Package ‘ergm’

March 25, 2016

Version 3.6.0

Date 2016-03-24

Title Fit, Simulate and Diagnose Exponential-Family Models for Networks

Depends statnet.common (>= 3.3), network (>= 1.13)

Imports robustbase (>= 0.9-10), coda (>= 0.18-1), trust, Matrix, lpSolve, parallel, methods, MASS

Suggests lattice, latticeExtra, sna, latentnet, rmarkdown, ergm.userterms, tergm, Rmpi

Description An integrated set of tools to analyze and simulate networks based on exponential-family random graph models (ERGM). ``ergm'' is a part of the ``statnet'' suite of packages for network analysis.

License GPL-3 + file LICENSE

URL http://statnet.org

VignetteBuilder rmarkdown

NeedsCompilation yes

Author Mark S. Handcock [aut],
David R. Hunter [aut],
Carter T. Butts [aut],
Steven M. Goodreau [aut],
Pavel N. Krivitsky [aut, cre],
Martina Morris [aut],
Li Wang [ctb],
Kirk Li [ctb],
Skye Bender-deMoll [ctb]

Maintainer Pavel N. Krivitsky <pavel@uow.edu.au>

Repository CRAN

Date/Publication 2016-03-25 09:12:20

R topics documented:

ergm-package *Fit, Simulate and Diagnose Exponential-Family Models for Networks*

Description

`ergm` is a collection of functions to plot, fit, diagnose, and simulate from exponential-family random graph models (ERGMs). 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, please cite the original authors as described in `citation(package="ergm")`.

All programs derived from this package must cite it.

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 implements maximum likelihood estimates of ERGMs to be calculated using Markov Chain Monte Carlo (via `ergm`). The package also provides tools for simulating networks (via `simulate.ergm`) and assessing model goodness-of-fit (see `mcmc.diagnostics` and `gof.ergm`).

A number of Statnet Project packages extend and enhance `ergm`. These include `tergm` (Temporal ERGM), which provides extensions for modeling evolution of networks over time; `ergm.count`, which facilitates exponential family modeling for networks whose dyadic measurements are counts; and `ergm.userterms`, which allows users to implement their own ERGM terms.

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

Author(s)

Mark S. Handcock <`handcock@stat.ucla.edu`>,
David R. Hunter <`dhunter@stat.psu.edu`>,
Carter T. Butts <`buttsc@uci.edu`>,
Steven M. Goodreau <`goodreau@u.washington.edu`>,
Pavel N. Krivitsky <`krivitsky@stat.psu.edu`>, and
Martina Morris <`morrism@u.washington.edu`>

Maintainer: Pavel N. Krivitsky <`krivitsky@stat.psu.edu`>

References

Admiraal R, Handcock MS (2007). **networksis**: Simulate bipartite graphs with fixed marginals through sequential importance sampling. Statnet Project, Seattle, WA. Version 1, `statnet.org`.

Bender-deMoll S, Morris M, Moody J (2008). Prototype Packages for Managing and Animating Longitudinal Network Data: **dynamicnetwork** and **rSoNIA**. *Journal of Statistical Software*, 24(7). `http://www.jstatsoft.org/v24/i07/`.

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

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

Butts CT (2007). **sna**: Tools for Social Network Analysis. R package version 2.3-2. `http://CRAN.R-project.org/package=sna`

Butts CT (2008). **network**: A Package for Managing Relational Data in R. *Journal of Statistical Software*, 24(2). `http://www.jstatsoft.org/v24/i02/`.

Butts C (2015). **network**: Classes for Relational Data. The Statnet Project (`http://www.statnet.org`). R package version 1.12.0, `http://CRAN.R-project.org/package=network`.

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

Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008a). A **statnet** Tutorial. *Journal of Statistical Software*, 24(8). `http://www.jstatsoft.org/v24/i08/`.

Goodreau SM, Kitts J, Morris M (2008b). Birds of a Feather, or Friend of a Friend? Using Exponential Random Graph Models to Investigate Adolescent Social Networks. *Demography*, 45, in press.

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`

Handcock MS (2003b). **degreenet**: Models for Skewed Count Distributions Relevant to Networks. Statnet Project, Seattle, WA. Version 1.0, `statnet.org`.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003a). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. Statnet Project, Seattle, WA. Version 3, `statnet.org`.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003b). **statnet**: Software Tools for the Statistical Modeling of Network Data. Statnet Project, Seattle, WA. Version 3, `statnet.org`.

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

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008b). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software*, 24(3). `http://www.jstatsoft.org/v24/i03/`.

Krivitsky PN, Handcock MS (2007). **latentnet**: Latent position and cluster models for statistical networks. Seattle, WA. Version 2, `http://statnet.org`.

Krivitsky PN (2012). Exponential-Family Random Graph Models for Valued Networks. *Electronic Journal of Statistics*, 2012, 6, 1100-1128. `doi:10.1214/12-EJS696`

Morris M, Handcock MS, Hunter DR (2008). Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 24(4). `http://www.jstatsoft.org/v24/i04/`.

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

`anova.ergm` *ANOVA for ERGM Fits*

Description

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

Usage

```
## S3 method for class 'ergm'
anova(object, ..., eval.loglik = FALSE)
## S3 method for class 'ergmlist'
anova(object, ..., eval.loglik = FALSE, scale = 0, test = "F")
```

Arguments

`object, ...`	objects of class `ergm`, usually, a result of a call to `ergm`.
`eval.loglik`	a logical specifying whether the log-likelihood will be evaluated if missing.
`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.

If any of the objects do not have estimated log-likelihoods, produces an error, unless `eval.loglik=TRUE`.

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, logLik.ergm for adding the log-likelihood to an existing ergm object.

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), eval.loglik=TRUE) # Note the eval.loglik argument.
anova(fit0, fit1)
anova(fit0, fit1, fit2)
```

---

approx.hotelling.diff.test *Approximate Hotelling T^2-Test for One Sample Means*

Description

A multivariate hypothesis test of difference between a vector of sample means and a vector of true means (mu). The null hypothesis assumes no differences for each pair of sample mean and true mean, the alternative hypothesis assumes not all pairs are equal. Smaller p-value would reject the null hypothesis. The distribution under the null hypothesis is an approximation to Hotelling's T^2 distribution, hence the test is an approximated test.

Hotelling's T^2 distribution

A multivariate method that is the multivariate counterpart of Student's t and which also forms the basis for certain multivariate control charts is based on Hotelling's T2 distribution, which was introduced by Hotelling (1947).

References

Hotelling, H. (1947). Multivariate Quality Control. In C. Eisenhart, M. W. Hastay, and W. A. Wallis, eds. Techniques of Statistical Analysis. New York: McGraw-Hill.

`as.edgelist` *Convert a network object into a numeric edgelist matrix*

Description

Constructs an edgelist in the format expected by ergm's internal functions

NOTE: the `as.edgelist` functions have been moved to the `network` package, and this help file may be removed in the future. See `as.edgelist`

Details

Constructs an edgelist matrix from a network, sorted tails-major order, with tails first, and, for undirected networks, tail < head.

The `as.matrix.network(nw, matrix.type="edgelist")` provides similar functionality but it does not enforce ordering..

Note

The `as.edgelist` functions have been moved to the `network` package. See `as.edgelist`

See Also

See also`as.edgelist`, `as.matrix.network.edgelist`

Examples

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

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

```
## S3 method for class 'numeric'
as.network(x, directed = TRUE,
    hyper = FALSE, loops = FALSE, multiple = FALSE, bipartite = FALSE,
    ignore.eval = TRUE, names.eval = NULL,
    edge.check = FALSE,
    density=NULL, init=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 `init` is 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.)
init	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`

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)
```

`check.ErgmTerm` *Ensures an Ergm Term and its Arguments Meet Appropriate Conditions*

Description

These are low-level functions not intended to be called by end users. For information on ergm terms, see ergm-terms

The `check.ErgmTerm` function ensures for the `InitErgmTerm.X` function that the term X:

- is applicable given the ’directed’ and ’bipartite’ attributes of the given network
- is not applied to a directed bipartite network
- has an appropiate number of arguments
- has correct argument types if arguments where provided
- has default values assigned if defaults are available

by halting execution if any of the first 3 criteria are not met.

The `ergm.checkargs` function ensures for the `InitErgm` function that the term X:

- has an appropiate number of arguments
- has correct argument types if arguments where provieded

- has default values assigned for non-required arguments

by halting execution if either of the first 2 criteria are not met

The the `ergm.checkdirected` function halts execution for the <InitErgm> functions with an error message if the given model term cannot be used with the network because of its state as (un)directed. (essentially it prints and formats the the error message)

Usage

```
check.ErgmTerm(nw, arglist, directed = NULL, bipartite = NULL,
               nonnegative = FALSE, varnames = NULL, vartypes = NULL,
               defaultvalues = list(), required = NULL, response = NULL)

ergm.checkargs(fname, arglist, varnames=NULL, vartypes=NULL,
                  defaultvalues=list(), required=NULL)

ergm.checkdirected(fname, nw.directedflag, requirement,
                                 extramessage="")
```

Arguments

`nw`	the network that term X is being checked against
`arglist`	the list of arguments for term X
`directed,`	logical, whether term X requires a directed network; default=NULL
`bipartite`	whether term X requires a bipartite network (T or F); default=NULL
`nonnegative`	whether term X requires a network with only nonnegative weights; default=FALSE
`varnames`	the vector of names of the possible arguments for term X; default=NULL
`vartypes`	the vector of types of the possible arguments for term X; default=NULL
`defaultvalues`	the list of default values for the possible arguments of term X; default=list()
`required`	the logical vector of whether each possible argument is required; default=NULL
`response`	edge attribute name?
`fname`	the name of the model term as a character string
`nw.directedflag`	logical,whether the network is directed
`requirement`	logical, whether the term requires a directed network
`extramessage`	additional messages to attach to the warning; default value = ""

Value

a list of the values for each possible argument of term X; user provided values are used when given, default values otherwise.

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)
```

`coef.length.model` *Extract Number of parameters in ergm Model*

Description

`coef.sublength` and `coef.length` are methods that extract the numbers of parameters for ergm model objects.

Usage

```
## S3 method for class 'model'
coef.length(object, ...)

## S3 method for class 'model'
coef.sublength(object, ...)
```

Arguments

`object`	an ergm model object
`...`	other arguments.

Value

`coef.sublength.model` returns a vector containing the number of model parameters corresponding to each model term. `coef.length.model` returns the sum of the values returned by `coef.sublength.model`.

`colMeans.mcmc.list` *utility operations for mcmc.list objects*

Description

Adaptations of `colMeans` and `sweep` for working with `mcmc.list` objects. The function mcmc.list is used to represent parallel runs of the same chain, with different starting values and random seeds.

Usage

```
colMeans.mcmc.list(x, ...)

sweep.mcmc.list(x, STATS, FUN = "-", check.margin = TRUE, ...)
```

Arguments

`x`	a `mcmc.list` object
`STATS`	the summary statistic which is to be swept out
`FUN`	the function to be used to carry out the sweep (default is '-', subtraction)
`check.margin`	logical. If TRUE (the default), warn if the length or dimensions of STATS do not match the specified dimensions of x. Set to FALSE for a small speed gain when you know that dimensions match.
`...`	additional arguments to `colMeans` or `sweep`

Details

`colMeans.mcmc.list` converts the mcmc list into a matrix and then runs `colMeans` on it

`sweep.mcmc.lists` modifies the values of the chains by computing some aggregate summary statistic function (`STATS`) on the entire chain and applying it to each value via `FUN` (default is subtraction). See `sweep` for additional examples.

Value

`colMeans.mcmc` returns a vector with length equal to the number of mcmc chains in `x` with the mean value for each chain. `sweep.mcmc.lists` returns an appropriately modified version of `x`

`control.ergm` *Auxiliary for Controlling ERGM Fitting*

Description

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

Usage

```
control.ergm(drop=TRUE,

             init=NULL,
             init.method=NULL,

             main.method=c("MCMLE","Robbins-Monro",
               "Stochastic-Approximation","Stepping"),
             force.main=FALSE,
             main.hessian=TRUE,

             MPLE.max.dyad.types=1e+6,
             MPLE.samplesize=50000,
             MPLE.type=c("glm", "penalized"),

             MCMC.prop.weights="default", MCMC.prop.args=list(),
             MCMC.interval=1024,
             MCMC.burnin=MCMC.interval*16,
             MCMC.samplesize=1024,

             MCMC.effectiveSize=NULL,
             MCMC.effectiveSize.damp=10,
             MCMC.effectiveSize.maxruns=1000,
             MCMC.effectiveSize.base=1/2,
             MCMC.effectiveSize.points=5,
             MCMC.effectiveSize.order=1,

             MCMC.return.stats=TRUE,
             MCMC.runtime.traceplot=FALSE,
             MCMC.init.maxedges=20000,
             MCMC.max.maxedges=Inf,
             MCMC.addto.se=TRUE,
             MCMC.compress=FALSE,
             MCMC.packagenames=c(),

             SAN.maxit=10,
             SAN.burnin.times=10,
             SAN.control=control.san(coef=init,
                         SAN.prop.weights=MCMC.prop.weights,
                         SAN.prop.args=MCMC.prop.args,
                         SAN.init.maxedges=MCMC.init.maxedges,
                         SAN.burnin=MCMC.burnin * SAN.burnin.times,
                         SAN.interval=MCMC.interval,
                         SAN.packagenames=MCMC.packagenames,
                         MPLE.max.dyad.types=MPLE.max.dyad.types,
                         parallel=parallel,
                         parallel.type=parallel.type,
                         parallel.version.check=parallel.version.check),

             MCMLE.termination=c("Hummel", "Hotelling", "precision", "none"),
             MCMLE.maxit=20,
             MCMLE.conv.min.pval=0.5,
             MCMLE.NR.maxit=100,
             MCMLE.NR.reltol=sqrt(.Machine$double.eps),
             obs.MCMC.samplesize=MCMC.samplesize,
             obs.MCMC.interval=MCMC.interval,
             obs.MCMC.burnin=MCMC.burnin,
             obs.MCMC.burnin.min=obs.MCMC.burnin/10,
           obs.MCMC.prop.weights=MCMC.prop.weights, obs.MCMC.prop.args=MCMC.prop.args,

             MCMLE.check.degeneracy=FALSE,
             MCMLE.MCMC.precision=0.005,
             MCMLE.MCMC.max.ESS.frac=0.1,
             MCMLE.metric=c("lognormal", "logtaylor",
                            "Median.Likelihood",
                            "EF.Likelihood", "naive"),
             MCMLE.method=c("BFGS","Nelder-Mead"),
             MCMLE.trustregion=20,
             MCMLE.dampening=FALSE,
             MCMLE.dampening.min.ess=20,
             MCMLE.dampening.level=0.1,
             MCMLE.steplength.margin=0.05,
             MCMLE.steplength=if(is.null(MCMLE.steplength.margin)) 0.5 else 1,
             MCMLE.adaptive.trustregion=3,
             MCMLE.sequential=TRUE,
             MCMLE.density.guard.min=10000,
             MCMLE.density.guard=exp(3),
             MCMLE.effectiveSize=NULL,
             MCMLE.last.boost=4,
             MCMLE.Hummel.esteq=TRUE,
             MCMLE.Hummel.miss.sample=100,
             MCMLE.Hummel.maxit=25,
             MCMLE.steplength.min=0.0001,

             SA.phase1_n=NULL,
             SA.initial_gain=NULL,
             SA.nsubphases=4,
             SA.niterations=NULL,
             SA.phase3_n=NULL,
             SA.trustregion=0.5,

             RM.phase1n_base=7,
             RM.phase2n_base=100,
             RM.phase2sub=7,
             RM.init_gain=0.5,
             RM.phase3n=500,

             Step.MCMC.samplesize=100,
             Step.maxit=50,
             Step.gridsize=100,

             CD.nsteps=8,
             CD.multiplicity=1,
             CD.nsteps.obs=128,
             CD.multiplicity.obs=1,
             CD.maxit=60,
             CD.conv.min.pval=0.5,
             CD.NR.maxit=100,
             CD.NR.reltol=sqrt(.Machine$double.eps),
             CD.metric=c("naive", "lognormal", "logtaylor",
               "Median.Likelihood",
               "EF.Likelihood"),
             CD.method=c("BFGS","Nelder-Mead"),
             CD.trustregion=20,
             CD.dampening=FALSE,
             CD.dampening.min.ess=20,
             CD.dampening.level=0.1,
             CD.steplength.margin=0.5,
             CD.steplength=1,
             CD.adaptive.trustregion=3,
             CD.adaptive.epsilon=0.01,
             CD.Hummel.esteq=TRUE,
             CD.Hummel.miss.sample=100,
             CD.Hummel.maxit=25,
             CD.steplength.min=0.0001,

             loglik.control=control.logLik.ergm(),

             seed=NULL,
             parallel=0,
             parallel.type=NULL,
             parallel.version.check=TRUE,
             ...)
```

Arguments

`drop`	Logical: If TRUE, terms whose observed statistic values are at the extremes of their possible ranges are dropped from the fit and their corresponding parameter estimates are set to plus or minus infinity, as appropriate. This is done because maximum likelihood estimates cannot exist when the vector of observed statistic lies on the boundary of the convex hull of possible statistic values.
`init`	numeric or `NA` vector equal in length to the number of parameters in the model or `NULL` (the default); the initial values for the estimation and coefficient offset terms. If `NULL` is passed, all of the initial values are computed using the method specified by `control$init.method`. If a numeric vector is given, the elements of the vector are interpreted as follows: • Elements corresponding to terms enclosed in `offset()` are used as the fixed offset coefficients. Note that offset coefficients alone can be more conveniently specified using `ergm` argument `offset.coef`. If both `offset.coef` and `init` arguments are given, values in `offset.coef` will take precedence. • Elements that do not correspond to offset terms and are not `NA` are used as starting values in the estimation. • Initial values for the elements that are `NA` are fit using the method specified by `control$init.method`. Passing `control.ergm(init=coef(prev.fit))` can be used to “resume” an uncoverged `ergm` run, but see `enformulate.curved`.
`init.method`	A chatacter vector or `NULL`. The default method depends on the reference measure used. For the binary (`"Bernoulli"`) ERGMs, it’s maximum pseudo-likelihood estimation (MPLE). Other valid values include `"zeros"` for a `0` vector of appropriate length and `"CD"` for contrastive divergence.

