R documentation

of all in 'man'

December 22, 2007

R topics documented:

`Getting.Started` *Getting Started with "ergm": Statistical Modeling of Network and Graph Data*

Description

`ergm` is a collection of functions to plot, fit, diagnose, and simulate from random graph models. For a list of functions type: help(package='ergm')

For a complete list of the functions, use `library(help="ergm")` or read the rest of the manual. For a simple demonstration, use `demo(packages="ergm")`.

When publishing results obtained using this package the original authors are to be cited as:

Mark S. Handcock, David R. Hunter, Carter T. Butts, Steven M. Goodreau, and Martina Morris. 2003 *ergm: An R package for the Statistical Modeling of Social Networks*
`http://www.csde.washington.edu/statnet`.

All programs derived from this package must cite it. For complete citation information, use `citation(package="ergm")`.

Details

Recent advances in the statistical modeling of random networks have had an impact on the empirical study of social networks. Statistical exponential family models (Strauss and Ikeda 1990) are a generalization of the Markov random network models introduced by Frank and Strauss (1986), which in turn derived from developments in spatial statistics (Besag, 1974). These models recognize the complex dependencies within relational data structures. To date, the use of stochastic network models for networks has been limited by three interrelated factors: the complexity of realistic models, the lack of simulation tools for inference and validation, and a poor understanding of the inferential properties of nontrivial models.

This manual introduces software tools for the representation, visualization, and analysis of network data that address each of these previous shortcomings. The package relies on the `network` package which allows networks to be represented in R. The `ergm` package allows maximum likelihood estimates of exponential random network models to be calculated using Markov Chain Monte Carlo. The package also provides tools for plotting networks, simulating networks and assessing model goodness-of-fit.

For detailed information on how to download and install the software, go to the `ergm` website: `http://www.csde.washington.edu/statnet`. A tutorial, support newsgroup, references and links to further resources are provided there.

Author(s)

Mark S. Handcock ⟨handcock@stat.washington.edu⟩,
David R. Hunter ⟨dhunter@stat.psu.edu⟩,
Carter T. Butts ⟨buttsc@uci.edu⟩,
Steven M. Goodreau ⟨goodreau@u.washington.edu⟩, and
Martina Morris ⟨morrism@u.washington.edu⟩

Maintainer: Mark S. Handcock ⟨handcock@stat.washington.edu⟩

References

Besag, J., 1974, Spatial interaction and the statistical analysis of lattice systems (with discussion), Journal of the Royal Statistical Society, B, 36, 192-236.

Frank, O., and Strauss, D.(1986). Markov graphs. Journal of the American Statistical Association, 81, 832-842.

Handcock, M. S., Hunter, D. R., Butts, C. T., Goodreau, S. M., and Morris, M. (2003), *statnet: An R package for the Statistical Modeling of Social Networks.*,
URL `http://www.csde.washington.edu/statnet`

Strauss, D., and Ikeda, M.(1990). Pseudolikelihood estimation for social networks. Journal of the American Statistical Association, 85, 204-212.

`anova.ergm` *ANOVA for Linear Model Fits*

Description

Compute an analysis of variance table for one or more linear model fits.

Usage

```
## S3 method for class 'ergm':
anova(object, ...)

anova.ergmlist(object, ..., scale = 0, test = "F")
```

Arguments

`object, ...`	objects of class `ergm`, usually, a result of a call to `ergm`.
`test`	a character string specifying the test statistic to be used. Can be one of `"F"`, `"Chisq"` or `"Cp"`, with partial matching allowed, or `NULL` for no test.
`scale`	numeric. An estimate of the noise variance σ^2. If zero this will be estimated from the largest model considered.

Details

Specifying a single object gives a sequential analysis of variance table for that fit. That is, the reductions in the residual sum of squares as each term of the formula is added in turn are given in the rows of a table, plus the residual sum of squares.

The table will contain F statistics (and P values) comparing the mean square for the row to the residual mean square.

If more than one object is specified, the table has a row for the residual degrees of freedom and sum of squares for each model. For all but the first model, the change in degrees of freedom and sum of squares is also given. (This only make statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

Optionally the table can include test statistics. Normally the F statistic is most appropriate, which compares the mean square for a row to the residual sum of squares for the largest model considered. If `scale` is specified chi-squared tests can be used. Mallows' C_p statistic is the residual sum of squares plus twice the estimate of σ^2 times the residual degrees of freedom.

Value

An object of class `"anova"` inheriting from class `"data.frame"`.

Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used, and `anova.ergmlist` will detect this with an error.

See Also

The model fitting function `ergm`, `anova`.

Examples

```
data(molecule)
molecule %v% "atomic type" <- c(1,1,1,1,1,1,2,2,2,2,2,2,2,3,3,3,3,3,3,3)
fit0 <- ergm(molecule ~ edges)
anova(fit0)
fit1 <- ergm(molecule ~ edges + nodefactor("atomic type"))
anova(fit1)
fit2 <- ergm(molecule ~ edges + nodefactor("atomic type") +  gwesp(0.5, fixed=TRUE))
anova(fit0, fit1)
anova(fit0, fit1, fit2)
```

`as.network.numeric` *Create a Simple Random network of a Given Size*

Description

`as.network.numeric` creates a random Bernoulli network of the given size as an object of class `network`.

Usage

```
as.network.numeric(x, directed = TRUE,
    hyper = FALSE, loops = FALSE, multiple = FALSE, bipartite = FALSE,
    ignore.eval = TRUE, names.eval = NULL,
    edge.check = FALSE,
    density=NULL, theta0=NULL, numedges=NULL, ...)
```

Arguments

`x`	count; the number of nodes in the network. If `bipartite=TRUE`, it is the number of events in the network.
`directed`	logical; should edges be interpreted as directed?
`hyper`	logical; are hyperedges allowed? Currently ignored.
`loops`	logical; should loops be allowed? Currently ignored.
`multiple`	logical; are multiplex edges allowed? Currently ignored.
`bipartite`	count; should the network be interpreted as bipartite? If present (i.e., non-NULL) it is the count of the number of actors in the bipartite network. In this case, the number of nodes is equal to the number of actors plus the number of events (with all actors preceding all events). The edges are then interpreted as nondirected.
`ignore.eval`	logical; ignore edge values? Currently ignored.
`names.eval`	optionally, the name of the attribute in which edge values should be stored. Currently ignored.
`edge.check`	logical; perform consistency checks on new edges?
`density`	numeric; the probability of a tie for Bernoulli networks. If neither density nor theta0 are given, it defaults to the number of nodes divided by the number of dyads (so the expected number of ties is the same as the number of nodes.)
`theta0`	numeric; the log-odds of a tie for Bernoulli networks. It is only used if density is not specified.
`numedges`	count; if present, sample the Bernoulli network conditional on this number of edges (rather than independently with the specified probability).
`...`	additional arguments

Details

The network will have not have vertex, edge or network attributes. These can be added with operators such as `%v%`, `%n%`, `%e%`.

Value

An object of class `network`

Author(s)

Carter T. Butts ⟨buttsc@uci.edu⟩ and Mark S. Handcock ⟨handcock@stat.washington.edu⟩

References

Butts, C.T. 2002. “Memory Structures for Relational Data in R: Classes and Interfaces” Working Paper.

See Also

`network`

Examples

```
#Draw a random directed network with 25 nodes
g<-network(25)
#Draw a random undirected network with density 0.1
g<-network(25, directed=FALSE, density=0.1)
#Draw a random bipartite network with 10 events and 5 actors and density 0.1
g<-network(5, bipartite=10, density=0.1)
```

`coef.ergm` *Extract Model Coefficients*

Description

`coef` is a Method which extracts model coefficients from objects returned by the `ergm` function. `coefficients` is an *alias* for it.

Usage

```
## S3 method for class 'ergm':
coef(object, ...)
## S3 method for class 'ergm':
coefficients(object, ...)
```

Arguments

`object`	an object for which the extraction of model coefficients is meaningful.
`...`	other arguments.

Value

Coefficients extracted from the model object `object`.

See Also

`fitted.values` and `residuals` for related methods; `glm`, `lm` for model fitting.

Examples

```
data(molecule)
molecule %v% "atomic type" <- c(1,1,1,1,1,1,2,2,2,2,2,2,2,3,3,3,3,3,3,3)
fit <- ergm(molecule ~ edges + nodefactor("atomic type"))
coef(fit)
```

`statnet-internal` *Internal statnet Objects*

Description

Internal ergm functions.

Details

Most of these are not to be called by the user (or in some cases are just waiting for proper documentation to be written :).

See Also

ergm, statnet-package

`ergm-package` *Statistical Modeling of Network and Graph Data*

Description

`ergm` is a collection of functions to plot, fit, diagnose, and simulate from random graph models. For a list of functions type: help(package='ergm')

For a complete list of the functions, use `library(help="ergm")` or read the rest of the manual. For a simple demonstration, use `demo(packages="ergm")`.

When publishing results obtained using this package the original authors are to be cited as:

Mark S. Handcock, David R. Hunter, Carter T. Butts, Steven M. Goodreau, and Martina Morris. 2003 *statnet: An R package for the Statistical Modeling of Social Networks* `http://www.csde.washington.edu/statnet`.

All programs derived from this package must cite it. For complete citation information, use `citation(package="ergm")`.

Details

Recent advances in the statistical modeling of random networks have had an impact on the empirical study of social networks. Statistical exponential family models (Strauss and Ikeda 1990) are a generalization of the Markov random network models introduced by Frank and Strauss (1986), which in turn derived from developments in spatial statistics (Besag, 1974). These models recognize the complex dependencies within relational data structures. To date, the use of stochastic network models for networks has been limited by three interrelated factors: the complexity of realistic models, the lack of simulation tools for inference and validation, and a poor understanding of the inferential properties of nontrivial models.

This manual introduces software tools for the representation, visualization, and analysis of network data that address each of these previous shortcomings. The package relies on the `network` package which allows networks to be represented in R. The `ergm` package allows maximum likelihood estimates of exponential random network models to be calculated using Markov Chain Monte Carlo. The package also provides tools for plotting networks, simulating networks and assessing model goodness-of-fit.

For detailed information on how to download and install the software, go to the `ergm` website: `http://www.csde.washington.edu/statnet`. A tutorial, support newsgroup, references and links to further resources are provided there.

Author(s)

Mark S. Handcock ⟨handcock@stat.washington.edu⟩,
David R. Hunter ⟨dhunter@stat.psu.edu⟩,
Carter T. Butts ⟨buttsc@uci.edu⟩,
Steven M. Goodreau ⟨goodreau@u.washington.edu⟩, and
Martina Morris ⟨morrism@u.washington.edu⟩

Maintainer: Mark S. Handcock ⟨handcock@stat.washington.edu⟩

References

Besag, J., 1974, Spatial interaction and the statistical analysis of lattice systems (with discussion), Journal of the Royal Statistical Society, B, 36, 192-236.

Frank, O., and Strauss, D.(1986). Markov graphs. Journal of the American Statistical Association, 81, 832-842.

Handcock, M. S., Hunter, D. R., Butts, C. T., Goodreau, S. M., and Morris, M. (2003), *statnet: An R package for the Statistical Modeling of Social Networks.*,
URL `http://www.csde.washington.edu/statnet`

Strauss, D., and Ikeda, M.(1990). Pseudolikelihood estimation for social networks. Journal of the American Statistical Association, 85, 204-212.

`ergm-terms` *Terms used in Exponential Family Random Graph Models*

Description

The function `ergm` is used to fit linear exponential random graph models, in which the probability of a given network, y, on a set of nodes is $\exp\{\theta{\cdot}g(y)\}/c(\theta)$, where $g(y)$ is a vector of network statistics for y, θ is a parameter vector of the same length and $c(\theta)$ is the normalizing constant for the distribution.

The network statistics $g(y)$ are entered as terms in the function call to `ergm`.

This page describes the possible terms (and hence network statistics).

Specifying models

Terms to `ergm` are specified by a formula to represent the network and network statistics. This is done via a `formula`, that is, an R formula object, of the form `y ~ <term 1> + <term 2> ...`, where `y` is a network object or a matrix that can be coerced to a network object, and `<term 1>`, `<term 2>`, etc, are each terms chosen from the list given below. To create a network object in R, use the `network` function, then add nodal attributes to it using the `%v%` operator if necessary.

Possible terms to represent network statistics

The `ergm` function allows the user to explore a large number of potential models for their network data. What follows is a list of model terms currently available by the program, and a brief description of each. In the formula for the model, the model terms are various function-like calls, some of which require arguments, separated by + signs.

Additional terms can be coded up by users via the `statnetuserterms` package.

The terms currently available are:

`absdiff(attrname)` *Absolute Difference:* The `attrname` argument is a character string giving the name of an attribute in the network's vertex attribute list. This term adds one network statistic to the model equaling the sum of `abs(attrname[i]-attrname[j])` for all edges (i,j) in the network.

`absdiffcat(attrname, base=NULL)` *Categorical Absolute Difference:* The `attrname` argument is a character string giving the name of an attribute in the network's vertex attribute list. This term adds one statistic for every possible nonzero distinct value of `abs(attrname[i]-attrname[j])` in the network; the value of each such statistic is the number of edges in the network with the corresponding absolute difference. The optional `base` argument is a vector indicating which nonzero differences, in order from smallest to largest, should be omitted from the model (i.e., treated like the zero-difference category). The `base` argument, if used, should contain indices, not differences themselves. For instance, if the possible values of `abs(attrname[i]-attrname[j])` are 0, 0.5, 3, 3.5, and 10, then to omit 0.5 and 10 one should set `base=c(1, 4)`.

`altkstar(lambda, fixed=FALSE)` *Alternating k-Star:* This term adds one network statistic to the model equal to a weighted alternating sequence of k-star statistics with weight parameter `lambda`. This is the version given in Snijders et al. (2006). We suggest using the `gwdegree` term instead. The `gwdegree` and `altkstar` produce mathematically equivalent models, as long as they are used together with the `edges` (or `kstar(1)`) term, yet the interpretation of the `gwdegree` parameters is slightly more straightforward than the interpretation of the `altkstar` parameters. For this reason, we recommend the use of the `gwdegree` instead of `altkstar`. See Section 3 and especially equation (13) of `http://www.sna.unimelb.edu.au/publications/cef4.pdf` for details. The optional argument `fixed` indicates whether the scale parameter `lambda` is to be fit as a curved exponential family model (see Hunter and Handcock, 2006). The default is `FALSE`, which means the scale parameter is not fixed and thus the model is a CEF model. This term can only be used with undirected networks.

`asymmetric` *Asymmetric Dyads:* This term adds one network statistic to the model, equaling the number of pairs of actors for which exactly one of $(i{\rightarrow}j)$ or $(j{\rightarrow}i)$ exists. This term can only be used with directed networks.

`b1concurrent(attrname)` *Concurrent node count for the first mode in a bipartite (aka two-mode) network:* This term adds one network statistic to the model, equal to the number of nodes in the first mode of the network with degree 2 or higher. The first mode of a bipartite network object is sometimes known as the "actor" mode. The optional term `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is the number of nodes in the first mode with ties to at least 2 other nodes with the same value for that attribute as the index node. This term can only be used with undirected networks.

