <- get.edgelist(layex$extd_graph)[,2] <= vcount(ex)
plot(layex$extd_graph, vertex.label.cex=0.5,
     edge.arrow.size=.5,
     vertex.size=ifelse(origvert, 5, 0),
     vertex.shape=ifelse(origvert, "square", "none"),
     vertex.label=ifelse(origvert, V(ex)$name, ""),
     edge.arrow.mode=ifelse(realedge, 2, 0))
```

leading.eigenvector.community

Community structure detecting based on the leading eigenvector of the community matrix

Description

This function tries to find densely connected subgraphs in a graph by calculating the leading non-negative eigenvector of the modularity matrix of the graph.

Usage

```
leading.eigenvector.community(graph, steps = -1, start = NULL,
       options = igraph.arpack.default, callback = NULL, extra = NULL,
       env = parent.frame())
community.le.to.membership(merges, steps, membership)
```

Arguments

`graph`	The input graph. Should be undirected as the method needs a symmetric matrix.
`steps`	The number of steps to take, this is actually the number of tries to make a step. It is not a particularly useful parameter. For `community.le.to.membership` the number of merges to produce from the supplied `membership` vector.
`start`	`NULL`, or a numeric membership vector, giving the start configuration of the algorithm.
`membership`	The starting community structure on which `steps` merges are performed.
`options`	A named list to override some ARPACK options.
`callback`	If not `NULL`, then it must be callback function. This is called after each iteration, after calculating the leading eigenvector of the modularity matrix. See details below.
`extra`	Additional argument to supply to the callback function.
`env`	The environment in which the callback function is evaluated.
`merges`	The merge matrix, possible from the result of `leading.eigenvector.community`.

Details

The function documented in these section implements the ‘leading eigenvector’ method developed by Mark Newman, see the reference below.

The heart of the method is the definition of the modularity matrix, `B`, which is `B=A-P`, `A` being the adjacency matrix of the (undirected) network, and `P` contains the probability that certain edges are present according to the ‘configuration model’. In other words, a `P[i,j]` element of `P` is the probability that there is an edge between vertices `i` and `j` in a random network in which the degrees of all vertices are the same as in the input graph.

The leading eigenvector method works by calculating the eigenvector of the modularity matrix for the largest positive eigenvalue and then separating vertices into two community based on the sign of the corresponding element in the eigenvector. If all elements in the eigenvector are of the same sign that means that the network has no underlying comuunity structure. Check Newman's paper to understand why this is a good method for detecting community structure.

`community.le.to.memberhip` creates a membership vector from the result of `leading.eigenvector.community`. It takes `membership` and permformes `steps` merges, according to the supplied `merges` matrix.

Value

`leading.eigenvector.community` returns a named list with the following members:

`membership`	The membership vector at the end of the algorithm, when no more splits are possible.
`merges`	The merges matrix starting from the state described by the `membership` member. This is a two-column matrix and each line describes a merge of two communities, the first line is the first merge and it creates community '`N`', `N` is the number of initial communities in the graph, the second line creates community `N+1`, etc.
`options`	Information about the underlying ARPACK computation, see `arpack` for details.

`community.le.to.membership` returns a named list with two components:

`membership`	A membership vector, a numerical vector indication which vertex belongs to which community. The communities are always numbered from one.
`csize`	A numeric vector giving the sizes of the communities.

Callback functions

The `callback` argument can be used to supply a function that is called after each eigenvector calculation. The following arguments are supplied to this function:

membership The actual membership vector, with zero-based indexing.

community The community that the algorithm just tried to split, community numbering starts with zero here.

value The eigenvalue belonging to the leading eigenvector the algorithm just found.

vector The leading eigenvector the algorithm just found.

multiplier An R function that can be used to multiple the actual modularity matrix with an arbitrary vector. Supply the vector as an argument to perform this multiplication. This function can be used with ARPACK.

extra The `extra` argument that was passed to `leading.eigenvector.community`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

MEJ Newman: Finding community structure using the eigenvectors of matrices, Physical Review E 74 036104, 2006.

See Also

`modularity`, `walktrap.community`, `edge.betweenness.community`, `fastgreedy.community`, `as.dendrogram`

Examples

```
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(1,6, 1,11, 6, 11))
lec <- leading.eigenvector.community(g)
lec

leading.eigenvector.community(g, start=membership(lec))
```

`line.graph` *Line graph of a graph*

Description

This function calculates the line graph of another graph.

Usage

```
line.graph(graph)
```

Arguments

`graph`	The input graph, it can be directed or undirected.

Details

The line graph `L(G)` of a `G` undirected graph is defined as follows. `L(G)` has one vertex for each edge in `G` and two vertices in `L(G)` are connected by an edge if their corresponding edges share an end point.

The line graph `L(G)` of a `G` directed graph is slightly different, `L(G)` has one vertex for each edge in `G` and two vertices in `L(G)` are connected by a directed edge if the target of the first vertex's corresponding edge is the same as the source of the second vertex's corresponding edge.

Value

A new graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>, the first version of the C code was written by Vincent Matossian.

Examples

```
# generate the first De-Bruijn graphs
g <- graph.full(2, directed=TRUE, loops=TRUE)
line.graph(g)
line.graph(line.graph(g))
line.graph(line.graph(line.graph(g)))
```

`maximum.cardinality.search`
Maximum cardinality search

Description

Maximum cardinality search is a simple ordering a vertices that is useful in determining the chordality of a graph.

Usage

```
maximum.cardinality.search(graph)
```

Arguments

`graph`	The input graph. It may be directed, but edge directions are ignored, as the algorithm is defined for undirected graphs.

Details

Maximum cardinality search visits the vertices in such an order that every time the vertex with the most already visited neighbors is visited. Ties are broken randomly.

The algorithm provides a simple basis for deciding whether a graph is chordal, see References below, and also `is.chordal`.

Value

A list with two components:

`alpha`	Numeric vector. The vertices ordered according to the maximum cardinality search.
`alpham1`	Numeric vector. The inverse of `alpha`.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Robert E Tarjan and Mihalis Yannakakis. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computation* 13, 566–579.

See Also

`is.chordal`

Examples

```
## The examples from the Tarjan-Yannakakis paper
g1 <- graph.formula(A-B:C:I, B-A:C:D, C-A:B:E:H, D-B:E:F,
                    E-C:D:F:H, F-D:E:G, G-F:H, H-C:E:G:I,
                    I-A:H)
maximum.cardinality.search(g1)
is.chordal(g1, fillin=TRUE)

g2 <- graph.formula(A-B:E, B-A:E:F:D, C-E:D:G, D-B:F:E:C:G,
                    E-A:B:C:D:F, F-B:D:E, G-C:D:H:I, H-G:I:J,
                    I-G:H:J, J-H:I)
maximum.cardinality.search(g2)
is.chordal(g2, fillin=TRUE)
```

minimal.st.separators *Minimum size vertex separators*

Description

List all vertex sets that are minimal (s,t) separators for some s and t, in an undirected graph.

Usage

```
minimal.st.separators(graph)
```

Arguments

graph	The input graph. It may be directed, but edge directions are ignored.

Details

A (s,t) vertex separator is a set of vertices, such that after their removal from the graph, there is no path between s and t in the graph.

A (s,t) vertex separator is minimal if none of its subsets is an (s,t) vertex separator.

Value

A list of numeric vectors. Each vector contains a vertex set (defined by vertex ids), each vector is an (s,t) separator of the input graph, for some s and t.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Anne Berry, Jean-Paul Bordat and Olivier Cogis: Generating All the Minimal Separators of a Graph, In: Peter Widmayer, Gabriele Neyer and Stephan Eidenbenz (editors): *Graph-theoretic concepts in computer science*, 1665, 167–172, 1999. Springer.

Examples

```
ring <- graph.ring(4)
minimal.st.separators(ring)

chvatal <- graph.famous("chvatal")
minimal.st.separators(chvatal)
```

`minimum.size.separators` *Minimum size vertex separators*

Description

Find all vertex sets of minimal size whose removal separates the graph into more components

Usage

```
minimum.size.separators (graph)
```

Arguments

`graph`	The input graph. It may be directed, but edge directions are ignored.

Details

This function implements the Kanevsky algorithm for finding all minimal-size vertex separators in an undirected graph. See the reference below for the details.

In the special case of a fully connected input graph with n vertices, all subsets of size $n-1$ are listed as the result.

Value

A list of numeric vectors. Each numeric vector is a vertex separator.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Arkady Kanevsky: Finding all minimum-size separating vertex sets in a graph. *Networks* 23 533–541, 1993.

JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, *Algorithmica* 15, 351–372, 1996.

J. Moody and D. R. White. Structural cohesion and embeddedness: A hierarchical concept of social groups. *American Sociological Review*, 68 103–127, Feb 2003.

See Also

`is.separator`

Examples

```
# The graph from the Moody-White paper
mw <- graph.formula(1-2:3:4:5:6, 2-3:4:5:7, 3-4:6:7, 4-5:6:7,
                    5-6:7:21, 6-7, 7-8:11:14:19, 8-9:11:14, 9-10,
                    10-12:13, 11-12:14, 12-16, 13-16, 14-15, 15-16,
                    17-18:19:20, 18-20:21, 19-20:22:23, 20-21,
                    21-22:23, 22-23)

# Cohesive subgraphs
mw1 <- induced.subgraph(mw, as.character(c(1:7, 17:23)))
mw2 <- induced.subgraph(mw, as.character(7:16))
mw3 <- induced.subgraph(mw, as.character(17:23))
mw4 <- induced.subgraph(mw, as.character(c(7,8,11,14)))
mw5 <- induced.subgraph(mw, as.character(1:7))

minimum.size.separators(mw)
minimum.size.separators(mw1)
minimum.size.separators(mw2)
minimum.size.separators(mw3)
minimum.size.separators(mw4)
minimum.size.separators(mw5)

# Another example, the science camp network
camp <- graph.formula(Harry:Steve:Don:Bert - Harry:Steve:Don:Bert,
                      Pam:Brazey:Carol:Pat - Pam:Brazey:Carol:Pat,
                      Holly   - Carol:Pat:Pam:Jennie:Bill,
                      Bill    - Pauline:Michael:Lee:Holly,
                      Pauline - Bill:Jennie:Ann,
                      Jennie  - Holly:Michael:Lee:Ann:Pauline,
                      Michael - Bill:Jennie:Ann:Lee:John,
                      Ann     - Michael:Jennie:Pauline,
                      Lee     - Michael:Bill:Jennie,
                      Gery    - Pat:Steve:Russ:John,
                      Russ    - Steve:Bert:Gery:John,
                      John    - Gery:Russ:Michael)
lapply(minimum.size.separators(camp), function(x) V(camp)[x])
```

`minimum.spanning.tree` *Minimum spanning tree*

Description

A subgraph of a connected graph is a *minimum spanning tree* if it is tree, and the sum of its edge weights are the minimal among all tree subgraphs of the graph. A minimum spanning forest of a graph is the graph consisting of the minimum spanning trees of its components.

Usage

```
minimum.spanning.tree(graph, weights=NULL, algorithm=NULL, ...)
```

Arguments

`graph`	The graph object to analyze.

`weights`	Numeric algorithm giving the weights of the edges in the graph. The order is determined by the edge ids. This is ignored if the `unweighted` algorithm is chosen
`algorithm`	The algorithm to use for calculation. `unweighted` can be used for unwieghted graphs, and `prim` runs Prim's algorithm for weighted graphs. If this is `NULL` then igraph tries to select the algorithm automatically: if the graph has an edge attribute called `weight` of the `weights` argument is not `NULL` then Prim's algorithm is chosen, otherwise the unwweighted algorithm is performed.
`...`	Additional arguments, unused.

Details

If the graph is unconnected a minimum spanning forest is returned.

Value

A graph object with the minimum spanning forest. (To check that it is a tree check that the number of its edges is `vcount(graph)-1`.) The edge and vertex attributes of the original graph are preserved in the result.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Prim, R.C. 1957. Shortest connection networks and some generalizations *Bell System Technical Journal*, 37 1389–1401.

See Also

`clusters`

Examples

```
g <- erdos.renyi.game(100, 3/100)
mst <- minimum.spanning.tree(g)
```

`modularity` *Modularity of a community structure of a graph*

Description

This function calculates how modular is a given division of a graph into subgraphs.

Usage

```
## S3 method for class 'igraph'
modularity(x, membership, weights = NULL, ...)
```

Arguments

`x`	The input graph.
`membership`	Numeric vector, for each vertex it gives its community. The communities are numbered from one.
`weights`	If not `NULL` then a numeric vector giving edge weights.
`...`	Additional arguments, none currently.

Details

The modularity of a graph with respect to some division (or vertex types) measures how good the division is, or how separated are the different vertex types from each other. It defined as

$$Q = \frac{1}{2m} \sum_{i,j} A_{ij} - \frac{k_i k_j}{2m} \delta(c_i, c_j),$$

here m is the number of edges, A_{ij} is the element of the A adjacency matrix in row i and column j, k_i is the degree of i, k_j is the degree of j, c_i is the type (or component) of i, c_j that of j, the sum goes over all i and j pairs of vertices, and $\delta(x, y)$ is 1 if $x = y$ and 0 otherwise.

If edge weights are given, then these are considered as the element of the A adjacency matrix, and k_i is the sum of weights of adjacent edges for vertex i.

Value

A numeric scalar, the modularity score of the given configuration.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

MEJ Newman and M Girvan: Finding and evaluating community structure in networks. Physical Review E 69 026113, 2004.

See Also

`walktrap.community`, `edge.betweenness.community`, `fastgreedy.community`, `spinglass.community` for various community detection methods.

Examples

```
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(1,6, 1,11, 6, 11))
wtc <- walktrap.community(g)
modularity(wtc)
modularity(g, membership(wtc))
```

`multilevel.community` *Finding community structure by multi-level optimization of modularity*

Description

This function implements the multi-level modularity optimization algorithm for finding community structure, see references below. It is based on the modularity measure and a hierarchial approach.

Usage

```
multilevel.community (graph, weights = NULL)
```

Arguments

`graph`	The input graph.
`weights`	Optional positive weight vector. If the graph has a `weight` edge attribute, then this is used by default. Supply `NA` here if the graph has a `weight` edge attribute, but you want to ignore it.

Details

This function implements the multi-level modularity optimization algorithm for finding community structure, see VD Blondel, J-L Guillaume, R Lambiotte and E Lefebvre: Fast unfolding of community hierarchies in large networks, `http://arxiv.org/abs/arXiv:0803.0476` for the details.

It is based on the modularity measure and a hierarchial approach. Initially, each vertex is assigned to a community on its own. In every step, vertices are re-assigned to communities in a local, greedy way: each vertex is moved to the community with which it achieves the highest contribution to modularity. When no vertices can be reassigned, each community is considered a vertex on its own, and the process starts again with the merged communities. The process stops when there is only a single vertex left or when the modularity cannot be increased any more in a step.

This function was contributed by Tom Gregorovic.

Value

`multilevel.community` returns a `communities` object, please see the `communities` manual page for details.

Author(s)

Tom Gregorovic, Tamas Nepusz `<tamas@cs.rhul.ac.uk>`

References

Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre: Fast unfolding of communities in large networks. J. Stat. Mech. (2008) P10008

See Also

See `communities` for extracting the membership, modularity scores, etc. from the results.

Other community detection algorithms: `walktrap.community`, `spinglass.community`, `leading.eigenvector.commu`
`edge.betweenness.community`, `fastgreedy.community`, `label.propagation.community`

Examples

```
# This is so simple that we will have only one level
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(1,6, 1,11, 6, 11))
multilevel.community(g)
```

`neighborhood` *Neighborhood of graph vertices*

Description

These functions find the vertices not farther than a given limit from another fixed vertex, these are called the neighborhood of the vertex.

Usage

```
neighborhood.size(graph, order, nodes=V(graph), mode=c("all", "out", "in"))
neighborhood(graph, order, nodes=V(graph), mode=c("all", "out", "in"))
graph.neighborhood(graph, order, nodes=V(graph), mode=c("all", "out", "in"))
connect.neighborhood(graph, order, mode=c("all", "out", "in", "total"))
```

Arguments

`graph`	The input graph.
`order`	Integer giving the order of the neighborhood.
`nodes`	The vertices for which the calculation is performed.
`mode`	Character constatnt, it specifies how to use the direction of the edges if a directed graph is analyzed. For 'out' only the outgoing edges are followed, so all vertices reachable from the source vertex in at most `order` steps are counted. For '"in"' all vertices from which the source vertex is reachable in at most `order` steps are counted. '"all"' ignores the direction of the edges. This argument is ignored for undirected graphs.

Details

The neighborhood of a given order `o` of a vertex `v` includes all vertices which are closer to `v` than the order. Ie. order 0 is always `v` itself, order 1 is `v` plus its immediate neighbors, order 2 is order 1 plus the immediate neighbors of the vertices in order 1, etc.

`neighborhood.size` calculates the size of the neighborhoods for the given vertices with the given order.

`neighborhood` calculates the neighborhoods of the given vertices with the given order parameter.

`graph.neighborhood` is creates (sub)graphs from all neighborhoods of the given vertices with the given order parameter. This function preserves the vertex, edge and graph attributes.

`connect.neighborhood` creates a new graph by connecting each vertex to all other vertices in its neighborhood.

Value

`neighborhood.size` returns with an integer vector.

`neighborhood` returns with a list of integer vectors.

`graph.neighborhood` returns with a list of graphs.

`connect.neighborhood` returns with a new graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>, the first version was done by Vincent Matossian

Examples

```
g <- graph.ring(10)
neighborhood.size(g, 0, 1:3)
neighborhood.size(g, 1, 1:3)
neighborhood.size(g, 2, 1:3)
neighborhood(g, 0, 1:3)
neighborhood(g, 1, 1:3)
neighborhood(g, 2, 1:3)

# attributes are preserved
V(g)$name <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")
graph.neighborhood(g, 2, 1:3)

# connecting to the neighborhood
g <- graph.ring(10)
g <- connect.neighborhood(g, 2)
```

---

nexus *Query and download from the Nexus network repository*

Description

The Nexus network repository is an online collection of network data sets. These functions can be used to query it and download data from it, directly as an igraph graph.

Usage

```
nexus.list(tags=NULL, offset=0, limit=10,
           operator=c("or", "and"),
           order=c("date", "name", "popularity"),
           nexus.url=getIgraphOpt("nexus.url"))
nexus.info(id, nexus.url=getIgraphOpt("nexus.url"))
nexus.get(id, offset=0, order=c("date", "name", "popularity"),
          nexus.url=getIgraphOpt("nexus.url"))
nexus.search(q, offset=0, limit=10,
             order=c("date", "name", "popularity"),
             nexus.url=getIgraphOpt("nexus.url"))

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

```
## S3 method for class 'nexusDatasetInfoList'
summary(object, ...)
## S3 method for class 'nexusDatasetInfoList'
print(x, ...)
```

Arguments

`tags`	A character vector, the tags that are searched. If not given (or `NULL`), then all datasets are listed.
`offset`	An offset to select part of the results. Results are listed from `offset+1`.
`limit`	The maximum number of results to return.
`operator`	A character scalar. If 'or' (the default), then all datasets that have at least one of the given tags, are returned. If it if 'and', then only datasets that have all the given tags, are returned.
`order`	The ordering of the results, possible values are: 'date', 'name', 'popularity'.
`id`	The numeric or character id of the data set to query or download. Instead of the data set ids, it is possible to supply a `nexusDatasetInfo` or `nexusDatasetInfoList` object here directly and then the query is done on the corresponding data set(s).
`q`	Nexus search string. See examples below. For the complete documentation please see the Nexus homepage at `http://nexus.igraph.org`.
`nexus.url`	The URL of the Nexus server. Don't change this from the default, unless you set up your own Nexus server.
`x,object`	The `nexusDatasetInfo` object to print.
`...`	Currently ignored.