Valid initial methods for a given reference can be queried using `ergm.init.methods`.

`main.method`
: One of "MCMLE" (default),"Robbins-Monro", "Stochastic-Approximation", or "Stepping". Chooses the estimation method used to find the MLE. `MCMLE` attempts to maximize an approximation to the log-likelihood function. `Robbins-Monro` and `Stochastic-Approximation` are both stochastic approximation algorithms that try to solve the method of moments equation that yields the MLE in the case of an exponential family model. Another alternative is a partial stepping algorithm (`Stepping`) as in Hummel et al. (2012). 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.

  Note that in recent versions of ERGM, the enhancements of `Stepping` have been folded into the default `MCMLE`, which is able to handle more modeling scenarios.

`force.main`
: Logical: If TRUE, then force MCMC-based estimation method, even if the exact MLE can be computed via maximum pseudolikelihood estimation.

`main.hessian`
: Logical: If TRUE, then an approximate Hessian matrix is used in the MCMC-based estimation method.

`MPLE.max.dyad.types`
: Maximum number of unique values of change statistic vectors, which are the predictors in a logistic regression used to calculate the MPLE. This calculation uses a compression algorithm that allocates space based on `MPLE.max.dyad.types`.

`MPLE.samplesize`
: Not currently documented; used in conditional-on-degree version of MPLE.

`MPLE.type`
: One of "glm" or "penalized". Chooses method of calculating MPLE. "glm" is the usual formal logistic regression, whereas "penalized" uses the bias-reduced method of Firth (1993) as originally implemented by Meinhard Ploner, Daniela Dunkler, Harry Southworth, and Georg Heinze in the "logistf" package.

`MCMC.prop.weights, obs.MCMC.prop.weights`
: Specifies the proposal distribution used in the MCMC Metropolis-Hastings algorithm. Possible choices depending on selected `reference` and `constraints` arguments of the `ergm` function, but often include `"TNT"` and `"random"`, and the `"default"` is to use the one with the highest priority available.

  The `TNT` (tie / no tie) option puts roughly equal weight on selecting a dyad with or without a tie as a candidate for toggling, whereas the `random` option puts equal weight on all possible dyads, though the interpretation of `random` may change according to the constraints in place. When no constraints are in place, the default is TNT, which appears to improve Markov chain mixing particularly for networks with a low edge density, as is typical of many realistic social networks.

  `obs.MCMC.prop.weights`, if given separately, specifies the weights to be used for the constrained MCMC when missing dyads are present, defaulting to the same as `MCMC.prop.weights`.

`MCMC.prop.args, obs.MCMC.prop.args`
: An alternative, direct way of specifying additional arguments to proposal. `obs.MCMC.prop.args`, if given separately, specifies the weights to be used for the constrained MCMC when missing dyads are present, defaulting to the same as `MCMC.prop.args`.

`MCMC.interval` Number of proposals between sampled statistics. Increasing interval will reduces the autocorrelation in the sample, and may increase the precision in estimates by reducing MCMC error, at the expense of time. Set the interval higher for larger networks.

`MCMC.burnin` Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.

`MCMC.samplesize`
Number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm. Increasing sample size may increase the precision in the estimates by reducing MCMC error, at the expense of time. Set it higher for larger networks, or when using parallel functionality.

`MCMLE.effectiveSize, MCMC.effectiveSize, MCMC.effectiveSize.damp, MCMC.effectiveSize.maxruns, MCMC.`
Set `MCMLE.effectiveSize` to non-NULL value to adaptively determine the burn-in and the MCMC length needed to get the specified effective size using the method of Sahlin (2011); 50 is a reasonable value. This feature is in experimental status until we verify the coverage of the standard errors.

`MCMC.return.stats`
Logical: If TRUE, return the matrix of MCMC-sampled network statistics. This matrix should have `MCMC.samplesize` rows. This matrix can be used directly by the coda package to assess MCMC convergence.

`MCMC.runtime.traceplot`
Logical: If TRUE, plot traceplots of the MCMC sample after every MCMC MLE iteration.

`MCMC.init.maxedges, MCMC.max.maxedges`
Maximum number of edges expected in network. Starting at `MCMC.init.maxedges`, it will be incremented by a factor of 10 if exceeded during fitting, up to `MCMC.max.maxedges`, at which point the process will stop with an error.

`MCMC.addto.se` Whether to add the standard errors induced by the MCMC algorithm to the estimates' standard errors.

`MCMC.compress` Logical: If TRUE, the matrix of sample statistics returned is compressed to the set of unique statistics with a column of frequencies post-pended.

`MCMC.packagenames`
Names of packages in which to look for change statistic functions in addition to those autodetected. This argument should not be needed outside of very strange setups.

`SAN.maxit` When `target.stats` argument is passed to `ergm`, the maximum number of attempts to use `san` to obtain a network with statistics close to those specified.

`SAN.burnin.times`
Multiplier for `SAN.burnin` relative to `MCMC.burnin`. This lets one control the amount of SAN burn-in (arguably, the most important of SAN parameters) without overriding the other SAN.control defaults.

`SAN.control` Control arguments to `san`. See `control.san` for details.

`MCMLE.termination`
The criterion used for terminating MCMLE estimation:

- "Hummel" Terminate when the Hummel step length is 1 for two consecutive iterations. For the last iteration, the sample size is boosted by a factor of `MCMLE.last.boost`. See Hummel et. al. (2012).
  Note that this criterion is incompatible with `MCMLE.steplength` $\neq$ 1 or `MCMLE.steplength.margin` = `NULL`.
- "Hotelling" After every MCMC sample, an autocorrelation-adjusted Hotelling's T^2 test for equality of MCMC-simulated network statistics to observed is conducted, and if its P-value exceeds `MCMLE.conv.min.pval`, the estimation is considered to have converged and finishes. This was the default option in `ergm` version 3.1.
- "precision" Terminate when the estimated loss in estimating precision due to using MCMC standard errors is below the precision bound specified by `MCMLE.MCMC.precision`, and the Hummel step length is 1 for two consecutive iterations. See `MCMLE.MCMC.precision` for details. This feature is in experimental status until we verify the coverage of the standard errors.
  Note that this criterion is incompatible with `MCMLE.steplength` $\neq$ 1 or `MCMLE.steplength.margin` = `NULL`.
- "none" Stop after `MCMLE.maxit` iterations.

`MCMLE.maxit` Maximum 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.

`MCMLE.conv.min.pval`
The P-value used in the Hotelling test for early termination.

`MCMLE.NR.maxit, MCMLE.NR.reltol`
The method, maximum number of iterations and relative tolerance to use within the `optim` rountine in the MLE optimization. Note that by default, ergm uses `trust`, and falls back to `optim` only when `trust` fails.

`obs.MCMC.samplesize, obs.MCMC.burnin, obs.MCMC.interval, obs.MCMC.burnin.min`
Sample size, burnin, and interval parameters for the MCMC sampling used when unobserved data are present in the estimation routine.

`MCMLE.check.degeneracy`
Logical: If TRUE, employ a check for model degeneracy.

`MCMLE.MCMC.precision, MCMLE.MCMC.max.ESS.frac`
`MCMLE.MCMC.precision` is a vector of upper bounds on the standard errors induced by the MCMC algorithm, expressed as a percentage of the total standard error. The MCMLE algorithm will terminate when the MCMC standard errors are below the precision bound, and the Hummel step length is 1 for two consecutive iterations. This is an experimental feature.
If effective sample size is used (see `MCMC.effectiveSize`), then ergm may increase the target ESS to reduce the MCMC standard error.

`MCMLE.metric` Method to calculate the loglikelihood approximation. See Hummel et al (2010) for an explanation of "lognormal" and "naive".

`MCMLE.method` Deprecated. By default, ergm uses `trust`, and falls back to `optim` with Nelder-Mead method when `trust` fails.

`MCMLE.trustregion`
Maximum increase the algorithm will allow for the approximated likelihood at a given iteration. See Snijders (2002) for details.

Note that not all metrics abide by it.

`MCMLE.dampening`
: (logical) Should likelihood dampening be used?

`MCMLE.dampening.min.ess`
: The effective sample size below which dampening is used.

`MCMLE.dampening.level`
: The proportional distance from boundary of the convex hull move.

`MCMLE.steplength.margin`
: The extra margin required for a Hummel step to count as being inside the convex hull of the sample. Set this to 0 if the step length gets stuck at the same value over several iteraions. Set it to `NULL` to use fixed step length. Note that this parameter is required to be non-`NULL` for MCMLE termination using Hummel or precision criteria.

`MCMLE.steplength`
: Multiplier for step length, which may (for values less than one) make fitting more stable at the cost of computational efficiency. Can be set to "adaptive"; see `MCMLE.adaptive.trustregion`.

  If `MCMLE.steplength.margin` is not `NULL`, the step length will be set using the algorithm of Hummel et al. (2010). In that case, it will serve as the maximum step length considered. However, setting it to anything other than 1 will preclude using Hummel or precision as termination criteria.

`MCMLE.adaptive.trustregion`
: Maximum increase the algorithm will allow for the approximated loglikelihood at a given iteration when `MCMLE.steplength="adaptive"`.

`MCMLE.sequential`
: Logical: If TRUE, the next iteration of the fit uses the last network sampled as the starting network. If FALSE, always use the initially passed network. The results should be similar (stochastically), but the TRUE option may help if the `target.stats` in the `ergm` function are far from the initial network.

`MCMLE.density.guard.min, MCMLE.density.guard`
: A simple heuristic to stop optimization if it finds itself in an overly dense region, which usually indicates ERGM degeneracy: if the sampler encounters a network configuration that has more than `MCMLE.density.guard.min` edges and whose number of edges is exceeds the observed network by more than `MCMLE.density.guard`, the optimization process will be stopped with an error.

`MCMLE.last.boost`
: For the Hummel termination criterion, increase the MCMC sample size of the last iteration by this factor.

`MCMLE.Hummel.esteq`
: For curved ERGMs, should the estimating function values be used to compute the Hummel step length? This allows the Hummel stepping algorithm converge when some sufficient statistics are at 0.

`MCMLE.steplength.min`
: Stops MCMLE estimation when the step length gets stuck below this minimum value.

`MCMLE.Hummel.miss.sample`
: In fitting the missing data MLE, the rules for step length become more complicated. In short, it is necessary for *all* points in the constrained sample to be in the convex hull of the unconstrained (though they may be on the border); and it is necessary for their centroid to be in its interior. This requires checking a large number of points against whether they are in the convex hull, so to speed up the procedure, a sample is taken of the points most likely to be outside it. This parameter specifies the sample size.

`MCMLE.Hummel.maxit`
: Maximum number of iterations in searching for the best step length.

`SA.phase1_n`
: 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 Snijders (2002) for details.

`SA.initial_gain`
: Initial gain to Phase 2 of the stochastic approximation algorithm. See Snijders (2002) for details.

`SA.nsubphases`
: Number of sub-phases in Phase 2 of the stochastic approximation algorithm. Defaults to `MCMLE.maxit`. See Snijders (2002) for details.

`SA.niterations`
: 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 Snijders (2002) for details.

`SA.phase3_n`
: Sample size for the MCMC sample in Phase 3 of the stochastic approximation algorithm. See Snijders (2002) for details.

`SA.trustregion`
: The trust region parameter for the likelihood functions, used in the stochastic approximation algorithm.

`RM.phase1n_base, RM.phase2n_base, RM.phase2sub, RM.init_gain, RM.phase3n`
: The Robbins-Monro control parameters are not yet documented.

`Step.MCMC.samplesize`
: MCMC sample size for the preliminary steps of the "Stepping" method of optimization. This is usually chosen to be smaller than the final MCMC sample size (which equals `MCMC.samplesize`). See Hummel et al. (2012) for details.

`Step.maxit`
: Maximum number of iterations (steps) allowed by the "Stepping" method.

`Step.gridsize`
: Integer N such that the "Stepping" style of optimization chooses a step length equal to the largest possible multiple of $1/N$. See Hummel et al. (2012) for details.

`CD.nsteps, CD.multiplicity`
: Main settings for contrastive divergence to obtain initial values for the estimation: respectively, the number of Metropolis–Hastings steps to take before reverting to the starting value and the number of tentative proposals per step. Computational experiments indicate that increasing `CD.multiplicity` improves the estimate faster than increasing `CD.nsteps` — up to a point — but it also samples from the wrong distribution, in the sense that while as `CD.nsteps`$\rightarrow \infty$, the CD estimate approaches the MLE, this is not the case for `CD.multiplicity`.

  In practice, MPLE, when available, usually outperforms CD for even a very high `CD.nsteps` (which is, in turn, not very stable), so CD is useful primarily when MPLE is not available. This feature is to be considered experimental and in flux.

	The default values have been set experimentally, providing a reasonably stable, if not great, starting values.
`CD.nsteps.obs, CD.multiplicity.obs`	When there are missing dyads, `CD.nsteps` and `CD.multiplicity` must be set to a relatively high value, as the network passed is not necessarily a good start for CD. Therefore, these settings are in effect if there are missing dyads in the observed network, using a higher default number of steps.
`CD.maxit, CD.conv.min.pval, CD.NR.maxit, CD.NR.reltol, CD.metric, CD.method, CD.trustregion, CD.dam`	Miscellaneous tuning parameters of the CD sampler and optimizer. These have the same meaning as their `MCMC.*` counterparts. Note that only the Hotelling's stopping criterion is implemented for CD.
`loglik.control`	See `control.ergm.bridge`
`seed`	Seed value (integer) for the random number generator. See `set.seed`
`parallel`	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
`parallel.type`	API to use for parallel processing. Supported values are `"MPI"` and `"PSOCK"`. Defaults to using the `parallel` package with PSOCK clusters. See `ergm-parallel`
`parallel.version.check`	Logical: If TRUE, check that the version of `ergm` running on the slave nodes is the same as that running on the master node.
`...`	Additional arguments, passed to other functions This argument is helpful because it collects any control parameters that have been deprecated; a warning message is printed in case of deprecated arguments.

Details

This function is only used within a call to the `ergm` function. See the usage section in `ergm` for details.

Value

A list with arguments as components.