`b1degree(d, attrname)` *Degree for the first mode in a bipartite (aka two-mode) network:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the number of nodes of degree `d[i]` in the first mode of a bipartite network, i.e. with exactly `d[i]` edges. The first mode of a bipartite network object is sometimes known as the "actor" mode. The optional term `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the degree count is the number of nodes with the same value of the attribute as the ego node. This term can only be used with undirected networks.

`b1factor(attrname, base=1)` *Factor Attribute Effect for the first mode in a bipartite (aka two-mode) network :* The `attrname` argument is a character string giving the name of a categorical attribute in the network's vertex attribute list. This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attrname` attribute. Each of these statistics gives the number of times a node with that attribute in the first mode of the network appears in an edge. The first mode of a bipartite network object is sometimes known as the "actor" mode. To include all attribute values is usually not a good idea, because the sum of all such statistics equals the number of edges and hence a linear dependency would arise in any model also including `edges`. Thus, the `base` argument tells which value(s), numbered in order according to the `sort` function, should be omitted. The default value, one, means that the smallest (i.e., first in sorted order) attribute value is omitted, making this value the reference category to which all other values are compared. For example, if the "fruit" factor has levels "orange", "apple", "banana", and "pear", then to add just two terms, one for "apple" and one for "pear", set "banana" and "orange" to the base (remember to sort the values first) by using `nodefactor("fruit", base=2:3)`. This term can only be used with undirected networks.

`b1star(k, attrname)` *k-Stars for the first mode in a bipartite (aka two-mode) network:* The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-stars whose center node is in the first mode of the network. The first mode of a bipartite network object is sometimes known as the "actor" mode. A k-star is defined to be a center node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $\{N, O_i\}$ exist for $i = 1, \dots, k$. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is over the number of k-stars (with center node in the first mode) where all nodes have the same value of the attribute. This term can only be used for undirected networks. Note that `b1star(1)` is equal to `b2star(1)` and to `edges`.

`b2concurrent(attrname)` *Concurrent node count for the second mode in a bipartite (aka two-mode) network:* This term adds one network statistic to the model, equal to the number of nodes in the second mode of the network with degree 2 or higher. The second mode of a bipartite network object is sometimes known as the "event" mode. The optional term `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is the number of nodes in the second mode with ties to at least 2 other nodes with the same value for that attribute as the index node. This term can only be used with undirected networks.

`b2degree(d, attrname)` *Degree for the second mode in a bipartite (aka two-mode) network:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the number of nodes of degree `d[i]` in the second mode of a bipartite network, i.e. with exactly `d[i]` edges. The second mode of a bipartite network object is sometimes known as the "event" mode. The optional term `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the degree count is the number of nodes with the same value of the attribute as the ego node. This term can only be used with undirected networks.

`b2factor(attrname, base=1)` *Factor Attribute Effect for the second mode in a bipartite (aka two-mode) network :* The `attrname` argument is a character string giving the name of a categorical attribute in the network's vertex attribute list. This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attrname` attribute. Each of these statistics gives the number of times a node with that attribute in the second mode of the network appears in an edge. The second mode of a bipartite network object is sometimes known as the "event" mode. To include all attribute values is usually not a good idea, because the sum of all such statistics equals the number of edges and hence a linear dependency would arise in any model also including `edges`. Thus, the `base` argument tells which value(s), numbered in order according to the `sort` function, should be omitted. The default value, one, means that the smallest (i.e., first in sorted order) attribute value is omitted, making this value the reference category to which all other values are compared. For example, if the "fruit" factor has levels "orange", "apple", "banana", and "pear", then to add just two terms, one for "apple" and one for "pear", set "banana" and "orange" to the base (remember to sort the values first) by using `nodefactor("fruit", base=2:3)`. This term can only be used with undirected networks.

`b2star(k, attrname)` *k-Stars for the second mode in a bipartite (aka two-mode) network:* The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-stars whose center node is in the second mode of the network. The second mode of a bipartite network object is sometimes known as the "event" mode. A k-star is defined to be a center node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $\{N, O_i\}$ exist for $i = 1, \dots, k$. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is over the number of k-stars (with center node in the second mode) where all nodes have the same value of the attribute. This term can only be used for undirected networks. Note that `b2star(1)` is equal to `b1star(1)` and to `edges`.

`bounded.degree(d, bound)` *Bounded Degree:* The `d` argument is a vector of distinct integers representing degrees. `bound` is a vector of upper bounds corresponding to each degree. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the minimum of `bound[i]` and the number of nodes in the network of degree exactly `d[i]`, i.e. with exactly `d[i]` edges. This term can only be used with undirected networks; for directed networks see `bounded.idegree` and `bounded.odegree`.

`bounded.idegree(d, bound)` *Bounded In-Degree:* The `d` argument is a vector of distinct integers representing in-degrees. `bound` is a vector of upper bounds corresponding to each degree. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the minimum of `bound[i]` and the number of nodes in the network of in-degree exactly `d[i]`, i.e.

with exactly `d[i]` in-edges. This term can only be used with directed networks; for undirected networks see `bounded.degree`.

`bounded.istar(k, bound)` ***Bounded In-Stars:*** The `k` argument is a vector of distinct integers. `bound` is a vector of upper bounds corresponding to each degree. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-instars in the network, where a k-instar is defined to be a node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $(N \leftarrow O_j)$ all exist for $j = 1, \dots, k$. The value is the minimum of this count and `bound[i]`. This term can only be used with directed networks; for undirected networks, see `bounded.kstar`.

`bounded.kstar(k, bound)` ***Bounded k-Stars:*** The `k` argument is a vector of distinct integers. `bound` is a vector of upper bounds corresponding to each degree. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-stars in the network, where a k-star is defined to be a node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the tie $\{N, O_j\}$ exists for $j = 1, \dots, k$. The value is the minimum of this count and `bound[i]`. This term can only be used with undirected networks; for directed networks, see `bounded.istar`, `bounded.ostar`.

`bounded.odegree(d, bound)` ***Bounded Out-Degree:*** The `d` argument is a vector of distinct integers representing in-degrees. `bound` is a vector of upper bounds corresponding to each degree. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the minimum of `bound[i]` and the number of nodes in the network of out-degree exactly `d[i]`, i.e. with exactly `d[i]` out-edges. This term can only be used with directed networks; for undirected networks see `bounded.degree`.

`bounded.ostar(k, bound)` ***Bounded Out-Stars:*** The `k` argument is a vector of distinct integers. `bound` is a vector of upper bounds corresponding to each degree. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-outstars in the network, where a k-instar is defined to be a node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $(N \rightarrow O_j)$ all exist for $j = 1, \dots, k$. The value is the minimum of this count and `bound[i]`. This term can only be used with directed networks; for undirected networks, see `bounded.kstar`.

`bounded.triangle(bound)` ***Bounded Triangles:*** This term adds one statistic to the model equal to the minimum of the bound and the number of triangles in the network. For an undirected network, a triangle is defined to be any set of three edges $\{i, j\}$, $\{j, k\}$, and $\{k, i\}$. For a directed network, a triangle is defined as any set of three edges $(i \rightarrow j)$, $(j \rightarrow k)$ and either $(k \rightarrow i)$ or $(i \rightarrow k)$.

`concurrent(attrname)` ***Concurrent node count:*** This term adds one network statistic to the model, equal to the number of nodes in the network with degree 2 or higher. The optional term `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is the number of nodes with ties to at least 2 other nodes with the same value for that attribute as the index node. This term can only be used with undirected networks.

`ctriple(attrname)` ***Cyclic Triples:*** This term adds one statistic to the model, equal to the number of cyclic triples in the network, defined as a set of edges of the form $\{(i \rightarrow j), (j \rightarrow k), (k \rightarrow i)\}$. Note that for all directed networks, `triangle` is equal to `ttriple+ctriple`, so at most two of these three terms can be in a model. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is over the number of cyclic triples where all three nodes have the same value of the attribute. This term can only be used with directed networks.

`cycle(k)` *Cycles:* The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`; the ith such statistic equals the number of cycles in the network with length exactly `k[i]`. The cycle statistic applies to both directed and undirected graphs. For directed networks, it counts directed cycles of length k, as opposed to undirected cycles in the undirected case. The directed cycle terms of lengths 2 and 3 are equivalent to `mutual` and `ctriple` (respectively). The undirected cycle term of length 3 is equivalent to `triangle`, and there is no undirected cycle term of length 2.

`degree(d, attrname)` *Degree:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the number of nodes in the network of degree `d[i]`, i.e. with exactly `d[i]` edges. The optional term `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the degree count is the number of nodes with the same value of the attribute as the ego node. This term can only be used with undirected networks; for directed networks see `idegree` and `odegree`.

`density` *Density:* This term adds one network statistic equal to the density of the network. For undirected networks, `density` equals `kstar(1)` or `edges` divided by $n(n-1)/2$; for directed networks, `density` equals `edges` or `istar(1)` or `ostar(1)` divided by $n(n-1)$.

`dsp(d)` *Dyadwise Shared Partners:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the number of dyads in the network with exactly `d[i]` shared partners. This term can be used with directed and undirected networks. For directed networks the count is over homogeneous shared partners only (i.e., only partners on a directed two-path connecting the nodes in the dyad and in the same direction).

`dyadcov(x, attrname)` *Dyadic Covariate:* If the network is directed, `x` is either a (symmetric) matrix of dyadic covariates, or an undirected network; if the latter, optional argument `attrname` provides the name of the edge attribute to use for edge values. This term adds three statistics to the model, representing the (polytomous) effect of the given covariate on the four possible dyad states (i.e., null, out-tie, in-tie, mutual). The statistics are the appearance of mutual, upper-triangular asymmetric, and lower-triangular asymmetric dyads (with the null state serving as a reference category). If the network is undirected, `x` is either a matrix of edgewise covariates, or a network; if the latter, optional argument `attrname` provides the name of the edge attribute to use for edge values. This term adds one statistic to the model, representing the effect of the given covariate on the appearance of edges. The `edgecov` and `dyadcov` terms are equivalent for undirected networks. `dyadcov` can be called more than once, to model the effects of multiple covariates.

`edgecov(x, attrname=NULL)` *Edge Covariate:* The `x` argument is either a matrix of edgewise covariates, or a network; if the latter, optional argument `attrname` provides the name of the edge attribute to use for edge values. This term adds one statistic to the model, representing the effect of the given covariate on the appearance of edges. The `edgecov` term applies to both directed and undirected networks. For undirected networks the covariates are also assumed to be undirected. The `edgecov` and `dyadcov` terms are equivalent for undirected networks. `edgecov` can be called more than once, to model the effects of multiple covariates.

`edges` *Edges:* This term adds one network statistic equal to the number of edges in the network. For undirected networks, `edges` is equal to `kstar(1)`; for directed networks, `edges` is equal to both `ostar(1)` and `istar(1)`.

`esp(d)` *Edgewise Shared Partners:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the number of edges in the network with exactly `d[i]` shared partners. This term can be used with directed and undirected networks. For directed networks the count is over homogeneous shared partners only (i.e., only partners on a directed two-path connecting the nodes in the edge in the same direction as the edge itself).

`gwb1degree(decay, fixed=FALSE)` *Geometrically Weighted Degree Distribution for the first mode in a bipartite (aka two-mode) network:* This term adds one network statistic to the model equal to the weighted degree distribution with weight parameter `decay`, for nodes in the first mode of a bipartite network. The first mode of a bipartite network object is sometimes known as the "actor" mode. This statistic is based on the version given as equation (14) in `http://www.sna.unimelb.edu.au/publications/cef4.pdf`. See the "Remark" in section 3 of that paper to see why it is used rather than the version given in Snijders et al. (2006). The optional argument `fixed` indicates whether the scale parameter `lambda` is to be fit as a curved exponential family model (see Hunter and Handcock, 2006). The default is `FALSE`, which means the scale parameter is not fixed and thus the model is a CEF model. This term can only be used with undirected networks.

`gwb2degree(decay, fixed=FALSE)` *Geometrically Weighted Degree Distribution for the second mode in a bipartite (aka two-mode) network:* This term adds one network statistic to the model equal to the weighted degree distribution with weight parameter `decay`, for nodes in the second mode of a bipartite network. The second mode of a bipartite network object is sometimes known as the "event" mode. This statistic is based on the version given as equation (14) in `http://www.sna.unimelb.edu.au/publications/cef4.pdf`. See the "Remark" in section 3 of that paper to see why it is used rather than the version given in Snijders et al. (2006). The optional argument `fixed` indicates whether the scale parameter `lambda` is to be fit as a curved exponential family model (see Hunter and Handcock, 2006). The default is `FALSE`, which means the scale parameter is not fixed and thus the model is a CEF model. This term can only be used with undirected networks.

`gwdegree(decay, fixed=FALSE)` *Geometrically Weighted Degree Distribution:* This term adds one network statistic to the model equal to the weighted degree distribution with weight parameter `decay`. This is the version given as equation (14) in `http://www.sna.unimelb.edu.au/publications/cef4.pdf`. See the "Remark" in section 3 of that paper to see why it is used rather than the version given in Snijders et al. (2006). The optional argument `fixed` indicates whether the scale parameter `lambda` is to be fit as a curved exponential family model (see Hunter and Handcock, 2006). The default is `FALSE`, which means the scale parameter is not fixed and thus the model is a CEF model. This term can only be used with undirected networks.

`gwdsp(alpha, fixed=FALSE)` *Geometrically Weighted Dyadwise Shared Partner Distribution:* This term adds one network statistic to the model equal to the geometrically weighted dyadwise shared partner distribution with weight parameter `alpha`. The optional argument `fixed` indicates whether the scale parameter `lambda` is to be fit as a curved exponential family model (see Hunter and Handcock, 2006). The default is `FALSE`, which means the scale parameter is not fixed and thus the model is a CEF model. This term can be used with directed and undirected networks. For directed networks the count is over homogeneous shared partners only (i.e., only partners on a directed two-path connecting the nodes in the dyad and in the same direction).

`gwesp(alpha, fixed=FALSE)` *Geometrically Weighted Edgewise Shared Partner Distribution:* This term adds one network statistic to the model equal to the geometrically weighted edgewise shared partner distribution with weight parameter `alpha`. The optional argument `fixed` indicates whether the scale parameter `lambda` is to be fit as a curved exponential family model (see Hunter and Handcock, 2006). The default is `FALSE`, which means the scale parameter is not fixed and thus the model is a CEF model. This term can be used with directed and undirected networks. For directed networks the geometric weighting is over homogeneous shared partners only (i.e., only partners on a directed two-path connecting the nodes in the edge in the same direction as the edge itself).

`gwidegree(decay, fixed=FALSE)` *Geometrically Weighted In-Degree Distribution:* This term adds one network statistic to the model equal to the weighted in-degree distribution with weight parameter `decay`. The optional argument `fixed` indicates whether the scale parameter `lambda` is to be fit as a curved exponential family model (see Hunter and Handcock, 2006). The default is `FALSE`, which means the scale parameter is not fixed and thus the model is a CEF model. This term can only be used with directed networks.