Details

Nexus is an online repository of networks, with an API that allow programatic queries against it, and programatic data download as well.

The `nexus.list` and `nexus.info` functions query the online database. They both return `nexusDatasetInfo` objects. `nexus.info` returns more information than `nexus.list`.

`nexus.search` searches Nexus, and returns a list of data sets, as `nexusDatasetInfo` objects. See below for some search examples.

`nexus.get` downloads a data set from Nexus, based on its numeric id, or based on a Nexus search string. For search strings, only the first search hit is downloaded, but see also the `offset` argument. (If there are not data sets found, then the function returns an error.)

The `nexusDatasetInfo` objects returned by `nexus.list` have the following fields:

id The numeric id of the dataset.

sid The character id of the dataset.

name Character scalar, the name of the dataset.

vertices/edges Character, the number of vertices and edges in the graph(s). Vertices and edges are separated by a slash, and if the data set consists of multiple networks, then they are separated by spaces.

tags Character vector, the tags of the dataset. Directed graph have the tags 'directed'. Undirected graphs are tagged as 'undirected'. Other common tags are: 'weighted', 'bipartite', 'social network', etc.

networks The ids and names of the networks in the data set. The numeric and character id are separated by a slash, and multiple networks are separated by spaces.

`nexusDatasetInfo` objects returned by `nexus.info` have the following additional fields:

date Character scalar, e.g. '2011-01-09', the date when the dataset was added to the database.

formats Character vector, the data formats in which the data set is available. The various formats are separated by semicolons.

licence Character scalar, the licence of the dataset.

licence url Character scalar, the URL of the licence of the dataset. Pleaase make sure you consult this before using a dataset.

summary Character scalar, the short description of the dataset, this is usually a single sentence.

description Character scalar, the full description of the dataset.

citation Character scalar, the paper(s) describing the dataset. Please cite these papers if you are using the dataset in your research, the licence of most datasets requires this.

attributes A list of lists, each list entry is a graph, vertex or edge attribute and has the following entries:

type Type of the attribute, either 'graph', 'vertex' or 'edge'.

datatype Data type of the attribute, currently it can be 'numeric' and 'string'.

name Character scalar, the name of the attribute.

description Character scalar, the description of the attribute.

The results of the Nexus queries are printed to the screen in a consise format, similar to the format of igraph graphs. A data set list (typically the result of `nexus.list` and `nexus.search`) looks like this:

```
NEXUS 1-5/18 -- data set list
[1] kaptail.4         39/109-223    #18 Kapferer tailor shop
[2] condmatcollab2003 31163/120029 #17 Condensed matter collaborations+
[3] condmatcollab     16726/47594  #16 Condensed matter collaborations+
[4] powergrid         4941/6594    #15 Western US power grid
[5] celegansneural    297/2359     #14 C. Elegans neural network
```

Each line here represents a data set, and the following information is given about them: the character id of the data set (e.g. `kaptail` or `powergrid`), the number of vertices and number of edges in the graph of the data sets. For data sets with multiple graphs, intervals are given here. Then the numeric id of the data set and the reamining space is filled with the name of the data set.

Summary information about an individual Nexus data set is printed as

```
NEXUS B--- 39 109-223 #18 kaptail -- Kapferer tailor shop
+ tags: directed; social network; undirected
+ nets: 1/KAPFTI2; 2/KAPFTS2; 3/KAPFTI1; 4/KAPFTS1
```

This is very similar to the header that is used for printing igraph graphs, but there are some differences as well. The four characters after the `NEXUS` word give the most important properties of the graph(s): the first is '`U`' for undirected and '`D`' for directed graphs, and '`B`' if the data set contains both directed and undirected graphs. The second is '`N`' named graphs. The third character is '`W`' for weighted graphs, the fourth is '`B`' if the data set contains bipartite graphs. Then the number of vertices and number of edges are printed, for data sets with multiple graphs, the smallest and the largest values are given. Then comes the numeric id, and the string id of the data set. The end of the first line contains the name of the data set. The second row lists the data set tags, and the third row the networks that are included in the data set.

Detailed data set information is printed similarly, but it contains more fields.

Value

`nexus.list` and `nexus.search` return a list of `nexusDatasetInfo` objects. The list also has these attributes:

size The number of data sets returned by the query.

totalsize The total number of data sets found for the query.

offset The offset parameter of the query.

limit The limit parameter of the query.

`nexus.info` returns a single `nexusDatasetInfo` object.

`nexus.get` returns an igraph graph object, or a list of graph objects, if the data set consists of multiple networks.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`http://nexus.igraph.org`

Examples

```
## Not run: nexus.list(tag="weighted")
nexus.list(limit=3, order="name")
nexus.list(limit=3, order="name")[[1]]
nexus.info(2)
g <- nexus.get(2)
summary(g)

## Data sets related to 'US':
nexus.search("US")

## Search for data sets that have 'network' in their name:
nexus.search("name:network")

## Any word can match
nexus.search("blog or US or karate")

## End(Not run)
```

`optimal.community` *Optimal community structure*

Description

This function calculates the optimal community structure of a graph, by maximizing the modularity measure over all possible partitions.

Usage

```
optimal.community(graph)
```

Arguments

`graph`	The input graph. Edge directions are ignored for directed graphs.

Details

This function calculates the optimal community structure for a graph, in terms of maximal modularity score.

The calculation is done by transforming the modularity maximization into an integer programming problem, and then calling the GLPK library to solve that. Please the reference below for details.

Note that modularity optimization is an NP-complete problem, and all known algorithms for it have exponential time complexity. This means that you probably don't want to run this function on larger graphs. Graphs with up to fifty vertices should be fine, graphs with a couple of hundred vertices might be possible.

Value

`optimal.community` returns a `communities` object, please see the `communities` manual page for details.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, Dorothea Wagner: On Modularity Clustering, *IEEE Transactions on Knowledge and Data Engineering* 20(2):172-188, 2008.

See Also

`communities` for the documentation of the result, `modularity`. See also `fastgreedy.community` for a fast greedy optimizer.

Examples

```
## Zachary's karate club
g <- graph.famous("Zachary")

## We put everything into a big 'try' block, in case
## igraph was compiled without GLPK support

try({
  ## The calculation only takes a couple of seconds
  oc <- optimal.community(g)

  ## Double check the result
  print(modularity(oc))
  print(modularity(g, membership(oc)))

  ## Compare to the greedy optimizer
  fc <- fastgreedy.community(g)
  print(modularity(fc))
}, silent=TRUE)
```

page.rank — *The Page Rank algorithm*

Description

Calculates the Google PageRank for the specified vertices.

Usage

```
page.rank (graph, vids = V(graph), directed = TRUE, damping = 0.85,
    personalized = NULL, weights = NULL,
    options = igraph.arpack.default)
page.rank.old (graph, vids = V(graph), directed = TRUE, niter = 1000,
    eps = 0.001, damping = 0.85, old = FALSE)
```

Arguments

`graph`	The graph object.
`vids`	The vertices of interest.
`directed`	Logical, if true directed paths will be considered for directed graphs. It is ignored for undirected graphs.
`damping`	The damping factor (‘d’ in the original paper).
`personalized`	Optional vector giving a probability distribution to calculate personalized PageRank. For personalized PageRank, the probability of jumping to a node when abandoning the random walk is not uniform, but it is given by this vector. The vector should contains an entry for each vertex and it will be rescaled to sum up to one.
`weights`	A numerical vector or `NULL`. This argument can be used to give edge weights for calculating the weighted PageRank of vertices. If this is `NULL` and the graph has a `weight` edge attribute then that is used. If `weights` is a numerical vector then it used, even if the graph has a `weights` edge attribute. If this is `NA`, then no edge weights are used (even if the graph has a `weight` edge attribute.
`options`	A named list, to override some ARPACK options. See `arpack` for details.
`niter`	The maximum number of iterations to perform.
`eps`	The algorithm will consider the calculation as complete if the difference of PageRank values between iterations change less than this value for every node.
`old`	A logical scalar, whether the old style (pre igraph 0.5) normalization to use. See details below.

Details

For the explanation of the PageRank algorithm, see the following webpage: `http://www-db.stanford.edu/~backrub/google.html`, or the following reference:

Sergey Brin and Larry Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Proceedings of the 7th World-Wide Web Conference, Brisbane, Australia, April 1998.

igraph 0.5 (and later) contains two PageRank calculation implementations. The `page.rank` function uses ARPACK to perform the calculation, see also `arpack`.

The `page.rank.old` function performs a simple power method, this is the implementation that was available under the name `page.rank` in pre 0.5 igraph versions. Note that `page.rank.old` has an argument called `old`. If this argument is `FALSE` (the default), then the proper PageRank algorithm is used, i.e. $(1-d)/n$ is added to the weighted PageRank of vertices to calculate the next iteration. If this argument is `TRUE` then $(1-d)$ is added, just like in the PageRank paper; d is the damping factor, and n is the total number of vertices. A further difference is that the old implementation does not renormalize the page rank vector after each iteration. Note that the `old=FALSE` method is not stable, is does not necessarily converge to a fixed point. It should be avoided for new code, it is only included for compatibility with old igraph versions.

Please note that the PageRank of a given vertex depends on the PageRank of all other vertices, so even if you want to calculate the PageRank for only some of the vertices, all of them must be calculated. Requesting the PageRank for only some of the vertices does not result in any performance increase at all.

Since the calculation is an iterative process, the algorithm is stopped after a given count of iterations or if the PageRank value differences between iterations are less than a predefined value.

Value

For `page.rank` a named list with entries:

`vector`	A numeric vector with the PageRank scores.
`value`	The eigenvalue corresponding to the eigenvector with the page rank scores. It should be always exactly one.
`options`	Some information about the underlying ARPACK calculation. See `arpack` for details.

For `page.rank.old` a numeric vector of Page Rank scores.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`> and Gabor Csardi <`csardi.gabor@gmail.com`>

References

Sergey Brin and Larry Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Proceedings of the 7th World-Wide Web Conference, Brisbane, Australia, April 1998.

See Also

Other centrality scores: `closeness`, `betweenness`, `degree`

Examples

```
g <- random.graph.game(20, 5/20, directed=TRUE)
page.rank(g)$vector

g2 <- graph.star(10)
page.rank(g2)$vector

# Personalized PageRank
g3 <- graph.ring(10)
page.rank(g3)$vector
reset <- seq(vcount(g3))
page.rank(g3, personalized=reset)$vector
```

permute.vertices *Permute the vertices of a graph*

Description

Create a new graph, by permuting vertex ids.

Usage

```
permute.vertices(graph, permutation)
```

Arguments

graph	The input graph, it can directed or undirected.
permutation	A numeric vector giving the permutation to apply. The first element is the new id of vertex 1, etc. Every number between one and vcount(graph) must appear exactly once.

Details

This function creates a new graph from the input graph by permuting its vertices according to the specified mapping. Call this function with the output of canonical.permutation to create the canonical form of a graph.

permute.vertices keeps all graph, vertex and edge attributes of the graph.

Value

A new graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

See Also

canonical.permutation

Examples

```
# Random permutation of a random graph
g <- random.graph.game(20, 50, type="gnm")
g2 <- permute.vertices(g, sample(vcount(g)))
graph.isomorphic(g, g2)

# Permutation keeps all attributes
g$name <- "Random graph, Gnm, 20, 50"
V(g)$name <- letters[1:vcount(g)]
E(g)$weight <- sample(1:5, ecount(g), replace=TRUE)
g2 <- permute.vertices(g, sample(vcount(g)))
graph.isomorphic(g, g2)
g2$name
V(g2)$name
E(g2)$weight
all(sort(E(g2)$weight) == sort(E(g)$weight))
```

`Pie charts as vertices` *Using pie charts as vertices in graph plots*

Description

More complex vertex images can be used to express addtional information about vertices. E.g. pie charts can be used as vertices, to denote vertex classes, fuzzy classification of vertices, etc.

Details

The vertex shape 'pie' makes igraph draw a pie chart for every vertex. There are some extra graphical vertex parameters that specify how the pie charts will look like:

pie Numeric vector, gives the sizes of the pie slices.

pie.color A list of color vectors to use for the pies. If it is a list of a single vector, then this is used for all pies. It the color vector is shorter than the number of areas in a pie, then it is recycled.

pie.border The color of the border line of the pie charts, in the same format as `pie.color`.

pie.angle The slope of shading lines, given as an angle in degrees (counter-clockwise).

pie.density The density of the shading lines, in lines per inch. Non-positive values inhibit the drawing of shading lines.

pie.lty The line type of the border of the slices.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`igraph.plotting`, `plot.igraph`

Examples

```
g <- graph.ring(10)
values <- lapply(1:10, function(x) sample(1:10,3))
if (interactive()) {
  plot(g, vertex.shape="pie", vertex.pie=values,
       vertex.pie.color=list(heat.colors(5)),
       vertex.size=seq(10,30,length=10), vertex.label=NA)
}
```

`plot.igraph` *Plotting of graphs*

Description

`plot.graph` is able to plot graphs to any R device. It is the non-interactive companion of the `tkplot` function.

Usage

```
## S3 method for class 'igraph'
plot(x, axes=FALSE, xlab="", ylab="", add=FALSE,
      xlim=c(-1,1), ylim=c(-1,1), main="", sub="",
      mark.groups=list(), mark.shape=1/2, mark.border=NA,
      mark.col=rainbow(length(mark.groups), alpha=0.3),
      mark.expand=15,
      ...)
```

Arguments

`x`	The graph to plot.
`axes`	Logical, whether to plot axes, defaults to FALSE.
`xlab`	The label of the horizontal axis. Defaults to the empty string.
`ylab`	The label of the vertical axis. Defaults to the empty string.
`add`	Logical scalar, whether to add the plot to the current device, or delete the device's current contents first.
`xlim`	The limits for the horizontal axis, it is unlikely that you want to modify this.
`ylim`	The limits for the vertical axis, it is unlikely that you want to modify this.
`main`	Main title.
`sub`	Subtitle.
`mark.groups`	A list of vertex id vectors. It is interpreted as a set of vertex groups. Each vertex group is highlighted, by plotting a colored smoothed polygon around and "under" it. See the arguments below to control the look of the polygons.
`mark.shape`	A numeric scalar or vector. Controls the smoothness of the vertex group marking polygons. This is basically the 'shape' parameter of the `xspline` function, its possible values are between -1 and 1. If it is a vector, then a different value is used for the different vertex groups.
`mark.border`	A scalar or vector giving the colors of the borders of the vertex group marking polygons. If it is `NA`, then no border is drawn.
`mark.col`	A scalar or vector giving the colors of marking the polygons, in any format accepted by `xspline`; e.g. numeric color ids, symbolic color names, or colors in RGB.
`mark.expand`	A numeric scalar or vector, the size of the border around the marked vertex groups. It is in the same units as the vertex sizes. If a vector is given, then different values are used for the different vertex groups.
`...`	Additional arguments, passed to `plot`.

Details

One convenient way to plot graphs is to plot with `tkplot` first, handtune the placement of the vertices, query the coordinates by the `tkplot.getcoords` function and use them with `plot` to plot the graph to any R device.

Value

Returns `NULL`, invisibly.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

See Also

`layout` for different layouts, `igraph.plotting` for the detailed description of the plotting parameters and `tkplot` and `rglplot` for other graph plotting functions.

Examples

```
g <- graph.ring(10)
## Not run: plot(g, layout=layout.kamada.kawai, vertex.color="green")
```

`power.law.fit` *Fitting a power-law distribution function to discrete data*

Description

`power.law.fit` fits a power-law distribution to a data set.

Usage

```
power.law.fit(x, xmin = NULL, start = 2, ...)
```

Arguments

`x`	The data to fit, a numeric vector containing integer values.
`xmin`	The lower bound for fitting the power-law. If NULL, the smallest value in `x` will be used. This argument makes it possible to fit only the tail of the distribution.
`start`	The initial value of the exponent for the minimizing function. Ususally it is safe to leave this untouched.
`...`	Additional arguments, passed to the maximum likelyhood optimizing function, `mle`.

Details

A power-law distribution is fitted with maximum likelyhood methods as recommended by Newman and (by default) the BFGS optimization (see mle) algorithm is applied.

The additional arguments are passed to the mle function, so it is possible to change the optimization method and/or its parameters.

Value

An object with class 'mle'. It can be used to calculate confidence intervals and log-likelihood. See mle-class for details.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Power laws, Pareto distributions and Zipf's law, M. E. J. Newman, *Contemporary Physics*, in press.

See Also

mle

Examples

```
# This should approximately yield the correct exponent 3
g <- barabasi.game(1000)    # increase this number to have a better estimation
d <- degree(g, mode="in")
power.law.fit(d+1, 20)
```

---

preference.game *Trait-based random generation*

Description

Generation of random graphs based on different vertex types.