References

- Snijders, T.A.B. (2002), Markov Chain Monte Carlo Estimation of Exponential Random Graph Models. Journal of Social Structure. Available from `http://www.cmu.edu/joss/content/articles/volume3/Snijders.pdf`.
- Firth (1993), Bias Reduction in Maximum Likelihood Estimates. Biometrika, 80: 27-38.
- 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.
- Hummel, R. M., Hunter, D. R., and Handcock, M. S. (2012), Improving Simulation-Based Algorithms for Fitting ERGMs, Journal of Computational and Graphical Statistics, 21: 920-939.
- Kristoffer Sahlin. Estimating convergence of Markov chain Monte Carlo simulations. Master's Thesis. Stockholm University, 2011. `http://www2.math.su.se/matstat/reports/master/2011/rep2/report.pdf`

See Also

`ergm`. The `control.simulate` function performs a similar function for `simulate.ergm`; `control.gof` performs a similar function for `gof`.

`control.ergm.bridge` *Auxiliary for Controlling ergm.bridge*

Description

Auxiliary function as user interface for fine-tuning ergm.bridge algorithm, which approximates log likelihood ratios using bridge sampling.

Usage

```
control.ergm.bridge(nsteps=20,
                    MCMC.burnin=10000,
                    MCMC.interval=100,
                    MCMC.samplesize=10000,
                    obs.MCMC.samplesize=MCMC.samplesize,
                    obs.MCMC.interval=MCMC.interval,
                    obs.MCMC.burnin=MCMC.burnin,

                    MCMC.prop.weights="default",
                    MCMC.prop.args=list(),

                    MCMC.init.maxedges=20000,
                    MCMC.packagenames=c(),

                    seed=NULL,
                    parallel=0,
                    parallel.type=NULL,
                    parallel.version.check=TRUE)
```

Arguments

`nsteps`	Number of geometric bridges to use.
`MCMC.burnin`	Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.
`MCMC.interval`	Number of proposals between sampled statistics.
`MCMC.samplesize`	Number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm.
`obs.MCMC.burnin, obs.MCMC.interval, obs.MCMC.samplesize`	The obs versions of these arguments are for the unobserved data simulation algorithm.

`MCMC.prop.weights`
: Specifies the proposal distribution used in the MCMC Metropolis-Hastings algorithm. Possible choices are `"TNT"` or `"random"`; the `"default"` is one of these two, depending on the constraints in place (as defined by the `constraints` argument of the `ergm` function), though not all weights may be used with all constraints. The `TNT` (tie / no tie) option puts roughly equal weight on selecting a dyad with or without a tie as a candidate for toggling, whereas the `random` option puts equal weight on all possible dyads, though the interpretation of `random` may change according to the constraints in place. When no constraints are in place, the default is TNT, which appears to improve Markov chain mixing particularly for networks with a low edge density, as is typical of many realistic social networks.

`MCMC.prop.args` An alternative, direct way of specifying additional arguments to proposal.

`MCMC.init.maxedges`
: Maximum number of edges expected in network.

`MCMC.packagenames`
: Names of packages in which to look for change statistic functions in addition to those autodetected. This argument should not be needed outside of very strange setups.

`seed` Seed value (integer) for the random number generator. See `set.seed`

`parallel` Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.

`parallel.type` API to use for parallel processing. Supported values are `"MPI"` and `"PSOCK"`. Defaults to using the `parallel` package with PSOCK clusters. See `ergm-parallel`

`parallel.version.check`
: Logical: If TRUE, check that the version of `ergm` running on the slave nodes is the same as that running on the master node.

Details

This function is only used within a call to the `ergm.bridge.llr` or `ergm.bridge.dindstart.llk` functions.

Value

A list with arguments as components.

See Also

`ergm.bridge.llr`, `ergm.bridge.dindstart.llk` control.gof *Auxiliary for Controlling ERGM Goodness-of-Fit Evaluation*

Description

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

Usage

```
control.gof.formula(nsim=100,
                    MCMC.burnin=10000,
                    MCMC.interval=1000,
                    MCMC.prop.weights="default",
                    MCMC.prop.args=list(),

                    MCMC.init.maxedges=20000,
                    MCMC.packagenames=c(),

                    MCMC.runtime.traceplot=FALSE,
                    network.output="network",

                    seed=NULL,
                    parallel=0,
                    parallel.type=NULL,
                    parallel.version.check=TRUE)

control.gof.ergm(nsim=100,
                 MCMC.burnin=NULL,
                 MCMC.interval=NULL,
                 MCMC.prop.weights=NULL,
                 MCMC.prop.args=NULL,

                 MCMC.init.maxedges=NULL,
                 MCMC.packagenames=NULL,

                 MCMC.runtime.traceplot=FALSE,
                 network.output="network",

                 seed=NULL,
                 parallel=0,
                 parallel.type=NULL,
                 parallel.version.check=TRUE)
```

Arguments

nsim	Number of networks to be randomly drawn using Markov chain Monte Carlo. This sample of networks provides the basis for comparing the model to the observed network.

`MCMC.burnin`	Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.
`MCMC.interval`	Number of proposals between sampled statistics.
`MCMC.prop.weights`	Specifies the proposal distribution used in the MCMC Metropolis-Hastings algorithm. Possible choices are `"TNT"` or `"random"`; the `"default"` is one of these two, depending on the constraints in place (as defined by the `constraints` argument of the `ergm` function), though not all weights may be used with all constraints. The `TNT` (tie / no tie) option puts roughly equal weight on selecting a dyad with or without a tie as a candidate for toggling, whereas the `random` option puts equal weight on all possible dyads, though the interpretation of `random` may change according to the constraints in place. When no constraints are in place, the default is TNT, which appears to improve Markov chain mixing particularly for networks with a low edge density, as is typical of many realistic social networks.
`MCMC.prop.args`	An alternative, direct way of specifying additional arguments to proposal.
`MCMC.init.maxedges`	Maximum number of edges expected in network.
`MCMC.packagenames`	Names of packages in which to look for change statistic functions in addition to those autodetected. This argument should not be needed outside of very strange setups.
`MCMC.runtime.traceplot`	Logical: If TRUE, plot traceplots of the MCMC sample after every MCMC MLE iteration.
`network.output`	R class with which to output networks. The options are "network" (default) and "edgelist.compressed" (which saves space but only supports networks without vertex attributes)
`seed`	Seed value (integer) for the random number generator. See `set.seed`
`parallel`	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
`parallel.type`	API to use for parallel processing. Supported values are `"MPI"` and `"PSOCK"`. Defaults to using the `parallel` package with PSOCK clusters. See `ergm-parallel`
`parallel.version.check`	Logical: If TRUE, check that the version of `ergm` running on the slave nodes is the same as that running on the master node.

Details

This function is only used within a call to the `gof` function. See the `usage` section in `gof` for details.

Value

A list with arguments as components.

See Also

`gof`. The `control.simulate` function performs a similar function for `simulate.ergm`; `control.ergm` performs a similar function for `ergm`.

---

control.logLik.ergm *Auxiliary for Controlling logLik.ergm*

Description

Auxiliary function as user interface for fine-tuning logLik.ergm algorithm, which approximates log likelihood values.

Usage

```
control.logLik.ergm(nsteps=20,
                    MCMC.burnin=NULL,
                    MCMC.interval=NULL,
                    MCMC.samplesize=NULL,
                    obs.MCMC.samplesize=MCMC.samplesize,
                    obs.MCMC.interval=MCMC.interval,
                    obs.MCMC.burnin=MCMC.burnin,

                    MCMC.prop.weights=NULL,
                    MCMC.prop.args=NULL,
                    warn.dyads=TRUE,

                    MCMC.init.maxedges=NULL,
                    MCMC.packagenames=NULL,

                    seed=NULL,
                    parallel = NULL,
                    parallel.type = NULL,
                    parallel.version.check = TRUE)
```

Arguments

`nsteps`	Number of geometric bridges to use.
`MCMC.burnin`	Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.
`MCMC.interval`	Number of proposals between sampled statistics.
`MCMC.samplesize`	Number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm.
`obs.MCMC.burnin, obs.MCMC.interval, obs.MCMC.samplesize`	The obs versions of these arguments are for the unobserved data simulation algorithm.

`MCMC.prop.weights`	Specifies the proposal distribution used in the MCMC Metropolis-Hastings algorithm. Possible choices are `"TNT"` or `"random"`; the `"default"` is one of these two, depending on the constraints in place (as defined by the `constraints` argument of the `ergm` function), though not all weights may be used with all constraints. The `TNT` (tie / no tie) option puts roughly equal weight on selecting a dyad with or without a tie as a candidate for toggling, whereas the `random` option puts equal weight on all possible dyads, though the interpretation of `random` may change according to the constraints in place. When no constraints are in place, the default is TNT, which appears to improve Markov chain mixing particularly for networks with a low edge density, as is typical of many realistic social networks.
`MCMC.prop.args`	An alternative, direct way of specifying additional arguments to proposal.
`warn.dyads`	Whether or not a warning should be issued when sample space constraints render the observed number of dyads ill-defined.
`MCMC.init.maxedges`	Maximum number of edges expected in network.
`MCMC.packagenames`	Names of packages in which to look for change statistic functions in addition to those autodetected. This argument should not be needed outside of very strange setups.
`seed`	Seed value (integer) for the random number generator. See `set.seed`
`parallel`	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
`parallel.type`	API to use for parallel processing. Supported values are `"MPI"` and `"PSOCK"`. Defaults to using the `parallel` package with PSOCK clusters. See `ergm-parallel`
`parallel.version.check`	Logical: If TRUE, check that the version of `ergm` running on the slave nodes is the same as that running on the master node.

Details

This function is only used within a call to the `logLik.ergm` function.

Value

A list with arguments as components.

See Also

`logLik.ergm`

control.san *Auxiliary for Controlling SAN*

Description

Auxiliary function as user interface for fine-tuning simulated annealing algorithm.

Usage

```
control.san(coef=NULL,

            SAN.tau=1,
            SAN.invcov=NULL,
            SAN.burnin=100000,
            SAN.interval=10000,
            SAN.init.maxedges=20000,

            SAN.prop.weights="default",
            SAN.prop.args=list(),
            SAN.packagenames=c(),

            MPLE.max.dyad.types=1e6,
            MPLE.samplesize = 50000,

            network.output="network",

            seed=NULL,
            parallel=0,
            parallel.type=NULL,
            parallel.version.check=TRUE)
```

Arguments

`coef`	Vector of model coefficients used for MCMC simulations, one for each model term.
`SAN.tau`	Currently unused.
`SAN.invcov`	Initial inverse covariance matrix used to calculate Mahalanobis distance in determining how far a proposed MCMC move is from the `target.stats` vector. If NULL, taken to be the covariance matrix returned when fitting the MPLE if `coef==NULL`, or the identity matrix otherwise.
`SAN.burnin`	Number of MCMC proposals before any sampling is done.
`SAN.interval`	Number of proposals between sampled statistics.
`SAN.init.maxedges`	Maximum number of edges expected in network.

`SAN.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. Other possible values are `"TNT"`, `"random"`, and `"nonobserved"`, though not all values may be used with all possible constraints.
`SAN.prop.args`	An alternative, direct way of specifying additional arguments to proposal.
`SAN.packagenames`	Names of packages in which to look for change statistic functions in addition to those autodetected. This argument should not be needed outside of very strange setups.
`MPLE.max.dyad.types`	Maximum number of unique values of change statistic vectors, which are the predictors in a logistic regression used to calculate the MPLE. This calculation uses a compression algorithm that allocates space based on `MPLE.max.dyad.types`
`MPLE.samplesize`	Not currently documented; used in conditional-on-degree version of MPLE.
`network.output`	R class with which to output networks. The options are "network" (default) and "edgelist.compressed" (which saves space but only supports networks without vertex attributes)
`seed`	Seed value (integer) for the random number generator. See `set.seed`
`parallel`	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
`parallel.type`	API to use for parallel processing. Supported values are `"MPI"` and `"PSOCK"`. Defaults to using the `parallel` package with PSOCK clusters. See `ergm-parallel`
`parallel.version.check`	Logical: If TRUE, check that the version of `ergm` running on the slave nodes is the same as that running on the master node.

Details

This function is only used within a call to the `san` function. See the `usage` section in `san` for details.

Value

A list with arguments as components.

See Also

`san`

`control.simulate` *Auxiliary for Controlling ERGM Simulation*

Description

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

Usage

```
control.simulate(MCMC.burnin=10000,
                 MCMC.interval=1000,
                 MCMC.prop.weights="default",
                 MCMC.prop.args=list(),
                 MCMC.init.maxedges=20000,
                 MCMC.packagenames=c(),
                 MCMC.runtime.traceplot=FALSE,
                 network.output="network",

                 parallel=0,
                 parallel.type=NULL,
                 parallel.version.check=TRUE,
                 ...)

control.simulate.formula(MCMC.burnin=10000,
                         MCMC.interval=1000,
                         MCMC.prop.weights="default",
                         MCMC.prop.args=list(),
                         MCMC.init.maxedges=20000,
                         MCMC.packagenames=c(),
                         MCMC.runtime.traceplot=FALSE,
                         network.output="network",

                         parallel=0,
                         parallel.type=NULL,
                         parallel.version.check=TRUE,
                         ...)

control.simulate.formula.ergm(MCMC.burnin=10000,
                              MCMC.interval=1000,
                              MCMC.prop.weights="default",
                              MCMC.prop.args=list(),
                              MCMC.init.maxedges=20000,
                              MCMC.packagenames=c(),
                              MCMC.runtime.traceplot=FALSE,
                              network.output="network",

                              parallel=0,
                                  parallel.type=NULL,
                                  parallel.version.check=TRUE,
                                  ...)

control.simulate.ergm(MCMC.burnin=NULL,
                      MCMC.interval=NULL,
                      MCMC.prop.weights=NULL,
                      MCMC.prop.args=NULL,
                      MCMC.init.maxedges=NULL,
                      MCMC.packagenames=NULL,
                      MCMC.runtime.traceplot=FALSE,
                      network.output="network",

                      parallel=0,
                      parallel.type=NULL,
                      parallel.version.check=TRUE,
                      ...)
```

Arguments

`MCMC.prop.weights`	Specifies the proposal distribution used in the MCMC Metropolis-Hastings algorithm. Possible choices are `"TNT"` or `"random"`; the `"default"` is one of these two, depending on the constraints in place (as defined by the `constraints` argument of the `ergm` function), though not all weights may be used with all constraints. The `TNT` (tie / no tie) option puts roughly equal weight on selecting a dyad with or without a tie as a candidate for toggling, whereas the `random` option puts equal weight on all possible dyads, though the interpretation of `random` may change according to the constraints in place. When no constraints are in place, the default is TNT, which appears to improve Markov chain mixing particularly for networks with a low edge density, as is typical of many realistic social networks.
`MCMC.prop.args`	An alternative, direct way of specifying additional arguments to proposal.
`MCMC.burnin`	Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.
`MCMC.interval`	Number of proposals between sampled statistics.
`MCMC.init.maxedges`	Maximum number of edges expected in network.
`MCMC.packagenames`	Names of packages in which to look for change statistic functions in addition to those autodetected. This argument should not be needed outside of very strange setups.
`MCMC.runtime.traceplot`	Logical: If TRUE, plot traceplots of the MCMC sample after every MCMC MLE iteration.
`network.output`	R class with which to output networks. The options are "network" (default) and "edgelist.compressed" (which saves space but only supports networks without vertex attributes)

`parallel`	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
`parallel.type`	API to use for parallel processing. Supported values are `"MPI"` and `"PSOCK"`. Defaults to using the `parallel` package with PSOCK clusters. See `ergm-parallel`
`parallel.version.check`	Logical: If TRUE, check that the version of `ergm` running on the slave nodes is the same as that running on the master node.
`...`	Additional arguments, passed to other functions This argument is helpful because it collects any control parameters that have been deprecated; a warning message is printed in case of deprecated arguments.

Details

This function is only used within a call to the `simulate` function. See the usage section in `simulate.ergm` for details.

Value

A list with arguments as components.

See Also

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

`degreedist` *Computes and Returns the Degree Distribution Information for a Given Network*

Description

The `degreedist` function computes and returns the degree distribution (number of vertices in the network with each degree value) for a given network.

The `degreedistfactor` function returns the cross table of the degree distribution for a network and a given factor (vertex attribute name)

Usage

```
degreedist(g, print = TRUE)
```

Arguments

`g`	a `network` object
`print`	logical, whether to print the degree distribution; default=TRUE

Details

Calculates the degrees using the appropriate ergm terms for for network type.