`gwodegree(decay, fixed=FALSE)` *Geometrically Weighted Out-Degree Distribution:* This term adds one network statistic to the model equal to the weighted out-degree distribution with weight parameter `decay`. The optional argument `fixed` indicates whether the scale parameter `lambda` is to be fit as a curved exponential family model (see Hunter and Handcock, 2006). The default is `FALSE`, which means the scale parameter is not fixed and thus the model is a CEF model. This term can only be used with directed networks.

`hamming(x)` *Hamming Distance:* This term adds one statistic to the model equal to the Hamming distance of the network from the network specified by `x`.

`idegree(d, attrname)` *In-Degree:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the number of nodes in the network of in-degree `d[i]`, i.e. the number of nodes with exactly `d[i]` in-edges. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count only considers edges in which both nodes have the same value of the attribute. This term can only be used with directed networks; for undirected networks see `degree`.

`isolates` *Isolates:* This term adds one statistic to the model equal to the number of isolates in the network. For an undirected network, an isolate is defined to be any node with degree zero. This term can only be used with undirected networks.

`istar(k, attrname)` *In-Stars:* The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-instars in the network, where a k-instar is defined to be a node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $(O_j \rightarrow N)$ exist for $j = 1, \dots, k$. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is over the number of k-instars where all nodes have the same value of the attribute. This term can only be used for directed networks; for undirected networks see `kstar`. Note that `istar(1)` is equal to both `ostar(1)` and `edges`.

`kstar(k, attrname)` *k-Stars:* The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-stars in the network, where a k-star is defined to be a node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $\{N, O_i\}$ exist for $i = 1, \dots, k$. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is over the number of k-stars where all nodes have the same value of the attribute. This term can only be used for undirected networks; for directed networks, see `istar`, `ostar`, `twopath` and `m2star`. Note that `kstar(1)` is equal to `edges`.

`localtriangle(x)` ***Triangles Within Neighborhoods:*** This term adds one statistic to the model equal to the number of triangles in the network between nodes "close to" each other. For an undirected network, a local triangle is defined to be any set of three edges between nodal pairs $\{(i,j),(j,k),(k,i)\}$ that are in the same neighborhood. For a directed network, a triangle is defined as any set of three edges $(i{\rightarrow}j)$, $(j{\rightarrow}k)$ and either $(k{\rightarrow}i)$ or $(k{\leftarrow}i)$ where again all nodes are within the same neighborhood. The argument `x` is a network or an adjacency matrix that specifies whether the two nodes are in the same neighborhood. Note that this is technically a special case of `triangle`.

`m2star` ***Mixed 2-Stars, a.k.a 2-Paths:*** This term can only be used with directed networks; for undirected networks see `kstar(2)`. This term adds one statistic to the model, equal to the number of mixed 2-stars in the network, defined as a pair of edges $(i{\rightarrow}j)$, $(j{\rightarrow}k)$. A mixed 2-star is sometimes called a 2-path because it is a directed path of length 2 from i to k via j. See also `twopath`.

`match(attrname, diff=FALSE)` ***Uniform Homophily and Differential Homophily:*** This is an alias for `nodematch(attrname, diff=FALSE)`.

`meandeg` ***Mean Vertex Degree:*** This term adds one network statistic to the model equal to the average degree of a node. Note that this term is a constant multiple of both `edges` and `density`.

`mutual` ***Mutuality:*** This term adds one network statistic to the model, equaling the number of pairs of actors i and j for which $(i{\rightarrow}j)$ and $(j{\rightarrow}i)$ both exist. This term can only be used with directed networks.

`nodefactor(attrname, base=1)` ***Factor Attribute Effect:*** The `attrname` argument is a character string giving the name of a categorical attribute in the network's vertex attribute list. This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attrname` attribute. Each of these statistics gives the number of times a node with that attribute appears in an edge in the network. In particular, for edges whose endpoints both have the same attribute value, this value is counted twice. To include all attribute values is usually not a good idea, because the sum of all such statistics equals twice the number of edges and hence a linear dependency would arise in any model also including `edges`. Thus, the `base` argument tells which value(s), numbered in order according to the `sort` function, should be omitted. The default value, one, means that the smallest (i.e., first in sorted order) attribute value is omitted, making this value the reference category to which all other values are compared. For example, if the "fruit" factor has levels "orange", "apple", "banana", and "pear", then to add just two terms, one for "apple" and one for "pear", then set "banana" and "orange" to the base (remember to sort the values first) by using `nodefactor("fruit", base=2:3)`. For an analogous term for quantitative vertex attributes, see `nodemain`.

`nodeifactor(attrname, base=1)` ***Factor Attribute Effect for in-edges:*** The `attrname` argument is a character string giving the name of a categorical attribute in the network's vertex attribute list. This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attrname` attribute. Each of these statistics gives the number of times a node with that attribute appears as the terminal node of a directed tie. The `base` argument tells which value(s), numbered in order according to the `sort` function, should be omitted. The default value, one, means that the smallest (i.e., first in sorted order) attribute value is omitted, making this value the reference category to which all other values are compared. For an example, see the `nodefactor` entry. The `nodeifactor` term may only be used with directed networks.

`nodemain(attrname)` *Main Effect of a Covariate:* The `attrname` argument is a character string giving the name of a quantitative (not categorical) attribute in the network's vertex attribute list. This term adds a single network statistic to the model equaling the sum of `attrname(i)` and `attrname(j)` for all edges (i, j) edges in the network. For categorical attributes, see `nodefactor`. Note that for directed networks, `nodemain` equals `receivercov` plus `sendercov`.

`nodematch(attrname, diff=FALSE)` *Uniform Homophily and Differential Homophily:* The `attrname` argument is a character string giving the name of an attribute in the network's vertex attribute list. When `diff=FALSE`, this term adds one network statistic to the model, which counts the number of edges (i, j) for which `attrname(i)==attrname(j)`. When `diff=TRUE`, p network statistics are added to the model, where p is the number of unique values of the `attrname` attribute. The kth such statistic counts the number of edges (i, j) for which `attrname(i) == attrname(j) == value(k)`, where `value(k)` is the kth smallest unique value of the attrname attribute.

`nodemix(attrname, contrast=FALSE)` *Nodal Attribute Mixing:* The `attrname` argument is a character string giving the name of a categorical attribute in the network's vertex attribute list. This term adds one network statistic to the model for each possible pairing of attribute values. The statistic equals the number of edges in the network in which the nodes have that pairing of values. In other words, this term produces one statistic for every entry in the mixing matrix for the attribute. The ordering of the attribute values is alphabetical. If the option `contrast=TRUE` is used, then a statistic for the first pairing is not included, making it the de facto reference category.

`nodeofactor(attrname, base=1)` *Factor Attribute Effect for out-edges:* The `attrname` argument is a character string giving the name of a categorical attribute in the network's vertex attribute list. This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attrname` attribute. Each of these statistics gives the number of times a node with that attribute appears as the node of origin of a directed tie. The `base` argument tells which value(s), numbered in order according to the `sort` function, should be omitted. The default value, one, means that the smallest (i.e., first in sorted order) attribute value is omitted, making this value the reference category to which all other values are compared. For an example, see the `nodefactor` entry. The `nodeofactor` term may only be used with directed networks.

`odegree(d, attrname)` *Out-Degree:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the number of nodes in the network of out-degree `d[i]`, i.e. the number of nodes with exactly `d[i]` out-edges. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count only considers edges in which both nodes have the same value of the attribute. This term can only be used with directed networks; for undirected networks see `degree`.

`ostar(k, attrname)` *k-Outstars:* The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-outstars in the network, where a k-outstar is defined to be a node N and a set of k different nodes $\{O_1, \ldots, O_k\}$ such that the ties $(N \to O_j)$ exist for $j = 1, \ldots, k$. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is the number of k-outstars where all nodes have the same value of the attribute. This term can only be used with directed networks; for undirected networks see `kstar`. Note that `ostar(1)` is equal to both `istar(1)` and `edges`.

`receiver` *Receiver Effect:* This term adds one network statistic for each node equal to the number of in-ties for that node. This measures the popularity of the node. The term for the first node is omitted because of redundancy, but the coefficient can be computed as the negative of the sum of the coefficients of all the other actors. That is, the average coefficient is zero, following the Holland-Leinhardt parametrization of the p_1 model. This term can only be used with a directed network. For undirected networks, see `sociality`.

`receivercov(attrname)` *Receiver Covariate Effect:* The `attrname` argument is a character string giving the name of an attribute in the network's vertex attribute list that takes numeric values. This term adds one network statistic to the model equaling the sum of the attribute values of the receivers of all ties. This term can only be used with a directed network. For undirected networks, see `nodemain`.

`sender` *Sender Effect:* This term adds one network statistic for each node equal to the number of out-ties for that node. This measures the activity of the node. The term for the first node is omitted because of redundancy, but the coefficient can be computed as the negative of the sum of the coefficients of all the other actors. That is, the average coefficient is zero, following the Holland-Leinhardt parametrization of the p_1 model. This term can only be used with a directed network. For undirected networks, see `sociality`.

`sendercov(attrname)` *Sender Covariate Effect:* The `attrname` argument is a character string giving the name of an attribute in the network's vertex attribute list that takes numeric values. This term adds one network statistic to the model equaling the sum of the attribute values of the senders of all ties. This term can only be used with a directed network. For undirected networks, see `nodemain`.

`smalldiff(attrname, cutoff)` *Small Difference:* The `attrname` argument is a character string giving the name of an attribute in the network's vertex attribute list and `cutoff` is any real number. This term adds one network statistic to the model, equal to the number of edges (i, j) for which `abs(attrname(i)-attrname(j))` is less than or equal to `cutoff`.

`sociality(attrname)` *Centralized Covariate Effect:* This term adds one network statistic for each node equal to the number of ties of that node. The optional `attrname` is a character string giving the name of an attribute in the network's vertex attribute list that takes categorical values. If provided, this term only counts ties between nodes with the same value of the attribute. This term can only be used with undirected networks. For directed networks, see `sender` and `receiver`.

`triangle(attrname)` *Triangles:* This term adds one statistic to the model equal to the number of triangles in the network. For an undirected network, a triangle is defined to be any set $\{(i, j), (j, k), (k, i)\}$ of three edges. For a directed network, a triangle is defined as any set of three edges $(i{\rightarrow}j)$ and $(j{\rightarrow}k)$ and either $(k{\rightarrow}i)$ or $(k{\leftarrow}i)$. Note that for directed networks, `triangle` equals `ttriple` plus `ctriple`, so at most two of these three terms can be in a model. The optional argument `attrname` restricts the count to those triples of nodes with equal values of the vertex attribute specified by `attrname`.

`tripercent(attrname)` *Triangle Percentage:* This term adds one statistic to the model equal to the percentage of triangles in the network relative to the number of potential triangles. For the definition of triangle, see `triangle`. A potential triangle is a 2-star. The optional argument `attrname` restricts the counts (both numerator and denominator) to those triples of nodes with equal values of the vertex attribute specified by `attrname`. This term can only be used with undirected networks.

`ttriple(attrname)` *Transitive Triples:* This term adds one statistic to the model, equal to the number of transitive triples in the network, defined as a set of edges $\{(i{\rightarrow}j), (j{\rightarrow}k), (i{\rightarrow}k)\}$. Note that `triangle` equals `ttriple+ctriple` for a directed network, so at most two of the three terms can be in a model. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is over the number of transitive triples where all three nodes have the same value of the attribute. This term can only be used with a directed network.

`twopath` *2-Paths:* This term adds one statistic to the model, equal to the number of 2-paths in the network. For directed network this is defined as a pair of edges $(i{\rightarrow}j), (j{\rightarrow}k)$. That is, it is a directed path of length 2 from i to k via j. For directed networks a 2-path is also a mixed 2-star. For undirected networks this is defined as a pair of edges $\{i, j\}, \{j, k\}$. That is, it is an undirected path of length 2 from i to k via j, also known as a 2-star.

References

Hunter, D. R. and M. S. Handcock (2006), Inference in curved exponential family models for networks, Journal of Computational and Graphical Statistics, 15: 565-583.

Hunter, D. R. (2007), Curved exponential family models for social networks, Social Networks, 29: 216-230.

Snijders, T. A. B., P. E. Pattison, G. L. Robins, and M. S. Handcock (2006), New specifications for exponential random graph models, Sociological Methodology, 36(1): 99-153.

See Also

ergm, network, %v%, %n%, summary.ergm, print.ergm

Examples

```
## Not run:
ergm(flomarriage ~ kstar(1:2) + absdiff("wealth") + triangle)

ergm(molecule ~ edges + kstar(2:3) + triangle
                      + nodematch("atomic type",diff=TRUE)
                      + triangle + absdiff("atomic type"))
## End(Not run)
```

`ergm` *Exponential Family Random Graph Models*

Description

`ergm` is used to fit linear exponential random network models, in which the probability of a given network, y, on a set of nodes is $\exp(\theta{\cdot}g(y))/c(\theta)$, where $g(y)$ is a vector of network statistics, θ is a parameter vector of the same length and $c(\theta)$ is the normalizing constant for the distribution. `ergm` can return either a maximum pseudo-likelihood estimate or an approximate maximum likelihood estimator based on a Monte Carlo scheme.

Usage

```
ergm(formula, theta0="MPLE",
     MPLEonly=FALSE, MLestimate=!MPLEonly, seed=NULL,
     burnin=10000, MCMCsamplesize=10000, interval=100, maxit=3,
     constraints=~.,
     control=ergm.control(),
     verbose=FALSE, ...)
```

Arguments

`formula`	formula; an R `formula` object, of the form `y ~ <model terms>`, where `y` is a `network` object or a matrix that can be coerced to a `network` object. For the details on the possible `<model terms>`, see `ergm-terms`. To create a `network` object in R, use the `network()` function, then add nodal attributes to it using the `%v%` operator if necessary.
`theta0`	vector; the parameter value used to generate the MCMC sample and as a starting value for the estimation. By default the MPLE is used (`theta0="MPLE"`).
`MPLEonly`	logical; `TRUE` if the maximum pseudo-likelihood estimate is to be computed and returned. Note that `MPLEonly=TRUE` will render moot most other parameters in this list.
`MLestimate`	logical; `TRUE` if only the Monte Carlo maximum likelihood estimate is to be computed and returned.
`burnin`	count; the number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.
`MCMCsamplesize`	count; the number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm.
`interval`	count; the number of proposals between sampled statistics.
`maxit`	count; the number of times the parameter for the MCMC should be updated by maximizing the MCMC likelihood. At each step the parameter is changed to the values that maximizes the MCMC likelihood based on the current sample.
`constraints`	A one-sided formula specifying one or more constraints on the support of the distribution of the networks being modeled, using syntax similar to the `formula` argument. Multiple constraints may be given, separated by "+" operators. Together with the model terms in the formula, the constraints define the distribution of networks being modeled. It is also possible to specify a proposal function directly by passing a string with the function's name. In that case, arguments to the proposal should be specified through the `prop.args` argument to `ergm.control`. The default is `~.`, for an unconstrained model. The constraint terms currently implemented are **`.` or `NULL`** A placeholder for no constraints: all networks of a particular size and type have non-zero probability. Cannot be combined with other constraints.

bd(attribs,maxout,maxin,minout,minin) Constrain maximum and minimum vertex degree. See "Placing Bounds on Degrees" section for more information.

degrees and nodedegrees Preserve the degree of each vertex of the given network: only networks whose vertex degrees are the same as those in the network passed in the model formula have non-zero probability.

degreedist Preserve the degree distribution of the given network: only networks whose degree distributions are the same as those in the network passed in the model formula have non-zero probability.

indegreedist and outdegreedist Preserve the (respectively) indegree or outdegree distribution of the given network.

edges Preserve the edge count of the given network: only networks having the same number of edges as the network passed in the model formula have non-zero probability.