Usage

```
preference.game(nodes, types, type.dist=rep(1, types), fixed.sizes=FALSE,
    pref.matrix=matrix(1, types, types), directed=FALSE, loops=FALSE)
asymmetric.preference.game(nodes, types,
    type.dist.matrix=matrix(1,types,types),
    pref.matrix = matrix(1, types, types), loops=FALSE)
```

Arguments

nodes	The number of vertices in the graphs.
types	The number of different vertex types.
type.dist	The distribution of the vertex types, a numeric vector of length 'types' containing non-negative numbers. The vector will be normed to obtain probabilities.
fixed.sizes	Fix the number of vertices with a given vertex type label. The type.dist argument gives the group sizes (i.e. number of vertices with the different labels) in this case.
type.dist.matrix	The joint distribution of the in- and out-vertex types.
pref.matrix	A square matrix giving the preferences of the vertex types. The matrix has 'types' rows and columns.

`directed`	Logical constant, whether to create a directed graph.
`loops`	Logical constant, whether self-loops are allowed in the graph.

Details

Both models generate random graphs with given vertex types. For `preference.game` the probability that two vertices will be connected depends on their type and is given by the 'pref.matrix' argument. This matrix should be symmetric to make sense but this is not checked. The distribution of the different vertes types is given by the 'type.dist' vector.

For `asymmetric.preference.game` each vertex has an in-type and an out-type and a directed graph is created. The probability that a directed edge is realized from a vertex with a given out-type to a vertex with a given in-type is given in the 'pref.matrix' argument, which can be asymmetric. The joint distribution for the in- and out-types is given in the 'type.dist.matrix' argument.

Value

An igraph graph.

Author(s)

Tamas Nepusz `<ntamas@gmail.com>` and Gabor Csardi `<csardi.gabor@gmail.com>` for the R interface

See Also

`establishment.game`. `callaway.traits.game`

Examples

```
pf <- matrix( c(1, 0, 0, 1), nr=2)
g <- preference.game(20, 2, pref.matrix=pf)
## Not run: tkplot(g, layout=layout.fruchterman.reingold)

pf <- matrix( c(0, 1, 0, 0), nr=2)
g <- asymmetric.preference.game(20, 2, pref.matrix=pf)
## Not run: tkplot(g, layout=layout.circle)
```

`print.igraph` *Print graphs to the terminal*

Description

These functions attempt to print a graph to the terminal in a human readable form.

Usage

```
## S3 method for class 'igraph'
print(x, full=getIgraphOpt("print.full"),
  graph.attributes=getIgraphOpt("print.graph.attributes"),
  vertex.attributes=getIgraphOpt("print.vertex.attributes"),
  edge.attributes=getIgraphOpt("print.edge.attributes"),
  names=TRUE, ...)
```

```
## S3 method for class 'igraph'
summary(object, ...)
## S3 method for class 'igraph'
str(object, ...)
```

Arguments

`x`	The graph to print.
`full`	Logical scalar, whether to print the graph structure itself as well.
`graph.attributes`	Logical constant, whether to print graph attributes.
`vertex.attributes`	Logical constant, whether to print vertex attributes.
`edge.attributes`	Logical constant, whether to print edge attributes.
`names`	Logical constant, whether to print symbolic vertex names (ie. the `name` vertex attribute) or vertex ids.
`object`	The graph of which the summary will be printed.
`...`	Additional agruments.

Details

`summary.igraph` prints the number of vertices, edges and whether the graph is directed.

`str.igraph` prints the same information, and also lists the edges, and optionally graph, vertex and/or edge attributes.

`print.igraph` behaves either as `summary.igraph` or `str.igraph` depending on the `full` argument. See also the 'print.full' igraph option and `getIgraphOpt`.

The graph summary printed by `summary.igraph` (and `print.igraph` and `str.igraph`) consists one or more lines. The first line contains the basic properties of the graph, and the rest contains its attributes. Here is an example, a small star graph with weighed directed edges and named vertices:

```
IGRAPH DNW- 10 9 -- In-star
+ attr: name (g/c), mode (g/c), center (g/n), name (v/c),
  weight (e/n)
```

The first line always starts with `IGRAPH`, showing you that the object is an igraph graph. Then a four letter long code string is printed. The first letter distinguishes between directed ('`D`') and undirected ('`U`') graphs. The second letter is '`N`' for named graphs, i.e. graphs with the `name` vertex attribute set. The third letter is '`W`' for weighted graphs, i.e. graphs with the `weight` edge attribute set. The fourth letter is '`B`' for bipartite graphs, i.e. for graphs with the `type` vertex attribute set.

Then, after two dashes, the name of the graph is printed, if it has one, i.e. if the `name` graph attribute is set.

From the second line, the attributes of the graph are listed, separated by a comma. After the attribute names, the kind of the attribute – graph, vertex or edge – is denoted, and the type of the attribute as well, character, numeric, or other ('`x`').

As of igraph 0.4 `str.igraph` (and `print.igraph`) uses the `max.print` option, see `options` for details.

Value

All these functions return the graph invisibly.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
g <- graph.ring(10)
g
summary(g)
```

`read.graph` *Reading foreign file formats*

Description

The `read.graph` function is able to read graphs in various representations from a file, or from a http connection. Currently some simple formats are supported.

Usage

```
read.graph(file, format = c("edgelist", "pajek", "ncol", "lgl",
        "graphml", "dimacs", "graphdb", "gml", "dl"), ...)
```

Arguments

`file`	The connection to read from. This can be a local file, or a `http` or `ftp` connection. It can also be a character string with the file name or URI.
`format`	Character constant giving the file format. Right now `edgelist`, `pajek`, `graphml`, `gml`, `ncol`, `lgl`, `dimacs` and `graphdb` are supported, the default is `edgelist`. As of igraph 0.4 this argument is case insensitive.
`...`	Additional arguments, see below.

Details

The `read.graph` function may have additional arguments depending on the file format (the `format` argument). See the details separately for each file format, below.

Value

A graph object.

Edge list format

This format is a simple text file with numeric vertex ids defining the edges. There is no need to have newline characters between the edges, a simple space will also do.

Additional arguments:

n The number of vertices in the graph. If it is smaller than or equal to the largest integer in the file, then it is ignored; so it is safe to set it to zero (the default).

directed Logical scalar, whether to create a directed graph. The default value is `TRUE`.

Pajek format

Pajek it a popular network analysis program for Windows. (See the Pajek homepage at http://vlado.fmf.uni-lj.si/pub/networks/pajek/.) It has a quite flexible but not very well documented file format, see the Pajek manual on the Pajek homepage for some information about the file format.

`igraph` implements only a subset of the Pajek format:

- Only .net files are supported, Pajek project files (which can contain many graph and also other type of data) are not. Poject files might be supported in a forthcoming igraph release if they turned out to be needed.
- Time events networks are not supported.
- Hypergraphs (graphs with non-binary edges) are not supported as igraph cannot handle them.
- Graphs containing both directed and undirected edges are not supported as igraph cannot represent them.
- Bipartite (also called affiliation) networks are not supported. The surrent igraph version imports the network structure correctly but vertex type information is omitted.
- Graph with multiple edge sets are not supported.

Vertex and edge attributes defined in the Pajek file will be also read and assigned to the graph object to be created. These are mainly parameters for graph visualization, but not exclusively, eg. the file might contain edge weights as well.

The following vertex attributes might be added:

igraph name	description, Pajek attribute
id	Vertex id
x, y, z	The ‘x’, ‘y’ and ‘z’ coordinate of the vertex
vertexsize	The size of the vertex when plotted (`size` in Pajek).
shape	The shape of the vertex when plotted.
color	Vertex color (`ic` in Pajek) if given with symbolic name
framecolor	Border color (`bc` in Pajek) if given with symbolic name
labelcolor	Label color (`lc` in Pajek) if given with symbolic name
xfact, yfact	The `x_fact` and `y_fact` Pajek attributes.
labeldist	The distance of the label from the vertex. (`lr` in Pajek.)
labeldegree, labeldegree2	The `la` and `lphi` Pajek attributes
framewidth	The width of the border (`bw` in Pajek).
fontsize	Size of the label font (`fos` in Pajek.)
rotation	The rotation of the vertex (`phi` in Pajek).
radius	Radius, for some vertex shapes (`r` in Pajek).
diamondratio	For the diamond shape (`q` in Pajek).

These igraph attributes are only created if there is at least one vertex in the Pajek file which has the corresponding associated information. Eg. if there are vertex coordinates for at least one vertex then the ‘x’, ‘y’ and possibly also ‘z’ vertex attributes will be created. For those vertices for which the attribute is not defined, `NaN` is assigned.

The following edge attributes might be added:

igraph name	description, Pajek attribute
weight	Edge weights.

label	`l` in Pajek.
color	Edge color, if the color is given with a symbolic name, `c` in Pajek.
color-red, color-green, color-blue	Edge color if it was given in RGB notation, `c` in Pajek.
edgewidth	`w` in Pajek.
arrowsize	`s` in Pajek.
hook1, hook2	`h1` and `h2` in Pajek.
angle1, angle2	`a1` and `a2` in Pajek, Bezier curve parameters.
velocity1, velocity2	`k1` and `k2` in Pajek, Bezier curve parameter.
arrowpos	`ap` in Pajek.
labelpos	`lp` in Pajek.
labelangle, labelangle2	`lr` and `lphi` in Pajek.
labeldegree	`la` in Pajek.
fontsize	`fos` in Pajek.
arrowtype	`a` in Pajek.
linepattern	`p` in Pajek.
labelcolor	`lc` in Pajek.

There are no additional arguments for this format.

GraphML file format

GraphML is an XML-based file format (an XML application in the XML terminology) to describe graphs. It is a modern format, and can store graphs with an extensible set of vertex and edge attributes, and generalized graphs which igraph cannot handle. Thus igraph supports only a subset of the GraphML language:

- Hypergraphs are not supported.
- Nested graphs are not supported.
- Mixed graphs, ie. graphs with both directed and undirected edges are not supported. read.graph() sets the graph directed if this is the default in the GraphML file, even if all the edges are in fact undirected.

See the GraphML homepage at `http://graphml.graphdrawing.org` for more information about the GraphML format.

Additional arguments:

index If the GraphML file contains more than one graphs, this argument can be used to select the graph to read. By default the first graph is read (index 0).

GML file format

GML is a simple textual format, see `http://www.infosun.fim.uni-passau.de/Graphlet/GML/` for details.

Although all syntactically correct GML can be parsed, we implement only a subset of this format, some attributes might be ignored. Here is a list of all the differences:

- Only `node` and `edge` attributes are used, and only if they have a simple type: integer, real or string. So if an attribute is an array or a record, then it is ignored. This is also true if only some values of the attribute are complex.

- Top level attributes except for `Version` and the first `graph` attribute are completely ignored.
- Graph attributes except for `node` and `edge` are completely ignored.
- There is no maximum line length.
- There is no maximum keyword length.
- Character entities in strings are not interpreted.
- We allow `inf` (infinity) and `nan` (not a number) as a real number. This is case insensitive, so `nan`, `NaN` and `NAN` are equal.

Please contact us if you cannot live with these limitations of the GML parser.

There are not additional argument for this format.

DL file format

The DL format is a simple textual file format used by the UCINET software. See `http://www.analytictech.com/networks/dataentry.htm` for examples. All formats mentioned here is supported by igraph.

Note the specification does not mention whether the format is case sensitive or not. For igraph DL files are case sensitive, i.e. 'Larry' and 'larry' are not the same.

Additional arguments:

directed Logical scalar, whether to create a directed graph. The default is to make the graph directed.

NCOL format

This format is used by the Large Graph Layout program (`http://bioinformatics.icmb.utexas.edu/lgl`), and it is simply a symbolic weighted edge list. It is a simple text file with one edge per line. An edge is defined by two symbolic vertex names separated by whitespace. (The symbolic vertex names themselves cannot contain whitespace.) They might followed by an optional number, this will be the weight of the edge; the number can be negative and can be in scientific notation. If there is no weight specified to an edge it is assumed to be zero.

The resulting graph is always undirected. LGL cannot deal with files which contain multiple or loop edges, this is however not checked here, as igraph is happy with these.

Additional arguments:

names Logical constant, whether to add the symbolic names as vertex attributes to the graph. If TRUE the name of the vertex attribute will be 'name'.

weights Character scalar, specifies whether edge weights should be added to the graph. Possible values are and their meaning are: 'no', edge weights will not be added; 'yes', edge weights will be added, if they are not present in the file, then all edges get zero weight; 'auto', edge weights will added if they are present in the file, otherwise not. The default is 'auto'.

directed Logical constant, whether to create a directed graph. The default is undirected.

LGL file format

The `lgl` format is used by the Large Graph Layout visualization software (`http://bioinformatics.icmb.utexas.edu/lgl`), it can describe undirected optionally weighted graphs. From the LGL manual: "The second format is the LGL file format (.lgl file suffix). This is yet another graph file format that tries to be as stingy as possible with space, yet keeping the edge file in a human readable (not binary) format. The format itself is like the following:

```
 # vertex1name
vertex2name [optionalWeight]
vertex3name [optionalWeight]
```

Here, the first vertex of an edge is preceded with a pound sign '\#'. Then each vertex that shares an edge with that vertex is listed one per line on subsequent lines."

LGL cannot handle loop and multiple edges or directed graphs, but in igraph it is not an error to have multiple and loop edges.

Additional arguments:

names Logical constant, whether to add the symbolic names as vertex attributes to the graph. If TRUE the name of the vertex attribute will be 'name'.

weights Character scalar, specifies whether edge weights should be added to the graph. Possible values are and their meaning are: 'no', edge weights will not be added; 'yes', edge weights will be added, if they are not present in the file, then all edges get zero weight; 'auto', edge weights will added if they are present in the file, otherwise not. The default is 'auto'.

DIMACS file format

The DIMACS file format, more specifically the version for network flow problems, see the files at `ftp://dimacs.rutgers.edu/pub/netflow/general-info/`

This is a line-oriented text file (ASCII) format. The first character of each line defines the type of the line. If the first character is `c` the line is a comment line and it is ignored. There is one problem line (`p`) in the file, it must appear before any node and arc descriptor lines. The problem line has three fields separated by spaces: the problem type (`min`, `max` or `asn`), the number of vertices and number of edges in the graph. Exactly two node identification lines are expected (`n`), one for the source, one for the target vertex. These have two fields: the id of the vertex and the type of the vertex, either `s` (=source) or `t` (=target). Arc lines start with `a` and have three fields: the source vertex, the target vertex and the edge capacity.

Vertex ids are numbered from 1.

The source vertex is assigned to the `source`, the target vertex to the `target` graph attribute. The edge capacities are assigned to the `capacity` edge attribute.

Additional arguments:

directed Logical scalar, whether to create a directed graph. By default a directed graph is created.

GraphDB format

This is a binary format, used in the graph database for isomorphism testing (`http://amalfi.dis.unina.it/graph/`) From the graph database homepage (`http://amalfi.dis.unina.it/graph/db/doc/graphdbat-2.html`):

The graphs are stored in a compact binary format, one graph per file. The file is composed of 16 bit words, which are represented using the so-called little-endian convention, i.e. the least significant byte of the word is stored first.

Then, for each node, the file contains the list of edges coming out of the node itself. The list is represented by a word encoding its length, followed by a word for each edge, representing the destination node of the edge. Node numeration is 0-based, so the first node of the graph has index 0.

See also `graph.graphdb`.

Only unlabelled graphs are implemented.

Additional attributes:

directed Logical scalar. Whether to create a directed graph.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`write.graph`

`reciprocity` *Reciprocity of graphs*

Description

Calculates the reciprocity of a directed graph.

Usage

```
reciprocity(graph, ignore.loops = TRUE, mode = c("default", "ratio"))
```

Arguments

`graph`	The graph object.
`ignore.loops`	Logical constant, whether to ignore loop edges.
`mode`	See below.

Details

The measure of reciprocity defines the proporsion of mutual connections, in a directed graph. It is most commonly defined as the probability that the opposite counterpart of a directed edge is also included in the graph. Or in adjacency matrix notation: $\sum_{ij}(A \cdot A')_{ij}$, where $A \cdot A'$ is the element-wise product of matrix A and its transpose. This measure is calculated if the `mode` argument is `default`.

Prior to igraph version 0.6, another measure was implemented, defined as the probability of mutual connection between a vertex pair, if we know that there is a (possibly non-mutual) connection between them. In other words, (unordered) vertex pairs are classified into three groups: (1) not-connected, (2) non-reciprocaly connected, (3) reciprocally connected. The result is the size of group (3), divided by the sum of group sizes (2)+(3). This measure is calculated if `mode` is `ratio`.

Value

A numeric scalar between zero and one.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`> and Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
g <- random.graph.game(20, 5/20, directed=TRUE)
reciprocity(g)
```

---

revolver *Measuring the driving force in evolving networks*

Description

These functions assume a simple evolving network model and measure the functional form of a so-called *attractiveness function* governing the evolution of the network.