Value

- if directed – a matrix of the distributions of in and out degrees; this is row bound and only contains degrees for which one of the in or out distributions has a positive count
- if bipartite – a list containing the degree distributions of b1 and b2
- otherwise – a vector of the positive values in the degree distribution

Examples

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

`ecoli` *Two versions of an E. Coli network dataset*

Description

This network data set comprises two versions of a biological network in which the nodes are operons in *Escherichia Coli* and a directed edge from one node to another indicates that the first encodes the transcription factor that regulates the second.

Usage

```
data(ecoli)
```

Details

The network object `ecoli1` is directed, with 423 nodes and 519 arcs. The object `ecoli2` is an undirected version of the same network, in which all arcs are treated as edges and the five isolated nodes (which exhibit only self-regulation in `ecoli1`) are removed, leaving 418 nodes.

Licenses and Citation

When publishing results obtained using this data set, the original authors (Salgado et al, 2001; Shen-Orr et al, 2002) should be cited, along with this `R` package.

Source

The data set is based on the RegulonDB network (Salgado et al, 2001) and was modified by Shen-Orr et al (2002).

References

Salgado et al (2001), Regulondb (version 3.2): Transcriptional Regulation and Operon Organization in Escherichia Coli K-12, *Nucleic Acids Research*, 29(1): 72-74.

Shen-Orr et al (2002), Network Motifs in the Transcriptional Regulation Network of Escerichia Coli, *Nature Genetics*, 31(1): 64-68.

---

enformulate.curved — *Convert a curved ERGM into a form suitable as initial values for the same ergm.*

Description

The generic `enformulate.curved` converts an `ergm` object or formula of a model with curved terms to the variant in which the curved parameters embedded into the formula and are removed from the parameter vector. This is the form required by `ergm` calls.

Usage

```
## S3 method for class 'ergm'
enformulate.curved(object, ...)
## S3 method for class 'formula'
enformulate.curved(object, theta, response=NULL, ...)
```

Arguments

`object`	An `ergm` object or an ERGM formula. The curved terms of the given formula (or the formula used in the fit) must have all of their arguments passed by name.
`theta`	Curved model parameter configuration.
`response`	Not for release.
`...`	Unused at this time.

Details

Because of a current kludge in `ergm`, output from one run cannot be directly passed as initial values (`control.ergm(init=)`) for the next run if any of the terms are curved. One workaround is to embed the curved parameters into the formula (while keeping `fixed=FALSE`) and remove them from `control.ergm(init=)`.

This function automates this process for curved ERGM terms included with the `ergm` package. It does not work with curved terms not included in ergm.

Value

A list with the following components:

`formula`	The formula with curved parameter estimates incorporated.
`theta`	The coefficient vector with curved parameter estimates removed.

See Also

ergm, simulate.ergm

Examples

```
data(sampson)
gest<-ergm(samplike~edges+gwesp(alpha=.5, fixed=FALSE),
    control=control.ergm(MCMLE.maxit=1))
# Error:
gest2<-try(ergm(gest$formula, control=control.ergm(init=coef(gest), MCMLE.maxit=2)))
print(gest2)

# Works:
tmp<-enformulate.curved(gest)
tmp
gest2<-try(ergm(tmp$formula, control=control.ergm(init=tmp$theta, MCMLE.maxit=2)))
summary(gest2)
```

---

ergm *Exponential-Family Random Graph Models*

Description

ergm is used to fit exponential-family random graph models (ERGMs), in which the probability of a given network, y, on a set of nodes is $h(y)\exp\{\eta(\theta)\cdot g(y)\}/c(\theta)$, where $h(y)$ is the reference measure (usually $h(y)=1$), $g(y)$ is a vector of network statistics for y, $\eta(\theta)$ is a natural parameter vector of the same length (with $\eta(\theta)=\theta$ for most terms), and $c(\theta)$ is the normalizing constant for the distribution. ergm can return a maximum pseudo-likelihood estimate, an approximate maximum likelihood estimate based on a Monte Carlo scheme, or an approximate contrastive divergence estimate based on a similar scheme.

(For an overview of the package, see ergm-package.)

Usage

```
ergm (formula,
      response=NULL,
      reference=~Bernoulli,
      constraints=~.,
      offset.coef=NULL,
      target.stats=NULL,
      eval.loglik=TRUE,
      estimate=c("MLE", "MPLE", "CD"),
      control=control.ergm(),
      verbose=FALSE,
      ...)
```

Arguments

`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` and Morris, Handcock and Hunter (2008) for binary ERGM terms and Krivitsky (2012) for valued ERGM terms (terms for weighted edges). To create a `network` object in R, use the `network()` function, then add nodal attributes to it using the `%v%` operator if necessary. Enclosing a model term in `offset()` fixes its value to one specified in `offset.coef`.
`response`	Name of the edge attribute whose value is to be modeled. Defaults to `NULL` for simple presence or absence, modeled via binary ERGM terms. Passing anything but `NULL` uses valued ERGM terms.
`reference`	A one-sided formula specifying the reference measure ($h(y)$) to be used. (Defaults to `~Bernoulli`.) See help for ERGM reference measures implemented in the `ergm` package.
`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 and the reference measure, 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 `control.ergm`. The default is `~.`, for an unconstrained model. See the ERGM constraints documentation for the constraints implemented in the `ergm` package. Other packages may add their own constraints. Note that not all possible combinations of constraints and reference measures are supported.
`offset.coef`	A vector of coefficients for the offset terms.
`target.stats`	vector of "observed network statistics," if these statistics are for some reason different than the actual statistics of the network on the left-hand side of `formula`. Equivalently, this vector is the mean-value parameter values for the model. If this is given, the algorithm finds the natural parameter values corresponding to these mean-value parameters. If `NULL`, the mean-value parameters used are the observed statistics of the network in the formula.
`eval.loglik`	Logical: For dyad-dependent models, if TRUE, use bridge sampling to evaluate the log-likelihoood associated with the fit. Has no effect for dyad-independent models. Since bridge sampling takes additional time, setting to FALSE may speed performance if likelihood values (and likelihood-based values like AIC and BIC) are not needed.
`estimate`	If "MPLE," then the maximum pseudolikelihood estimator is returned. If "MLE" (the default), then an approximate maximum likelihood estimator is returned. For certain models, the MPLE and MLE are equivalent, in which case this argument is ignored. (To force MCMC-based approximate likelihood calculation even when the MLE and MPLE are the same, see the `force.main` argument

	of `control.ergm`. If "CD" (*EXPERIMENTAL*), the Monte-Carlo contrastive divergence estimate is returned.)
`control`	A list of control parameters for algorithm tuning. Constructed using `control.ergm`.
`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.

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.
`sample.obs`	As `sample`, but for the constrained sample.
`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 `estimate="MPLE"` then `MCMCtheta` equals the MPLE.
`loglikelihood`	The approximate change in log-likelihood in the last iteration. 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.
`covar`	Approximate covariance matrix for the MLE, based on the inverse Hessian of the approximated loglikelihood evaluated at the maximizer.
`failure`	Logical: Did the MCMC estimation fail?
`network`	Original network
`newnetwork`	The final network at the end of the MCMC simulation
`coef.init`	The initial value of θ.
`est.cov`	The covariance matrix of the model statistics in the final MCMC sample.
`coef.hist, steplen.hist, stats.hist, stats.obs.hist`	For the MCMLE method, the history of coefficients, Hummel step lengths, and average model statistics for each iteration..
`control`	The control list passed to the call.
`etamap`	The set of functions mapping the true parameter theta to the canonical parameter eta (irrelevant except in a curved exponential family model)
`formula`	The original `formula` entered into the `ergm` function.
`target.stats`	The target.stats used during estimation (passed through from the Arguments)

`target.esteq`	Used for curved models to preserve the target mean values of the curved terms. It is identical to target.stats for non-curved models.
`constrained`	The list of constraints implied by the constraints used by original `ergm` call
`constraints`	Constraints used during estimation (passed through from the Arguments)
`reference`	The reference measure used during estimation (passed through from the Arguments)
`estimate`	The estimation method used (passed through from the Arguments).
`offset`	vector of logical telling which model parameters are to be set at a fixed value (i.e., not estimated).
`drop`	If `control$drop=TRUE`, a numeric vector indicating which terms were dropped due to to extreme values of the corresponding statistics on the observed network, and how: `0` The term was not dropped. `-1` The term was at its minimum and the coefficient was fixed at `-Inf`. `+1` The term was at its maximum and the coefficient was fixed at `+Inf`.
`estimable`	A logical vector indicating which terms could not be estimated due to a `constraints` constraint fixing that term at a constant value.
`null.lik`	Log-likelihood of the null model. Valid only for unconstrained models.
`mle.lik`	The approximate log-likelihood for the MLE. The value is only approximate because it is estimated based on the MCMC random sample.
`degeneracy.value`	Score calculated to assess the degree of degeneracy in the model. Only shows when MCMLE.check.degeneracy is TRUE in `control.ergm`.
`degeneracy.type`	Supporting output for `degeneracy.value`. Only shows when MCMLE.check.degeneracy is TRUE in `control.ergm`. Mainly for internal use.

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.

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 `control.ergm(style="Robbins-Monro")` option.

The mechanism for proposing new networks for the MCMC sampling scheme, which is a Metropolis-Hastings algorithm, depends on two things: The `constraints`, which define the set of possible networks that could be proposed in a particular Markov chain step, and the weights placed on these possible steps by the proposal distribution. The former may be controlled using the `constraints` argument described above. The latter may be controlled using the `prop.weights` argument to the `control.ergm` function.

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

References

Admiraal R, Handcock MS (2007). **networksis**: Simulate bipartite graphs with fixed marginals through sequential importance sampling. Statnet Project, Seattle, WA. Version 1. `statnet.org`.

Bender-deMoll S, Morris M, Moody J (2008). Prototype Packages for Managing and Animating Longitudinal Network Data: **dynamicnetwork** and **rSoNIA**. *Journal of Statistical Software*, 24(7). `http://www.jstatsoft.org/v24/i07/`.

Butts CT (2007). **sna**: Tools for Social Network Analysis. R package version 2.3-2. `http://CRAN.R-project.org/package=sna`.

Butts CT (2008). **network**: A Package for Managing Relational Data in R. *Journal of Statistical Software*, 24(2). `http://www.jstatsoft.org/v24/i02/`.

Butts C (2015). **network**: The Statnet Project (http://www.statnet.org). R package version 1.12.0, `http://CRAN.R-project.org/package=network`.

Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008a). A **statnet** Tutorial. *Journal of Statistical Software*, 24(8). `http://www.jstatsoft.org/v24/i08/`.

Goodreau SM, Kitts J, Morris M (2008b). Birds of a Feather, or Friend of a Friend? Using Exponential Random Graph Models to Investigate Adolescent Social Networks. *Demography*, 45, in press.

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`

Handcock MS (2003b). **degreenet**: Models for Skewed Count Distributions Relevant to Networks. Statnet Project, Seattle, WA. Version 1.0, `statnet.org`.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003a). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. Statnet Project, Seattle, WA. Version 2, `statnet.org`.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003b). **statnet**: Software Tools for the Statistical Modeling of Network Data. Statnet Project, Seattle, WA. Version 2, `statnet.org`.

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

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008b). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software*, 24(3). `http://www.jstatsoft.org/v24/i03/`.

Krivitsky PN (2012). Exponential-Family Random Graph Models for Valued Networks. *Electronic Journal of Statistics*, 2012, 6, 1100-1128. `doi:10.1214/12-EJS696`

Morris M, Handcock MS, Hunter DR (2008). Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 24(4). `http://www.jstatsoft.org/v24/i04/`.

Snijders, T.A.B. (2002), Markov Chain Monte Carlo Estimation of Exponential Random Graph Models. Journal of Social Structure. Available from http://www.cmu.edu/joss/content/articles/volume3/Snijders.pdf.

See Also

network, %v%, %n%, ergm-terms, ergmMPLE, 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=flomarriage %v% "wealth" / 20, 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"),
  control=control.ergm(MCMC.samplesize=10000))
summary(gest)

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

---

ergm-constraints *Sample Space Constraints for Exponential-Family Random Graph Models*

Description

ergm is used to fit exponential-family random graph models (ERGMs), in which the probability of a given network, y, on a set of nodes is $h(y)\exp\{\eta(\theta)\cdot g(y)\}/c(\theta)$, where $h(y)$ is the reference measure (usually $h(y) = 1$), $g(y)$ is a vector of network statistics for y, $\eta(\theta)$ is a natural parameter vector of the same length (with $\eta(\theta) = \theta$ for most terms), and $c(\theta)$ is the normalizing constant for the distribution.

This page describes the constraints (the networks y for which $h(y) > 0$) that are included with the ergm package. Other packages may add new constraints.

Constraints implemented in the `ergm` package

`.` **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.

`blockdiag(attrname)` Force a block-diagonal structure (and its bipartite analogue) on the network. Only dyads (i, j) for which `attrname(i)==attrname(j)` can have edges.

Note that the current implementation requires that blocks be contiguous for “unipartite” graphs, and for bipartite graphs, they must be contiguous within a partition and must have the same ordering in both partitions. (They do not, however, require that all blocks be represented in both partitions, but those that overlap must have the same order.)

`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. If the network is directed, both indegree and outdegree are preserved.

`odegrees`**,** `idegrees`**,** `b1degrees`**,** `b2degrees` For directed networks, `odegrees` preserves the outdegree of each vertex of the given network, while allowing indegree to vary, and conversely for `idegrees`. `b1degrees` and `b2degrees` perform a similar function for bipartite networks.

`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.

`idegreedist` **and** `odegreedist` 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.

`observed` Preserve the observed dyads of the given network.

`fixedas(present,absent)` Preserve the edges in ’present’ and preclude the edges in ’absent’. Both ’present’ and ’absent’ can take input object as edgelist and network, the latter will convert to the corresponding edgelist.

`fixallbut(free.dyads)` Preserve the dyad status in all but free.dyads. free.dyads can take input object as edgelist and network, the latter will convert to the corresponding edgelist.

Not all combinations of the above are supported.

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.

References

Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008a). A **statnet** Tutorial. *Journal of Statistical Software*, 24(8). `http://www.jstatsoft.org/v24/i08/`.

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

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008b). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software*, 24(3). `http://www.jstatsoft.org/v24/i03/`.

Krivitsky PN (2012). Exponential-Family Random Graph Models for Valued Networks. *Electronic Journal of Statistics*, 2012, 6, 1100-1128. `doi:10.1214/12-EJS696`

Morris M, Handcock MS, Hunter DR (2008). Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 24(4). `http://www.jstatsoft.org/v24/i04/`.

`ergm-parallel` *Parallel Processing in the `ergm` Package*

Description

For estimation that require MCMC, `ergm` can take advantage of multiple CPUs or CPU cores on the system on which it runs, as well as computing clusters. It uses package `parallel` and `snow` to facilitate this, and supports all cluster types that they does.

The number of nodes used and the parallel API are controlled using the `parallel` and `parallel.type` arguments passed to the control functions, such as `control.ergm`.

The `ergm.getCluster` function is usually called internally by the ergm process (in `ergm.getMCMCsample`) and will attempt to start the appropriate type of cluster indicated by the `control.ergm` settings. The `ergm.stopCluster` is helpful if the user has directly created a cluster.

Further details on the various cluster types are included below.

Usage

```
ergm.getCluster(control, verbose=FALSE)

ergm.stopCluster(object, ...)
```

Arguments

`control`	a `control.ergm` list of parameter values from which the parallel settings should be read
`object`	an object, probably of class "cluster"
`verbose`	logical, should detailed status info be printed to console
`...`	not currently used

PSOCK clusters

The `parallel` package is used with PSOCK clusters by default, to utilize multiple cores on a system. The number of cores on a system can be determined with the `detectCores` function.

This method works with the base installation of R on all platforms, and does not require additional software.

For more advanced applications, such as clusters that span multiple machines on a network, the clusters can be initialized manually, and passed into `ergm` using the `parallel` control argument. See the second example below.

MPI clusters

To use MPI to accelerate ERGM sampling, pass the control parameter `parallel.type="MPI"`. `ergm` requires the `snow` and `Rmpi` packages to communicate with an MPI cluster.

Using MPI clusters requires the system to have an existing MPI installation. See the MPI documentation for your particular platform for instructions.

To use ergm across multiple machines in a high performance computing environment, see the section "User initiated clusters" below.