Not all combinations of the above are supported.

`control`	A list of control parameters for algorithm tuning. Constructed using `ergm.control`.
`seed`	integer; random number integer seed. Defaults to `NULL` to use whatever the state of the random number generator is at the time of the call.
`verbose`	logical; if this is `TRUE`, the program will print out additional information, including goodness of fit statistics.
`...`	Additional arguments, to be passed to lower-level functions in the future.

Value

`ergm` returns an object of class `ergm` that is a list consisting of the following elements:

`coef`	The Monte Carlo maximum likelihood estimate of θ, the vector of coefficients for the model parameters.
`sample`	The $n \times p$ matrix of network statistics, where n is the sample size and p is the number of network statistics specified in the model, that is used in the maximum likelihood estimation routine.
`iterations`	The number of Newton-Raphson iterations required before convergence.
`MCMCtheta`	The value of θ used to produce the Markov chain Monte Carlo sample. As long as the Markov chain mixes sufficiently well, `sample` is roughly a random sample from the distribution of network statistics specified by the model with the parameter equal to `MCMCtheta`. If `MPLEonly=TRUE` then `MCMCtheta` equals the MPLE.
`loglikelihood`	The approximate log-likelihood for the MLE. The value is only approximate because it is estimated based on the MCMC random sample.
`gradient`	The value of the gradient vector of the approximated loglikelihood function, evaluated at the maximizer. This vector should be very close to zero.
`hessian`	The Hessian matrix of the approximated loglikelihood function, evaluated at the maximizer. This matrix may be inverted to give an approximate covariance matrix for the MLE.

`samplesize`	The size of the MCMC sample
`formula`	The original `formula` entered into the `ergm` function.
`statsmatrix`	If the option `returnMCMCstats=TRUE`, this is the the matrix of change statistics from the MCMC run.
`newnetwork`	The network generated at the end of the MCMC sampling.
`proposal`	The structure containing information about the Metropolis-Hasting proposal used.

See the method `print.ergm` for details on how an `ergm` object is printed. Note that the method `summary.ergm` returns a summary of the relevant parts of the `ergm` object in concise summary format.

Model Terms

The `ergm` function allows the user to explore a large number of potential models for their network data. The terms currently supported by the program, and a brief description of each is given in the documentation `ergm-terms`. In the `formula` for the model, the model terms are various function-like calls, some of which require arguments, separated by + signs. See `ergm-terms` for details.

Notes on model specification

Although each of the statistics in a given model is a summary statistic for the entire network, it is rarely necessary to calculate statistics for an entire network in a proposed Metropolis-Hastings step. Thus, for example, if the triangle term is included in the model, a census of all triangles in the observed network is never taken; instead, only the change in the number of triangles is recorded for each edge toggle.

In the implementation of `ergm`, the model is initialized in R, then all the model information is passed to a C program that generates the sample of network statistics using MCMC. This sample is then returned to R, which implements a simple Newton-Raphson algorithm to approximate the MLE. An alternative style of maximum likelihood estimation is to use a stochastic approximation algorithm. This can be chosen with the `algorithm.control(style="Robbins-Monro")` option.

The default mechanism for proposing new networks for the MCMC sample space is the Metropolis-Hastings algorithm, which simply chooses a dyad at random and proposes to toggle that edge; each possible dyad is equally likely. The `proposaltype` option allow many more complex proposals to be specified. We have developed and implemented a wide range of algorithms. These are described in the documentation for `proposaltype`. For example, we have included proposal functions that condition on maintaining the absolute degree distribution for the observed network. Each proposal network will have exactly the same number of nodes with each degree as does the original network; this means that if the proposal network removes an edge between a node of degree 3 and a node of degree 5, it must also add an edge between a node of degree 2 and a node of degree 4. Note that one or both of the latter nodes may be the same as the former nodes.

The package is designed so that the user can add additional proposal types.

Placing Bounds on Degrees:

There are many times when one may wish to condition on the number of inedges or outedges possessed by a node, either as a consequence of some intrinsic property of that node (e.g., to control for activity or popularity processes), to account for known outliers of some kind, and thus we wish to limit its indegree, an intrinsic property of the sampling scheme whence came our data (e.g., the survey asked everyone to name only three friends total) or as a function of the attributes of the nodes to which a node has edges (e.g., we specify that nodes designated "male" have a maximum number of outdegrees to nodes designated "female"). To accomplish this we use the `constraints` term `bd`.

Let's consider the simple cases first. Suppose you want to condition on the total number of degrees regardless of attributes. That is, if you had a survey that asked respondents to name three alters and no more, then you might want to limit your maximal outdegree to three without regard to any of the alters' attributes. The argument is then:

`constraints=~bd(maxout=3)`

Similar calls are used to restrict the number of indegrees (`maxin`), the minimum number of outdegrees (`minout`), and the minimum number of indegrees (`minin`).

You can also set ego specific limits. For example:

`constraints=bd(maxout=rep(c(3,4),c(36,35)))`

limits the first 36 to 3 and the other 35 to 4 outdegrees.

Multiple restrictions can be combined. `bd` is very flexible. In general, the `bd` term can contain up to five arguments:

```
    bd(attribs=attribs,
       maxout=maxout,
       maxin=maxin,
       minout=minout,
       minin=minin)
```

Omitted arguments are unrestricted, and arguments of length 1 are replicated out to all nodes (as above). If an individual entry in `maxout`,..., `minin` is `NA` then no restriction of that kind is applied to that actor.

In general, `attribs` is a matrix of the attributes on which we are conditioning. The dimensions of `attribs` are `n_nodes` rows by `attrcount` columns, where `attrcount` is the number of distinct attribute values on which we want to condition (i.e., a separate column is required for "male" and "female" if we want to condition on the number of ties to both "male" and "female" partners). The value of `attribs[n, i]`, therefore, is `TRUE` if node `n` has attribute value i, and `FALSE` otherwise. (Note that, since each column represents only a single value of a single attribute, the values of this matrix are all Boolean (`TRUE` or `FALSE`).) It is important to note that `attribs` is a matrix of nodal attributes, not alter attributes.

So, for instance, if we wanted to construct an `attribs` matrix with two columns, one each for male and female attribute values (we are conditioning on these values of the attribute "sex"), and the attribute sex is represented in ads.sex as an `n_node`-long vector of 0s and 1s (men and women), then our code would look as follows:

```
 # male column: bit vector, TRUE for males
 attrsex1 <- (ads.sex == 0)
 # female column: bit vector, TRUE for females
 attrsex2 <- (ads.sex == 1)
 # now create attribs matrix
 attribs <- matrix(ncol=2,nrow=71, data=c(attrsex1,attrsex2))
```

`maxout` is a matrix of alter attributes, with the same dimensions as the `attribs` matrix. `maxout` is `n_nodes` rows by `attrcount` columns. The value of `maxout[n,i]`, therefore, is the maximum number of outdegrees permitted from node `n` to nodes with the attribute `i` (where a `NA` means there is no maximum).

For example: if we wanted to create a `maxout` matrix to work with our `attribs` matrix above, with a maximum from every node of five outedges to males and five outedges to females, our code would look like this:

```
# every node has maximum of 5 outdegrees to male alters
maxoutsex1 <- c(rep(5,71))
# every node has maximum of 5 outdegrees to female alters
maxoutsex2 <- c(rep(5,71))
# now create maxout matrix
maxout <- cbind(maxoutsex1,maxoutsex2)
```

The `maxin`, `minout`, and `minin` matrices are constructed exactly like the `maxout` matrix, except for the maximum allowed indegree, the minimum allowed outdegree, and the minimum allowed indegree, respectively. Note that in an undirected network, we only look at the outdegree matrices; `maxin` and `minin` will both be ignored in this case.

```
attribs[n][0] = 1 # just the ego values
maxout[n][0] = minout[n][0] = observed outdegree of n in network
maxin[n][0] = minin[n][0] = observed indegree of n in network
```

Dealing with degeneracy

In order to begin the process of estimating network coefficients, we need starting values - guesses at the true values of the network statistic coefficients. The default is to begin the MCMC estimation process at the deterministic MPLE values. These values are often taken as good-enough final answers by many other applications. However recent work has indicated that they are sub-optimal and can be dramatically bad.

In using the MPLE values, MCMC MLE often runs into problems caused by the inherent instability of the natural parameter space of the models (Handcock 2000, 2002, 2003). If the initial values for the parameter coefficients are off by a very small amount in the wrong direction, the result is often a sample of networks that are degenerate - that is, networks that are entirely full or entirely empty, or that are otherwise less-than-representative of the sample of network space our process is attempting to explore. (In part, this is an indication of why one should not rely solely on the MPLE). The package contains many algorithmic tools to obtain quality inference. If the sample of networks is degenerate, our algorithm will fail in its calculation of an MLE for our data (usually in constructing the Hessian matrix). See the references for details, especially Handcock (2003) and Hunter and Handcock (2006).

References

Boer, P., Huisman, M., Snijders, T.A.B., and Zeggelink, E.P.H. (2003). *StOCNET: an open software system for the advanced statistical analysis of social networks.* Version 1.4. Groningen: ProGAMMA / ICS

Handcock, M.S. (2000) *Progress in Statistical Modeling of Drug User and Sexual Networks*, Center for Statistics and the Social Sciences, University of Washington.

Handcock, M. S. (2002) *Degeneracy and inference for social network models* Paper presented at the Sunbelt XXII International Social Network Conference in New Orleans, LA.

Handcock, M. S. (2003) *Assessing Degeneracy in Statistical Models of Social Networks*, Working Paper #39, Center for Statistics and the Social Sciences, University of Washington. www.csss.washington.edu/Papers/wp39.pdf

Hunter, D. R. and Handcock, M. S. (2006) *Inference in curved exponential family models for networks*, Journal of Computational and Graphical Statistics.

See Also

network, %v%, %n%, ergm-terms, summary.ergm, print.ergm

Examples

```
#
# load the Florentine marriage data matrix
#
data(flo)
#
# attach the sociomatrix for the Florentine marriage data
# This is not yet a network object.
#
flo
#
# Create a network object out of the adjacency matrix
#
flomarriage <- network(flo,directed=FALSE)
flomarriage
#
# print out the sociomatrix for the Florentine marriage data
#
flomarriage[,]
#
# create a vector indicating the wealth of each family (in thousands of lira)
# and add it as a covariate to the network object
#
flomarriage %v% "wealth" <- c(10,36,27,146,55,44,20,8,42,103,48,49,10,48,32,3)
flomarriage
#
# create a plot of the social network
#
plot(flomarriage)
#
# now make the vertex size proportional to their wealth
#
plot(flomarriage, vertex.cex="wealth", main="Marriage Ties")
#
# Use 'data(package = "ergm")' to list the data sets in a
#
```

```
data(package="ergm")
#
# Load a network object of the Florentine data
#
data(florentine)
#
# Fit a model where the propensity to form ties between
# families depends on the absolute difference in wealth
#
gest <- ergm(flomarriage ~ edges + absdiff("wealth"))
summary(gest)
#
# add terms for the propensity to form 2-stars and triangles
# of families
#
gest <- ergm(flomarriage ~ kstar(1:2) + absdiff("wealth") + triangle)
summary(gest)

# import synthetic network that looks like a molecule
data(molecule)
# Add a attribute to it to mimic the atomic type
molecule %v% "atomic type" <- c(1,1,1,1,1,1,2,2,2,2,2,2,2,3,3,3,3,3,3,3)
#
# create a plot of the social network
# colored by atomic type
#
plot(molecule, vertex.col="atomic type",vertex.cex=3)

# measure tendency to match within each atomic type
gest <- ergm(molecule ~ edges + kstar(2) + triangle + nodematch("atomic type"),
  MCMCsamplesize=10000)
summary(gest)

# compare it to differential homophily by atomic type
gest <- ergm(molecule ~ edges + kstar(2) + triangle
                              + nodematch("atomic type",diff=TRUE),
  MCMCsamplesize=10000)
summary(gest)
```

---

ergm.control *Auxiliary for Controlling ERGM Fitting*

Description

Auxiliary function as user interface for fine-tuning 'ergm' fitting.

Usage

```
ergm.control(prop.weights = "default", prop.args = NULL,
              prop.weights.diss = "default", prop.args.diss = NULL, nr.maxit =
              100, calc.mcmc.se = TRUE, hessian = FALSE, compress = FALSE,
              maxNumDyadTypes = 10000, maxedges = 20000, maxchanges = 1e+06,
              MPLEsamplesize = 50000, MPLEtype=c("glm", "penalized"), trace = 0,
              steplength = 0.5, drop = TRUE, force.mcmc = FALSE, mcmc.precision =
              0.05, metric = c("Likelihood", "raw"), method = c("BFGS", "Nelder-Meac
              trustregion = 20, initial.loglik = NULL,
              style = c("Newton-Raphson", "Robbins-Monro",
              "Stochastic-Approximation"), phase1_n = NULL, initial_gain = NULL,
              nsubphases = "maxit", niterations = NULL, phase3_n = NULL,
              RobMon.phase1n_base = 7, RobMon.phase2n_base = 7, RobMon.phase2sub
              = 4, RobMon.init_gain = 0.4, RobMon.phase3n = 500, dyninterval =
              1000, parallel = 0, returnMCMCstats = TRUE)
```

Arguments

`prop.weights`	Specifies the method to allocate probabilities of being proposed to dyads. Defaults to `"default"`, which picks a reasonable default for the specified constraint. Possible values are `"TNT"`, `"random"`, and `"nonobserved"`, though not all values may be used with all possible constraints (in the `ergm` function).
`prop.args`	An alternative, direct way of specifying additional arguments to proposal.
`prop.weights.diss`	As `prop.weights`, for dissolution model.
`prop.args.diss`	As `prop.args`, for dissolution model.
`nr.maxit`	count; The maximum number of iterations in the Newton-Raphson optimization. Defaults to `100`. `maxit` gives the total number of likelihood function evaluations.
`calc.mcmc.se`	logical; should the contribution to the standard errors of the estimator incurred by the MCMC sampling be computed. Default is `TRUE`.
`hessian`	logical; Should the Hessian matrix of the likelihood function be computed. Default is `TRUE`.
`compress`	logical; Should the matrix of sample statistics returned be compressed to the set of unique statistics with a column of frequencies post-pended. This also uses a compression algorithm in the computation of the maximum psuedo-likelihood estimate that will dramatically speed it for large networks. Default is `FALSE`.
`maxNumDyadTypes`	count; The maximum number of unique pseudolikelihood change statistics to be allowed if `compress=TRUE`. It is only relevant in that case. Default is `10000`.
`maxedges`	Maximum number of edges for which to allocate space.
`maxchanges`	Maximum number of changes in dynamic network simulation for which to allocate space.
`MPLEsamplesize`	count; the sample size to use for endogenous sampling in the pseudolikelihood computation. Default is `50000`.