Usage

```
evolver.d (nodes, kernel, outseq = NULL, outdist = NULL, m = 1,
           directed = TRUE)

revolver.d (graph, niter=5, sd=FALSE, norm=FALSE,
           cites=FALSE, expected=FALSE, error=TRUE, debug=numeric())
revolver.ad (graph, niter=5, agebins=max(vcount(graph)/7100, 10),
            sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE, error=TRUE,
            debug=matrix(ncol=2, nrow=0))
revolver.ade (graph, cats, niter=5, agebins=max(vcount(graph)/7100, 10),
             sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
             error=TRUE, debug=matrix(ncol=2, nrow=0))
revolver.e (graph, cats, niter=5, st=FALSE,
           sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
           error=TRUE, debug=numeric())
revolver.de (graph, cats, niter=5,
            sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
            error=TRUE, debug=numeric())
revolver.l (graph, niter=5, agebins=max(vcount(graph)/7100, 10),
           sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
           error=TRUE, debug=numeric())
revolver.dl (graph, niter=5, agebins=max(vcount(graph)/7100, 10),
            sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
            error=TRUE, debug=numeric())
revolver.el (graph, cats, niter=5, agebins=max(vcount(graph)/7100, 10),
            sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
            error=TRUE, debug=numeric())
revolver.r (graph, window, niter=5, sd=FALSE, norm=FALSE,
           cites=FALSE, expected=FALSE, error=TRUE, debug=numeric())
revolver.ar (graph, window, niter=5, agebins=max(vcount(graph)/7100, 10),
            sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE, error=TRUE,
            debug=matrix(ncol=2, nrow=0))
revolver.di (graph, cats, niter=5,
            sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
            error=TRUE, debug=numeric())
revolver.adi (graph, cats, niter=5, agebins=max(vcount(graph)/7100, 10),
             sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
             error=TRUE, debug=matrix(ncol=2, nrow=0))
```

```
revolver.il (graph, cats, niter=5, agebins=max(vcount(graph)/7100, 10),
            sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
            error=TRUE, debug=numeric())
revolver.ir (graph, cats, window, niter=5,
            sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
            error=TRUE, debug=numeric())
revolver.air (graph, cats, window,
             niter=5, agebins=max(vcount(graph)/7100, 10),
             sd=FALSE, norm=FALSE, cites=FALSE, expected=FALSE,
             error=TRUE, debug=matrix(ncol=2, nrow=0))
revolver.d.d (graph, vtime = V(graph)$time, etime = E(graph)$time, niter = 5,
             sd = FALSE, norm = FALSE, cites = FALSE, expected = FALSE,
             error = TRUE, debug = matrix(ncol = 2, nrow = 0))
revolver.p.p (graph, events = get.graph.attribute(graph, "events"),
             vtime = V(graph)$time, etime = E(graph)$time, niter = 5, sd = FALSE,
             norm = FALSE, cites = FALSE, expected = FALSE, error = TRUE,
             debug = matrix(ncol = 2, nrow = 0))
revolver.error.d (graph, kernel)
revolver.error.ad (graph, kernel)
revolver.error.ade (graph, kernel, cats)
revolver.error.adi (graph, kernel, cats)
revolver.error.air (graph, kernel, cats, window)
revolver.error.ar (graph, kernel, window)
revolver.error.de (graph, kernel, cats)
revolver.error.di (graph, kernel, cats)
revolver.error.dl (graph, kernel)
revolver.error.e (graph, kernel, cats)
revolver.error.el (graph, kernel, cats)
revolver.error.il (graph, kernel, cats)
revolver.error.ir (graph, kernel, cats, window)
revolver.error.l (graph, kernel)
revolver.error.r (graph, kernel, window)

revolver.ml.ade (graph, niter, cats, agebins = 300, delta = 1e-10,
    filter = NULL)
revolver.ml.d (graph, niter, delta = 1e-10, filter = NULL)
revolver.ml.de (graph, niter, cats, delta = 1e-10, filter = NULL)
revolver.ml.df (graph, niter, delta = 1e-10)
revolver.ml.f (graph, niter, delta = 1e-10)
revolver.ml.l (graph, niter, agebins = 300, delta = 1e-10)

revolver.ml.AD.alpha.a.beta (graph, alpha, a, beta, abstol = 1e-08,
    reltol = 1e-08, maxit = 1000, agebins = 300, filter = NULL)
revolver.ml.AD.dpareto (graph, alpha, a, paralpha, parbeta, parscale,
    abstol = 1e-08, reltol = 1e-08, maxit = 1000, agebins = 300, filter
    = NULL)
revolver.ml.ADE.alpha.a.beta (graph, cats, alpha, a, beta, coeffs,
    abstol = 1e-08, reltol = 1e-08, maxit = 1000, agebins = 300, filter
    = NULL)
revolver.ml.ADE.dpareto (graph, cats, alpha, a, paralpha, parbeta, parscale,
    coeffs, abstol = 1e-08, reltol = 1e-08, maxit = 1000, agebins = 300,
    filter = NULL)
```

```
revolver.ml.D.alpha (graph, alpha, abstol = 1e-08, reltol = 1e-08, maxit
    = 1000, filter = NULL)
revolver.ml.D.alpha.a (graph, alpha, a, abstol = 1e-08, reltol = 1e-08,
    maxit = 1000, filter = NULL)
revolver.ml.DE.alpha.a (graph, cats, alpha, a, coeffs, abstol = 1e-08,
    reltol = 1e-08, maxit = 1000, filter = NULL)

revolver.ml.AD.dpareto.eval (graph, alpha, a, paralpha, parbeta,
    parscale, agebins = 300, filter = NULL)
revolver.ml.ADE.dpareto.eval (graph, cats, alpha, a, paralpha, parbeta,
    parscale, coeffs, agebins = 300, filter = NULL)
revolver.ml.ADE.dpareto.evalf (graph, cats, par, agebins, filter = NULL)

revolver.probs.ad (graph, kernel, ntk = FALSE)
revolver.probs.ade (graph, kernel, cats)
revolver.probs.d (graph, kernel, ntk = FALSE)
revolver.probs.de (graph, kernel, cats)
revolver.probs.ADE.dpareto (graph, par, cats, gcats, agebins)
```

Arguments

`nodes`	The number of vertices in the generated network.
`kernel`	The kernel function, a vector, matrix or array, depending on the number of model parameters.
`outseq`	The out-degree sequence, or `NULL` if no out-degree sequence is used.
`outdist`	The out-degree distribution, or `NULL` if all vertices have the same out-degree. This argument is ignored if the `outseq` argument is not `NULL`.
`m`	Numeric scalar, the out-degree of the verticec. It is ignored if at least one of `outseq` and `outdist` is not `NULL`.
`directed`	Logical scalar, whether to create a directed graph.
`graph`	The input graph.
`niter`	The number of iterations to perform.
`sd`	Logical scalar, whether to return the standard deviation of the estimates.
`norm`	Logical scalar, whether to return the normalizing factors.
`cites`	Logical scalar, whether to return the number of citations to the different vertex types.
`expected`	Logical scalar, whether to return the expected number of citations for the different vertex types.
`error`	Logical scalar, whether to return the error of the fit.
`debug`	Currently not used.
`agebins`	The number of bins for vertex age.
`cats`	The number of categories to use.
`window`	The width of the time window to use, measured in number of vertices.
`vtime`	Numeric vector, the time steps when the vertices where added to the network.
`etime`	Numeric vector, the time steps when the edges where added to the network.
`events`	A list of numeric vectors, each vector represents an event, with the participation of the listed vertices.

`delta`	Real scalar, the error margin that is allowed for the convergence.
`filter`	Logical vector, length is the number of vertices. Only vertices corresponding to `TRUE` entries are used in the fitting.
`alpha`	Starting value for the '`alpha`' parameter.
`a`	Starting value for the '`a`' parameter.
`paralpha`	Starting value for the '`paralpha`' (Pareto alpha) parameter.
`parbeta`	Starting value for the '`parbeta`' (Pareto beta) parameter.
`parscale`	Starting value for the '`parscale`' (Pareto scale) parameter.
`abstol`	Real scalar, absolute tolerance for the ML fitting.
`reltol`	Real scalar, relative tolerance for the ML fitting.
`maxit`	Numeric scalar, the maximum number of iterations.
`beta`	Real scalar, starting value for the '`beta`' parameter.
`coeffs`	Numeric vector, starting values for the coefficients.
`par`	Pareto parameters for the different vertex types, in a matrix.
`ntk`	Logical scalar, whether to return the Ntk values.
`gcats`	Numeric vector, the vertex types.
`st`	Logical scalar, whether to return the S(t) values.

Details

The functions should be considered as experimental, so no detailed documentation yet. Sorry.

Value

A named list.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

`rewire` *Graph rewiring*

Description

Randomly rewires a graph while preserving the degree distribution.

Usage

```
rewire(graph, mode = "simple", niter = 100)
```

Arguments

`graph`	The graph to be rewired.
`mode`	The rewiring algorithm to be used. It can be one of the following: `simple`: simple rewiring algorithm which chooses two arbitrary edges in each step (namely (a,b) and (c,d)) and substitutes them with (a,d) and (c,b) if they don't yet exist.
`niter`	Number of rewiring trials to perform.

Details

This function generates a new graph based on the original one by randomly rewiring edges while preserving the original graph's degree distribution.

Value

A new graph object.

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com>

See Also

degree.sequence.game

Examples

```
g <- graph.ring(20)
g2 <- rewire(g, niter=3)
```

---

rewire.edges *Rewires the endpoints of the edges of a graph randomly*

Description

This function rewires the endpoints of the edges with a constant probability uniformly randomly to a new vertex in a graph.

Usage

```
rewire.edges(graph, prob, loops=FALSE, multiple=FALSE)
```

Arguments

graph	The input graph
prob	The rewiring probability, a real number between zero and one.
loops	Logical scalar, whether loop edges are allowed in the rewired graph.
multiple	Logical scalar, whether multiple edges are allowed int the generated graph.

Details

Note that this function might create graphs with multiple and/or loop edges.

Value

A new graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
# Some random shortcuts shorten the distances on a lattice
g <- graph.lattice( length=100, dim=1, nei=5 )
average.path.length(g)
g <- rewire.edges( g, prob=0.05 )
average.path.length(g)
```

`rglplot` *3D plotting of graphs with OpenGL*

Description

Using the `rgl` package, `rglplot` plots a graph in 3D. The plot can be zoomed, rotated, shifted, etc. but the coordinates of the vertices is fixed.

Usage

```
rglplot(x, ...)
```

Arguments

`x`	The graph to plot.
`...`	Additional arguments, see `igraph.plotting` fo the details

Details

Note that `rglplot` is considered to be highly experimental. It is not very useful either. See `igraph.plotting` for the possible arguments.

Value

`NULL`, invisibly.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`igraph.plotting`, `plot.igraph` for the 2D version, `tkplot` for interactive graph drawing in 2D.

Examples

```
## Not run:
g <- graph.lattice( c(5,5,5) )
coords <- layout.fruchterman.reingold(g, dim=3)
rglplot(g, layout=coords)

## End(Not run)
```

running.mean — *Running mean of a time series*

Description

running.mean calculates the running mean in a vector with the given bin width.

Usage

```
running.mean(v, binwidth)
```

Arguments

v	The numeric vector.
binwidth	Numeric constant, the size of the bin, should be meaningful, ie. smaller than the length of v.

Details

The running mean of v is a w vector of length length(v)-binwidth+1. The first element of w id the average of the first binwidth elements of v, the second element of w is the average of elements 2:(binwidth+1), etc.

Value

A numeric vector of length length(v)-binwidth+1

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
running.mean(1:100, 10)
```

scg — *All-in-one Function for the SCG of Matrices and Graphs*

Description

This function handles all the steps involved in the Spectral Coarse Graining (SCG) of some matrices and graphs as described in the reference below.

Usage

```
scg(X, ev, nt, groups = NULL, mtype = c("symmetric", "laplacian",
      "stochastic"), algo = c("optimum", "interv_km", "interv",
      "exact_scg"), norm = c("row", "col"), direction = c("default",
      "left", "right"), evec = NULL, p = NULL, use.arpack = FALSE,
    maxiter = 300, sparse = getIgraphOpt("sparsematrices"), output =
    c("default", "matrix", "graph"), semproj = FALSE, epairs = FALSE,
    stat.prob = FALSE)
```

Arguments

`X`	The input graph or square matrix. Can be of class `igraph`, `matrix` or `Matrix`.
`ev`	A vector of positive integers giving the indexes of the eigenpairs to be preserved. For real eigenpairs, 1 designates the eigenvalue with largest algebraic value, 2 the one with second largest algebraic value, etc. In the complex case, it is the magnitude that matters.
`nt`	A vector of positive integers of length one or equal to `length(ev)`. When `algo` = "optimum", `nt` contains the number of groups used to partition each eigenvector separately. When `algo` is equal to "interv_km" or "interv", `nt` contains the number of intervals used to partition each eigenvector. The same partition size or number of intervals is used for each eigenvector if `nt` is a single integer. When `algo` = "exact_cg" this parameter is ignored.
`groups`	A vector of `nrow(X)` or `vcount(X)` integers labeling each group vertex in the partition. If this parameter is supplied most part of the function is bypassed.
`mtype`	Character scalar. The type of semi-projector to be used for the SCG. For now "symmetric", "laplacian" and "stochastic" are available.
`algo`	Character scalar. The algorithm used to solve the SCG problem. Possible values are "optimum", "interv_km", "interv" and "exact_scg".
`norm`	Character scalar. Either "row" or "col". If set to "row" the rows of the Laplacian matrix sum up to zero and the rows of the stochastic matrix sum up to one; otherwise it is the columns.
`direction`	Character scalar. When set to "right", resp. "left", the parameters `ev` and `evec` refer to right, resp. left eigenvectors. When passed "default" it is the SCG described in the reference below that is applied (common usage). This argument is currently not implemented, and right eigenvectors are always used.
`evec`	A numeric matrix of (eigen)vectors to be preserved by the coarse graining (the vectors are to be stored column-wise in `evec`). If supplied, the eigenvectors should correspond to the indexes in `ev` as no cross-check will be done.
`p`	A probability vector of length `nrow(X)` (or `vcount(X)`). `p` is the stationary probability distribution of a Markov chain when `mtype` = "stochastic". This parameter is ignored in all other cases.
`use.arpack`	Logical scalar. When set to `TRUE` uses the function `arpack` to compute eigenpairs. This parameter should be set to `TRUE` if one deals with large (over a few thousands) AND sparse graphs or matrices. This argument is not implemented currently and LAPACK is used for solving the eigenproblems.
`maxiter`	A positive integer giving the maximum number of iterations for the k-means algorithm when `algo` = "interv_km". This parameter is ignored in all other cases.
`sparse`	Logical scalar. Whether to return sparse matrices in the result, if matrices are requested.
`output`	Character scalar. Set this parameter to "default" to retrieve a coarse-grained object of the same class as `X`.
`semproj`	Logical scalar. Set this parameter to `TRUE` to retrieve the semi-projectors of the SCG.
`epairs`	Logical scalar. Set this to `TRUE` to collect the eigenpairs computed by `scg`.
`stat.prob`	Logical scalar. This is to collect the stationary probability `p` when dealing with stochastic matrices.

Details

Please see SCG for an introduction.

In the following V is the matrix of eigenvectors for which the SCG is solved. V is calculated from `X`, if it is not given in the `evec` argument.

The algorithm "optimum" solves exactly the SCG problem for each eigenvector in `V`. The running time of this algorithm is $O(\max nt \cdot m^2)$ for the symmetric and laplacian matrix problems (i.e. when `mtype` is "symmetric" or "laplacian". It is $O(m^3)$ for the stochastic problem. Here m is the number of rows in `V`. In all three cases, the memory usage is $O(m^2)$.

The algorithms "interv" and "interv_km" solve approximately the SCG problem by performing a (for now) constant binning of the components of the eigenvectors, that is `nt[i]` constant-size bins are used to partition `V[,i]`. When `algo` = "interv_km", the (Lloyd) k-means algorithm is run on each partition obtained by "interv" to improve accuracy.

Once a minimizing partition (either exact or approximate) has been found for each eigenvector, the final grouping is worked out as follows: two vertices are grouped together in the final partition if they are grouped together in each minimizing partition. In general the size of the final partition is not known in advance when `ncol(V)`>1.

Finally, the algorithm "exact_scg" groups the vertices with equal components in each eigenvector. The last three algorithms essentially have linear running time and memory load.

Value

`Xt`	The coarse-grained graph, or matrix, possibly a sparse matrix.
`groups`	A vector of `nrow(X)` or `vcount(X)` integers giving the group label of each object (vertex) in the partition.
`L`	The semi-projector L if `semproj = TRUE`.
`R`	The semi-projector R if `semproj = TRUE`.
`values`	The computed eigenvalues if `epairs = TRUE`.
`vectors`	The computed or supplied eigenvectors if `epairs = TRUE`.
`p`	The stationary probability vector if `mtype = stochastic` and `stat.prob = TRUE`. For other matrix types this is missing.

Author(s)

David Morton de Lachapelle <`david.morton@epfl.ch`>, <`david.mortondelachapelle@swissquote.ch`>

References

D. Morton de Lachapelle, D. Gfeller, and P. De Los Rios, Shrinking Matrices while Preserving their Eigenpairs with Application to the Spectral Coarse Graining of Graphs. Submitted to *SIAM Journal on Matrix Analysis and Applications*, 2008. `http://people.epfl.ch/david.morton`

See Also

SCG for an introduction. `scgNormEps`, `scgGrouping` and `scgSemiProjectors`.

Examples

```
# SCG of a toy network
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(1,6, 1,11, 6, 11))
cg <- scg(g, 1, 3, algo="exact_scg")

#plot the result
layout <- layout.kamada.kawai(g)
nt <- vcount(cg$Xt)
col <- rainbow(nt)
vsize <- table(cg$groups)
ewidth <- round(E(cg$Xt)$weight,2)
## Not run:
op <- par(mfrow=c(1,2))
plot(g, vertex.color = col[cg$groups], vertex.size = 20,
vertex.label = NA, layout = layout)
plot(cg$Xt, edge.width = ewidth, edge.label = ewidth,
vertex.color = col, vertex.size = 20*vsize/max(vsize),
vertex.label=NA, layout = layout.kamada.kawai)
par(op)

## End(Not run)

## SCG of real-world network
library(igraphdata)
data(immuno)
summary(immuno)
n <- vcount(immuno)
interv <- c(100,100,50,25,12,6,3,2,2)
cg <- scg(immuno, ev= n-(1:9), nt=interv, mtype="laplacian",
                        algo="interv", epairs=TRUE)

## are the eigenvalues well-preserved?
gt <- cg$Xt
nt <- vcount(gt)
Lt <- graph.laplacian(gt)
evalt <- eigen(Lt, only.values=TRUE)$values[nt-(1:9)]
res <- cbind(interv, cg$values, evalt)
res <- round(res,5)
colnames(res) <- c("interv","lambda_i","lambda_tilde_i")
rownames(res) <- c("N-1","N-2","N-3","N-4","N-5","N-6","N-7","N-8","N-9")
print(res)

## use SCG to get the communities
com <- scg(graph.laplacian(immuno), ev=n-c(1,2), nt=2)$groups
col <- rainbow(max(com))
layout <- layout.auto(immuno)
## Not run:
plot(immuno, layout=layout, vertex.size=3, vertex.color=col[com],
                vertex.label=NA)

## End(Not run)

## display the coarse-grained graph
gt <- simplify(as.undirected(gt))
layout.cg <- layout.kamada.kawai(gt)
```

```
com.cg <- scg(graph.laplacian(gt), nt-c(1,2), 2)$groups
vsize <- sqrt(as.vector(table(cg$groups)))
## Not run:
op <- par(mfrow=c(1,2))
plot(immuno, layout=layout, vertex.size=3, vertex.color=col[com],
              vertex.label=NA)
plot(gt, layout=layout.cg, vertex.size=15*vsize/max(vsize),
              vertex.color=col[com.cg],vertex.label=NA)
par(op)

## End(Not run)
```

---

scgExtra *SCG Extra Functions*

Description

Some useful functions to perform general actions in Spectral Coarse Graining (SCG).