User initiated clusters

A cluster can be passed into `ergm` with the `parallel` control parameter. `ergm` will detect the number of nodes in the cluster, and use all of them for MCMC sampling. This method is flexible: it will accept any cluster type that is compatible with `snow` or `parallel` packages. Usage examples for a multiple-machine high performance MPI cluster can be found at the statnet wiki: `https://statnet.csde.washington.edu/trac/wiki/ergmParallel`

Examples

```
# Uses 2 SOCK clusters for MCMLE estimation
data(faux.mesa.high)
nw <- faux.mesa.high
fauxmodel.01 <- ergm(nw ~ edges + isolates + gwesp(0.2, fixed=TRUE),
                     control=control.ergm(parallel=2, parallel.type="PSOCK"))
summary(fauxmodel.01)
```

`ergm-references` *Reference Measures for Exponential-Family Random Graph Models*

Description

This page describes the possible reference measures (baseline distributions) for found in the `ergm` package, particularly the default (Bernoulli) reference measure for binary ERGMs.

The reference measure is specified on the RHS of a one-sided formula passed as the `reference` argument to `ergm`. See the `ergm` documentation for a complete description of how reference measures are specified.

Possible reference measures to represent baseline distributions

Reference measures currently available are:

`Bernoulli` *Bernoulli-reference ERGM:* Specifies each dyad's baseline distribution to be Bernoulli with probability of the tie being 0.5. This is the only reference measure used in binary mode.

`DiscUnif(a,b)` *Discrete-Uniform-reference ERGM:* Specifies each dyad's baseline distribution to be discrete uniform between a and b (both inclusive): $h(y) = 1$, with the support being a,a+1,...,b-1,b. At this time, both a and b must be finite.

`Unif(a,b)` *Coninuous-Uniform-reference ERGM:* Specifies each dyad's baseline distribution to be continuous uniform between a and b (both inclusive): $h(y) = 1$, with the support being [a,b]. At this time, both a and b must be finite.

`StdNormal` *Standard-Normal-reference ERGM:* Specifies each dyad's baseline distribution to be the normal distribution with mean 0 and variance 1.

References

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008b). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software*, 24(3). `http://www.jstatsoft.org/v24/i03/`.

Krivitsky PN (2012). Exponential-Family Random Graph Models for Valued Networks. *Electronic Journal of Statistics*, 2012, 6, 1100-1128. `doi:10.1214/12-EJS696`

See Also

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

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

Description

The function ergm is used to fit exponential random graph models, in which the probability of a given network, y, on a set of nodes is $h(y)\exp\{\eta(\theta)\cdot g(y)\}/c(\theta)$, where $h(y)$ is the reference measure (for valued network models), $g(y)$ is a vector of network statistics for y, $\eta(\theta)$ is a natural parameter vector of the same length (with $\eta(\theta) = \theta$ for most terms), 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) included in ergm package. Other packages may add their own terms, and package ergm.userterms provides tools for implementing them.

The current recommendation for any package implementing additional terms is to create a help file with a name or alias `ergm-terms`, so that `help("ergm-terms")` will list ERGM terms available from all loaded packages.

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.

Binary and valued ERGM terms

ergm functions such as ergm and simulate (for ERGMs) may operate in two modes: binary and weighted/valued, with the latter activated by passing a non-NULL value as the `response` argument, giving the edge attribute name to be modeled/simulated.

Binary ERGM statistics cannot be used in valued mode and vice versa. However, a substantial number of binary ERGM statistics — particularly the ones with dyadic indepenence — have simple generalizations to valued ERGMs, and have been adapted in ergm. They have the same form as their binary ERGM counterparts, with an additional argument: `form`, which, at this time, has two possible values: `"sum"` (the default) and `"nonzero"`. The former creates a statistic of the form $\sum_{i,j} x_{i,j} y_{i,j}$, where $y_{i,j}$ is the value of dyad (i, j) and $x_{i,j}$ is the term's covariate associated with it. The latter computes the binary version, with the edge considered to be present if its value is not 0.

Valued version of some binary ERGM terms have an argument `threshold`, which sets the value above which a dyad is conidered to have a tie. (Value less than or equal to `threshold` is considered a nontie.)

Covariate transformations

Some terms taking nodal or dyadic covariates take optional `transform` and `transformname` arguments. `transform` should be a function with one argument, taking a data structure of the same mode as the covariate and returning a similarly structured data structure, transforming the covariate as needed.

For example, `nodecov("a", transform=function(x) x^2)` will add a nodal covariate having the square of the value of the nodal attribute `"a"`.

`transformname`, if given, will be added to the term's name to help identify it.

Terms to represent network statistics included in the `ergm` package

A cross-referenced html version of the term documentation is is available via `vignette('ergm-term-crossRef')` and terms can also be searched via `search.ergmTerms`.

`absdiff(attrname, pow=1)` **(binary) (dyad-independent) (frequently-used) (directed) (undirected) (quantitative nod**
: *Absolute difference:* The `attrname` argument is a character string giving the name of a quantitative 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])^pow` for all edges (i,j) in the network.

`absdiffcat(attrname, base=NULL)` **(binary) (dyad-independent) (directed) (undirected) (categorical nodal attribute**
: *Categorical absolute difference:* The `attrname` argument is a character string giving the name of a quantitative 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)`. Note that this term should generally be used only when the quantitative attribute has a limited number of possible values; an example is the `"Grade"` attribute of the `faux.mesa.high` or `faux.magnolia.high` datasets.

`altkstar(lambda, fixed=FALSE)` **(binary) (undirected) (curved) (categorical nodal attribute)**
: *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). 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 Hunter (2007) 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(attrname=NULL, diff=FALSE, keep=NULL)` **(binary) (directed) (dyad-independent) (triad-related)**
: *Asymmetric dyads:* This term adds one network statistic to the model equal to 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. If the optional `attrname` argument is used, only asymmetric pairs that match on the named vertex attribute are counted. The optional modifiers `diff` and `keep` are used in the same way as for the `nodematch` term; refer to this term for details and an example.

`atleast(threshold=0)` **(valued) (directed) (undirected) (dyad-independent)** *Number of dyads with values greater than or equal to a threshold* Adds one statistic equaling to the number of dyads whose values equal or exceed `threshold`.

`atmost(threshold=0)` **(valued) (directed) (undirected) (dyad-independent)** *Number of dyads with values less than or equal to a threshold* Adds one statistic equaling to the number of dyads whose values equal or are exceeded by `threshold`.

`b1concurrent(by=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)** *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 argument `by` is a character string giving the name of an attribute in the network's vertex attribute list; it functions just like the `by` argument of the `b1degree` term. Without the optional argument, this statistic is equivalent to `b1mindegree(2)`. This term can only be used with undirected bipartite networks.

`b1cov(attrname, transform, transformname)` **(binary) (undirected) (bipartite) (dyadic-independent) (quantitative** *Main effect of a covariate for the first mode in a bipartite (aka two-mode) network:* The `attrname` argument is a character string giving the name of a numeric (not categorical) attribute in the network's vertex attribute list. This term adds a single network statistic to the model equaling the total value of `attrname(i)` for all edges (i, j) in the network. This term may only be used with bipartite networks. For categorical attributes, see `b1factor`.

`b1degrange(from, to=+Inf, by=NULL, homophily=FALSE)` **(binary) (bipartite) (undirected)** *Degree range for the first mode in a bipartite (a.k.a. two-mode) network:* The `from` and `to` arguments are vectors of distinct integers (or `+Inf`, for `to` (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the ith such statistic equals the number of nodes of the first mode ("actors") in the network of degree greater than or equal to `from[i]` but strictly less than `to[i]`, i.e. with edge count in semiopen interval `[from,to)`. The optional argument `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and `homophily` is `FALSE` (the default), then separate degree range statistics are calculated for nodes having each separate value of the attribute.

This term can only be used with bipartite networks; for directed networks see `idegrange` and `odegrange`. For undirected networks, see `degrange`, and see `b2degrange` for degrees of the second mode ("events").

`b1degree(d, by=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute) (frequently-used)** *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 argument `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then each node's degree is tabulated only with other nodes having the same value of the by attribute. This term can only be used with undirected bipartite networks.

b1factor(attrname, base=1) **(binary) (bipartite) (undirected) (dyad-independent) (frequently-used) (categorical no**

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, base=1, means that the smallest (i.e., first in sorted order) attribute value is omitted. 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). This term can only be used with undirected bipartite networks.

b1mindegree(d) **(binary) (bipartite) (undirected)** *Minimum 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 in the first mode of a bipartite network with at least degree d[i]. The first mode of a bipartite network object is sometimes known as the "actor" mode. This term can only be used with undirected bipartite networks.

b1nodematch(attrname, diff=FALSE, keep=NULL, by=NULL, alpha=1, beta=1, byb2attr=NULL) **(binary) (bipart**

Nodal attribute-based homophily effect for the first mode in a bipartite (aka two-mode) network: This term is introduced in Bomiriya et al (2014). The attrname argument is a character string giving the name of a categorical attribute in the network's vertex attribute list. Out of the two arguments (discount parameters) alpha and beta, both which takes values from [0,1], only one should be set at a time. If none is set to a value other than 1, this term will simply be a homophily based two-star statistic. This term adds one statistic to the model unless diff is set to TRUE, in which case the term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the attrname attribute. To include only the attribute values you wish, use the keep argument. If an alpha discount parameter is used, each of these statistics gives the sum of the number of common second-mode nodes raised to the power alpha for each pair of first-mode nodes with that attribute. If a beta discount parameter is used, each of these statistics gives half the sum of the number of two-paths with two first-mode nodes with that attribute as the two ends of the two path raised to the power beta for each edge in the network. The byb2attr argument is a character string giving the name of a second mode categorical attribute in the network's attribute list. Setting this argument will separate the orginal statistics based on the values of the set second mode attribute— i.e. for example, if diff is FALSE, then the sum of all the statistics for each level of this second-mode attribute will be equal to the original b1nodematch statistic where byb2attr set to NULL. This term can only be used with undirected bipartite networks.

b1star(k, attrname=NULL) **(binary) (bipartite) (undirected) (categorical nodal attribute)** *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 $\{O_1, \ldots, O_k\}$ such that the ties $\{N, O_i\}$ exist for $i = 1, \ldots, k$. 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 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 bipartite networks. Note that `b1star(1)` is equal to `b2star(1)` and to `edges`.

`b1starmix(k, attrname, base=NULL, diff=TRUE)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**
Mixing matrix for k-stars centered on the first mode of a bipartite network: Only a single value of k is allowed. This term counts all k-stars in which the b2 nodes (called events in some contexts) are homophilous in the sense that they all share the same value of `attrname`. However, the `b1` node (in some contexts, the actor) at the center of the k-star does NOT have to have the same value as the b2 nodes; indeed, the values taken by the `b1` nodes may be completely distinct from those of the b2 nodes, which allows for the use of this term in cases where there are two separate nodal attributes, one for the `b1` nodes and another for the b2 nodes (in this case, however, these two attributes should be combined to form a single nodal attribute called `attrname`. A different statistic is created for each value of `attrname` seen in a `b1` node, even if no k-stars are observed with this value. Whether a different statistic is created for each value seen in a `b2` node depends on the value of the `diff` argument: When `diff=TRUE`, the default, a different statistic is created for each value and thus the behavior of this term is reminiscent of the `nodemix` term, from which it takes its name; when `diff=FALSE`, all homophilous k-stars are counted together, though these k-stars are still categorized according to the value of the central `b1` node. The `base` term may be used to control which of the possible terms are left out of the model: By default, all terms are included, but if `base` is set to a vector of indices then the corresponding terms (in the order they would be created when `base=NULL`) are left out.

`b1twostar(b1attrname, b2attrname, base=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**
Two-star census for central nodes centered on the first mode of a bipartite network: This term takes two nodal attribute names, one for `b1` nodes (actors in some contexts) and one for b2 nodes (events in some contexts). Only `b1attrname` is required; if `b2attrname` is not passed, it is assumed to be the same as `b1attrname`. Assuming that there are n_1 values of `b1attrname` among the b1 nodes and n_2 values of `b2attrname` among the b2 nodes, then the total number of distinct categories of two stars according to these two attributes is $n_1(n_2)(n_2 + 1)/2$. This model term creates a distinct statistic counting each of these categories. The `base` term may be used to leave some of these categories out; when passed as a vector of integer indices (in the order the statistics would be created when `base=NULL`), the corresponding terms will be left out.

`b2concurrent(by=NULL)` **(binary) (bipartite) (undirected) (frequently-used)** *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 argument `by` is a character string giving the name of an attribute in the network's vertex attribute list; it functions just like the `by` argument of the `b2degree` term. Without the optional argument, this statistic is equivalent to `b2mindegree(2)`. This term can only be used with undirected bipartite networks.

`b2cov(attrname, transform, transformname)` **(binary) (undirected) (bipartite) (dyadic-independent) (quantitative**
Main effect of a covariate for the second mode in a bipartite (aka two-mode) network: The `attrname` argument is a character string giving the name of a numeric (not categorical) attribute in the network's vertex attribute list. This term adds a single network statistic to the

model equaling the total value of `attrname(j)` for all edges (i, j) in the network. This term may only be used with bipartite networks. For categorical attributes, see `b2factor`.

`b2degrange(from, to=+Inf, by=NULL, homophily=FALSE)` **(binary) (bipartite) (undirected)**
Degree range for the second mode in a bipartite (a.k.a. two-mode) network: The `from` and `to` arguments are vectors of distinct integers (or `+Inf`, for `to` (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the ith such statistic equals the number of nodes of the second mode ("events") in the network of degree greater than or equal to `from[i]` but strictly less than `to[i]`, i.e. with edge count in semiopen interval `[from,to)`. The optional argument `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and `homophily` is `FALSE` (the default), then separate degree range statistics are calculated for nodes having each separate value of the attribute.

This term can only be used with bipartite networks; for directed networks see `idegrange` and `odegrange`. For undirected networks, see `degrange`, and see `b1degrange` for degrees of the first mode ("actors").

`b2degree(d, by=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute) (frequently-used)**
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 `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then each node's degree is tabulated only with other nodes having the same value of the `by` attribute. This term can only be used with undirected bipartite networks.

`b2factor(attrname, base=1)` **(binary) (bipartite) (undirected) (dyad-independent) (categorical nodal attribute) (fre**
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, `base=1`, means that the smallest (i.e., first in sorted order) attribute value is omitted. 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)`. This term can only be used with undirected bipartite networks.

`b2mindegree(d)` **(binary) (bipartite) (undirected)** *Minimum 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 in the second mode of a bipartite network with at least degree `d[i]`. The second mode of a bipartite network object is sometimes known as the "event" mode. This term can only be used with undirected bipartite networks.

`b2nodematch(attrname, diff=FALSE, keep=NULL, by=NULL, alpha=1, beta=1, byb1attr=NULL)` **(binary) (bipart**
Nodal attribute-based homophily effect for the second mode in a bipartite (aka two-mode) network: This term is introduced in Bomiriya et al (2014). The `attrname` argument is a character string giving the name of a categorical attribute in the network's vertex attribute list. Out of the two arguments (discount parameters) `alpha` and `beta`, both which takes values from [0,1], only one should be set at a time. If none is set to a value other than 1, this term will simply be a homophily based two-star statistic. This term adds one statistic to the model unless `diff` is set to `TRUE`, in which case the term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attrname` attribute. To include only the attribute values you wish, use the `keep` argument. If an `alpha` discount parameter is used, each of these statistics gives the sum of the number of common first-mode nodes raised to the power `alpha` for each pair of second-mode nodes with that attribute. If a `beta` discount parameter is used, each of these statistics gives half the sum of the number of two-paths with two second-mode nodes with that attribute as the two ends of the two path raised to the power `beta` for each edge in the network. The `byb1attr` argument is a character string giving the name of a first mode categorical attribute in the network's attribute list. Setting this argument will separate the orginal statistics based on the values of the set first mode attribute— i.e. for example, if `diff` is `FALSE`, then the sum of all the statistics for each level of this first-mode attribute will be equal to the original `b2nodematch` statistic where `byb1attr` set to `NULL`. This term can only be used with undirected bipartite networks.

`b2star(k, attrname=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)** *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 bipartite networks. Note that `b2star(1)` is equal to `b1star(1)` and to `edges`.