`MPLEtype`	one of "glm" or "penalized"; method to use for psuedolikelihood. "glm" is the usual formal logistic regression. "penalized" uses the bias-reduced method of Firth (1983) as originally implemented by Meinhard Ploner, Daniela Dunkler, Harry Southworth, and Georg Heinze in the "logistf" package. Default is "glm".
`trace`	non-negative integer; If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information: for method `"L-BFGS-B"` there are six levels of tracing. (To understand exactly what these do see the source code for `optim`: higher levels give more detail.)
`steplength`	Multiplier for step length, to make fitting more stable at the cost of efficiency.
`drop`	logical; Should the degenerate terms in the model be dropped from the fit? If statistics occur on the extreme of their range they correspond to infinite parameter estimates. Default is `TRUE`.
`force.mcmc`	logical; should MCMC maximum likelihood be used? Only relevant for dyadic independent networks, in which the MLE could be found using MPLE instead.
`mcmc.precision`	vector; upper bounds on the precision of the standard errors induced by the MCMC algorithm. Defaults to `0.05`.
`metric`	character; The name of the optimization metric to use. Defaults to `"Likelihood"`.
`method`	character; The name of the optimization method to use. See `optim` for the options. The default method `"BFGS"` is a quasi-Newton method (also known as a variable metric algorithm). It is attributed to Broyden, Fletcher, Goldfarb and Shanno. This uses function values and gradients to build up a picture of the surface to be optimized.
`trustregion`	numeric; The maximum amount the algorithm will allow the approximated likelihood to be increased at a given iteration. Defaults to 20. See Boer, Huisman, Snijders, and Zeggelink (2003) for details.
`initial.loglik`	Initial value of loglikelihood, if known.
`style`	character; The style of maximum likelihood estimation to use. The default is optimization of an MCMC estimate of the log-likelihood. An alternative is to use a form of stochastic approximation (`"Robbins-Monro"`). The direct use of the likelihood function has many theoretical advantages over stochastic approximation, but the choice will depend on the model and data being fit. See Handcock (2000) and Hunter and Handcock (2006) for details.
`phase1_n`	count; The number of MCMC samples to draw in Phase 1 of the stochastic approximation algorithm. Defaults to 7 plus 3 times the number of terms in the model. See Boer, Huisman, Snijders, and Zeggelink (2003) for details.
`initial_gain`	numeric; The initial gain to Phase 2 of the stochastic approximation algorithm. Defaults to 0.1. See Boer, Huisman, Snijders, and Zeggelink (2003) for details.
`nsubphases`	count; The number of sub-phases in Phase 2 of the stochastic approximation algorithm. Defaults to `maxit`. See Boer, Huisman, Snijders, and Zeggelink (2003) for details.
`niterations`	count; The number of MCMC samples to draw in Phase 2 of the stochastic approximation algorithm. Defaults to 7 plus the number of terms in the model. See Boer, Huisman, Snijders, and Zeggelink (2003) for details.

`phase3_n`	count; The sample size for the MCMC sample in Phase 3 of the stochastic approximation algorithm. Defaults to 1000. See Boer, Huisman, Snijders, and Zeggelink (2003) for details.
`RobMon.phase1n_base`	Robbins-Monro control parameter
`RobMon.phase2n_base`	Robbins-Monro control parameter
`RobMon.phase2sub`	Robbins-Monro control parameter
`RobMon.init_gain`	Robbins-Monro control parameter
`RobMon.phase3n`	Robbins-Monro control parameter
`returnMCMCstats`	logical; If this is `TRUE` (the default) the matrix of change statistics from the MCMC run is returned as component `sample`. This matrix is actually an object of class `mcmc` and can be used directly in the `CODA` package to assess MCMC convergence.
`dyninterval`	Number of Metropolis-Hastings proposal for each phase in the dynamic network simulation.
`parallel`	Number of threads in which to run the sampling.

Value

A list with arguments as components.

See Also

`ergm`, `glm.control` performs a similar function for `glm`

`ergmuserterms-package` *Add Statistics Terms for the 'ergm' Package*

Description

The `ergm` package is capable of fitting a wide range of exponential random network models, in which the probability of a given network, y, on a set of nodes is $\exp(\theta \cdot g(y))/c(\theta)$, where $g(y)$ is a vector of network statistics, θ is a parameter vector of the same length and $c(\theta)$ is the normalizing constant for the distribution. The `ergm` function fits these models when they are expressed via an R `formula` object, of the form `y ~ <model terms>`, where `y` is a network object or a matrix that can be coerced to a network object. To create a network object in R, use the `network()` function, then add nodal attributes to it using the `%v%` operator if necessary.

The `ergm` package contains a wide range of terms. For the details on the possible `<model terms>`, see `ergm-terms`.

This package can be modified by users to add user-defined terms to `ergm` models. The terms can be used throughout the `ergm` package and behave identically to the supplied terms.

Details

The `ergmuserterms` package is available from the `statnet` website (`http://csde.washington.edu/statnet`).

The code contains some simple examples and templates. These include:

`m2star` *Mixed 2-stars, a.k.a. 2-paths.* This option can only be specified with a directed network; for undirected graphs see `kstar(2)`. This option adds one statistic to the model, equal to the number of mixed-2-stars in the network, defined as a pair of edges $\{(i{\to}j), (j{\to}k)\}$.

`testme` *A clone of Edges.* This is included for purposes of an example. This option adds one graph statistic equal to the number of edges in the graph. For undirected graphs, `edges` is isomorphic to `kstar(1)`; for directed networks, `edges` is isomorphic to both `ostar(1)` and `istar(1)`.

In the implementation of `ergm`, the model is initialized in R, then all the model information is passed to a `C` program that generates the sample of graph statistics using MCMC. This sample is then returned to R, which then approximates the MLE.

See Also

ergm, network, ergm-terms

Examples

```
## Not run:
library(ergmuserterms)
data(sampson)
monk.fit <- ergm(samplike~m2star)
summary(monk.fit)

monk.fit <- ergm(samplike ~ m2star + testme)
summary(monk.fit)
## End(Not run)
```

`faux.magnolia.high` *Goodreau's Faux Magnolia High School as a network object*

Description

This data set represents a simulation of an in-school friendship network. The network is named faux.magnolia.high because the school commnunities on which it is based are large and located in the southern US.

Usage

```
data(faux.magnolia.high)
```

Format

`faux.magnolia.high` is a `network` object with 1461 vertices (students, in this case) and 974 undirected edges (mutual friendships). To obtain additional summary information about it, type `summary(faux.magnolia.high)`.

The vertex attributes are `Grade`, `Sex`, and `Race`. The `Grade` attribute has values 7 through 12, indicating each student's grade in school. The `Sex` attribute has values 1 for male and 2 for female. The `Race` attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes values 1 through 6: 1 = White (non-Hisp.); 2 = Black (non-Hisp.); 3 = Hispanic; 4 = Asian (non-Hisp.); 5 = Native American (non-Hisp.); and 6 = Other (non-Hisp.)

Licenses and Citation

If the source of the data set does not specified otherwise, this data set is protected by the Creative Commons License `http://creativecommons.org/licenses/by-nc-nd/2.5/`.

When publishing results obtained using this data set, the original authors (Resnick et al, 1997) should be cited. In addition this package should be cited as:

Mark S. Handcock, David R. Hunter, Carter T. Butts, Steven M. Goodreau, and Martina Morris. 2003 *statnet: An R package for the Statistical Modeling of Social Networks*
`http://www.csde.washington.edu/statnet`.

Source

The data set is based upon a model fit to data from two school communities from the AddHealth Study, Wave I (Resnick et al., 1997). It was constructed as follows:

The two schools in question (a junior and senior high school in the same community) were combined into a single network dataset. Students who did not take the AddHealth survey or who were not listed on the schools' student rosters were eliminated, then an undirected link was established between any two individuals who both named each other as a friend. All missing race, grade, and sex values were replaced by a random draw with weights determined by the size of the attribute classes in the school.

The following `ergm` model was fit to the original data:

```
magnolia.fit <- ergm (magnolia ~ edges + nodematch("Grade",diff=T)
 + nodematch("Race",diff=T) + nodematch("Sex",diff=F)
 + absdiff("Grade") + gwesp(0.25,fixed=T), burnin=10000,
 interval=1000, MCMCsamplesize=2500, maxit=25,
 control=ergm.control(steplength=0.25))
```

Then the faux.magnolia.high dataset was created by simulating a single network from the above model fit:

```
faux.magnolia.high <- simulate (magnolia.fit, nsim=1, burnin=100000000,
 constraint = "edges")
```

References

Resnick M.D., Bearman, P.S., Blum R.W. et al. (1997). *Protecting adolescents from harm. Findings from the National Longitudinal Study on Adolescent Health*, *Journal of the American Medical Association*, 278: 823-32.

See Also

`network`, `plot.network`, `ergm`

`faux.mesa.high` *Goodreau's Faux Mesa High School as a network object*

Description

This data set (formerly called "fauxhigh") represents a simulation of an in-school friendship network. The network is named `faux.mesa.high` because the school commnunity on which it is based is in the rural western US, with a student body that is largely Hispanic and Native American.

Usage

```
data(faux.mesa.high)
```

Format

`faux.mesa.high` is a `network` object with 205 vertices (students, in this case) and 203 undirected edges (mutual friendships). To obtain additional summary information about it, type `summary(faux.mesa.high)`

The vertex attributes are `Grade`, `Sex`, and `Race`. The `Grade` attribute has values 7 through 12, indicating each student's grade in school. The `Sex` attribute has values 1 for male and 2 for female. The `Race` attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes values 1 through 6: 1 = White (non-Hisp.); 2 = Black (non-Hisp.); 3 = Hispanic; 4 = Asian (non-Hisp.); 5 = Native American (non-Hisp.); and 6 = Other (non-Hisp.)

Licenses and Citation

If the source of the data set does not specified otherwise, this data set is protected by the Creative Commons License `http://creativecommons.org/licenses/by-nc-nd/2.5/`.

When publishing results obtained using this data set, the original authors (Resnick et al, 1997) should be cited. In addition this package should be cited as:

Mark S. Handcock, David R. Hunter, Carter T. Butts, Steven M. Goodreau, and Martina Morris. 2003 *statnet: An R package for the Statistical Modeling of Social Networks*
`http://www.csde.washington.edu/statnet`.

Source

The data set is based upon a model fit to data from one school community from the AddHealth Study, Wave I (Resnick et al., 1997). It was constructed as follows:

A vector representing the sex of each student in the school was randomly re-ordered. The same was done with the students' response to questions on race and grade. These three attribute vectors were permuted independently. Missing values for each were randomly assigned with weights determined by the size of the attribute classes in the school.

The following `ergm` formula was used to fit a model to the original data:

```
~ edges + nodefactor("Grade") + nodefactor("Race") + nodefactor("Sex")
 + nodematch("Grade",diff=T) + nodematch("Race",diff=T)
 + nodematch("Sex",diff=F) + gwdegree(1.0,fixed=T)
 + gwesp(1.0,fixed=T) + gwdsp(1.0,fixed=T)
```

The resulting model fit was then applied to a network with actors possessing the permuted attributes and with the same number of edges as in the original data.

The processes for handling missing data and defining the race attribute are described in Hunter, Goodreau & Handcock (2007).

References

Hunter D.R., Goodreau S.M. and Handcock M.S. (2007). *Goodness of Fit of Social Network Models*, *Journal of the American Statistical Association*.

Resnick M.D., Bearman, P.S., Blum R.W. et al. (1997). *Protecting adolescents from harm. Findings from the National Longitudinal Study on Adolescent Health*, *Journal of the American Medical Association*, 278: 823-32.

See Also

`network`, `plot.network`, `ergm`

`flobusiness` *Florentine Family Business Ties Data as a "network" object*

Description

This is a data set of business ties among Renaissance Florentine families. The data is originally from Padgett (1994) via `UCINET` and stored as a `network` object.

Breiger & Pattison (1986), in their discussion of local role analysis, use a subset of data on the social relations among Renaissance Florentine families (person aggregates) collected by John Padgett from historical documents. The relations are business ties (`flobusiness` - specifically, recorded financial ties such as loans, credits and joint partnerships).

As Breiger & Pattison point out, the original data are symmetrically coded. This is acceptable perhaps for marital ties, but is unfortunate for the financial ties (which are almost certainly directed).

To remedy this, the financial ties can be recoded as directed relations using some external measure of power - for instance, a measure of wealth. Vertex information is provided (1) `wealth` each family's net wealth in 1427 (in thousands of lira); (2) `priorates` the number of priorates (seats on the civic council) held between 1282- 1344; and (3) `totalties` the total number of business or marriage ties in the total dataset of 116 families (see Breiger & Pattison (1986), p 239).

Substantively, the data include families who were locked in a struggle for political control of the city of Florence around 1430. Two factions were dominant in this struggle: one revolved around the infamous Medicis (9), the other around the powerful Strozzis (15).

Usage

```
data(florentine)
```

Source

Padgett, John F. 1994. Marriage and Elite Structure in Renaissance Florence, 1282-1500. Paper delivered to the Social Science History Association.

References

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

Breiger R. and Pattison P. (1986). *Cumulated social roles: The duality of persons and their algebras*, Social Networks, 8, 215-256.

See Also

flo, network, plot.network, ergm, flomarriage

`flomarriage` *Florentine Family Marriage Ties Data as a "network" object*

Description

This is a data set of marriage ties among Renaissance Florentine families. The data is originally from Padgett (1994) via `UCINET` and stored as a `network` object.

Breiger & Pattison (1986), in their discussion of local role analysis, use a subset of data on the social relations among Renaissance Florentine families (person aggregates) collected by John Padgett from historical documents. The relations are marriage alliances (`flomarriage` betwween the families.

As Breiger & Pattison point out, the original data are symmetrically coded. This is perhaps acceptable perhaps for marital ties. Vertex information is provided on (1) `wealth` each family's net wealth in 1427 (in thousands of lira); (2) `priorates` the number of priorates (seats on the civic council) held between 1282- 1344; and (3) `totalties` the total number of business or marriage ties in the total dataset of 116 families (see Breiger & Pattison (1986), p 239).

Substantively, the data include families who were locked in a struggle for political control of the city of Florence around 1430. Two factions were dominant in this struggle: one revolved around the infamous Medicis (9), the other around the powerful Strozzis (15).

Usage

```
data(florentine)
```

Source

Padgett, John F. 1994. Marriage and Elite Structure in Renaissance Florence, 1282-1500. Paper delivered to the Social Science History Association.