Usage

```
scgNormEps(V, groups, mtype = c("symmetric", "laplacian",
          "stochastic"), p = NULL, norm = c("row", "col"))
```

Arguments

V	A numeric matrix of (eigen)vectors assumed normalized. The vectors are to be stored column-wise in V).
groups	A vector of nrow(V) integers labeling each group vertex in the partition.
mtype	The type of semi-projector used for the SCG. For now “symmetric”, “laplacian” and “stochastic” are available.
p	A probability vector of length nrow(V). p is the stationary probability distribution of a Markov chain when mtype = “stochastic”. This parameter is ignored otherwise.
norm	Either “row” or “col”. If set to “row” the rows of the Laplacian matrix sum to zero and the rows of the stochastic matrix sum to one; otherwise it is the columns.

Details

scgNormEps computes $\|v_i - Pv_i\|$, where v_i is the ith eigenvector in V and P is the projector corresponding to the mtype argument.

Value

normEps returns with a numeric vector whose ith component is $\|v_i - Pv_i\|$ (see Details).

Author(s)

David Morton de Lachapelle <david.morton@epfl.ch>, <david.mortondelachapelle@swissquote.ch>

References

D. Morton de Lachapelle, D. Gfeller, and P. De Los Rios, Shrinking Matrices while Preserving their Eigenpairs with Application to the Spectral Coarse Graining of Graphs. Submitted to *SIAM Journal on Matrix Analysis and Applications*, 2008. `http://people.epfl.ch/david.morton`

See Also

SCG and `scg`.

Examples

```
v <- rexp(20)
km <- kmeans(v,5)
sum(km$withinss)
scgNormEps(cbind(v), km$cluster)^2
```

`scgGrouping` *SCG Problem Solver*

Description

This function solves the Spectral Coarse Graining (SCG) problem; either exactly, or approximately but faster.

Usage

```
scgGrouping(V, nt, mtype = c("symmetric", "laplacian",
           "stochastic"), algo = c("optimum", "interv_km",
           "interv","exact_scg"), p = NULL, maxiter = 100)
```

Arguments

`V`	A numeric matrix of (eigen)vectors to be preserved by the coarse graining (the vectors are to be stored column-wise in `V`).
`nt`	A vector of positive integers of length one or equal to `length(ev)`. When `algo` = “optimum”, `nt` contains the number of groups used to partition each eigenvector separately. When `algo` is equal to “interv_km” or “interv”, `nt` contains the number of intervals used to partition each eigenvector. The same partition size or number of intervals is used for each eigenvector if `nt` is a single integer. When `algo` = “exact_cg” this parameter is ignored.
`mtype`	The type of semi-projectors used in the SCG. For now “symmetric”, “laplacian” and “stochastic” are available.
`algo`	The algorithm used to solve the SCG problem. Possible values are “optimum”, “interv_km”, “interv” and “exact_scg”.
`p`	A probability vector of length equal to `nrow(V)`. `p` is the stationary probability distribution of a Markov chain when `mtype` = “stochastic”. This parameter is ignored in all other cases.
`maxiter`	A positive integer giving the maximum number of iterations of the k-means algorithm when `algo` = “interv_km”. This parameter is ignored in all other cases.

Details

The algorithm "optimum" solves exactly the SCG problem for each eigenvector in `V`. The running time of this algorithm is $O(\max nt \cdot m^2)$ for the symmetric and laplacian matrix problems (i.e. when `mtype` is "symmetric" or "laplacian". It is $O(m^3)$ for the stochastic problem. Here m is the number of rows in `V`. In all three cases, the memory usage is $O(m^2)$.

The algorithms "interv" and "interv_km" solve approximately the SCG problem by performing a (for now) constant binning of the components of the eigenvectors, that is `nt[i]` constant-size bins are used to partition `V[,i]`. When `algo` = "interv_km", the (Lloyd) k-means algorithm is run on each partition obtained by "interv" to improve accuracy.

Once a minimizing partition (either exact or approximate) has been found for each eigenvector, the final grouping is worked out as follows: two vertices are grouped together in the final partition if they are grouped together in each minimizing partition. In general the size of the final partition is not known in advance when `ncol(V)`>1.

Finally, the algorithm "exact_scg" groups the vertices with equal components in each eigenvector. The last three algorithms essentially have linear running time and memory load.

Value

A vector of `nrow(V)` integers giving the group label of each object (vertex) in the partition.

Author(s)

David Morton de Lachapelle <`david.morton@epfl.ch`>, <`david.mortondelachapelle@swissquote.ch`>

References

D. Morton de Lachapelle, D. Gfeller, and P. De Los Rios, Shrinking Matrices while Preserving their Eigenpairs with Application to the Spectral Coarse Graining of Graphs. Submitted to *SIAM Journal on Matrix Analysis and Applications*, 2008. `http://people.epfl.ch/david.morton`

See Also

SCG for a detailed introduction. `scg`, `scgNormEps`

Examples

```
# eigenvectors of a random symmetric matrix
M <- matrix(rexp(10^6), 10^3, 10^3)
M <- (M + t(M))/2
V <- eigen(M, symmetric=TRUE)$vectors[,c(1,2)]

# displays size of the groups in the final partition
gr <- scgGrouping(V, nt=c(2,3))
col <- rainbow(max(gr))
plot(table(gr), col=col, main="Group size", xlab="group", ylab="size")

## comparison with the grouping obtained by kmeans
## for a partition of same size
gr.km <- kmeans(V,centers=max(gr), iter.max=100, nstart=100)$cluster
op <- par(mfrow=c(1,2))
plot(V[,1], V[,2], col=col[gr],
main = "SCG grouping",
xlab = "1st eigenvector",
ylab = "2nd eigenvector")
```

```
plot(V[,1], V[,2], col=col[gr.km],
main = "K-means grouping",
xlab = "1st eigenvector",
ylab = "2nd eigenvector")
par(op)
## kmeans disregards the first eigenvector as it
## spreads a much smaller range of values than the second one

### comparing optimal and k-means solutions
### in the one-dimensional case.
x <- rexp(2000, 2)
gr.true <- scgGrouping(cbind(x), 100)
gr.km <- kmeans(x, 100, 100, 300)$cluster
scgNormEps(cbind(x), gr.true)
scgNormEps(cbind(x), gr.km)
```

---

scgSemiProjectors *Semi-Projectors*

Description

A function to compute the L and R semi-projectors for a given partition of the vertices.

Usage

```
scgSemiProjectors(groups, mtype = c("symmetric", "laplacian",
                      "stochastic"), p = NULL, norm = c("row", "col"),
                    sparse = getIgraphOpt("sparsematrices"))
```

Arguments

groups	A vector of `nrow(X)` or `vcount(X)` integers giving the group label of every vertex in the partition.
mtype	The type of semi-projectors. For now “symmetric”, “laplacian” and “stochastic” are available.
p	A probability vector of length `length(gr)`. `p` is the stationary probability distribution of a Markov chain when `mtype` = “stochastic”. This parameter is ignored in all other cases.
norm	Either “row” or “col”. If set to “row” the rows of the Laplacian matrix sum up to zero and the rows of the stochastic sum up to one; otherwise it is the columns.
sparse	Logical scalar, whether to return sparse matrices.

Details

The three types of semi-projectors are defined as follows. Let $\gamma(j)$ label the group of vertex j in a partition of all the vertices.

The symmetric semi-projectors are defined as

$$L_{\alpha j} = R_{\alpha j} = \frac{1}{\sqrt{|\alpha|}}\delta_{\alpha\gamma(j)},$$

the (row) Laplacian semi-projectors as

$$L_{\alpha j} = \frac{1}{|\alpha|}\delta_{\alpha\gamma(j)} \quad \text{and} \quad R_{\alpha j} = \delta_{\alpha\gamma(j)},$$

and the (row) stochastic semi-projectors as

$$L_{\alpha j} = \frac{p_1(j)}{\sum_{k\in\gamma(j)} p_1(k)} \quad \text{and} \quad R_{\alpha j} = \delta_{\alpha\gamma(j)\delta_{\alpha\gamma(j)}},$$

where p_1 is the (left) eigenvector associated with the one-eigenvalue of the stochastic matrix. L and R are defined in a symmetric way when `norm = col`. All these semi-projectors verify various properties described in the reference.

Value

`L`	The semi-projector L.
`R`	The semi-projector R.

Author(s)

David Morton de Lachapelle <`david.morton@epfl.ch`>, <`david.mortondelachapelle@swissquote.ch`>

References

D. Morton de Lachapelle, D. Gfeller, and P. De Los Rios, Shrinking Matrices while Preserving their Eigenpairs with Application to the Spectral Coarse Graining of Graphs. Submitted to *SIAM Journal on Matrix Analysis and Applications*, 2008. `http://people.epfl.ch/david.morton`

See Also

SCG for a detailed introduction. `scg`, `scgNormEps`, `scgGrouping`

Examples

```
# compute the semi-projectors and projector for the partition
# provided by a community detection method
g <- barabasi.game(20, m=1.5)
eb <- edge.betweenness.community(g)
memb <- membership(eb)
lr <- scgSemiProjectors(memb)
#In the symmetric case L = R
tcrossprod(lr$R)  # same as lr$R %*% t(lr$R)
P <- crossprod(lr$R)  # same as t(lr$R) %*% lr$R
#P is an orthogonal projector
isSymmetric(P)
sum( (P %*% P-P)^2 )

## use L and R to coarse-grain the graph Laplacian
lr <- scgSemiProjectors(memb, mtype="laplacian")
L <- graph.laplacian(g)
Lt <- lr$L %*% L %*% t(lr$R)
## or better lr$L %*% tcrossprod(L,lr$R)
rowSums(Lt)
```

shortest.paths *Shortest (directed or undirected) paths between vertices*

Description

`shortest.paths` calculates the length of all the shortest paths from or to the vertices in the network. `get.shortest.paths` calculates one shortest path (the path itself, and not just its length) from or to the given vertex.

Usage

```
shortest.paths(graph, v=V(graph), to=V(graph),
      mode = c("all", "out", "in"),
      weights = NULL, algorithm = c("automatic", "unweighted",
                                    "dijkstra", "bellman-ford",
                                    "johnson"))
get.shortest.paths(graph, from, to=V(graph), mode = c("out", "all",
      "in"), weights = NULL, output=c("vpath", "epath", "both"))
get.all.shortest.paths(graph, from, to = V(graph), mode = c("out",
      "all", "in"), weights=NULL)
average.path.length(graph, directed=TRUE, unconnected=TRUE)
path.length.hist (graph, directed = TRUE)
```

Arguments

`graph`	The graph to work on.
`v`	Numeric vector, the vertices from which the shortest paths will be calculated.
`to`	Numeric vector, the vertices to which the shortest paths will be calculated. By default it includes all vertices. Note that for `shortest.paths` every vertex must be included here at most once. (This is not required for `get.shortest.paths`.
`mode`	Character constant, gives whether the shortest paths to or from the given vertices should be calculated for directed graphs. If `out` then the shortest paths *from* the vertex, if `in` then *to* it will be considered. If `all`, the default, then the corresponding undirected graph will be used, ie. not directed paths are searched. This argument is ignored for undirected graphs.
`weights`	Possibly a numeric vector giving edge weights. If this is `NULL` and the graph has a `weight` edge attribute, then the attribute is used. If this is `NA` then no weights are used (even if the graph has a `weight` attribute).
`algorithm`	Which algorithm to use for the calculation. By default igraph tries to select the fastest suitable algorithm. If there are no weights, then an unweighted breadth-first search is used, otherwise if all weights are positive, then Dijkstra's algorithm is used. If there are negative weights and we do the calculation for more than 100 sources, then Johnson's algorithm is used. Otherwise the Bellman-Ford algorithm is used. You can override igraph's choice by explicitly giving this parameter. Note that the igraph C core might still override your choice in obvious cases, i.e. if there are no edge weights, then the unweighted algorithm will be used, regardless of this argument.
`from`	Numeric constant, the vertex from or to the shortest paths will be calculated. Note that right now this is not a vector of vertex ids, but only a single vertex.

output	Character scalar, defines how to report the shortest paths. "vpath" means that the vertices along the paths are reported, this form was used prior to igraph version 0.6. "epath" means that the edges along the paths are reported. "both" means that both forms are returned, in a named list with components "vpath" and "epath".
directed	Whether to consider directed paths in directed graphs, this argument is ignored for undirected graphs.
unconnected	What to do if the graph is unconnected (not strongly connected if directed paths are considered). If TRUE only the lengths of the existing paths are considered and averaged; if FALSE the length of the missing paths are counted having length vcount(graph), one longer than the longest possible geodesic in the network.

Details

The shortest path, or geodesic between two pair of vertices is a path with the minimal number of vertices. The functions documented in this manual page all calculate shortest paths between vertex pairs.

shortest.paths calculates the lengths of pairwise shortest paths from a set of vertices (from) to another set of vertices (to). It uses different algorithms, depending on the argorithm argument and the weight edge attribute of the graph. The implemented algorithms are breadth-first search ('unweighted'), this only works for unweighted graphs; the Dijkstra algorithm ('dijkstra'), this works for graphs with non-negative edge weights; the Bellman-Ford algorithm ('bellman-ford'), and Johnson's algorithm ('"johnson"'). The latter two algorithms work with arbitrary edge weights, but (naturally) only for graphs that don't have a negative cycle.

igraph can choose automatically between algorithms, and chooses the most efficient one that is appropriate for the supplied weights (if any). For automatic algorithm selection, supply 'automatic' as the algorithm argument. (This is also the default.)

get.shortest.paths calculates a single shortest path (i.e. the path itself, not just its length) between the source vertex given in from, to the target vertices given in to. get.shortest.paths uses breadth-first search for unweighted graphs and Dijkstra's algorithm for weighted graphs. The latter only works if the edge weights are non-negative.

get.all.shortest.paths calculates *all* shortest paths between pairs of vertices. More precisely, between the from vertex to the vertices given in to. It uses a breadth-first search for unweighted graphs and Dijkstra's algorithm for weighted ones. The latter only supports non-negative edge weights.

average.path.length calculates the average path length in a graph, by calculating the shortest paths between all pairs of vertices (both ways for directed graphs). This function does not consider edge weights currently and uses a breadth-first search.

path.length.hist calculates a histogram, by calculating the shortest path length between each pair of vertices. For directed graphs both directions are considered, so every pair of vertices appears twice in the histogram.

Value

For shortest.paths a numeric matrix with length(to) columns and length(v) rows. The shortest path length from a vertex to itself is always zero. For unreachable vertices Inf is included.

For get.shortest.paths the return value depends on the output parameter. If this is "vpath", then a list of length vcount(graph) is returned. List element i contains the vertex ids on the path from vertex from to vertex i (or the other way for directed graphs depending on the mode argument).

The vector also contains `from` and `i` as the first and last elements. If `from` is the same as `i` then it is only included once. If there is no path between two vertices then a numeric vector of length zero is returned as the list element.

If `output` is “epath”, then a similar list is returned, but the vectors in the list contain the edge ids along the shortest paths, instead of the vertex ids.

If `output` is “both”, then both lists are returned, in a named list with entries named as “vpath” and “epath”.

For `get.all.shortest.paths` a list is returned, each list element contains a shortest path from `from` to a vertex in `to`. The shortest paths to the same vertex are collected into consecutive elements of the list.

For `average.path.length` a single number is returned.

`path.length.hist` returns a named list with two entries: `res` is a numeric vector, the histogram of distances, `unconnected` is a numeric scalar, the number of pairs for which the first vertex is not reachable from the second. The sum of the two entries is always $n(n-1)$ for directed graphs and $n(n-1)/2$ for undirected graphs.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

West, D.B. (1996). *Introduction to Graph Theory.* Upper Saddle River, N.J.: Prentice Hall.

Examples

```
g <- graph.ring(10)
shortest.paths(g)
get.shortest.paths(g, 5)
get.all.shortest.paths(g, 1, 6:8)
average.path.length(g)
## Weighted shortest paths
el <- matrix(nc=3, byrow=TRUE,
             c(1,2,0, 1,3,2, 1,4,1, 2,3,0, 2,5,5, 2,6,2, 3,2,1, 3,4,1,
               3,7,1, 4,3,0, 4,7,2, 5,6,2, 5,8,8, 6,3,2, 6,7,1, 6,9,1,
               6,10,3, 8,6,1, 8,9,1, 9,10,4) )
g2 <- add.edges(graph.empty(10), t(el[,1:2]), weight=el[,3])
shortest.paths(g2, mode="out")
```

`similarity` *Similarity measures of two vertices*

Description

These functions calculates similarity scores for vertices based on their connection patterns.

Usage

```
similarity.jaccard(graph, vids = V(graph), mode = c("all", "out", "in",
      "total"), loops = FALSE)
similarity.dice(graph, vids = V(graph), mode = c("all", "out", "in",
      "total"), loops = FALSE)
similarity.invlogweighted(graph, vids = V(graph),
       mode = c("all", "out", "in", "total"))
```

Arguments

graph	The input graph.
vids	The vertex ids for which the similarity is calculated.
mode	The type of neighboring vertices to use for the calculation, possible values: 'out', 'in', 'all'.
loops	Whether to include vertices themselves in the neighbor sets.

Details

The Jaccard similarity coefficient of two vertices is the number of common neighbors divided by the number of vertices that are neighbors of at least one of the two vertices being considered. similarity.jaccard calculates the pairwise Jaccard similarities for some (or all) of the vertices.

The Dice similarity coefficient of two vertices is twice the number of common neighbors divided by the sum of the degrees of the vertices. similarity.dice calculates the pairwise Dice similarities for some (or all) of the vertices.