`b2starmix(k, attrname, base=NULL, diff=TRUE)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**
Mixing matrix for k-stars centered on the second mode of a bipartite network: This term is exactly the same as `b1starmix` except that the roles of `b1` and `b2` are reversed.

`b2twostar(b1attrname, b2attrname, base=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**
Two-star census for central nodes centered on the second mode of a bipartite network: This term is exactly the same as `b1twostar` except that the roles of `b1` and `b2` are reversed.

`balance` **(binary) (triad-related) (directed) (undirected)** *Balanced triads:* This term adds one network statistic to the model equal to the number of triads in the network that are balanced. The balanced triads are those of type `102` or `300` in the categorization of Davis and Leinhardt (1972). For details on the 16 possible triad types, see `?triad.classify` in the `{sna}` package. For an undirected network, the balanced triads are those with an even number of ties (i.e., 0 and 2).

`coincidence(d=NULL,active=0)` **(binary) (bipartite) (undirected)** *Coincident node count for the second mode in a bipartite (aka two-mode) network:* By default this term adds one network statistic to the model for each pair of nodes of mode two. It is equal to the number of (first mode) mutual partners of that pair. The first mode of a bipartite network object is sometimes known as the "actor" mode and the seconds as the "event" mode. So this is the number of actors going to both events in the pair. The optional argument `d` is a two-column matrix of (row-wise) pairs indices where the first row is less than the second row. The second optional argument, `active`, selects pairs for which the observed count is at least `active`. This term can only be used with undirected bipartite networks.

`concurrent(by=NULL)` **(binary) (undirected) (categorical nodal attribute)** *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 argument `by` is a character string giving the name of an attribute in the network's vertex attribute list; it functions just like the `by` argument of the `degree` term. This term can only be used with undirected networks.

`concurrentties(by=NULL)` **(binary) (undirected) (categorical nodal attribute)** *Concurrent tie count:* This term adds one network statistic to the model, equal to the number of ties incident on each actor beyond the first. The optional argument `by` is a character string giving the name of an attribute in the network's vertex attribute list; it functions just like the `by` argument of the `degree` term. This term can only be used with undirected networks.

`ctriple(attrname=NULL)` **(binary) (directed) (triad-related) (categorical nodal attribute) , a.k.a.** `ctriad` **(binary) (dir**
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)` **(binary) (directed) (undirected)** *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 networks. 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.

`cyclicalties(attrname=NULL)` **(binary) (directed),** `cyclicalties(threshold=0)` **(valued) (directed) (undirected)**
Cyclical ties: This term adds one statistic, equal to the number of ties $i \rightarrow j$ such that there exists a two-path from i to j. (Related to the `ttriple` term.) The binary version takes a nodal attribute `attrname`, and, if given, all three nodes involved (i, j, and the node on the two-path) must match on this attribute in order for $i \rightarrow j$ to be counted. The binary version of this term can only be used with directed networks. The valued version can be used with both directed and undirected.

`cyclicalweights(twopath="min",combine="max",affect="min")` **(valued) (directed) (undirected)**
Cyclical weights: This statistic implements the cyclical weights statistic, like that defined by Krivitsky (2012), Equation 13, but with the focus dyad being $y_{j,i}$ rather than $y_{i,j}$. The currently implemented options for `twopath` is the minimum of the constituent dyads (`"min"`) or their geometric mean (`"geomean"`); for `combine`, the maximum of the 2-path strengths (`"max"`) or their sum (`"sum"`); and for `affect`, the minimum of the focus dyad and the combined strength of the two paths (`"min"`) or their geometric mean (`"geomean"`). For each of

these options, the first (and the default) is more stable but also more conservative, while the second is more sensitive but more likely to induce a multimodal distribution of networks.

`ddsp(d, type="OTP")` **(binary) (directed)** *Directed dyadwise shared partners:* This term adds one network statistic to the model for each element in `d` where the ith such statistic equals the number of *dyads* in the network with exactly `d[i]` shared partners. This term can only be used with directed networks. Multiple shared partner definitions are possible; the `type` argument may be used to select the type of shared partner to be counted (see below for type codes). By default, outgoing two-paths are employed.

While there is only one shared partner configuration in the undirected case, nine distinct configurations are possible for directed graphs. Currently, edgewise shared partner terms may be defined with respect to five of these configurations; they are defined here as follows (using terminology from Butts (2008) and the `relevent` package):

Outgoing Two-path (OTP) vertex k is an OTP shared partner of ordered pair (i, j) iff $i \rightarrow k \rightarrow j$. Also known as "transitive shared partner".

Incoming Two-path (ITP) vertex k is an ITP shared partner of ordered pair (i, j) iff $j \rightarrow k \rightarrow i$. Also known as "cyclical shared partner"

Outgoing Shared Partner (OSP) vertex k is an OSP shared partner of ordered pair (i, j) iff $i \rightarrow k, j \rightarrow k$.

Incoming Shared Partner (ISP) vertex k is an ISP shared partner of ordered pair (i, j) iff $k \rightarrow i, k \rightarrow j$.

Note that Robins et al. (2009) define closely related statistics to several of the above, using slightly different terminology.

`degrange(from, to=+Inf, by=NULL, homophily=FALSE)` **(binary) (undirected) (categorical nodal attribute)** *Degree range:* The `from` and `to` arguments are vectors of distinct integers (or `+Inf`, for `to` (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the ith such statistic equals the number of nodes in the network of degree greater than or equal to `from[i]` but strictly less than `to[i]`, i.e. with edges in semiopen interval `[from,to)`. The optional argument `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and `homophily` is `FALSE` (the default), then separate degree range statistics are calculated for nodes having each separate value of the attribute.

This term can only be used with undirected networks; for directed networks see `idegrange` and `odegrange`. This term can be used with bipartite networks, and will count nodes of both first and second mode in the specified degree range. To count only nodes of the first mode ("actors"), use `b1degrange` and to count only those fo the second mode ("events"), use `b2degrange`.

`degree(d, by=NULL, homophily=FALSE)` **(binary) (undirected) (categorical nodal attribute) (frequently-used)** *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 argument `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and homophily is FALSE (the default), then separate degree statistics are calculated for nodes having each separate value of the attribute. This term can only be used with undirected networks; for directed networks see idegree and odegree.

degreepopularity **(binary) (undirected)** *Degree popularity:* This term adds one network statistic to the model equaling the sum over the actors of each actor's degree taken to the 3/2 power (or, equivalently, multiplied by its square root). This term is an undirected analog to the terms of Snijders et al. (2010), equations (11) and (12). This term can only be used with undirected networks.

degcrossprod **(binary) (undirected)** *Degree Cross-Product:* This term adds one network statistic equal to the mean of the cross-products of the degrees of all pairs of nodes in the network which are tied. Only coded for undirected networks.

degcor **(binary) (undirected)** *Degree Correlation:* This term adds one network statistic equal to the correlation of the degrees of all pairs of nodes in the network which are tied. Only coded for undirected networks.

density **(binary) (dyad-independent) (directed) (undirected)** *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 $n(n-1)$.

desp(d, type="OTP") **(binary) (directed)** *Directed edgewise shared partners:* This term adds one network statistic to the model for each element in d where the ith such statistic equals the number of *edges* in the network with exactly d[i] shared partners. This term can only be used with directed networks. Multiple shared partner definitions are possible; the type argument may be used to select the type of shared partner to be counted (see ddsp for type codes). By default, outgoing two-paths are employed.

dgwdsp(alpha=0, fixed=FALSE, cutoff=30, type="OTP") **(binary) (directed)** *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 > 0. Note that the GWDSP statistic is equal to the sum of GWNSP plus GWESP. 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. For a directed network, multiple shared partner definitions are possible; the type argument may be used to select the type of shared partner to employ (see ddsp for definitions). By default, outgoing two-paths are employed. The optional argument cutoff sets the number of underlying DSP terms to use in computing the statistics to reduce the computational burden.

dgwesp(alpha=0, fixed=FALSE, cutoff=30, type="OTP") **(binary) (directed)** *Geometrically weighted edgewise shared partner distribution:* This term adds a statistic equal to the geometrically weighted *edgewise* (not 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. For a directed network, multiple shared partner definitions are possible; the type argument may be used to select the type of shared partner to employ (see ddsp for definitions). By default, outgoing two-paths are employed. The optional argument cutoff sets the number of underlying ESP terms to use in computing the statistics to reduce the computational burden.

dgwnsp(alpha=0, fixed=FALSE, cutoff=30, type="OTP") **(binary) (directed)** *Geometrically weighted non-edgewise shared partner distribution:* This term is just like gwesp and gwdsp except it adds a statistic equal to the geometrically weighted nonedgewise (that is, over dyads that do not have an edge) 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. For a directed network, multiple shared partner definitions are possible; the `type` argument may be used to select the type of shared partner to employ (see `ddsp` for definitions). By default, outgoing two-paths are employed. The optional argument `cutoff` sets the number of underlying NSP terms to use in computing the statistics to reduce the computational burden.

`dnsp(d, type="OTP")` **(binary) (directed)** *Directed non-edgewise shared partners:* This term adds one network statistic to the model for each element in `d` where the ith such statistic equals the number of *non-edges* in the network with exactly `d[i]` shared partners. This term can only be used with directed networks. Multiple shared partner definitions are possible; the `type` argument may be used to select the type of shared partner to be counted (see `ddsp` for type codes). By default, outgoing two-paths are employed.

`dsp(d)` **(binary) (directed) (undirected)** *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).

`dyadcov(x, attrname=NULL)` **(binary) (dyad-independent) (directed) (undirected) (categorical nodal attribute)** *Dyadic covariate:* If the network is directed, `x` is either a (symmetric) matrix of covariates, one for each possible dyad (i, j), or an undirected network; if the latter, optional argument `attrname` provides the name of the quantitative edge attribute to use for covariate values (in this case, missing edges in `x` are assigned a covariate value of zero). This term adds three statistics to the model, each equal to the sum of the covariate values for all dyads occupying one of the three possible non-empty dyad states (mutual, upper-triangular asymmetric, and lower-triangular asymmetric dyads, respectively), with the empty or 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, equal to the sum of the covariate values for each edge appearing in the network. The `edgecov` and `dyadcov` terms are equivalent for undirected networks.

`edgecov(x, attrname=NULL)` **(binary) (dyad-independent) (directed) (undirected) (frequently-used) ,** `edgecov(x, at`
Edge covariate: The `x` argument is either a square matrix of covariates, one for each possible edge in the network, the name of a network attribute of covariates, or a network; if the latter, optional argument `attrname` provides the name of the quantitative edge attribute to use for covariate values (in this case, missing edges in `x` are assigned a covariate value of zero). This term adds one statistic to the model, equal to the sum of the covariate values for each edge appearing in the network. 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.

`edges` **(binary) (valued) (dyad-independent) (directed) (undirected) (frequently-used) , a.k.a** `nonzero` **(valued) (direct**
Edges: This term adds one network statistic equal to the number of edges (i.e. nonzero values) 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)` **(binary) (directed) (undirected)** *Edgewise shared partners:* This is just like the `dsp` term, except this term adds one network statistic to the model for each element in `d` where the ith such statistic equals the number of *edges* (rather than 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 edge and in the same direction).

`equalto(value=0, tolerance=0)` **(valued) (directed) (undirected) (dyadic-independent)** *Number of dyads with values equal to a specific value (within tolerance):* Adds one statistic equal to the number of dyads whose values are within `tolerance` of `value`, i.e., between `value-tolerance` and `value+tolerance`, inclusive.

`greaterthan(threshold=0)` **(valued) (directed) (undirected) (dyadic-independent)** *Number of dyads with values strictly greater than a threshold:* Adds one statistic equal to the number of dyads whose values exceed `threshold`.

`gwb1degree(decay, fixed=FALSE, cutoff=30)` **(binary) (bipartite) (undirected) (curved)** *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 decay controlled by the `decay` parameter, 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. The `decay` parameter is the same as theta_s in equation (14) in Hunter (2007). The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used as merely the starting value for the estimation in a curved exponential family model (the default). The optional argument `cutoff` is only relevant if `fixed=FALSE`. In that case it only uses this number of terms in computing the statistics to reduce the computational burden. This term can only be used with undirected bipartite networks.

`gwb2degree(decay, fixed=FALSE, cutoff=30)` **(binary) (bipartite) (undirected) (curved)** *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 decay controlled by the `decay` parameter, 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. The `decay` parameter is the same as theta_s in equation (14) in Hunter (2007). The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used as merely the starting value for the estimation in a curved exponential family model (the default). The optional argument `cutoff` is only relevant if `fixed=FALSE`. In that case it only uses this number of terms in computing the statistics to reduce the computational burden. This term can only be used with undirected bipartite networks.

`gwdegree(decay, fixed=FALSE, cutoff=30)` **(binary) (undirected) (curved) (frequently-used)** *Geometrically weighted degree distribution:* This term adds one network statistic to the model equal to the weighted degree distribution with decay controlled by the `decay` parameter. The `decay` parameter is the same as theta_s in equation (14) in Hunter (2007). The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used as merely the starting value for the estimation in a curved exponential family model (the default). The optional argument `cutoff` is only relevant if `fixed=FALSE`. In that case it only uses this number of terms in computing the statistics to reduce the computational burden. This term can only be used with undirected networks.

`gwdsp(alpha=0, fixed=FALSE, cutoff=30)` **(binary) (directed) (undirected) (curved)** *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` > 0. 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). The optional argument `cutoff` is only relevant if `fixed=FALSE`. In that case it only uses this number of terms in computing the statistics to reduce the computational burden.

`gwesp(alpha=0, fixed=FALSE, cutoff=30)` **(binary) (frequently-used) (directed) (undirected) (curved)** *Geometrically weighted edgewise shared partner distribution:* This term is just like `gwdsp` except it adds a statistic equal to the geometrically weighted *edgewise* (not 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 geometric weighting is over homogeneous shared partners only (i.e., only partners on a directed two-path connecting the nodes in the edge and in the same direction). The optional argument `cutoff` is only relevant if `fixed=FALSE`. In that case it only uses this number of terms in computing the statistics to reduce the computational burden.

`gwidegree(decay, fixed=FALSE, cutoff=30)` **(binary) (directed) (curved)** *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. The optional argument `cutoff` is only relevant if `fixed=FALSE`. In that case it only uses this number of terms in computing the statistics to reduce the computational burden.

`gwnsp(alpha=0, fixed=FALSE, cutoff=30)` **(binary) (directed) (undirected) (curved)** *Geometrically weighted nonedgewise shared partner distribution:* This term is just like `gwesp` and `gwdsp` except it adds a statistic equal to the geometrically weighted *nonedgewise* (that is, over dyads that do not have an edge) 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 non-edge and in the same direction). The optional argument `cutoff` is only relevant if `fixed=FALSE`. In that case it only uses this number of terms in computing the statistics to reduce the computational burden.

`gwodegree(decay, fixed=FALSE, cutoff=30)` **(binary) (directed) (curved)** *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. The optional argument `cutoff` is only relevant if `fixed=FALSE`. In that case it only uses this number of terms in computing the statistics to reduce the computational burden.

`hamming(x, cov, attrname=NULL)` **(binary) (dyad-independent) (directed) (undirected)** *Hamming distance:* This term adds one statistic to the model equal to the weighted or unweighted Hamming distance of the network from the network specified by `x`. (If no argument is given, `x` is taken to be the observed network, i.e., the network on the left side of the $\sim$ in the formula that defines the ERGM.) Unweighted Hamming distance is defined as the total number of pairs (i, j) (ordered or unordered, depending on whether the network is directed or undirected) on which the two networks differ. If the optional argument `cov` is specified, then the weighted Hamming distance is computed instead, where each pair (i, j) contributes a pre-specified weight toward the distance when the two networks differ on that pair. The argument `cov` is either a matrix of edgewise weights or a network; if the latter, the optional argument `attrname` provides the name of the edge attribute to use for weight values.

`hammingmix(attrname, x, base=0)` **(binary) (directed) (dyad-independent)** *Hamming distance within mixing:* This term adds one statistic to the model for every possible pairing of attribute values of the network for the vertex attribute named `attrname`. Each such statistic is the Hamming distance (i.e., the number of differences) between the appropriate subset of dyads in the network and the corresponding subset in `x`. The ordering of the attribute values is alphabetical. The option `base` gives the index of statistics to be omitted from the tabulation. For example `base=2` will omit the second statistic, making it the de facto reference category. This term can only be used with directed networks.