References

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

Breiger R. and Pattison P. (1986). *Cumulated social roles: The duality of persons and their algebras*, Social Networks, 8, 215-256.

See Also

flobusiness, flo, network, plot.network, ergm

`florentine` *Florentine Family Marriage and Business Ties Data as a "network" object*

Description

This is a data set of marriage and business ties among Renaissance Florentine families. The data is originally from Padgett (1994) via `UCINET` and stored as a `network` object.

Breiger & Pattison (1986), in their discussion of local role analysis, use a subset of data on the social relations among Renaissance Florentine families (person aggregates) collected by John Padgett from historical documents. The two relations are business ties (`flobusiness` - specifically, recorded financial ties such as loans, credits and joint partnerships) and marriage alliances (`flomarriage`).

As Breiger & Pattison point out, the original data are symmetrically coded. This is acceptable perhaps for marital ties, but is unfortunate for the financial ties (which are almost certainly directed). To remedy this, the financial ties can be recoded as directed relations using some external measure of power - for instance, a measure of wealth. Both graphs provide vertex information on (1) `wealth` each family's net wealth in 1427 (in thousands of lira); (2) `priorates` the number of priorates (seats on the civic council) held between 1282- 1344; and (3) `totalties` the total number of business or marriage ties in the total dataset of 116 families (see Breiger & Pattison (1986), p 239).

Substantively, the data include families who were locked in a struggle for political control of the city of Florence around 1430. Two factions were dominant in this struggle: one revolved around the infamous Medicis (9), the other around the powerful Strozzis (15).

Usage

```
data(florentine)
```

Source

Padgett, John F. 1994. Marriage and Elite Structure in Renaissance Florence, 1282-1500. Paper delivered to the Social Science History Association.

References

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

Breiger R. and Pattison P. (1986). *Cumulated social roles: The duality of persons and their algebras*, Social Networks, 8, 215-256.

See Also

flo, network, plot.network, ergm

`g4` *Goodreau's four node network as a "network" object*

Description

This is an example thought of by Steve Goodreau. It is a directed network of four nodes and five ties stored as a `network` object.

It is interesting because the maximum likelihood estimator of the model with out degree 3 in it exists, but the maximum psuedolikelihood estimator does not.

Usage

```
data(g4)
```

Source

Steve Goodreau

See Also

florentine, network, plot.network, ergm

Examples

```
data(g4)
summary(ergm(g4 ~ odegree(3), MPLEonly=TRUE))
summary(ergm(g4 ~ odegree(3), theta0=0))
```

gof.ergm.control *Auxiliary for Controlling ERGM Goodness-of-Fit Evaluation*

Description

Auxiliary function as user interface for fine-tuning ERGM Goodness-of-Fit Evaluation.

Usage

```
gof.formula.control(prop.weights = "default", prop.args = NULL, drop =
TRUE, summarizestats = FALSE, maxchanges = 1e+06)

gof.ergm.control(prop.weights = NULL, prop.args = NULL, drop = TRUE, summarizestats
```

Arguments

`prop.weights`	Specifies the method to allocate probabilities of being proposed to dyads. For the `simulate.formula` variant, defaults to `"default"`, which picks a reasonable default for the specified constraint. For `simulate.ergm` variant, defaults to `NULL`, to reuse the weights with which the given `ergm.object` was fitted. Other possible values are `"TNT"`, `"random"`, and `"nonobserved"`, though not all values may be used with all possible constraints.
`prop.args`	An alternative, direct way of specifying additional arguments to proposal.
`drop`	logical; Should the degenerate terms in the model be dropped from the fit? If statistics occur on the extreme of their range they correspond to infinite parameter estimates. Default is `TRUE`.
`summarizestats`	logical; Print out a summary of the sufficient statistics of the generated network. This is useful as a diagnostic. Default is `FALSE`.
`maxchanges`	Maximum number of changes in dynamic network simulation for which to allocate space.

Value

A list with arguments as components.

See Also

`gof.formula`, `gof.ergm`, `glm.control` performs a similar function for `glm`

gof.ergm — *Conduct Goodness-of-Fit Diagnostics on a Exponential Family Random Graph Model*

Description

gof calculates p-values for geodesic distance, degree, and reachability summaries to diagnose the goodness-of-fit of exponential family random graph models. See ergm for more information on these models.

Usage

```
## Default S3 method:
gof(object,...)
## S3 method for class 'formula':
gof(formula, ..., theta0=NULL,
         nsim=100, burnin=10000, interval=1000,
         GOF=~degree+espartners+distance,
         constraints=~.,
         control=gof.formula.control(),
         seed=NULL,
         verbose=FALSE)
## S3 method for class 'ergm':
gof(object, ...,
         nsim=100,
         GOF=~degree+espartners+distance,
         burnin=10000, interval=1000,
         constraints=NULL,
         control=gof.ergm.control(),
         seed=NULL,
         theta0=NULL, verbose=FALSE)
```

Arguments

`object`	an R object. Either a formula or an ergm object. See documentation for ergm.
`formula`	formula; An R formula object, of the form `y ~ <model terms>`, where `y` is a network object or a matrix that can be coerced to a network object. This specifies the model to simulate from. For the details on the possible `<model terms>`, see ergm-terms. To create a network object in R, use the `network()` function, then add nodal attributes to it using the `%v%` operator if necessary.
`theta0`	When given either a formula or an object of class ergm, theta0 are the parameters from which the sample is drawn. By default set to a vector of 0.
`nsim`	The number of simulations to use for the MCMC p-values. This is the size of the sample of graphs to be randomly drawn from the distribution specified by the object on the set of all graphs.

`GOF`	formula; an R formula object, of the form `~ <model terms>` specifying the statistics to use to diagnosis the goodness-of-fit of the model. They do not need to be in the model formula specified in `formula`, and typically are not. Examples are the degree distribution ("degree"), minimum geodesic distances ("dist"), and shared partner distributions ("espartners" and "dspartners"). For the details on the possible `<model terms>`, see `ergm-terms`.
`burnin`	Number of proposed edge toggles before any MCMC sampling is done. If the model is correct this can be 0. Currently, there is no support for any check of the Markov chain mixing, so burnin should be set to a fairly large number.
`interval`	Number of proposed edge toggles between sampled statistics. The program prints a warning if too few proposed toggles are being accepted (if the number of proposed toggles between sampled observations ever equals an integral multiple of 100*(1+the number of toggles accepted)).
`constraints`	A one-sided formula specifying one or more constraints on the support of the distribution of the networks being modeled. See the help for similarly-named argument in `ergm` for more information. For `gof.formula`, defaults to unconstrained. For `gof.ergm`, defaults to the constraints with which `object` was fitted.
`control`	A list to control parameters, constructed using `gof.formula.control` or `gof.ergm.control` (which have different defaults).
`seed`	integer; random number integer seed. Defaults to `NULL` to use whatever the state of the random number generater is at the time of the call.
`verbose`	Provide verbose information on the progress of the simulation.
`...`	Additional arguments, to be passed to lower-level functions in the future.

Details

A sample of graphs is randomly drawn from the specified model. The first argument is typically the output of a call to `ergm` and the model used for that call is the one fit.

A plot of the summary measures is plotted. More information can be found by looking at the documentation of `ergm`.

Value

`gof`, `gof.ergm`, and `gof.formula` return an object of class `gofobject`. This is a list of the tables of statistics and p-values. This is typically plotted using `plot.gofobject`.

See Also

ergm, network, simulate.ergm, summary.ergm, plot.gofobject

Examples

```
#
data(florentine)
#
# test the gof.ergm function
```

```
#
gest <- ergm(flomarriage ~ edges + kstar(2))
gest
summary(gest)

#
# Plot the probabilities first
#
gofflo <- gof(gest)
gofflo
#
# Place all three on the same page
# with nice margins
#
par(mfrow=c(1,3))
par(oma=c(0.5,2,1,0.5))
#
plot(gofflo)
#
# And now the odds
#
plot(gofflo, plotlogodds=TRUE)
#
# Use the formula version
#
plot(gof(flomarriage ~ edges + kstar(2), theta0=c(-1.6339, 0.0049)))
```

---

mcmc.diagnostics.ergm

Conduct MCMC diagnostics on an ergm fit

Description

This function creates simple diagnostic plots for the MCMC sampled statistics produced from a fit. It also prints the Raftery-Lewis diagnostics, indicates if they are sufficient, and suggests the run length required.

Usage

```
## S3 method for class 'ergm':
mcmc.diagnostics (object, sample = "sample", smooth = TRUE,
                  r = 0.0125, digits = 6, maxplot = 1000, verbose = TRUE,
                  center = TRUE, main = "Summary of MCMC samples", xlab =
                  "Iterations", ylab = "", check.degeneracy = TRUE, ...)
```

Arguments

`object`	An object. See documentation for `ergm`.

`sample`	The component of `object` on which the diagnosis is based. The two usuals ones are `thetasample` from the auxilary sample of the natural parameter and `sample` the (default) sample of the sufficient statistics from the model.
`smooth`	Draw a smooth line through trace plots
`r`	Percentile of the distribution to estimate
`digits`	Number of digits to print
`maxplot`	Maximum number of statistics to plot
`verbose`	If this is `TRUE`, print out more information about the MCMC runs including lag correlations.
`center`	logical; should the samples be centered on the observed statistics.
`main`	Figure title for the diagnostic plots.
`xlab`	X-axis label for diagnostic plots
`ylab`	Y-axis label for diagnostic plots
`check.degeneracy`	Logical: Should the diagnostics include a check for model degeneracy?
`...`	Additional arguments, to be passed to lower-level functions in the future.

Details

The plots produced are a trace of the sampled output and a density estimate for each variable in the chain.

The Raftery-Lewis diagnostic is a run length control diagnostic based on a criterion of accuracy of estimation of the quantile q. It is intended for use on a short pilot run of a Markov chain. The number of iterations required to estimate the quantile q to within an accuracy of +/- r with probability p is calculated. Separate calculations are performed for each variable within each chain.

In fact, an `object` contains the matrix of statistics from the MCMC run as component `$sample`. This matrix is actually an object of class `mcmc` and can be used directly in the `CODA` package to assess MCMC convergence. *Hence all MCMC diagnostic methods available in* `coda` *are available directly.* See the examples and `http://www.mrc-bsu.cam.ac.uk/bugs/classic/coda04/readme.shtml`.

More information can be found by looking at the documentation of `ergm`.

Value

`mcmc.diagnostics.ergm` returns a matrix of Raftery-Lewis diagnostics.

Details of output

M The number of `burn in` iterations to be discarded (total over all chains).

N The number of iterations after burn in required to estimate the quantile q to within an accuracy of +/- r with probability p (total over all chains). The overall number of iterations required (M + N).

Total Overall number of iterations required (M + N).

Nmin The minimum required sample size for a chain with no correlation between consecutive samples. Positive autocorrelation will increase the required sample size above this minimum value.

I An estimate (the `dependence factor`) of the extent to which auto-correlation inflates the required sample size. Values of `I` larger than 5 indicate strong autocorrelation which may be due to a poor choice of starting value, high posterior correlations, or `stickiness` of the MCMC algorithm.

Author(s)

Mark S. Handcock, ⟨handcock@stat.washington.edu⟩ based on the `coda` package and also ideas from `mcgibbsit` by Gregory R. Warnes ⟨gregory_r_warnes@groton.pfizer.com⟩. It is based on the the R function `raftery.diag` in `coda`. `raftery.diag`, in turn, is based on the FORTRAN program `gibbsit` written by Steven Lewis which is available from the Statlib archive.

References

Warnes, G.W. (2000). Multi-Chain and Parallel Algorithms for Markov Chain Monte Carlo. Dissertation, Department of Biostatistics, University of Washington,

Raftery, A.E. and Lewis, S.M. (1992). One long run with diagnostics: Implementation strategies for Markov chain Monte Carlo. Statistical Science, 7, 493-497.

Raftery, A.E. and Lewis, S.M. (1995). The number of iterations, convergence diagnostics and generic Metropolis algorithms. In Practical Markov Chain Monte Carlo (W.R. Gilks, D.J. Spiegelhalter and S. Richardson, eds.). London, U.K.: Chapman and Hall.

See Also

`ergm`, `network`, `coda`, `mcgibbsit`, `summary.ergm`

Examples

```
#
data(florentine)
#
# test the mcmc.diagnostics function
#
gest <- ergm(flomarriage ~ edges + kstar(2))
summary(gest)

#
# Plot the probabilities first
#
mcmc.diagnostics(gest)
#
# Use coda directly
#
library(coda)
#
plot(gest$sample, ask=FALSE)
ergm.raftery.diag(gest$sample, r=0.1)
```

```
#
# A full range of diagnostics are available
# using codamenu()
#
```

`molecule` *Synthetic network with 20 nodes and 28 edges*

Description

This is a synthetic network of 20 nodes that is used as an example within the `ergm` documentation. It has an interesting elongated shape - reminencent of a chemical molecule. It is stored as a `network` object.

Usage

```
 data(molecule)
```

See Also

florentine, sampson, network, plot.network, ergm

`network.update` *Replaces the sociomatrix in a network object*

Description

Replaces the sociomatrix in a network object with the sociomatrix specified by `newmatrix`. See `ergm` for more information.

Usage

```
network.update(nw, newmatrix, matrix.type=NULL)
```

Arguments

`nw`	a `network` object. See documentation for the `network` package.
`newmatrix`	Either an adjacency matrix (a matrix of zeros and ones indicating the presence of a tie from i to j) or an edgelist (a two-column matrix listing origin and destination node numbers for each edge; note that in an undirected matrix, the first column should be the smaller of the two numbers).
`matrix.type`	One of "adjacency" or "edgelist" telling which type of matrix `newmatrix` is. Default is to use the `which.matrix.type` function.

Value

network.update returns a network object.

See Also

ergm, network

Examples

```
#
data(florentine)
#
# test the network.update function
#
# Create a Bernoulli network
rand.net <- network(network.size(flomarriage))
# store the sociomatrix
rand.mat <- rand.net[,]
# Update the network
network.update(flomarriage, rand.mat)
# Try this with an edgelist
rand.mat <- as.matrix.network.edgelist(flomarriage)[1:5,]
network.update(flomarriage, rand.mat)
```

---

plot.ergm *Plotting Method for class ergm*

Description

plot.ergm is the plotting method for ergm objects. It plots the MCMC diagnostics via the mcmc.diagnostics function. See ergm for more information on how to fit these models.