The inverse log-weighted similarity of two vertices is the number of their common neighbors, weighted by the inverse logarithm of their degrees. It is based on the assumption that two vertices should be considered more similar if they share a low-degree common neighbor, since high-degree common neighbors are more likely to appear even by pure chance. Isolated vertices will have zero similarity to any other vertex. Self-similarities are not calculated. See the following paper for more details: Lada A. Adamic and Eytan Adar: Friends and neighbors on the Web. Social Networks, 25(3):211-230, 2003.

Value

A length(vids) by length(vids) numeric matrix containing the similarity scores.

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com> for the manual page.

References

Lada A. Adamic and Eytan Adar: Friends and neighbors on the Web. *Social Networks*, 25(3):211-230, 2003.

See Also

cocitation and bibcoupling

Examples

```
g <- graph.ring(5)
similarity.dice(g)
similarity.jaccard(g)
```

---

simplify *Simple graphs*

Description

Simple graphs are graphs which do not contain loop and multiple edges.

Usage

```
simplify(graph, remove.multiple = TRUE, remove.loops = TRUE,
         edge.attr.comb = getIgraphOpt("edge.attr.comb"))
is.simple(graph)
```

Arguments

`graph`	The graph to work on.
`remove.loops`	Logical, whether the loop edges are to be removed.
`remove.multiple`	Logical, whether the multiple edges are to be removed.
`edge.attr.comb`	Specifies what to do with edge attributes, if `remove.multiple=TRUE`. In this case many edges might be mapped to a single one in the new graph, and their attributes are combined. Please see `attribute.combination` for details on this.

Details

A loop edge is an edge for which the two endpoints are the same vertex. Two edges are multiple edges if they have exactly the same two endpoints (for directed graphs order does matter). A graph is simple is it does not contain loop edges and multiple edges.

`is.simple` checks whether a graph is simple.

`simplify` removes the loop and/or multiple edges from a graph. If both `remove.loops` and `remove.multiple` are `TRUE` the function returns a simple graph.

Value

A new graph object with the edges deleted.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`is.loop`, `is.multiple` and `count.multiple`, `delete.edges`, `delete.vertices`

Examples

```
g <- graph( c(1,2,1,2,3,3) )
is.simple(g)
is.simple(simplify(g, remove.loops=FALSE))
is.simple(simplify(g, remove.multiple=FALSE))
is.simple(simplify(g))
```

`Spectral coarse graining` *Spectral Coarse Graining*

Description

Functions to perform the Spectral Coarse Graining (SCG) of matrices and graphs.

Introduction

The SCG functions provide a framework, called Spectral Coarse Graining (SCG), for reducing large graphs while preserving their *spectral-related features*, that is features closely related with the eigenvalues and eigenvectors of a graph matrix (which for now can be the adjacency, the stochastic, or the Laplacian matrix).

Common examples of such features comprise the first-passage-time of random walkers on Markovian graphs, thermodynamic properties of lattice models in statistical physics (e.g. Ising model), and the epidemic threshold of epidemic network models (SIR and SIS models).

SCG differs from traditional clustering schemes by producing a *coarse-grained graph* (not just a partition of the vertices), representative of the original one. As shown in [1], Principal Component Analysis can be viewed as a particular SCG, called *exact SCG*, where the matrix to be coarse-grained is the covariance matrix of some data set.

SCG should be of interest to practitioners of various fields dealing with problems where matrix eigenpairs play an important role, as for instance is the case of dynamical processes on networks.

SCG in brief

The main idea of SCG is to operate on a matrix a shrinkage operation specifically designed to preserve some of the matrix eigenpairs while not altering other important matrix features (such as its structure). Mathematically, this idea was expressed as follows. Consider a (complex) $n \times n$ matrix M and form the product

$$\widetilde{M} = LMR^*,$$

where $\tilde{n} < n$ and $L, R \in \mathbf{C}^{\tilde{n}\times n}$ are such that $LR^* = I_{\tilde{n}}$ (R^* denotes the conjugate transpose of R). Under these assumptions, it can be shown that $P = R^*L$ is an $\tilde{n}$-rank projector and that, if (λ, v) is a (right) eigenpair of M (i.e. $Mv = \lambda v$) and P is orthogonal, there exists an eigenvalue $\tilde{\lambda}$ of $\widetilde{M}$ such that

$$|\lambda - \tilde{\lambda}| \leq \text{const}\|e_P(v)\|[1 + O(\|e_P(v)\|^2)],$$

where $\|e_P(v)\| = \|v - Pv\|$. Hence, if P (or equivalently L, R) is chosen so as to make $\|e_P(v)\|$ as small as possible, one can preserve to any desired level the original eigenvalue λ in the coarse-grained matrix $\widetilde{M}$; under extra assumptions on M, this result can be generalized to eigenvectors [1]. This leads to the following generic definition of a SCG problem.

Given $M \in \mathbf{C}^{n\times n}$ and (λ, v) a (right) eigenpair of M to be preserved by the coarse graining, the problem is to find a projector $\widehat{P}$ solving

$$\min_{P\in\Omega} \|e_P(v)\|,$$

where Ω is a set of projectors in $\mathbf{C}^{n\times n}$ described by some ad hoc constraints $c_1, \ldots c_r$ (e.g. $c_1 : P \in \mathbf{R}^{n\times n}, c_2 : P = P^t, c_3 : P_{ij} \geq 0$, etc).

Choosing pertinent constraints to solve the SCG problem is of great importance in applications. For instance, in the absence of constraints the SCG problem is solved trivially by $\widehat{P} = vv^*$ (v is assumed normalized). We have designed a particular constraint, called *homogeneous mixing*, which ensures that vertices belonging to the same group are merged consistently from a physical point of view (see [1] for details). Under this constraint the SCG problem reduces to finding the partition of $\{1, \ldots, n\}$ (labeling the original vertices) minimizing

$$\|e_P(v)\|^2 = \sum_{\alpha=1}^{\tilde{n}} \sum_{i\in\alpha} [v(i) - (Pv)(i)]^2,$$

where α denotes a group (i.e. a block) in a partition of $\{1, \ldots, n\}$, and $|\alpha|$ is the number of elements in α.

If M is symmetric or stochastic, for instance, then it may be desirable (or mandatory) to choose L, R so that $\widetilde{M}$ is symmetric or stochastic as well. This *structural constraint* has led to the construction of particular semi-projectors for symmetric [1], stochastic [3] and Laplacian [2] matrices, that are made available.

In short, the coarse graining of matrices and graphs involves:

1. Retrieving a matrix or a graph matrix M from the problem.
2. Computing the eigenpairs of M to be preserved in the coarse-grained graph or matrix.
3. Setting some problem-specific constraints (e.g. dimension of the coarse-grained object).
4. Solving the constrained SCG problem, that is finding $\widehat{P}$.
5. Computing from $\widehat{P}$ two semi-projectors $\widehat{L}$ and $\widehat{R}$ (e.g. following the method proposed in [1]).
6. Working out the product $\widetilde{M} = \widehat{L}M\widehat{R}^*$ and, if needed, defining from $\widetilde{M}$ a coarse-grained graph.

Functions for performing SCG

The main function is the "all-in-one" `scg`. This function handles all the steps involved in the Spectral Coarse Graining (SCG) of some particular matrices and graphs as described above and in reference [1]. In more details, `scg` computes some prescribed eigenpairs of a matrix or a graph matrix (for now adjacency, Laplacian and stochastic matrices are available), works out an optimal partition to preserve the eigenpairs, and finally outputs a coarse-grained matrix or graph along with other useful information.

These steps can also be carried out independently: (1) Use `get.adjacency`, `graph.laplacian` or `get.stochastic` to compute a matrix M. (2) Work out some prescribed eigenpairs of M e.g. by means of `eigen` or `arpack`. (3) Invoke one the four algorithms of the function `scgGrouping` to get a partition that will preserve the eigenpairs in the coarse-grained matrix. (4) Compute the semi-projectors L and R using `scgSemiProjectors` and from there the coarse-grained matrix $\widetilde{M} = LMR^*$. If necessary, construct a coarse-grained graph from $\widetilde{M}$ (e.g. as in [1]).

Author(s)

David Morton de Lachapelle <david.morton@epfl.ch>, <david.mortondelachapelle@swissquote.ch>

References

D. Morton de Lachapelle, D. Gfeller, and P. De Los Rios, Shrinking Matrices while Preserving their Eigenpairs with Application to the Spectral Coarse Graining of Graphs. Submitted to *SIAM Journal on Matrix Analysis and Applications*, 2008. `http://people.epfl.ch/david.morton`

D. Gfeller, and P. De Los Rios, Spectral Coarse Graining and Synchronization in Oscillator Networks. *Physical Review Letters*, **100**(17), 2008. `http://arxiv.org/abs/0708.2055`

D. Gfeller, and P. De Los Rios, Spectral Coarse Graining of Complex Networks, *Physical Review Letters*, **99**(3), 2007. `http://arxiv.org/abs/0706.0812`

`spinglass.community` *Finding communities in graphs based on statistical meachanics*

Description

This function tries to find communities in graphs via a spin-glass model and simulated annealing.

Usage

```
spinglass.community(graph, weights=NULL, vertex=NULL, spins=25,
                    parupdate=FALSE, start.temp=1, stop.temp=0.01,
                    cool.fact=0.99, update.rule=c("config", "random",
                    "simple"), gamma=1, implementation=c("orig", "neg"),
                    gamma.minus=1)
```

Arguments

`graph`	The input graph, can be directed but the direction of the edges is neglected.
`weights`	The weights of the edges. Either a numeric vector or `NULL`. If it is null and the input graph has a 'weight' edge attribute then that will be used. If `NULL` and no such attribute is present then the edges will have equal weights. Set this to `NA` if the graph was a 'weight' edge attribute, but you don't want to use it for community detection.
`vertex`	This parameter can be used to calculate the community of a given vertex without calculating all communities. Note that if this argument is present then some other arguments are ignored.
`spins`	Integer constant, the number of spins to use. This is the upper limit for the number of communities. It is not a problem to supply a (reasonably) big number here, in which case some spin states will be unpopulated.
`parupdate`	Logical constant, whether to update the spins of the vertices in parallel (synchronously) or not. This argument is ignored if the second form of the function is used (ie. the '`vertex`' argument is present).
`start.temp`	Real constant, the start temperature. This argument is ignored if the second form of the function is used (ie. the '`vertex`' argument is present).
`stop.temp`	Real constant, the stop temperature. The simulation terminates if the temperature lowers below this level. This argument is ignored if the second form of the function is used (ie. the '`vertex`' argument is present).
`cool.fact`	Cooling factor for the simulated annealing. This argument is ignored if the second form of the function is used (ie. the '`vertex`' argument is present).

`update.rule`	Character constant giving the 'null-model' of the simulation. Possible values: "simple" and "config". "simple" uses a random graph with the same number of edges as the baseline probability and "config" uses a random graph with the same vertex degrees as the input graph.
`gamma`	Real constant, the gamma argument of the algorithm. This specifies the balance between the importance of present and non-present edges in a community. Roughly, a comunity is a set of vertices having many edges inside the community and few edges outside the community. The default 1.0 value makes existing and non-existing links equally important. Smaller values make the existing links, greater values the missing links more important.
`implementation`	Character scalar. Currently igraph contains two implementations for the Spinglass community finding algorithm. The faster original implementation is the default. The other implementation, that takes into account negative weights, can be chosen by supplying 'neg' here.
`gamma.minus`	Real constant, the gamma.minus parameter of the algorithm. This specifies the balance between the importance of present and non-present negative weighted edges in a community. Smaller values of gamma.minus, leads to communities with lesser negative intra-connectivity. If this argument is set to zero, the algorithm reduces to a graph coloring algorithm, using the number of spins as the number of colors. This argument is ignored if the 'orig' implementation is chosen.

Details

This function tries to find communities in a graph. A community is a set of nodes with many edges inside the community and few edges between outside it (i.e. between the community itself and the rest of the graph.)

This idea is reversed for edges having a negative weight, ie. few negative edges inside a community and many negative edges between communities. Note that only the 'neg' implementation supports negative edge weights.

The `spinglass.cummunity` function can solve two problems related to community detection. If the `vertex` argument is not given (or it is `NULL`), then the regular community detection problem is solved (approximately), i.e. partitioning the vertices into communities, by optimizing the an energy function.

If the `vertex` argument is given and it is not `NULL`, then it must be a vertex id, and the same energy function is used to find the community of the the given vertex. See also the examples below.

Value

If the `vertex` argument is not given, ie. the first form is used then a `spinglass.community` returns a `communities` object.

If the `vertex` argument is present, ie. the second form is used then a named list is returned with the following components:

`community`	Numeric vector giving the ids of the vertices in the same community as `vertex`.
`cohesion`	The cohesion score of the result, see references.
`adhesion`	The adhesion score of the result, see references.
`inner.links`	The number of edges within the community of `vertex`.
`outer.links`	The number of edges between the community of `vertex` and the rest of the graph.

Author(s)

Jorg Reichardt <lastname@physik.uni-wuerzburg.de> for the original code and Gabor Csardi <csardi.gabor@gmail.com> for the igraph glue code.

Changes to the original function for including the possibility of negative ties were implemented by Vincent Traag <vtraag@f-m.fm>.

References

J. Reichardt and S. Bornholdt: Statistical Mechanics of Community Detection, *Phys. Rev. E*, 74, 016110 (2006), http://arxiv.org/abs/cond-mat/0603718

M. E. J. Newman and M. Girvan: Finding and evaluating community structure in networks, *Phys. Rev. E* 69, 026113 (2004)

V.A. Traag and Jeroen Bruggeman: Community detection in networks with positive and negative links, http://arxiv.org/abs/0811.2329 (2008).

See Also

communities, clusters

Examples

```
g <- erdos.renyi.game(10, 5/10) %du% erdos.renyi.game(9, 5/9)
g <- add.edges(g, c(1, 12))
g <- induced.subgraph(g, subcomponent(g, 1))
spinglass.community(g, spins=2)
spinglass.community(g, vertex=1)
```

---

static.fitness.game *Random graphs from vertex fitness scores*

Description

This function generates a non-growing random graph with edge probabilities proportional to node fitness scores.

Usage

```
static.fitness.game (no.of.edges, fitness.out, fitness.in,
                     loops = FALSE, multiple = FALSE)
```

Arguments

no.of.edges	The number of edges in the generated graph.
fitness.out	A numeric vector containing the fitness of each vertex. For directed graphs, this specifies the out-fitness of each vertex.
fitness.in	If NULL (the default), the generated graph will be undirected. If not NULL, then it should be a numeric vector and it specifies the in-fitness of each vertex. If this argument is not NULL, then a directed graph is generated, otherwise an undirected one.
loops	Logical scalar, whether to allow loop edges in the graph.
multiple	Logical scalar, whether to allow multiple edges in the graph.

Details

This game generates a directed or undirected random graph where the probability of an edge between vertices i and j depends on the fitness scores of the two vertices involved. For undirected graphs, each vertex has a single fitness score. For directed graphs, each vertex has an out- and an in-fitness, and the probability of an edge from i to j depends on the out-fitness of vertex i and the in-fitness of vertex j.

The generation process goes as follows. We start from N disconnected nodes (where N is given by the length of the fitness vector). Then we randomly select two vertices i and j, with probabilities proportional to their fitnesses. (When the generated graph is directed, i is selected according to the out-fitnesses and j is selected according to the in-fitnesses). If the vertices are not connected yet (or if multiple edges are allowed), we connect them; otherwise we select a new pair. This is repeated until the desired number of links are created.

It can be shown that the *expected* degree of each vertex will be proportional to its fitness, although the actual, observed degree will not be. If you need to generate a graph with an exact degree sequence, consider `degree.sequence.game` instead.

This model is commonly used to generate static scale-free networks. To achieve this, you have to draw the fitness scores from the desired power-law distribution. Alternatively, you may use `static.power.law.game` which generates the fitnesses for you with a given exponent.

Value

An igraph graph, directed or undirected.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`>

References

Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. *Phys Rev Lett* 87(27):278701, 2001.

Examples

```
N <- 10000
g <- static.fitness.game(5*N, sample((1:50)^-2, N, replace=TRUE))
degree.distribution(g)
## Not run: plot(degree.distribution(g, cumulative=TRUE), log="xy")
```

`static.power.law.game` *Scale-free random graphs, from vertex fitness scores*

Description

This function generates a non-growing random graph with expected power-law degree distributions.

Usage

```
static.power.law.game (no.of.nodes, no.of.edges, exponent.out,
                       exponent.in = -1, loops = FALSE,
                       multiple = FALSE, finite.size.correction = TRUE)
```

Arguments

`no.of.nodes`	The number of vertices in the generated graph.
`no.of.edges`	The number of edges in the generated graph.
`exponent.out`	Numeric scalar, the power law exponent of the degree distribution. For directed graphs, this specifies the exponent of the out-degree distribution. It must be greater than or equal to 2. If you pass `Inf` here, you will get back an Erdos-Renyi random network.
`exponent.in`	Numeric scalar. If negative, the generated graph will be undirected. If greater than or equal to 2, this argument specifies the exponent of the in-degree distribution. If non-negative but less than 2, an error will be generated.
`loops`	Logical scalar, whether to allow loop edges in the generated graph.
`multiple`	Logical scalar, whether to allow multiple edges in the generated graph.
`finite.size.correction`	Logical scalar, whether to use the proposed finite size correction of Cho et al., see references below.

Details

This game generates a directed or undirected random graph where the degrees of vertices follow power-law distributions with prescribed exponents. For directed graphs, the exponents of the in- and out-degree distributions may be specified separately.

The game simply uses `static.fitness.game` with appropriately constructed fitness vectors. In particular, the fitness of vertex i is i^{-alpha}, where $alpha = 1/(gamma - 1)$ and gamma is the exponent given in the arguments.

To remove correlations between in- and out-degrees in case of directed graphs, the in-fitness vector will be shuffled after it has been set up and before `static.fitness.game` is called.

Note that significant finite size effects may be observed for exponents smaller than 3 in the original formulation of the game. This function provides an argument that lets you remove the finite size effects by assuming that the fitness of vertex i is $(i + i_0 - 1)^{-alpha}$ where i_0 is a constant chosen appropriately to ensure that the maximum degree is less than the square root of the number of edges times the average degree; see the paper of Chung and Lu, and Cho et al for more details.