`idegrange(from, to=+Inf, by=NULL, homophily=FALSE)` **(binary) (directed) (categorical nodal attribute)** *In-degree range:* The `from` and `to` arguments are vectors of distinct integers (or `+Inf`, for `to` (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the ith such statistic equals the number of nodes in the network of in-degree greater than or equal to `from[i]` but strictly less than `to[i]`, i.e. with in-edge count in semiopen interval `[from,to)`. The optional argument `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and `homophily` is `FALSE` (the default), then separate degree range statistics are calculated for nodes having each separate value of the attribute.

This term can only be used with directed networks; for undirected networks (bipartite and not) see `degrange`. For degrees of specific modes of bipartite networks, see `b1degrange` and `b2degrange`. For in-degrees, see `idegrange`.

`idegree(d, by=NULL, homophily=FALSE)` **(binary) (directed) (categorical nodal attribute) (frequently-used)** *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 term `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and `homophily` is `FALSE` (the default), then separate degree statistics are calculated for nodes having each separate value of the attribute. This term can only be used with directed networks; for undirected networks see `degree`.

`idegreepopularity` **(binary) (directed)** *In-degree popularity:* This term adds one network statistic to the model equaling the sum over the actors of each actor's in-degree taken to the 3/2 power (or, equivalently, multiplied by its square root). This term is analogous to the term of Snijders et al. (2010), equation (11). This term can only be used with directed networks.

`ininterval(lower=-Inf, upper=+Inf, open=c(TRUE,TRUE))` **(valued) (directed) (undirected) (dyadic-independent)** *Number of dyads whose values are in an interval* Adds one statistic equaling to the number of dyads whose values are between `lower` and `upper`. Argument `open` is a `logical` vector of length 2 that controls whether the interval is open (exclusive) on the lower and on the upper end, respectively.

`intransitive` **(binary) (directed) (triad-related)** *Intransitive triads:* This term adds one statistic to the model, equal to the number of triads in the network that are intransitive. The intransitive triads are those of type `111D`, `201`, `111U`, `021C`, or `030C` in the categorization of Davis and Leinhardt (1972). For details on the 16 possible triad types, see `triad.classify` in the `sna` package. Note the distinction from the `ctriple` term. This term can only be used with directed networks.

`isolates` **(binary) (directed) (undirected) (frequently-used)** *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. For a directed network, an isolate is any node with both in-degree and out-degree equal to zero.

`istar(k, attrname=NULL)` **(binary) (directed) (categorical nodal attribute)** *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 $(O_j \to N)$ 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=NULL)` **(binary) (undirected) (categorical nodal attribute)** *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`.

`smallerthan(threshold=0)` **(valued) (directed) (undirected) (dyadic-independent)** *Number of dyads with values strictly smaller than a threshold:* Adds one statistic equaling to the number of dyads whose values exceeded by `threshold`.

`localtriangle(x)` **(binary) (triad-related) (directed) (undirected)** *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 $(i \to j), (j \to k)$ and either $(k \to i)$ or $(k \leftarrow i)$ where again all nodes are within the same neighborhood. The argument `x` is an undirected network or an symmetric adjacency matrix that specifies whether the two nodes are in the same neighborhood. Note that `triangle`, with or without an argument, is a special case of `localtriangle`.

`m2star` **(binary) (directed)** *Mixed 2-stars, a.k.a 2-paths:* This term adds one statistic to the model, equal to the number of mixed 2-stars in the network, where a mixed 2-star is a pair of distinct edges $(i \to j), (j \to k)$. 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. However, in the case of a 2-path the focus is usually on the endpoints i and k, whereas for a mixed 2-star the focus is usually on the midpoint j. This term can only be used with directed networks; for undirected networks see `kstar(2)`. See also `twopath`.

`meandeg` **(binary) (dyad-independent) (directed) (undirected)** *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(same=NULL, diff=FALSE, by=NULL, keep=NULL)` **(binary) (directed) (dyad-independent) (frequently-used),**
Mutuality: In binary ERGMs, equal to the number of pairs of actors i and j for which $(i \to j)$ and $(j \to i)$ both exist. For valued ERGMs, equal to $\sum_{i<j} m(y_{i,j}, y_{j,i})$, where m is determined by `form` argument: `"min"` for $\min(y_{i,j}, y_{j,i})$, `"nabsdiff"` for $-|y_{i,j}, y_{j,i}|$, `"product"` for $y_{i,j}y_{j,i}$, and `"geometric"` for $\sqrt{y_{i,j}}\sqrt{y_{j,i}}$. See Krivitsky (2012) for a discussion of these statistics. `form="threshold"` simply computes the binary `mutuality` after thresholding at `threshold`.

This term can only be used with directed networks. The binary version also has the following capabilities: if the optional `same` argument is passed the name of a vertex attribute, only mutual pairs that match on the attribute are counted; separate counts for each unique matching value can be obtained by using `diff=TRUE` with `same`; and if `by` is passed the name of a vertex attribute, then each node is counted separately for each mutual pair in which it occurs and the counts are tabulated by unique values of the attribute. This means that the sum of the mutual statistics when `by` is used will equal twice the standard mutual statistic. Only one of `same` or `by` may be used, and only the former is affected by `diff`; if both `same` and `by` are passed, `by` is ignored. Finally, if `keep` is passed a numerical vector, this vector of integers tells which statistics should be kept whenever the `mutual` term would ordinarily result in multiple statistics.

`nearsimmelian` **(binary) (directed) (triad-related)** *Near simmelian triads:* This term adds one statistic to the model equal to the number of near Simmelian triads, as defined by Krackhardt and Handcock (2007). This is a sub-graph of size three which is exactly one tie short of being complete. This term can only be used with directed networks.

`nodecov(attrname, transform, transformname)` **(binary) (dyad-independent) (frequently-used) (directed) (undirec**
Main effect of a covariate: The `attrname` argument is a character string giving the name of a numeric (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) in the network. For categorical attributes, see `nodefactor`. Note that for directed networks, `nodecov` equals `nodeicov` plus `nodeocov`.

`nodecovar` **(valued) (directed) (undirected) (quantitative nodal attribute)** *Uncentered covariance of dyad values incident on each actor:* This term adds one statistic equal to $\sum_{i,j,k}(y_{i,j}y_{i,k} + y_{k,j}y_{k,j})$. This can be viewed as a valued analog of the `kstar(2)` statistic.

`nodefactor(attrname, base=1)` **(binary) (dyad-independent) (directed) (undirected) (categorical nodal attribute) (f**
Factor attribute effect: The `attrname` argument is a character vector giving one or more names of categorical attributes 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 (or each combination of the attributes given). Each of these statistics gives the number of times a node with that attribute or those attributes appears in an edge in the network. In particular, for edges whose endpoints both have the same attribute values, this value is counted

twice. To include all attribute values is usually not a good idea – though this may be accomplished if desired by setting `base=0` – 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, `base=1`, means that the smallest (i.e., first in sorted order) attribute value is omitted. 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 `nodecov`.

`nodeicov(attrname, transform, transformname)` **(binary) (directed) (quantitative nodal attribute) (frequently-use**
Main effect of a covariate for in-edges: The `attrname` argument is a character string giving the name of a numeric (not categorical) attribute in the network's vertex attribute list. This term adds a single network statistic to the model equaling the total value of `attrname(j)` for all edges (i, j) in the network. This term may only be used with directed networks. For categorical attributes, see `nodeifactor`.

`nodeicovar` **(valued) (directed) (quantitative nodal attribute)** *Uncentered covariance of in-dyad values incident on each actor:* This term adds one statistic equal to $\sum_{i,j,k} y_{k,j} y_{k,j}$. This can be viewed as a valued analog of the `istar(2)` statistic.

`nodeifactor(attrname, base=1)` **(binary) (dyad-independent) (directed) (categorical nodal attribute) (frequently-u**
Factor attribute effect for in-edges: The `attrname` argument is a character vector giving one or more names 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 (or each combination of the attributes given). Each of these statistics gives the number of times a node with that attribute or those attributes appears as the terminal node of a directed tie. To include all attribute values is usually not a good idea – though this may be accomplished if desired by setting `base=0` – 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, `base=1`, means that the smallest (i.e., first in sorted order) attribute value is omitted. 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 `nodeicov`.

`nodeisqrtcovar` **(valued) (directed) (non-negative) (quantitative nodal attribute)** *Uncentered covariance of square roots of in-dyad values incident on each actor:* This term adds one statistic equal to $\sum_{i,j,k} \sqrt{y_{i,j}}\sqrt{y_{k,j}}$. This can be viewed as a valued analog of the `istar(2)` statistic.

`nodematch(attrname, diff=FALSE, keep=NULL)` **(binary) (dyad-independent) (frequently-used) (directed) (undirect**
Uniform homophily and differential homophily: The `attrname` argument is a character vector giving one or more names of attributes 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 multiple names are given, the statistic counts only those on which all the named attributes match.) 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. If set to non-NULL, the optional keep argument should be a vector of integers giving the values of k that should be considered for matches; other values are ignored (this works for both diff=FALSE and diff=TRUE). For instance, to add two statistics, counting the matches for just the 2nd and 4th categories, use nodematch with diff=TRUE and keep=c(2,4).

nodemix(attrname, base=NULL) **(binary) (dyad-independent) (frequently-used) (directed) (undirected) (categorical**
Nodal attribute mixing: The attrname argument is a character vector giving the names of categorical attributes in the network's vertex attribute list. By default, 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. (When multiple names are given, a statistic is added for each combination of attribute values for those names.) In other words, this term produces one statistic for every entry in the mixing matrix for the attribute(s). The ordering of the attribute values is alphabetical (for nominal categories) or numerical (for ordered categories). The optional base argument is a vector of integers corresponding to the pairings that should not be included. If base contains only negative integers, then these integers correspond to the only pairings that should be included. By default (i.e., with base=NULL or base=0), all pairings are included.

nodeocov(attrname, transform, transformname) **(binary) (directed) (dyadic-independent)(quantitative nodal attr**
Main effect of a covariate for out-edges: The attrname argument is a character string giving the name of a numeric (not categorical) attribute in the network's vertex attribute list. This term adds a single network statistic to the model equaling the total value of attrname(i) for all edges (i, j) in the network. This term may only be used with directed networks. For categorical attributes, see nodeofactor.

nodeocovar **(valued) (directed) (quantitative nodal attribute)** *Uncentered covariance of out-dyad values incident on each actor:* This term adds one statistic equal to $\sum_{i,j,k} y_{i,j}y_{i,k}$. This can be viewed as a valued analog of the ostar(2) statistic.

nodeofactor(attrname, base=1) **(binary) (dyad-independent) (directed) (categorical nodal attribute) ,** nodeofactor
Factor attribute effect for out-edges: The attrname argument is a character string giving one or more names of categorical attributes 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 (or each combination of the attributes given). Each of these statistics gives the number of times a node with that attribute or those attributes appears as the node of origin of a directed tie. To include all attribute values is usually not a good idea – though this may be accomplished if desired by setting base=0 – 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, base=1, means that the smallest (i.e., first in sorted order) attribute value is omitted. 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 nodeocov.

nodeosqrtcovar **(valued) (directed) (non-negative) (quantitative nodal attribute)** *Uncentered covariance of square roots of out-dyad values incident on each actor:* This term adds one statistic equal to $\sum_{i,j,k} \sqrt{y_{i,j}}\sqrt{y_{i,k}}$. This can be viewed as a valued analog of the ostar(2) statistic.

nodesqrtcovar(center=TRUE) **(valued) (non-negative) (directed) (undirected) (quantitative nodal attribute)**

Covariance of square roots of dyad values incident on each actor: This term adds one statistic equal to $\sum_{i,j,k}(\sqrt{y_{i,j}}\sqrt{y_{i,k}} + \sqrt{y_{k,j}}\sqrt{y_{k,j}})$ if `center=FALSE`. This can be viewed as a valued analog of the `kstar(2)` statistic. If `center=FALSE` (the default), the statistic is instead $\sum_{i,j,k}((\sqrt{y_{i,j}} - \bar{\sqrt{y}})(\sqrt{y_{i,k}} - \bar{\sqrt{y}}) + (\sqrt{y_{k,j}} - \bar{\sqrt{y}})(\sqrt{y_{k,j}} - \bar{\sqrt{y}}))$, where $\bar{\sqrt{y}}$ is the mean of the square root of dyad values.

`nsp(d)` **(binary) (directed) (undirected)** *Nonedgewise shared partners:* This is just like the `dsp` and `esp` terms, except this term adds one network statistic to the model for each element in `d` where the ith such statistic equals the number of *non-edges* (that is, dyads that do not have an edge) 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 non-edge and in the same direction).

`odegrange(from, to=+Inf, by=NULL, homophily=FALSE)` **(binary) (directed) (categorical nodal attribute)** *Out-degree range:* The `from` and `to` arguments are vectors of distinct integers (or `+Inf`, for `to` (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the ith such statistic equals the number of nodes in the network of out-degree greater than or equal to `from[i]` but strictly less than `to[i]`, i.e. with out-edge count in semiopen interval `[from,to)`. The optional argument `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and `homophily` is `FALSE` (the default), then separate degree range statistics are calculated for nodes having each separate value of the attribute.

This term can only be used with directed networks; for undirected networks (bipartite and not) see `degrange`. For degrees of specific modes of bipartite networks, see `b1degrange` and `b2degrange`. For in-degrees, see `idegrange`.

`odegree(d, by=NULL, homophily=FALSE)` **(binary) (directed) (categorical nodal attribute) (frequently-used)** *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 `by` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and `homophily` is `FALSE` (the default), then separate degree statistics are calculated for nodes having each separate value of the attribute. This term can only be used with directed networks; for undirected networks see `degree`.

`odegreepopularity` **(binary) (directed)** *Out-degree popularity:* This term adds one network statistic to the model equaling the sum over the actors of each actor's outdegree taken to the 3/2 power (or, equivalently, multiplied by its square root). This term is analogous to the term of Snijders et al. (2010), equation (12). This term can only be used with directed networks.

`opentriad` **(binary) (undirected) (triad-related)** *Open triads:* This term adds one statistic to the model equal to the number of 2-stars minus three times the number of triangles in the network. It is currently only implemented for undirected networks.

`ostar(k, attrname=NULL)` **(binary) (directed) (categorical nodal attribute)** *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 $\{O_1, \dots, O_k\}$ such that the ties $(N{\rightarrow}O_j)$ 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 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(base=1)` **(binary) (directed) (dyad-independent)** *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 by default because of linear dependence that arises if this term is used together with `edges`, but its 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 (Holland and Leinhardt, 1981). The `base` argument allows the user to determine which nodes' statistics should be omitted. The `base` argument can also be a vector of negative indices, to specify which should be added instead of deleted, and `base=0` specifies that all statistics should be included. This term can only be used with directed networks. For undirected networks, see `sociality`.

`sender(base=1)` **(binary) (directed) (dyad-independent)** *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 by default because of linear dependence that arises if this term is used together with `edges`, but its 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 (Holland and Leinhardt, 1981). The `base` argument allows the user to determine which nodes' statistics should be omitted. The `base` argument can also be a vector of negative indices, to specify which should be added instead of deleted, and `base=0` specifies that all statistics should be included. This term can only be used with directed networks. For undirected networks, see `sociality`.

`simmelian` **(binary) (directed) (triad-related)** *Simmelian triads:* This term adds one statistic to the model equal to the number of Simmelian triads, as defined by Krackhardt and Handcock (2007). This is a complete sub-graph of size three. This term can only be used with directed networks.

`simmelianties` **(binary) (triad-related) (directed)** *Ties in simmelian triads:* This term adds one statistic to the model equal to the number of ties in the network that are associated with Simmelian triads, as defined by Krackhardt and Handcock (2007). Each Simmelian has six ties in it but, because Simmelians can overlap in terms of nodes (and associated ties), the total number of ties in these Simmelians is less than six times the number of Simmelians. Hence this is a measure of the clustering of Simmelians (given the number of Simmelians). This term can only be used with directed networks.

`smalldiff(attrname, cutoff)` **(binary) (dyad-independent) (directed) (undirected) (quantitative nodal attribute)** *Number of ties between actors with similar (but not necessarily identical) attribute values:* The `attrname` argument is a character string giving the name of a quantitative attribute in the network's vertex attribute list. This term adds one statistic, having as its value the number of edges in the network for which the incident actors' attribute values differ less than `cotoff`; that is, number of edges between `i` to `j` such that `abs(attrname[i]-attrname[j])<cutoff`.