Usage

```
## S3 method for class 'ergm':
plot(x, ..., mle=FALSE, comp.mat = NULL,
          label = NULL, label.col = "black",
          xlab, ylab, main, label.cex = 0.8, edge.lwd = 1,
          edge.col=1, al = 0.1,
          contours=0, density=FALSE, only.subdens = FALSE,
          drawarrows=FALSE,
          contour.color=1, plotnetwork=FALSE, pie = FALSE, piesize=0.07,
          vertex.col=1, vertex.pch=19, vertex.cex=2,
          mycol=c("black","red","green","blue","cyan",
                  "magenta","orange","yellow","purple"),
          mypch=15:19, mycex=2:10)
```

Arguments

`x`	an R object of class `ergm`. See documentation for `ergm`.
`mle`	Plots the network using the MLE of the positions for latent models.
`pie`	For latent clustering models, each node is drawn as a pie chart representing the probabilities of cluster membership.
`piesize`	The size of the pie charts.
`contours`	For latent models, plots a contours by contours array of the network with one contour per network corresponding to the posterior distribution of each of the nodes.
`contour.color`	Color of the contour lines.
`density`	If density=TRUE, plots the density of the posterior position of the nodes. If density=c(nr,nc), plots a nr by nc array of density estimates for each cluster.
`only.subdens`	If density=c(nr,nc), only plots the densities of the clusters, not the overall density.
`drawarrows`	If density=TRUE, draws the ties on the density plot.
`plotnetwork`	If density=c(nr,nc), a plot of the network is also shown.
`comp.mat`	For latent models, the positions are Procrustes transformed to look like comp.mat.
`label`	A vector of the same length as the number of nodes containing the labels of the nodes.
`label.col`	The color to be used for plotting the labels.
`label.cex`	The size of the node labels.
`xlab`	Title for the x axis.
`ylab`	Title for the y axis.
`main`	The main title for the network.
`edge.lwd`	The line width for the arrows between nodes.
`edge.col`	The color of the arrows between nodes.
`al`	The length of the arrow heads.
`vertex.col`	The color of the nodes as defined by `mycol`. Can be specified as an attribute of the network used in the model.
`vertex.pch`	The plotting character of the nodes as defined by `mypch`. Can be specified as an attribute of the network used in the model. By default it is 15 - a red square.
`vertex.cex`	The size of the nodes as defined by `mycex`. Can be specified as an attribute of the network used in the model.
`mycol`	Vector of colors to be used. Defaults to: c("black","red","green","blue","cyan", "magenta","orange","yellow","purple")
`mypch`	Vector of plotting characters to be used. Defaults to:
`mycex`	Vector of character expansion values.
`...`	Other optional arguments to be used by the plot function.

Details

Plots the results of an ergm fit.

More information can be found by looking at the documentation of `ergm`.

Value

NULL

See Also

ergm, network, plot.network, plot, add.contours

Examples

```
## Not run:
#
# The example assumes you have the 'latentnet' package installed.
#
# Using Sampson's Monk data, lets fit a
# simple latent position model
#
data(sampson)
#
# Get the group labels
#
samp.labs <- substr(get.vertex.attribute(samplike,"group"),1,1)
#
samp.fit <- ergm(samplike ~ latent(k=2), burnin=10000,
                 MCMCsamplesize=2000, interval=30)
#
# See if we have convergence in the MCMC
mcmc.diagnostics(samp.fit)
#
# Plot the fit
#
plot(samp.fit,label=samp.labs, vertex.col="group")
#
# Using Sampson's Monk data, lets fit a latent clustering model
#
samp.fit <- ergm(samplike ~ latentcluster(k=2, ngroups=3), burnin=10000,
                 MCMCsamplesize=2000, interval=30)
#
# See if we have convergence in the MCMC
mcmc.diagnostics(samp.fit)
#
# Lets look at the goodness of fit:
#
plot(samp.fit,label=samp.labs, vertex.col="group")
plot(samp.fit,pie=TRUE,label=samp.labs)
plot(samp.fit,density=c(2,2))
plot(samp.fit,contours=5,contour.color="red")
plot(samp.fit,density=TRUE,drawarrows=TRUE)
add.contours(samp.fit,nlevels=8,lwd=2)
points(samp.fit$Z.mkl,pch=19,col=samp.fit$class)
## End(Not run)
```

plot.gofobject *Plot Goodness-of-Fit Diagnostics on a Exponential Family Random Graph Model*

Description

plot.gofobject plots diagnostics such as the degree distribution, geodesic distances, shared partner distributions, and reachability for the goodness-of-fit of exponential family random graph models. See ergm for more information on these models.

Usage

```
## S3 method for class 'gofobject':
plot(x, ...,
          cex.axis=0.7, plotlogodds=FALSE,
          main = "Goodness-of-fit diagnostics",
          normalize.reachability=FALSE,
          verbose=FALSE)
```

Arguments

x	an object of class gofobject, typically produced by the gof.ergm or gof.formula functions. See the documentation for these.
cex.axis	Character expansion of the axis labels relative to that for the plot.
plotlogodds	Plot the odds of a dyad having given characteristics (e.g., reachability, minimum geodesic distance, shared partners). This is an alternative to the probability of a dyad having the same property.
main	Title for the goodness-of-fit plots.
normalize.reachability	Should the reachability proportion be normalized to make it more comparable with the other geodesic distance proportions.
verbose	Provide verbose information on the progress of the plotting.
...	Additional arguments, to be passed to the plot function.

Details

gof.ergm produces a sample of networks randomly drawn from the specified model. This function produces a plot of the summary measures.

Value

none

See Also

gof.ergm, gof.formula, ergm, network, simulate.ergm

Examples

```
#
data(florentine)
#
# test the gof.ergm function
#
gest <- ergm(flomarriage ~ edges + kstar(2))
gest
summary(gest)

#
# Plot the probabilities first
#
gofflo <- gof(gest)
gofflo
plot(gofflo)
#
# And now the odds
#
plot(gofflo, plotlogodds=TRUE)
#
# Use the formula version
#
gof(flomarriage ~ edges + kstar(2), theta0=c(-1.6339, 0.0049))
```

`plot.network.statnet` *Two-Dimensional Visualization of Networks*

Description

`plot.network.statnet` produces a simple two-dimensional plot of the network object `x`. A variety of options are available to control vertex placement, display details, color, etc. The function is based on the plotting capabilities of the `network` package with additional pre-processing of arguments. Some of the capabilites require the `latentnet` package. See `plot.network` in the `network` package for details.

Usage

```
## S3 method for class 'statnet':
plot.network(x,
    attrname=NULL,
    label=network.vertex.names(x),
    coord=NULL,
    jitter=TRUE,
    thresh=0,
    usearrows=TRUE,
    mode="fruchtermanreingold",
    displayisolates=TRUE,
    interactive=FALSE,
    xlab=NULL,
    ylab=NULL,
    xlim=NULL,
    ylim=NULL,
    pad=0.2,
    label.pad=0.5,
    displaylabels=FALSE,
    boxed.labels=TRUE,
    label.pos=0,
    label.bg="white",
    vertex.sides=8,
    vertex.rot=0,
    arrowhead.cex=1,
    label.cex=1,
    loop.cex=1,
    vertex.cex=1,
    edge.col=1,
    label.col=1,
    vertex.col=2,
    label.border=1,
    vertex.border=1,
    edge.lty=1,
    label.lty=NULL,
    vertex.lty=1,
    edge.lwd=0,
    label.lwd=par("lwd"),
    edge.len=0.5,
    edge.curve=0.1,
    edge.steps=50,
    loop.steps=20,
    object.scale=0.01,
    uselen=FALSE,
    usecurve=FALSE,
    suppress.axes=TRUE,
    vertices.last=TRUE,
    new=TRUE,
    layout.par=NULL,
    cex.main=par("cex.main"),
    cex.sub=par("cex.sub"),
    latent.control=list(maxit=500,trace=0,dyadsample=10000,
                        penalty.sigma=c(5,0.5), nsubsample=200),
    colornames=colors(),
    verbose=FALSE, latent=FALSE,
       ...)
```

Arguments

`x`	an object of class `network`.
`attrname`	an optional edge attribute, to be used to set edge values.
`label`	a vector of vertex labels, if desired; defaults to the vertex labels returned by `network.vertex.names`.
`coord`	user-specified vertex coordinates, in an NCOL(dat)x2 matrix. Where this is specified, it will override the `mode` setting.
`jitter`	boolean; should the output be jittered?
`thresh`	real number indicating the lower threshold for tie values. Only ties of value >`thresh` are displayed. By default, `thresh`=0.
`usearrows`	boolean; should arrows (rather than line segments) be used to indicate edges?
`mode`	the vertex placement algorithm; this must correspond to a `network.layout` function. These include `"latent"`, `"latentPrior"`, and `"fruchtermanreingold"`.
`displayisolates`	boolean; should isolates be displayed?
`interactive`	boolean; should interactive adjustment of vertex placement be attempted?
`xlab`	x axis label.
`ylab`	y axis label.
`xlim`	the x limits (min, max) of the plot.
`ylim`	the y limits of the plot.
`pad`	amount to pad the plotting range; useful if labels are being clipped.
`label.pad`	amount to pad label boxes (if `boxed.labels==TRUE`), in character size units.
`displaylabels`	boolean; should vertex labels be displayed?
`boxed.labels`	boolean; place vertex labels within boxes?
`label.pos`	position at which labels should be placed, relative to vertices. `0` results in labels which are placed away from the center of the plotting region; `1`, `2`, `3`, and `4` result in labels being placed below, to the left of, above, and to the right of vertices (respectively); and `label.pos>=5` results in labels which are plotted with no offset (i.e., at the vertex positions).
`label.bg`	background color for label boxes (if `boxed.labels==TRUE`); may be a vector, if boxes are to be of different colors.
`vertex.sides`	number of polygon sides for vertices; may be given as a vector or a vertex attribute name, if vertices are to be of different types.
`vertex.rot`	angle of rotation for vertices (in degrees); may be given as a vector or a vertex attribute name, if vertices are to be rotated differently.
`arrowhead.cex`	expansion factor for edge arrowheads.

`label.cex`	character expansion factor for label text.
`loop.cex`	expansion factor for loops; may be given as a vector or a vertex attribute name, if loops are to be of different sizes.
`vertex.cex`	expansion factor for vertices; may be given as a vector or a vertex attribute name, if vertices are to be of different sizes.
`edge.col`	color for edges; may be given as a vector, adjacency matrix, or edge attribute name, if edges are to be of different colors.
`label.col`	color for vertex labels; may be given as a vector or a vertex attribute name, if labels are to be of different colors.
`vertex.col`	color for vertices; may be given as a vector or a vertex attribute name, if vertices are to be of different colors.
`label.border`	label border colors (if `boxed.labels==TRUE`); may be given as a vector, if label boxes are to have different colors.
`vertex.border`	border color for vertices; may be given as a vector or a vertex attribute name, if vertex borders are to be of different colors.
`edge.lty`	line type for edge borders; may be given as a vector, adjacency matrix, or edge attribute name, if edge borders are to have different line types.
`label.lty`	line type for label boxes (if `boxed.labels==TRUE`); may be given as a vector, if label boxes are to have different line types.
`vertex.lty`	line type for vertex borders; may be given as a vector or a vertex attribute name, if vertex borders are to have different line types.
`edge.lwd`	line width scale for edges; if set greater than 0, edge widths are scaled by `edge.lwd*dat`. May be given as a vector, adjacency matrix, or edge attribute name, if edges are to have different line widths.
`label.lwd`	line width for label boxes (if `boxed.labels==TRUE`); may be given as a vector, if label boxes are to have different line widths.
`edge.len`	if `uselen==TRUE`, curved edge lengths are scaled by `edge.len`.
`edge.curve`	if `usecurve==TRUE`, the extent of edge curvature is controlled by `edge.curv`. May be given as a fixed value, vector, adjacency matrix, or edge attribute name, if edges are to have different levels of curvature.
`edge.steps`	for curved edges (excluding loops), the number of line segments to use for the curve approximation.
`loop.steps`	for loops, the number of line segments to use for the curve approximation.
`object.scale`	base length for plotting objects, as a fraction of the linear scale of the plotting region. Defaults to 0.01.
`uselen`	boolean; should we use `edge.len` to rescale edge lengths?
`usecurve`	boolean; should we use `edge.curve`?
`suppress.axes`	boolean; suppress plotting of axes?
`vertices.last`	boolean; plot vertices after plotting edges?

`new`	boolean; create a new plot? If `new==FALSE`, vertices and edges will be added to the existing plot.
`layout.par`	parameters to the `network.layout` function specified in `mode`.
`cex.main`	Character expansion for the plot title.
`cex.sub`	Character expansion for the plot sub-title.
`latent.control`	A list of parameters to control the `latent` and `latentPrior` models, `dyadsample` determines the size above which to sample the latent dyads; see `ergm` and `optim` for details.
`colornames`	A vector of color names that can be selected by index for the plot. By default it is `colors()`.
`verbose`	logical; if this is `TRUE`, we will print out more information as we run the function.
`latent`	logical; use a two-dimensional latent space model based on the MLE fit. See documentation for `ergmm()` in `latentnet`.
`...`	additional arguments to `plot`.

Details

`plot.network` is a version of the standard network visualization tool within the `sna` library. By means of clever selection of display parameters, a fair amount of display flexibility can be obtained. Network layout – if not specified directly using `coord` – is determined via one of the various available algorithms. These are (briefly) as follows:

1. `latentPrior`: Use a two-dimensional latent space model based on a Bayesian minimum Kullback-Leibler fit. See documentation for `latent()` in `ergm`.
2. `random`: Vertices are placed (uniformly) randomly within a square region about the origin.
3. `circle`: Vertices are placed evenly about the unit circle.
4. `circrand`: Vertices are placed in a "Gaussian donut," with distance from the origin following a normal distribution and angle relative to the X axis chosen (uniformly) randomly.
5. `eigen`, `princoord`: Vertices are placed via (the real components of) the first two eigenvectors of:
   (a) `eigen`: the matrix of correlations among (concatenated) rows/columns of the adjacency matrix
   (b) `princoord`: the raw adjacency matrix.
6. `mds`, `rmds`, `geodist`, `adj`, `seham`: Vertices are placed by a metric MDS. The distance matrix used is given by:
   (a) `mds`: absolute row/column differences within the adjacency matrix
   (b) `rmds`: Euclidean distances between rows of the adjacency matrix
   (c) `geodist`: geodesic distances between vertices within the network
   (d) `adj`: $(\max A) - A$, where A is the raw adjacency matrix
   (e) `seham`: structural (dis)equivalence distances (i.e., as per `sedist` in the package `sna`) based on the Hamming metric

7. `spring`, `springrepulse`: Vertices are placed using a simple spring embedder. Parameters for the embedding model are given by `embedder.params`, in the following order: vertex mass; equilibrium extension; spring coefficient; repulsion equilibrium distance; and base coefficient of friction. Initial vertex positions are in random order around a circle, and simulation proceeds – increasing the coefficient of friction by the specified base value per unit time – until "motion" within the system ceases. If `springrepulse` is specified, then an inverse-cube repulsion force between vertices is also simulated; this force is calibrated so as to be exactly equal to the force of a unit spring extension at a distance specified by the repulsion equilibrium distance.

Value

None.

Requires

`mva`

Author(s)

Carter T. Butts ⟨buttsc@uci.edu⟩

References

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

See Also

`plot`

Examples

```
data(florentine)
plot(flomarriage)  #Plot the Florentine Marriage data
plot(network(10))  #Plot a random network
## Not run: plot(flomarriage,interactive="points")
```

`print.ergm` *Exponential Random Graph Models*

Description

`print.ergm` is the method used to print an `ergm` object created by the `ergm` function.