Value

An igraph graph, directed or undirected.

Author(s)

Tamas Nepusz <`ntamas@gmail.com`>

References

Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. *Phys Rev Lett* 87(27):278701, 2001.

Chung F and Lu L: Connected components in a random graph with given degree sequences. *Annals of Combinatorics* 6, 125-145, 2002.

Cho YS, Kim JS, Park J, Kahng B, Kim D: Percolation transitions in scale-free networks under the Achlioptas process. *Phys Rev Lett* 103:135702, 2009.

Examples

```
g <- static.power.law.game(10000, 30000, 2.2, 2.3)
## Not run: plot(degree.distribution(g, cumulative=TRUE, mode="out"), log="xy")
```

`stCuts` *List all (s,t)-cuts of a graph*

Description

List all (s,t)-cuts in a directed graph.

Usage

```
stCuts(graph, source, target)
```

Arguments

`graph`	The input graph. It must be directed.
`source`	The source vertex.
`target`	The target vertex.

Details

Given a G directed graph and two, different and non-ajacent vertices, s and t, an (s, t)-cut is a set of edges, such that after removing these edges from G there is no directed path from s to t.

Value

A list with entries:

`cuts`	A list of numeric vectors containing edge ids. Each vector is an (s, t)-cut.
`partition1s`	A list of numeric vectors containing vertex ids, they correspond to the edge cuts. Each vertex set is a generator of the corresponding cut, i.e. in the graph $G = (V, E)$, the vertex set X and its complementer $V - X$, generates the cut that contains exactly the edges that go from X to $V - X$.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, *Algorithmica* 15, 351–372, 1996.

See Also

`stMincuts` to list all minimum cuts.

Examples

```
# A very simple graph
g <- graph.formula(a -+ b -+ c -+ d -+ e)
stCuts(g, source="a", target="e")

# A somewhat more difficult graph
g2 <- graph.formula(s --+ a:b, a:b --+ t,
                    a --+ 1:2:3, 1:2:3 --+ b)
stCuts(g2, source="s", target="t")
```

`stMincuts` *List all minimum (s, t)-cuts of a graph*

Description

Listing all minimum (s, t)-cuts of a directed graph, for given s and t.

Usage

```
stMincuts(graph, source, target, capacity = NULL)
```

Arguments

`graph`	The input graph. It must be directed.
`source`	The id of the source vertex.
`target`	The id of the target vertex.
`capacity`	Numeric vector giving the edge capacities. If this is `NULL` and the graph has a `weight` edge attribute, then this attribute defines the edge capacities. For forcing unit edge capacities, even for graphs that have a `weight` edge attribute, supply `NA` here.

Details

Given a G directed graph and two, different and non-ajacent vertices, s and t, an (s, t)-cut is a set of edges, such that after removing these edges from G there is no directed path from s to t.

The size of an (s, t)-cut is defined as the sum of the capacities (or weights) in the cut. For unweighed (=equally weighted) graphs, this is simply the number of edges.

An (s, t)-cut is minimum if it is of the smallest possible size.

Value

A list with entries:

`value`	Numeric scalar, the size of the minimum cut(s).
`cuts`	A list of numeric vectors containing edge ids. Each vector is a minimum (s, t)-cut.
`partition1s`	A list of numeric vectors containing vertex ids, they correspond to the edge cuts. Each vertex set is a generator of the corresponding cut, i.e. in the graph $G = (V, E)$, the vertex set X and its complementer $V - X$, generates the cut that contains exactly the edges that go from X to $V - X$.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

JS Provan and DR Shier: A Paradigm for listing (s,t)-cuts in graphs, *Algorithmica* 15, 351–372, 1996.

See Also

stCuts, minimum.size.separators

Examples

```
# A difficult graph, from the Provan-Shier paper
g <- graph.formula(s --+ a:b, a:b --+ t,
                   a --+ 1:2:3:4:5, 1:2:3:4:5 --+ b)
stMincuts(g, source="s", target="t")
```

---

structure.info *Gaining information about graph structure*

Description

Functions for exploring the basic structure of a network: number of vertices and edges, the neighbors of a node, test whether two vertices are connected by an edge.

Usage

```
vcount(graph)
ecount(graph)
neighbors(graph, v, mode = 1)
incident(graph, v, mode=c("all", "out", "in", "total"))
is.directed(graph)
are.connected(graph, v1, v2)
get.edge(graph, id)
get.edges(graph, es)
```

Arguments

graph	The graph.
v	The vertex of which the adjacent vertices or incident edges are queried.
mode	Character string, specifying the type of adjacent vertices or incident edges to list in a directed graph. If “out”, then only outgoing edges (or their corresponding vertices) are considered; “in” considers incoming edges; ‘all’ ignores edge directions. This argument is ignored for undirected graphs.
v1	The id of the first vertex. For directed graphs only edges pointing from v1 to v2 are searched.
v2	The id of the second vertex. For directed graphs only edges pointing from v1 to v2 are searched.
id	A numeric edge id.
es	An edge sequence.

Details

These functions provide the basic structural information of a graph.

`vcount` gives the number of vertices in the graph.

`ecount` gives the number of edges in the graph.

`neighbors` gives the neighbors of a vertex. The vertices connected by multiple edges are listed as many times as the number of connecting edges.

`incident` gives the incident edges of a vertex.

`is.directed` gives whether the graph is directed or not. It just gives its `directed` attribute.

`are.connected` decides whether there is an edge from `v1` to `v2`.

`get.edge` returns the end points of the edge with the supplied edge id. For directed graph the source vertex comes first, for undirected graphs, the order is arbitrary.

`get.edges` returns a matrix with the endpoints of the edges in the edge sequence argument.

Value

`vcount` and `ecount` return integer constants. `neighbors` returns an integer vector. `is.directed` and `are.connected` return boolean constants. `get.edge` returns a numeric vector of length two. `get.edges` returns a two-column matrix.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`graph`

Examples

```
g <- graph.ring(10)
vcount(g)
ecount(g)
neighbors(g, 5)
incident(g, 5)
are.connected(g, 1, 2)
are.connected(g, 2, 4)
get.edges(g, 1:6)
```

`subgraph` *Subgraph of a graph*

Description

`subgraph` creates a subgraph of a graph, containing only the specified vertices and all the edges among them.

Usage

```
induced.subgraph(graph, vids, impl=c("auto", "copy_and_delete",
     "create_from_scratch"))
subgraph.edges(graph, eids, delete.vertices = TRUE)
subgraph(graph, v)
```

Arguments

`graph`	The original graph.
`vids,v`	Numeric vector, the vertices of the original graph which will form the subgraph.
`impl`	Character scalar, to choose between two implementation of the subgraph calculation. '`copy_and_delete`' copies the graph first, and then deletes the vertices and edges that are not included in the result graph. '`create_from_scratch`' searches for all vertices and edges that must be kept and then uses them to create the graph from scratch. '`auto`' chooses between the two implementations automatically, using heuristics based on the size of the original and the result graph.
`eids`	The edge ids of the edges that will be kept in the result graph.
`delete.vertices`	Logical scalar, whether to remove vertices that do not have any adjacent edges in `eids`.

Details

`induced.subgraph` calculates the induced subgraph of a set of vertices in a graph. This means that exactly the specified vertices and all the edges between then will be kept in the result graph.

`subgraph.edges` calculates the subgraph of a graph. For this function one can specify the vertices and edges to keep. This function will be renamed to `subgraph` in the next major version of igraph.

The `subgraph` function does the same as `induced.graph` currently (assuming '`auto`' as the `impl` argument), but it is deprecated and will be removed in the next major version of igraph.

Value

A new graph object.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
g <- graph.ring(10)
g2 <- induced.subgraph(g, 1:7)
g3 <- subgraph.edges(g, 1:5, 1:5)
```

`subgraph.centrality` *Find subgraph centrality scores of network positions*

Description

Subgraph centrality of a vertex measures the number of subgraphs a vertex participates in, weighting them according to their size.

Usage

```
subgraph.centrality (graph, diag=FALSE)
```

Arguments

`graph`	The input graph, it should be undirected, but the implementation does not check this currently.
`diag`	Boolean scalar, whether to include the diagonal of the adjacency matrix in the analysis. Giving `FALSE` here effectively eliminates the loops edges from the graph before the calculation.

Details

The subgraph centrality of a vertex is defined as the number of closed loops originating at the vertex, where longer loops are exponentially downweighted.

Currently the calculation is performed by explicitly calculating all eigenvalues and eigenvectors of the adjacency matrix of the graph. This effectively means that the measure can only be calculated for small graphs.

Value

A numeric vector, the subgraph centrality scores of the vertices.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`> based on the Matlab code by Ernesto Estrada

References

Ernesto Estrada, Juan A. Rodriguez-Velazquez: Subgraph centrality in Complex Networks. *Physical Review E* 71, 056103 (2005).

See Also

`evcent`, `page.rank`

Examples

```
g <- ba.game(100, m=4, dir=FALSE)
sc <- subgraph.centrality(g)
cor(degree(g), sc)
```

tkigraph *Experimental basic igraph GUI*

Description

This functions starts an experimental GUI to some igraph functions. The GUI was written in Tcl/Tk, so it is cross platform.

Usage

```
tkigraph()
```

Details

`tkigraph` has its own online help system, please see that for the details about how to use it.

Value

Returns `NULL`, invisibly.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`tkplot` for interactive plotting of graphs.

tkplot *Interactive plotting of graphs*

Description

`tkplot` and its companion functions serve as an interactive graph drawing facility. Not all parameters of the plot can be changed interactively right now though, eg. the colors of vertices, edges, and also others have to be pre-defined.

Usage

```
tkplot(graph, canvas.width=450, canvas.height=450, ...)

tkplot.close(tkp.id, window.close = TRUE)
tkplot.off()
tkplot.fit.to.screen(tkp.id, width = NULL, height = NULL)
tkplot.reshape(tkp.id, newlayout, ...)
tkplot.export.postscript(tkp.id)
tkplot.getcoords(tkp.id, norm = FALSE)
tkplot.center(tkp.id)
tkplot.rotate(tkp.id, degree = NULL, rad = NULL)
```

Arguments

`graph`	The `graph` to plot.
`canvas.width,canvas.height`	The size of the tkplot drawing area.
`tkp.id`	The id of the tkplot window to close/reshape/etc.
`window.close`	Leave this on the default value.
`width`	The width of the rectangle for generating new coordinates.
`height`	The height of the rectangle for generating new coordinates.
`newlayout`	The new layout, see the `layout` parameter of tkplot.
`norm`	Logical, should we norm the coordinates.
`degree`	The degree to rotate the plot.
`rad`	The degree to rotate the plot, in radian.
`...`	Not used right now.

Details

`tkplot` is an interactive graph drawing facility. It is not very well developed at this stage, but it should be still useful.

It's handling should be quite straightforward most of the time, here are some remarks and hints.

There are different popup menus, activated by the right mouse button, for vertices and edges. Both operate on the current selection if the vertex/edge under the cursor is part of the selection and operate on the vertex/edge under the cursor if it is not.

One selection can be active at a time, either a vertex or an edge selection. A vertex/edge can be added to a selection by holding the `control` key while clicking on it with the left mouse button. Doing this again deselect the vertex/edge.

Selections can be made also from the `Select` menu. The 'Select some vertices' dialog allows to give an expression for the vertices to be selected: this can be a list of numeric R expessions separated by commas, like '`1,2:10,12,14,15`' for example. Similarly in the 'Select some edges' dialog two such lists can be given and all edges connecting a vertex in the first list to one in the second list will be selected.

In the color dialog a color name like 'orange' or RGB notation can also be used.

The `tkplot` command creates a new Tk window with the graphical representation of `graph`. The command returns an integer number, the tkplot id. The other commands utilize this id to be able to query or manipulate the plot.

`tkplot.close` closes the Tk plot with id `tkp.id`.

`tkplot.off` closes all Tk plots.

`tkplot.fit.to.screen` fits the plot to the given rectange (`width` and `height`), if some of these are `NULL` the actual phisical width od height of the plot window is used.

`tkplot.reshape` applies a new layout to the plot, its optional parameters will be collected to a list analogous to `layout.par`.

`tkplot.export.postscript` creates a dialog window for saving the plot in postscript format.

`tkplot.getcoords` returns the coordinates of the vertices in a matrix. Each row corresponds to one vertex.

`tkplot.center` shifts the figure to the center of its plot window.

`tkplot.rotate` rotates the figure, its parameter can be given either in degrees or in radians.

Value

`tkplot` returns an integer, the id of the plot, this can be used to manipulate it from the command line.

`tkplot.getcoords` returns a matrix with the coordinates.

`tkplot.close`, `tkplot.off`, `tkplot.fit.to.screen`, `tkplot.reshape`, `tkplot.export.postscript`, `tkplot.center` and `tkplot.rotate` return `NULL` invisibly.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

See Also

`plot.igraph`, `layout`

Examples

```
g <- graph.ring(10)
## Not run: tkplot(g)
```

`topological.sort` *Topological sorting of vertices in a graph*

Description

A topological sorting of a directed acyclic graph is a linear ordering of its nodes where each node comes before all nodes to which it has edges.

Usage

```
topological.sort(graph, mode=c("out", "all", "in"))
```

Arguments

`graph`	The input graph, should be directed
`mode`	Specifies how to use the direction of the edges. For "`out`", the sorting order ensures that each node comes before all nodes to which it has edges, so nodes with no incoming edges go first. For "`in`", it is quite the opposite: each node comes before all nodes from which it receives edges. Nodes with no outgoing edges go first.

Details

Every DAG has at least one topological sort, and may have many. This function returns a possible topological sort among them. If the graph is not acyclic (it has at least one cycle), a partial topological sort is returned and a warning is issued.

Value

A numeric vector containing vertex ids in topologically sorted order.

Author(s)

Tamas Nepusz <ntamas@gmail.com> and Gabor Csardi <csardi.gabor@gmail.com> for the R interface

Examples

```
g <- barabasi.game(100)
topological.sort(g)
```

---

traits *Graph generation based on different vertex types*

Description

These functions implement evolving network models based on different vertex types.

Usage

```
callaway.traits.game (nodes, types, edge.per.step = 1, type.dist = rep(1,
    types), pref.matrix = matrix(1, types, types), directed = FALSE)
establishment.game(nodes, types, k = 1, type.dist = rep(1, types),
    pref.matrix = matrix(1, types, types), directed = FALSE)
```

Arguments

`nodes`	The number of vertices in the graph.
`types`	The number of different vertex types.
`edge.per.step`	The number of edges to add to the graph per time step.
`type.dist`	The distribution of the vertex types. This is assumed to be stationary in time.
`pref.matrix`	A matrix giving the preferences of the given vertex types. These should be probabilities, ie. numbers between zero and one.
`directed`	Logical constant, whether to generate directed graphs.
`k`	The number of trials per time step, see details below.

Details

For `callaway.traits.game` the simulation goes like this: in each discrete time step a new vertex is added to the graph. The type of this vertex is generated based on `type.dist`. Then two vertices are selected uniformly randomly from the graph. The probability that they will be connected depends on the types of these vertices and is taken from `pref.matrix`. Then another two vertices are selected and this is repeated `edges.per.step` times in each time step.

For `establishment.game` the simulation goes like this: a single vertex is added at each time step. This new vertex tries to connect to `k` vertices in the graph. The probability that such a connection is realized depends on the types of the vertices involved and is taken from `pref.matrix`.

Value

A new graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

Examples

```
# two types of vertices, they like only themselves
g1 <- callaway.traits.game(1000, 2, pref.matrix=matrix( c(1,0,0,1), nc=2))
g2 <- establishment.game(1000, 2, k=2, pref.matrix=matrix( c(1,0,0,1), nc=2))
```

---

transitivity *Transitivity of a graph*

Description

Transitivity measures the probability that the adjacent vertices of a vertex are connected. This is sometimes also called the clustering coefficient.

Usage

```
transitivity(graph, type=c("undirected", "global", "globalundirected",
       "localundirected", "local", "average", "localaverage",
       "localaverageundirected", "barrat", "weighted"), vids=NULL,
       weights=NULL, isolates=c("NaN", "zero"))
```

Arguments

`graph`	The graph to analyze.
`type`	The type of the transitivity to calculate. Possible values: `global` The global transitivity of an undirected graph (directed graphs are considered as undirected ones as well). This is simply the ratio of the triangles and the connected triples in the graph. For directed graph the direction of the edges is ignored. `local` The local transitivity of an undirected graph, this is calculated for each vertex given in the `vids` argument. The local transitivity of a vertex is the ratio of the triangles connected to the vertex and the triples centered on the vertex. For directed graph the direction of the edges is ignored. `undirected` This is the same as `global`. `globalundirected` This is the same as `global`. `localundirected` This is the same as `local`. `barrat` The weighted transitivity as defined A. Barrat. See details below. `weighted` The same as `barrat`.
`vids`	The vertex ids for the local transitivity will be calculated. This will be ignored for global transitivity types. The default value is `NULL`, in this case all vertices are considered. It is slightly faster to supply `NULL` here than `V(graph)`.
`weights`	Optional weights for weighted transitivity. It is ignored for other transitivity measures. If it is `NULL` (the default) and the graph has a `weight` edge attribute, then it is used automatically.

`isolates`	Character scalar, defines how to treat vertices with degree zero and one. If it is '`NaN`' then they local transitivity is reported as `NaN` and they are not included in the averaging, for the transitivity types that calculate an average. If there are no vertices with degree two or higher, then the averaging will still result `NaN`. If it is '`zero`', then we report 0 transitivity for them, and they are included in the averaging, if an average is calculated.

Details

Note that there are essentially two classes of transitivity measures, one is a vertex-level, the other a graph level property.

There are several generalizations of transitivity to weighted graphs, here we use the definition by A. Barrat, this is a local vertex-level quantity, its formula is

$$C_i^w = \frac{1}{s_i(k_i-1)} \sum_{j,h} \frac{w_{ij}+w_{ih}}{2} a_{ij}a_{ih}a_{jh}$$

s_i is the strength of vertex i, see `graph.strength`, a_{ij} are elements of the adjacency matrix, k_i is the vertex degree, w_{ij} are the weights.

This formula gives back the normal not-weighted local transitivity if all the edge weights are the same.

The `barrat` type of transitivity does not work for graphs with multiple and/or loop edges. If you want to calculate it for a directed graph, call `as.undirected` with the `collapse` mode first.

Value

For '`global`' a single number, or `NaN` if there are no connected triples in the graph.

For '`local`' a vector of transitivity scores, one for each vertex in '`vids`'.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

Wasserman, S., and Faust, K. (1994). *Social Network Analysis: Methods and Applications.* Cambridge: Cambridge University Press.

Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, Alessandro Vespignani: The architecture of complex weighted networks, Proc. Natl. Acad. Sci. USA 101, 3747 (2004)

Examples

```
g <- graph.ring(10)
transitivity(g)
g2 <- erdos.renyi.game(1000, 10/1000)
transitivity(g2)   # this is about 10/1000

# Weighted version, the figure from the Barrat paper
gw <- graph.formula(A-B:C:D:E, B-C:D, C-D)
E(gw)$weight <- 1
E(gw)[ V(gw)[name == "A"] %--% V(gw)[name == "E" ] ]$weight <- 5
transitivity(gw, vids=V(gw)[name=="A"], type="weighted")
```

```
# Weighted reduces to "local" if weights are the same
gw2 <- erdos.renyi.game(1000, 10/1000)
E(gw2)$weight <- 1
all(transitivity(gw2, type="local") == transitivity(gw2, type="weighted"))
all( transitivity(gw2, vids=2:vcount(g)-1, type="local") ==
     transitivity(gw2, vids=2:vcount(g)-1, type="weighted") )
```

triad.census *Triad census, subgraphs with three vertices*

Description

This function counts the different subgraphs of three vertices in a graph.

Usage

```
triad.census(graph)
```

Arguments

graph	The input graph, it should be directed. An undirected graph results a warning, and undefined results.

Details

Triad census was defined by David and Leinhardt (see References below). Every triple of vertices (A, B, C) are classified into the 16 possible states:

003 A,B,C, the empty graph.

012 A->B, C, the graph with a single directed edge.

102 A<->B, C, the graph with a mutual connection between two vertices.

021D A<-B->C, the out-star.

021U A->B<-C, the in-star.

021C A->B->C, directed line.

111D A<->B<-C.

111U A<->B->C.

030T A->B<-C, A->C.

030C A<-B<-C, A->C.

201 A<->B<->C.

120D A<-B->C, A<->C.

120U A->B<-C, A<->C.

120C A->B->C, A<->C.

210 A->B<->C, A<->C.

300 A<->B<->C, A<->C, the complete graph.

This functions uses the RANDESU motif finder algorithm to find and count the subgraphs, see `graph.motifs`.

Value

A numeric vector, the subgraph counts, in the order given in the above description.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

References

See also Davis, J.A. and Leinhardt, S. (1972). The Structure of Positive Interpersonal Relations in Small Groups. In J. Berger (Ed.), Sociological Theories in Progress, Volume 2, 218-251. Boston: Houghton Mifflin.

See Also

`dyad.census` for classifying binary relationships, `graph.motifs` for the underlying implementation.

Examples

```
g <- erdos.renyi.game(15, 45, type="gnm", dir=TRUE)
triad.census(g)
```

`unfold.tree` *Convert a general graph into a forest*

Description

Perform a breadth-first search on a graph and convert it into a tree or forest by replicating vertices that were found more than once.

Usage

```
unfold.tree(graph, mode = c("all", "out", "in", "total"), roots)
```

Arguments

`graph`	The input graph, it can be either directed or undirected.
`mode`	Character string, defined the types of the paths used for the breadth-first search. "out" follows the outgoing, "in" the incoming edges, "all" and "total" both of them. This argument is ignored for undirected graphs.
`roots`	A vector giving the vertices from which the breadth-first search is performed. Typically it contains one vertex per component.

Details

A forest is a graph, whose components are trees.

The `roots` vector can be calculated by simply doing a topological sort in all components of the graph, see the examples below.

Value

A list with two components:

`tree`	The result, an `igraph` object, a tree or a forest.
`vertex_index`	A numeric vector, it gives a mapping from the vertices of the new graph to the vertices of the old graph.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

Examples

```
g <- graph.tree(10)
V(g)$id <- seq_len(vcount(g))-1
roots <- sapply(decompose.graph(g), function(x) {
            V(x)$id[ topological.sort(x)[1]+1 ] })
tree <- unfold.tree(g, roots=roots)
```

`Vertex shapes` *Various vertex shapes when plotting igraph graphs*

Description

Starting from version 0.5.1 igraph supports different vertex shapes when plotting graphs.

Details

Note that the current vertex shape implementation is experimental and it might change in the future. Currently vertex shapes are implemented only for `plot.igraph`.

In igraph a vertex shape is defined by a function that 1) provides information about the size of the shape for clipping the edges and 2) plots the shape if requested. These functions are called "shape functions" in the rest of this manual page.

Shape functions have a '`mode`' argument that decides in which mode they should operate. '`clip`' selects clipping mode and '`plot`' selects plotting mode.

In clipping mode a shape function has the following arguments:

coords A matrix with four columns, it contains the coordinates of the vertices for the edge list supplied in the `el` argument.

el A matrix with two columns, the edges of which some end points will be clipped. It should have the same number of rows as `coords`.

mode "`clip`" for choosing clipping mode.

params This is a function object that can be called to query vertex/edge/plot graphical parameters. The first argument of the function is "`vertex`", "`edge`" or "`plot`" to decide the type of the parameter, the second is a character string giving the name of the parameter. E.g.

```
params("vertex", "size")
```

end Character string, it gives which end points will be used. Possible values are "`both`", "`from`" and "`to`". If "`from`" the function is expected to clip the first column in the `el` edge list, "`to`" selects the second column, "`both`" selects both.

In 'clipping' mode, a shape function should return a matrix with the same number of rows as the `el` arguments. If `end` is `both` then the matrix must have four columns, otherwise two. The matrix contains the modified coordinates, with the clipping applied.

In plotting mode the following arguments are supplied to the shape function:

coords The coordinates of the vertices, a matrix with two columns.

v The ids of the vertices to plot. It should match the number of rows in the `coords` argument.

mode "`plot`" for choosing plotting mode.

params The same as in clipping mode, see above.

In 'plotting' mode the return value of the shape function is not used.

Author(s)

Gabor Csardi `<csardi.gabor@gmail.com>`

See Also

`igraph.plotting`, `plot.igraph`

Examples

```
## Not run:
g <- graph.ring(10, dir=TRUE, mut=TRUE)
plot(g, vertex.shape="rectangle", layout=layout.circle)

## End(Not run)
```

`vertex.connectivity` *Vertex connectivity.*

Description

The vertex connectivity of a graph or two vertices, this is recently also called group cohesion.

Usage

```
vertex.connectivity(graph, source=NULL, target=NULL, checks=TRUE)
vertex.disjoint.paths(graph, source, target)
graph.cohesion(graph, checks=TRUE)
```

Arguments

`graph`	The input graph.
`source`	The id of the source vertex, for `vertex.connectivity` it can be `NULL`, see details below.
`target`	The id of the target vertex, for `vertex.connectivity` it can be `NULL`, see details below.
`checks`	Logical constant. Whether to check that the graph is connected and also the degree of the vertices. If the graph is not (strongly) connected then the connectivity is obviously zero. Otherwise if the minimum degree is one then the vertex connectivity is also one. It is a good idea to perform these checks, as they can be done quickly compared to the connectivity calculation itself. They were suggested by Peter McMahan, thanks Peter.

Details

The vertex connectivity of two vertices (`source` and `target`) in a directed graph is the minimum number of vertices needed to remove from the graph to eliminate all (directed) paths from `source` to `target`. `vertex.connectivity` calculates this quantity if both the `source` and `target` arguments are given and they're not `NULL`.

The vertex connectivity of a graph is the minimum vertex connectivity of all (ordered) pairs of vertices in the graph. In other words this is the minimum number of vertices needed to remove to make the graph not strongly connected. (If the graph is not strongly connected then this is zero.) `vertex.connectivity` calculates this quantitty if neither the `source` nor `target` arguments are given. (Ie. they are both `NULL`.)

A set of vertex disjoint directed paths from `source` to `vertex` is a set of directed paths between them whose vertices do not contain common vertices (apart from `source` and `target`). The maximum number of vertex disjoint paths between two vertices is the same as their vertex connectivity in most cases (if the two vertices are not connected by an edge).

The cohesion of a graph (as defined by White and Harary, see references), is the vertex connectivity of the graph. This is calculated by `graph.cohesion`.

These three functions essentially calculate the same measure(s), more precisely `vertex.connectivity` is the most general, the other two are included only for the ease of using more descriptive function names.

Value

A scalar real value.

Author(s)

Gabor Csardi <`csardi.gabor@gmail.com`>

References

White, Douglas R and Frank Harary 2001. The Cohesiveness of Blocks In Social Networks: Node Connectivity and Conditional Density. *Sociological Methodology* 31 (1) : 305-359.

See Also

`graph.maxflow`, `edge.connectivity`, `edge.disjoint.paths`, `graph.adhesion`

Examples

```
g <- barabasi.game(100, m=1)
g <- delete.edges(g, E(g)[ 100 %--% 1 ])
g2 <- barabasi.game(100, m=5)
g2 <- delete.edges(g2, E(g2)[ 100 %--% 1])
vertex.connectivity(g, 100, 1)
vertex.connectivity(g2, 100, 1)
vertex.disjoint.paths(g2, 100, 1)

g <- erdos.renyi.game(50, 5/50)
g <- as.directed(g)
g <- induced.subgraph(g, subcomponent(g, 1))
graph.cohesion(g)
```

`walktrap.community` *Community strucure via short random walks*

Description

This function tries to find densely connected subgraphs, also called communities in a graph via random walks. The idea is that short random walks tend to stay in the same community.

Usage

```
walktrap.community(graph, weights = E(graph)$weight, steps = 4, merges =
           TRUE, modularity = TRUE, membership = TRUE)
```

Arguments

`graph`	The input graph, edge directions are ignored in directed graphs.
`weights`	The edge weights.
`steps`	The length of the random walks to perform.
`merges`	Logical scalar, whether to include the merge matrix in the result.
`modularity`	Logical scalar, whether to include the vector of the modularity scores in the result. If the `membership` argument is true, then it will be always calculated.
`membership`	Logical scalar, whether to calculate the membership vector for the split corresponding to the highest modularity value.

Details

This function is the implementation of the Walktrap community finding algorithm, see Pascal Pons, Matthieu Latapy: Computing communities in large networks using random walks, http://arxiv.org/abs/physics/0512106

Value

`walktrap.community` returns a `communities` object, please see the `communities` manual page for details.

Author(s)

Pascal Pons <`google@for.it`> and Gabor Csardi <`csardi.gabor@gmail.com`> for the R and igraph interface

References

Pascal Pons, Matthieu Latapy: Computing communities in large networks using random walks, http://arxiv.org/abs/physics/0512106

See Also

See `communities` on getting the actual membership vector, merge matrix, modularity score, etc.

`modularity` and `fastgreedy.community`, `spinglass.community`, `leading.eigenvector.community`, `edge.betweenness.community` for other community detection methods.

Examples

```
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(1,6, 1,11, 6, 11))
walktrap.community(g)
```

`watts.strogatz.game` *The Watts-Strogatz small-world model*

Description

Generate a graph according to the Watts-Strogatz network model.

Usage

```
watts.strogatz.game(dim, size, nei, p, loops = FALSE, multiple = FALSE)
```

Arguments

`dim`	Integer constant, the dimension of the starting lattice.
`size`	Integer constant, the size of the lattice along each dimension.
`nei`	Integer constant, the neighborhood within which the vertices of the lattice will be connected.
`p`	Real constant between zero and one, the rewiring probability.
`loops`	Logical scalar, whether loops edges are allowed in the generated graph.
`multiple`	Logical scalar, whether multiple edges are allowed int the generated graph.

Details

First a lattice is created with the given `dim`, `size` and `nei` arguments. Then the edges of the lattice are rewired uniformly randomly with probability `p`.

Note that this function might create graphs with loops and/or multiple edges. You can use `simplify` to get rid of these.

Value

A graph object.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Duncan J Watts and Steven H Strogatz: Collective dynamics of 'small world' networks, Nature 393, 440-442, 1998.

See Also

graph.lattice, rewire.edges

Examples

```
g <- watts.strogatz.game(1, 100, 5, 0.05)
average.path.length(g)
transitivity(g, type="average")
```

---

write.graph *Writing the graph to a file in some format*

Description

write.graph is a general function for exporting graphs to foreign file formats, however not many formats are implemented right now.

Usage

```
write.graph(graph, file, format=c("edgelist", "pajek", "ncol",
       "lgl", "graphml", "dimacs", "gml", "dot", "leda"), ...)
```

Arguments

graph	The graph to export.
file	A connection or a string giving the file name to write the graph to.
format	Character string giving the file format. Right now pajek, graphml, dot, gml, edgelist, lgl, ncol and dimacs are implemented. As of igraph 0.4 this argument is case insensitive.
...	Other, format specific arguments, see below.

Value

A NULL, invisibly.

Edge list format

The edgelist format is a simple text file, with one edge in a line, the two vertex ids separated by a space character. The file is sorted by the first and the second column. This format has no additional arguments.

Pajek format

The Pajek format is a text file, see `read.graph` for details. Appropriate vertex and edge attributes are also written to the file. This format has no additional arguments.

GraphML format

The GraphML format is a flexible XML based format. See `read.graph` for GraphML details. Vertex and edge attributes are also written to the file. This format has no additional arguments.

Dot format

The dot format is used by the popular GraphViz program. Vertex and edge attributes are written to the file. There are no additional arguments for this format.

LGL format

The `lgl` format is also a simple text file, this is the format expected by the 'Large Graph Layout' layout generator software. See read.graph for details. Additional arguments:

names If you want to write symbolic vertex names instead of vertex ids, supply the name of the vertex attribute containing the symbolic names here. By default the 'name' attribute is used if there is one. Supply `NULL` if you want to use numeric vertex ids even if there is a 'name' vertex attribute.

weights If you want to write edge weights to the file, supply the name of the edge attribute here. By defaults the vertex attribute 'weights' are used if they are installed. Supply `NULL` here if you want to omit the weights.

isolates Logical, if `TRUE` the isolate vertices are also written to the file, they are omitted by default.

NCOL format

The `ncol` format is also used by LGL, it is a text file, see read.graph for details. Additional arguments:

names If you want to write symbolic vertex names instead of vertex ids, supply the name of the vertex attribute containing the symbolic names here. By default the 'name' attribute is used if there is one. Supply `NULL` if you want to use numeric vertex ids even if there is a 'name' vertex attribute.

weights If you want to write edge weights to the file, supply the name of the edge attribute here. By defaults the vertex attribute 'weights' are used if they are installed. Supply `NULL` here if you want to omit the weights.

Dimacs format

The `dimacs` file format, more specifically the version for network flow problems, see the files at `ftp://dimacs.rutgers.edu/pub/netflow/general-info/`

This is a line-oriented text file (ASCII) format. The first character of each line defines the type of the line. If the first character is `c` the line is a comment line and it is ignored. There is one problem line (`p` in the file, it must appear before any node and arc descriptor lines. The problem line has three fields separated by spaces: the problem type (`min`, `max` or `asn`), the number of vertices and number of edges in the graph. Exactly two node identification lines are expected (`n`), one for the source, one for the target vertex. These have two fields: the id of the vertex and the type of the vertex, either `s` (=source) or `t` (=target). Arc lines start with `a` and have three fields: the source vertex, the target vertex and the edge capacity.

Vertex ids are numbered from 1.

Additional arguments:

source The id of the source vertex, if `NULL` (the default) then it is taken from the `source` graph attribute.

target The id of the target vertex, if `NULL` (the default) then it is taken from the `target` graph attribute.

capacity A numeric vector giving the edge capacities. If `NULL` (the default) then it is taken from the `capacity` edge attribute.

GML file format

GML is a quite general textual format, see `http://www.infosun.fim.uni-passau.de/Graphlet/GML/` for details.

The graph, vertex and edges attributes are written to the file as well, if they are numeric of string.

As igraph is more forgiving about attribute names, it might be neccessary to simplify the them before writing to the GML file. This way we'll have a syntactically correct GML file. The following simple procedure is performed on each attribute name: first the alphanumeric characters are extracted, the others are ignored. Then if the first character is not a letter then the attribute name is prefixed with <quote>igraph</quote>. Note that this might result identical names for two attributes, igraph does not check this.

The "id" vertex attribute is treated specially. If the `id` argument is not `NULL` then it should be a numeric vector with the vertex ids and the "id" vertex attribute is ignored (if there is one). If `id` is 0 and there is a numeric `id` vertex attribute that is used instead. If ids are not specified in either way then the regular igraph vertex ids are used.

Note that whichever way vertex ids are specified, their uniqueness is not checked.

If the graph has edge attributes named "source" or "target" they're silently ignored. GML uses these attributes to specify the edges, so we cannot write them to the file. Rename them before calling this function if you want to preserve them.

Additional arguments:

id `NULL` or a numeric vector giving the vertex ids. See details above.

creator A character scalar to be added to the "Creator" line in the GML file. If this is `NULL` (the default) then the current date and time is added.

LEDA file format

LEDA is a library for efficient data types and algorithms. The support for the LEDA format is very basic at the moment; igraph writes only the LEDA graph section which supports one selected vertex and edge attribute and no layout information or visual attributes. See `http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html` for the details of this format.

Additional arguments:

vertex.attr The name of the vertex attribute whose values are to be stored in the output or `NULL` if no vertex attribute has to be stored.

edge.attr The name of the edge attribute whose values are to be stored in the output or `NULL` if no edge attribute has to be stored.

Author(s)

Gabor Csardi <csardi.gabor@gmail.com>

References

Adai AT, Date SV, Wieland S, Marcotte EM. LGL: creating a map of protein function with an algorithm for visualizing very large biological networks. *J Mol Biol.* 2004 Jun 25;340(1):179-90.

See Also

read.graph

Examples

```
g <- graph.ring(10)
## Not run: write.graph(g, "/tmp/g.txt", "edgelist")
```

Index