Usage

```
print.ergm (x, digits = max(3, getOption("digits") - 3), ...)
```

Arguments

`x`	An `ergm` object. See documentation for `ergm`.
`digits`	Significant digits for coefficients
`...`	Additional arguments, to be passed to lower-level functions in the future.

Details

Automatically called when an object of class `ergm` is printed. Currently, `print.ergm` summarizes the number of Newton-Raphson iterations required, the size of the MCMC sample, the theta vector governing the selection of the sample, and the Monte Carlo MLE.

Value

The value returned is the `ergm` object itself.

See Also

network, ergm

Examples

```
data(florentine)

x <- ergm(flomarriage ~ density)
class(x)
x
```

`samplk` *Longitudinal networks of positive affection within a monastery as a “network" object*

Description

Sampson (1969) recorded the social interactions among a group of monks while resident as an experimenter on vision, and collected numerous sociometric rankings. During his stay, a political “crisis in the cloister" resulted in the expulsion of four monks (Nos. 2, 3, 17, and 18) and the voluntary departure of several others - most immediately, Nos. 1, 7, 14, 15, and 16. (In the end, only 5, 6, 9, and 11 remained). Of particular interest is the data on positive affect relations (“liking"), in which each monk was asked if they had positive relations to each of the other monks.

The data were gathered at three times to capture changes in group sentiment over time: `samplk1`, `samplk2`, and `samplk3`. They represent three time points in the period during which a new cohort entered the monastery near the end of the study but before the major conflict began. Each member ranked only his top three choices on “liking." (Some subjects offered tied ranks for their top four choices). A tie from monk A to monk B exists if A nominated B as one of his three best friends at that that time point.

`samplk3` is a data set of Hoff, Raftery and Handcock (2002).

See also the data set `sampson` containing the time-aggregated graph `samplike`. It is the cumulative tie for “liking" over the three periods. For this, a tie from monk A to monk B exists if A nominated B as one of his three best friends at any of the three time points.

All graphs are stored as `network` objects.

This data set is standard in the social network analysis literature, having been modeled by Holland and Leinhardt (1981), Reitz (1982), Holland, Laskey and Leinhardt (1983), and Fienberg, Meyer, and Wasserman (1981), Hoff, Raftery, and Handcock (2002), etc. This is only a small piece of the data collected by Sampson.

Usage

```
data(samplk)
```

Source

Sampson, S. F. (1968), *A novitiate in a period of change: An experimental and case study of relationships,* Unpublished Ph.D. dissertation, Department of Sociology, Cornell University.

References

White, H.C., Boorman, S.A. and Breiger, R.L. (1976). *Social structure from multiple networks. I. Blockmodels of roles and positions.* American Journal of Sociology, 81(4), 730-780.

See Also

sampson, florentine, network, plot.network, ergm

`sampson` *Cumulative network of positive affection within a monastery as a “network" object*

Description

Sampson (1969) recorded the social interactions among a group of monks while resident as an experimenter on vision, and collected numerous sociometric rankings. During his stay, a political “crisis in the cloister" resulted in the expulsion of four monks (Nos. 2, 3, 17, and 18) and the voluntary departure of several others - most immediately, Nos. 1, 7, 14, 15, and 16. (In the end, only 5, 6, 9, and 11 remained). Of particular interest is the data on positive affect relations (“liking"), in which each monk was asked if they had positive relations to each of the other monks.

The data were gathered at three times to capture changes in group sentiment over time. They represent three time points in the period during which a new cohort entered the monastery near the end of the study but before the major conflict began. Each member ranked only his top three choices on “liking." (Some subjects offered tied ranks for their top four choices). A tie from monk A to monk B exists if A nominated B as one of his three best friends at that that time point.

`samplike` is the time-aggregated graph. It is the cumulative tie for “liking" over the three periods. For this, a tie from monk A to monk B exists if A nominated B as one of his three best friends at any of the three time points.

All graphs are stored as network objects.

This data set is standard in the social network analysis literature, having been modeled by Holland and Leinhardt (1981), Reitz (1982), Holland, Laskey and Leinhardt (1983), and Fienberg, Meyer, and Wasserman (1981), Hoff, Raftery, and Handcock (2002), etc. This is only a small piece of the data collected by Sampson.

Usage

```
data(sampson)
```

Source

Sampson, S. F. (1968), *A novitiate in a period of change: An experimental and case study of relationships,* Unpublished Ph.D. dissertation, Department of Sociology, Cornell University.

References

White, H.C., Boorman, S.A. and Breiger, R.L. (1976). *Social structure from multiple networks. I. Blockmodels of roles and positions.* American Journal of Sociology, 81(4), 730-780.

See Also

florentine, network, plot.network, ergm

`simulate.ergm` *Draw from the distribution of an Exponential Family Random Graph Model*

Description

simulate is used to draw from exponential family random network models in their natural parameterizations. See ergm for more information on these models.

Usage

```
## S3 method for class 'formula':
simulate(object, nsim=1, seed=NULL, ..., theta0,
                          burnin=1000, interval=1000,
                          basis=NULL,
                          sequential=TRUE,
                          constraints = ~.,
                          control = simulate.formula.control(),
                          verbose=FALSE)
## S3 method for class 'ergm':
simulate(object, nsim=1, seed=NULL, ..., theta0,
                       burnin=1000, interval=1000,
                       sequential=TRUE,
                       constraints = NULL,
                       control = simulate.ergm.control(),
                       verbose=FALSE)
```

Arguments

`object`	an R object. Either a `formula` or an `ergm` object. The `formula` should be of the form `y ~ <model terms>`, where `y` is a network object or a matrix that can be coerced to a `network` object. For the details on the possible `<model terms>`, see `ergm-terms`. To create a `network` object in R, use the `network()` function, then add nodal attributes to it using the `%v%` operator if necessary.
`nsim`	Number of networks to be randomly drawn from the given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm.
`seed`	Random number integer seed. The default is `sample(10000000, size=1)`.
`theta0`	For Bernoulli networks this is the log-odds of a tie, however it is only used if `prob` is not specified. When given either a `formula` or an object of class `ergm`, `theta0` are the parameters from which the sample is drawn.
`burnin`	The number of proposed proposals before any MCMC sampling is done. Currently, there is no support for any check of the Markov chain mixing, so `burnin` should be set to a fairly large number.
`interval`	The number of proposals between sampled statistics. The program prints a warning if too few proposals are being accepted (if the number of proposals between sampled observations ever equals an integral multiple of 100(1+the number of proposals accepted)).
`basis`	An optional `network` object to start the MCMC algorithm from. This is used only if a `network` object is not specified on the right-hand-side of the `formula`. If neither is specified an error is given as it is used to specify the nature of the network requested (e.g., size and directionality).
`constraints`	A one-sided formula specifying one or more constraints on the support of the distribution of the networks being simulated. See the documentation for a similar argument for `ergm` for more information. For `simulate.formula`, defaults to no constraints. For `simulate.ergm`, defaults to using the same constraints as those with which `object` was fitted.
`control`	A list of control parameters for algorithm tuning. Constructed using `simulate.ergm.control` or `simulate.formula.control`, which have different defaults.
`sequential`	Should the returned draws use the prior draw as the starting network or always use the initially passed network? For random draws the results should be similar (stochastically), but the `sequential=TRUE` option is useful for dynamic draws.
`verbose`	If this is `TRUE`, we will print out more information as we run the program, including (currently) some goodness of fit statistics.
`...`	further arguments passed to or used by methods.

Details

A sample of networks is randomly drawn from the specified model. The model is specified by the first argument of the function. If the first argument is a `formula` then this defines the model. If the first argument is the output of a call to `ergm` then the model used for that call is the one fit - and unless `theta0` is specified, the sample is from the MLE of the parameters. If neither of those are given as the first argument then a Bernoulli network is generated with the probability of ties defined by `prob` or `theta0`.

Note that the first network is sampled after `burnin` + `interval` steps, and any subsequent networks are sampled each `interval` steps after the first.

More information can be found by looking at the documentation of `ergm`.

Value

`simulate` returns an object of class `network.series` that is a list consisting of the following elements:

`formula`	The `formula` used to generate the sample.
`networks`	A list of the generated networks.
`stats`	The $n \times p$ matrix of network change statistics, where n is the sample size and p is the number of network change statistics specified in the model.

See Also

ergm, network, print.network

Examples

```
#
# Let's draw from a Bernoulli model with 16 nodes
# and density 0.5 (i.e., theta0 = c(0,0))
#
g.sim <- simulate(network(16) ~ edges + mutual)
#
# What are the statistics like?
#
summary(g.sim ~ edges + mutual)
#
# Now simulate a network with higher mutuality
#
g.sim <- simulate(network(16) ~ edges + mutual, theta0=c(0,2))
#
# How do the statistics like?
#
summary(g.sim ~ edges + mutual)
#
# Let's draw from a Bernoulli model with 16 nodes
# and tie probability 0.1
#
g.use <- network(16,density=0.1,directed=FALSE)
#
```

```
# Starting from this network let's draw 5 realizations
# of a edges and 2-star network
#
g.sim <- simulate(~edges+kstar(2),nsim=5,theta0=c(-1.8,0.03),
               basis=g.use,
               burnin=100000,interval=1000)
g.sim
#
# attach the Florentine Marriage data
#
data(florentine)
#
# fit an edges and 2-star model using the ergm function
#
gest <- ergm(flomarriage ~ edges + kstar(2))
summary(gest)
#
# Draw from the fitted model
#
g.sim <- simulate(gest,nsim=100,burnin=1000,interval=1000)
g.sim
```

`simulate.ergm.control`

Auxiliary for Controlling ERGM Simulation

Description

Auxiliary function as user interface for fine-tuning ERGM simulation.

Usage

```
simulate.formula.control(prop.weights = "default", prop.args = NULL,
drop = FALSE, summarizestats = FALSE, maxchanges = 1e+06, parallel=0)

simulate.ergm.control(prop.weights = NULL, prop.args = NULL, drop = FALSE, summariz
```

Arguments

`prop.weights`	Specifies the method to allocate probabilities of being proposed to dyads. For the `simulate.formula` variant, defaults to `"default"`, which picks a reasonable default for the specified constraint. For `simulate.ergm` variant, defaults to `NULL`, to reuse the weights with which the given `ergm.object` was fitted. Other possible values are `"TNT"`, `"random"`, and `"nonobserved"`, though not all values may be used with all possible constraints.
`prop.args`	An alternative, direct way of specifying additional arguments to proposal.
`drop`	logical; Should the degenerate terms in the model be dropped from the fit? If statistics occur on the extreme of their range they correspond to infinite parameter estimates. Default is `FALSE`.

`summarizestats`	logical; Print out a summary of the sufficient statistics of the generated network. This is useful as a diagnostic. Default is `FALSE`.
`maxchanges`	Maximum number of changes in dynamic network simulation for which to allocate space.
`parallel`	Number of threads in which to run the sampling.

Value

A list with arguments as components.

References

`simulate.formula`, `simulate.ergm`, `glm.control` performs a similar function for `glm`

See Also

`simulate.ergm`, `simulate.formula`, `glm.control` performs a similar function for `glm`

`summary.ergm` *Summarizing ERGM Model Fits*

Description

`summary` method for class `"ergm"`.

Usage

```
## S3 method for class 'ergm':
summary(object, ..., check.degeneracy = FALSE,
                correlation = FALSE, covariance = FALSE)
```

Arguments

`object`	an object of class `"ergm"`, usually, a result of a call to `ergm`.
`check.degeneracy`	Logical: Should the output include a check for model degeneracy?
`correlation`	logical; if `TRUE`, the correlation matrix of the estimated parameters is returned and printed.
`covariance`	logical; if `TRUE`, the covariance matrix of the estimated parameters is returned and printed.
`...`	further arguments passed to or from other methods.

Details

`summary.ergm` tries to be smart about formatting the coefficients, standard errors, etc.

Value

The function summary.ergm computes and returns a list of summary statistics of the fitted ergm model given in object.

See Also

network, ergm, print.ergm. The model fitting function ergm, summary.

Function coef will extract the matrix of coefficients with standard errors, t-statistics and p-values.

Examples

```
data(florentine)

x <- ergm(flomarriage ~ density)
summary(x)
```

---

summary.gofobject *Summaries the Goodness-of-Fit Diagnostics on a Exponential Family Random Graph Model*

Description

summary.gofobject summaries the diagnostics such as the degree distribution, geodesic distances, shared partner distributions, and reachability for the goodness-of-fit of exponential family random graph models. See ergm for more information on these models.

Usage

```
## S3 method for class 'gofobject':
summary(object, ...)
```

Arguments

object	an object of class gofobject, typically produced by the gof.ergm or gof.formula functions. See the documentation for these.
...	Additional arguments, to be passed to the plot function.

Details

gof.ergm produces a sample of networks randomly drawn from the specified model. This function produces a print out the summary measures.

Value

none

See Also

gof.ergm, gof.formula, ergm, network, simulate.ergm

Examples

```
#
data(florentine)
#
# test the gof.ergm function
#
gest <- ergm(flomarriage ~ edges + kstar(2))
gest
summary(gest)

#
# Plot the probabilities first
#
gofflo <- gof(gest)
gofflo
summary(gofflo)
```

`summary.statistics` *Calculation of network or graph statistics*

Description

Used to calculate the specified statistics for an observed network if its argument is a formula for an `ergm`. See `ergm-terms` for more information on the statistics that may be specified.

Usage

```
## S3 method for class 'formula':
summary.statistics(object, ..., drop=FALSE, basis=NULL)
## S3 method for class 'ergm':
summary.statistics(object, ..., drop=FALSE, basis=NULL)
```

Arguments

`object`	an R object. It is either an R `formula` object (see above) or an `ergm` model object. In the latter case, `summary.statistics` is called for the `object$formula` object. In the former case, `object` is of the form `y ~ <model terms>`, where `y` is a `network` object or a matrix that can be coerced to a `network` object. For the details on the possible `<model terms>`, see `ergm-terms`. To create a `network` object in R, use the `network()` function, then add nodal attributes to it using the `%v%` operator if necessary.
`drop`	logical: Should terms whose observed statistics are extreme among the set of all possible network statistics (which result in nonexistent MLEs) be dropped?

`basis`	An optional `network` object relative to which the global statistics should be calculated.
`...`	further arguments passed to or used by methods.

Details

If `object` is of class `formula`, then `summary` may be used in lieu of `summary.statistics` because `summary.formula` calls the `summary.statistics` function. The function actually cumulates the change statistics when removing edges from the observed network one by one until the empty network results. Since each model term has a prespecified value (zero by default) for the corresponding statistic(s) on an empty network, these change statistics give the absolute statistics on the original network.

Value

A vector of statistics measured on the network.

See Also

ergm, network, ergm-terms

Examples

```
#
# Lets look at the Florentine marriage data
#
data(florentine)
#
# test the summary.statistics function
#
summary(flomarriage ~ edges + kstar(2))
m <- as.matrix(flomarriage)
summary(m ~ edges)  # twice as large as it should be
summary(m ~ edges, directed=FALSE) # Now it's correct
```

Index