`sociality(attrname=NULL, base=1)` **(binary) (undirected) (categorical nodal attribute)** *Undirected* *degree:* This term adds one network statistic for each node equal to the number of ties of that node. The optional `attrname` argument 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 (an actor-specific version of the `nodematch` term). This term can only be used with undirected networks. For directed networks, see `sender` and `receiver`. By default, `base=1` means that the statistic for the first node will be omitted, but this argument may be changed to control which statistics are included just as for the `sender` and `receiver` terms.

`sum(pow=1)` **(valued) (directed) (undirected)** *Sum of dyad values (optionally taken to a power):* This term adds one statistic equal to the sum of dyad values taken to the power `pow`, which defaults to 1.

`threetrail(keep=1:4)` **(binary) (directed) (undirected) (triad-related),** *Three-trails:* a.k.a. `threepath`. For an undirected network, this term adds one statistic equal to the number of 3-trails, where a 3-trail is defined as a "trail" of length three that traverses three distinct edges. Note that a 3-trail need not include four distinct nodes; in particular, a triangle counts as three 3-trails. For a directed network, this term adds four statistics (or some subset of these four specified by the `keep` argument), one for each of the four distinct types of directed three-paths. If the nodes of the path are written from left to right such that the middle edge points to the right (R), then the four types are RRR, RRL, LRR, and LRL. That is, an RRR 3-trail is of the form $i \rightarrow j \rightarrow k \rightarrow l$, and RRL 3-trail is of the form $i \rightarrow j \rightarrow k \leftarrow l$, etc. Like in the undirected case, there is no requirement that the nodes be distinct in a directed 3-trail. However, the three edges must all be distinct. Thus, a mutual tie $i \leftrightarrow j$ does not count as a 3-trail of the form $i \rightarrow j \rightarrow i \leftarrow j$; however, in the subnetwork $i \leftrightarrow j \rightarrow k$, there are two directed 3-trails, one LRR ($k \leftarrow j \rightarrow i \leftarrow j$) and one RRR ($j \rightarrow i \rightarrow j \leftarrow k$).

This term used to be (inaccurately) called `threepath`. That name has been deprecated and may be removed in a future version.

`transitive` **(binary) (directed) (triad-related)** *Transitive triads:* This term adds one statistic to the model, equal to the number of triads in the network that are transitive. The transitive triads are those of type `120D`, `030T`, `120U`, or `300` in the categorization of Davis and Leinhardt (1972). For details on the 16 possible triad types, see `triad.classify` in the `sna` package. Note the distinction from the `ttriple` term. This term can only be used with directed networks.

`transitiveties(attrname=NULL)` **(binary) (directed) (triad-related) (categorical nodal attribute) ,** `transitiveties(`
Transitive ties: This term adds one statistic, equal to the number of ties $i \rightarrow j$ such that there exists a two-path from i to j. (Related to the `ttriple` term.) The binary version takes a nodal attribute `attrname`, and, if given, all three nodes involved (i, j, and the node on the two-path) must match on this attribute in order for $i \rightarrow j$ to be counted. The binary version of this term can only be used with directed networks. The valued version can be used with both directed and undirected.

`transitiveweights(twopath="min",combine="max",affect="min")` **(valued) (directed) (undirected) (non-negative)**
Transitive weights: This statistic implements the transitive weights statistic defined by Krivitsky (2012), Equation 13. The currently implemented options for `twopath` is the minimum of the constituent dyads (`"min"`) or their geometric mean (`"geomean"`); for `combine`, the maximum of the 2-path strengths (`"max"`) or their sum (`"sum"`); and for `affect`, the minimum of the focus dyad and the combined strength of the two paths (`"min"`) or their geometric mean (`"geomean"`). For each of these options, the first (and the default) is more stable but also more conservative, while the second is more sensitive but more likely to induce a multimodal distribution of networks.

`triadcensus(d)` **(binary) (triad-related) (directed) (undirected)** *Triad census:* For a directed network, this term adds one network statistic for each of an arbitrary subset of the 16 possible types of triads categorized by Davis and Leinhardt (1972) as `003, 012, 102, 021D, 021U, 021C, 111D, 111U, 0` and `300`. Note that at least one category should be dropped; otherwise a linear dependency will exist among the 16 statistics, since they must sum to the total number of three-node sets. By default, the category `003`, which is the category of completely empty three-node sets, is dropped. This is considered category zero, and the others are numbered 1 through 15 in the order given above. By specifying a numeric vector of integers from 0 to 15 as the `d` argument, the user may specify a set of terms to add other than the default value of `1:15`. Each statistic is the count of the corresponding triad type in the network. For details on the 16 types, see `?triad.classify` in the `{sna}` package, on which this code is based. For an undirected network, the triad census is over the four types defined by the number of ties (i.e., 0, 1, 2, and 3), and the default is to add `1:3`, which is to say that the 0 is dropped; however, this too may be controlled by changing the `d` argument to a numeric vector giving a subset of $\{0, 1, 2, 3\}$.

`triangle(attrname=NULL)` **(binary) (frequently-used) (triad-related) (directed) (undirected) (categorical nodal attrib**
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 $(i{\to}j)$ and $(j{\to}k)$ and either $(k{\to}i)$ or $(k{\leftarrow}i)$. The former case is called a "transitive triple" and the latter is called a "cyclic triple", so in the case of a directed network, `triangle` equals `ttriple` plus `ctriple` — thus 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=NULL)` **(binary) (undirected) (triad-related) (categorical nodal attribute)**
Triangle percentage: This term adds one statistic to the model equal to 100 times the ratio of the number of triangles in the network to the sum of the number of triangles and the number of 2-stars not in triangles (the latter is considered a potential but incomplete triangle). In case the denominator equals zero, the statistic is defined to be zero. For the definition of triangle, see `triangle`. 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 is often called the mean correlation coefficient. This term can only be used with undirected networks; for directed networks, it is difficult to define the numerator and denominator in a consistent and meaningful way.

`ttriple(attrname=NULL)` **(binary) (directed) (triad-related) (categorical nodal attribute) , a.k.a.** `ttriad` **(binary) (dir**
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 $\{(i{\to}j), (j{\to}k), (i{\to}k)\}$. 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 directed networks.

`twopath` **(binary) (directed) (undirected)** *2-Paths:* This term adds one statistic to the model, equal to the number of 2-paths in the network. For a directed network this is defined as a pair of edges $(i{\to}j), (j{\to}k)$, where i and j must be distinct. 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 but the interpretation is usually different; see `m2star`. For undirected networks a twopath 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

- Bomiriya, R. P, Bansal, S., and Hunter, D. R. (2014). Modeling Homophily in ERGMs for Bipartite Networks. Submitted.
- Butts, CT. (2008). “A Relational Event Framework for Social Action.” *Sociological Methodology,* 38(1).
- Davis, J.A. and Leinhardt, S. (1972). The Structure of Positive Interpersonal Relations in Small Groups. In J. Berger (Ed.), *Sociological Theories in Progress, Volume 2*, 218–251. Boston: Houghton Mifflin.
- Holland, P. W. and S. Leinhardt (1981). An exponential family of probability distributions for directed graphs. *Journal of the American Statistical Association*, 76: 33–50.
- 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.
- Krackhardt, D. and Handcock, M. S. (2007). Heider versus Simmel: Emergent Features in Dynamic Structures. *Lecture Notes in Computer Science*, 4503, 14–27.
- Krivitsky P. N. (2012). Exponential-Family Random Graph Models for Valued Networks. *Electronic Journal of Statistics*, 2012, 6, 1100-1128. `doi:10.1214/12-EJS696`
- Robins, G; Pattison, P; and Wang, P. (2009). “Closure, Connectivity, and Degree Distributions: Exponential Random Graph (p*) Models for Directed Social Networks.” *Social Networks,* 31:105-117.
- Snijders T. A. B., G. G. van de Bunt, and C. E. G. Steglich. Introduction to Stochastic Actor-Based Models for Network Dynamics. *Social Networks*, 2010, 32(1), 44-60. `doi:10.1016/j.socnet.2009.02.004`
- Morris M, Handcock MS, and Hunter DR. Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 2008, 24(4), 1-24. `http://www.jstatsoft.org/v24/i04`
- 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` package, `search.ergmTerms`, `ergm`, `network`, `%v%`, `%n%`

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.allstats *Calculate all possible vectors of statistics on a network for an ERGM*

Description

`ergm.allstats` produces a matrix of network statistics for an arbitrary `statnet` exponential-family random graph model. One possible use for this function is to calculate the exact loglikelihood function for a small network via the `ergm.exact` function.

Usage

```
ergm.allstats (formula, zeroobs = TRUE, force = FALSE,
               maxNumChangeStatVectors = 2^16, ...)
```

Arguments

`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.
`zeroobs`	Logical: Should the vectors be centered so that the network passed in the `formula` has the zero vector as its statistics?
`force`	Logical: Should the algorithm be run even if it is determined that the problem may be very large, thus bypassing the warning message that normally terminates the function in such cases?
`maxNumChangeStatVectors`	Maximum possible number of distinct values of the vector of statistics. It's good to use a power of 2 for this.
`...`	further arguments; not currently used.

Details

The mechanism for doing this is a recursive algorithm, where the number of levels of recursion is equal to the number of possible dyads that can be changed from 0 to 1 and back again. The algorithm starts with the network passed in `formula`, then recursively toggles each edge twice so that every possible network is visited.

`ergm.allstats` should only be used for small networks, since the number of possible networks grows extremely fast with the number of nodes. An error results if it is used on a directed network of more than 6 nodes or an undirected network of more than 8 nodes; use `force=TRUE` to override this error.

Value

Returns a list object with these two elements:

`weights`	integer of counts, one for each row of `statmat` telling how many networks share the corresponding vector of statistics.
`statmat`	matrix in which each row is a unique vector of statistics.

See Also

`ergm.exact`

Examples

```
# Count by brute force all the edge statistics possible for a 7-node
# undirected network
mynw <- network(matrix(0,7,7),dir=FALSE)
unix.time(a <- ergm.allstats(mynw~edges))

# Summarize results
rbind(t(a$statmat),a$weights)

# Each value of a$weights is equal to 21-choose-k,
# where k is the corresponding statistic (and 21 is
# the number of dyads in an 7-node undirected network).
# Here's a check of that fact:
as.vector(a$weights - choose(21, t(a$statmat)))

# Simple ergm.exact outpuf for this network.
# We know that the loglikelihood for my empty 7-node network
# should simply be -21*log(1+exp(eta)), so we may check that
# the following two values agree:
-21*log(1+exp(.1234))
ergm.exact(.1234, mynw~edges, statmat=a$statmat, weights=a$weights)
```

`ergm.bounddeg` *initializes the parameters to bound degree during sampling*

Description

Not normally called directly by user, `ergm.bounddeg` initializes the list of parameters used to bound the degree during the Metropolis Hastings sampling process, and issues warnings if the original network doesn't meet the constraints specified by 'bounddeg'.

Usage

```
ergm.bounddeg(bounddeg, nw)
```

Arguments

`bounddeg`	a list of parameters which may contain the following for a network of size n nodes: • attribs: an nxp matrix, where entry ij is TRUE if node i has attribute j, and FALSE otherwise; default=an nx1 matrix of 1's • maxout : an nxp matrix, where entry ij is the maximum number of out degrees for node i to nodes with attribute j; default=an nxp matrix of the value (n-1) • maxin : defined similarly to maxout, but ignored for undirected networks; default=an nxp matrix of the value (n-1) • minout : defined similarly to maxout; default=an nxp matrix of 0's • minin : defined similarly to maxout, but ignored for undirected networks; default=an nxp matrix of 0's
`nw`	the orginal `network` specified to `ergm` in 'formula'

Details

In some modeling situations, the degree of certain nodes are constrained to lie in a certain range (rather than their theoretically possible range of 0 to n-1). Such sample space constraints may be incorporated into the ergm modeling process, and if so then the MCMC routine is prevented from visiting network states that violate any of these bounds.

In case there are categories of nodes and degree bounds for each set of categories, such constraints may be incorporated as well. For instance, if the nodes are girls and boys, and there is a maximum of 5 out-ties to boys and a maximum of 5 out-ties to girls for each node, we would define p to be 2, and the nxp matrix attribs would have TRUE in the first column (say) for exactly those nodes that are boys and TRUE in the second column for only the girls. The maxout matrix would consist of all 5s in this case, and the other arguments would be left as their default values.

Since the observed network is generally the beginning of the Markov chain, it must satisfy all of the degree constraints itself; thus, this function returns an error message if any bound is violated by the observed network.

Value

a list of parameters used to bound degree during sampling

- condAllDegExact: always FALSE
- attribs : as defined above
- maxout : as defined above
- maxin : as defined above
- minout : as defined above
- minin : as defined above

See Also

`ergm_MH_proposals`

ergm.bridge.dindstart.llk

Bridge sampling to estiamte log-likelihood of an ERGM, using a dyad-independent ERGM as a staring point.

Description

This function is a wrapper around `ergm.bridge.llr` that uses a dyad-independent ERGM as a starting point for bridge sampling to estimate the log-likelihood for a given dyad-dependent model and parameter configuration. The dyad-independent model may be specified or can be chosen adaptively.

Usage

```
ergm.bridge.dindstart.llk(object,
                          response=NULL,
                          constraints=~.,
                          coef,
                          dind=NULL,
                          coef.dind=NULL,
                          basis=NULL,
                          ...,
                          llkonly=TRUE,
                          control=control.ergm.bridge())
```

Arguments

`object`	A model formula. See `ergm` for details.
`response`	The name of the edge attribute that is the response. Note that it's included solely for consistency, since `ergm.bridge.dindstart.llk` can only handle binary ERGMs.
`constraints`	A model constraints formula. See `ergm` for details. Note that only constraints that do not induce dyadic dependence can be handled by `ergm.bridge.dindstart.llk`.
`coef`	A vector of coefficients for the configuration of interest.
`dind`	A one-sided formula with the dyad-independent model to use as a starting point. Defaults to the dyad-independent terms found in the formula `object` with an overal density term (`edges`) added if not redundant.
`coef.dind`	Parameter configuration for the dyad-independent starting point. Defaults to the MLE of `dind`.
`basis`	An optional `network` object to start the Markov chain. If omitted, the default is the left-hand-side of the `object`.
`...`	Further arguments to `ergm.bridge.llr` and `simulate.formula.ergm`.
`llkonly`	Whether only the estiamted log-likelihood should be returned. (Defaults to `TRUE`.)
`control`	Control parameters. See `control.ergm.bridge`.

Value

If `llkonly=TRUE`, returns the scalar log-likelihood. Otherwise, returns a copy of the list returned by `ergm.bridge.llr` in addition to the following components:

`llk.dind`	The log-likelihood of the dyad-independence model.
`llk`	The estimated log-likelihood.

References

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

See Also

`ergm.bridge.llr`, `simulate.formula.ergm`

`ergm.bridge.llr` *A simple implementation of bridge sampling to evaluate log-likelihood-ratio between two ERGM configurations*

Description

`ergm.bridge.llr` uses bridge sampling with geometric spacing to estimate the difference between the log-likelihoods of two parameter vectors for an ERGM via repeated calls to `simulate.formula.ergm`.

`ergm.bridge.0.llk` is a convenience wrapper around `ergm.bridge.llr`: returns the log-likelihood of configuration 'theta' *relative to the reference measure*. That is, the configuration with theta=0 is defined as having log-likelihood of 0

See also `ergm.bridge.dindstart.llk` to use dyad-independent ERGM as a staring point.

Usage

```
ergm.bridge.llr(object,
                response=NULL,
                constraints=~.,
                from,
                to,
                basis=NULL,
                verbose=FALSE,
                ...,
                llronly=FALSE,
                control=control.ergm.bridge())

ergm.bridge.0.llk(object,
                  response=response,
                  coef,
                  ...,
                  llkonly=TRUE,
                  control=control.ergm.bridge())
```

Arguments

`object`	A model formula. See `ergm` for details.
`response`	Not for release.
`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.
`from, to`	The initial and final parameter vectors.
`basis`	An optional `network` object to start the Markov chain. If omitted, the default is the left-hand-side of the `object`.
`verbose`	Logical: If TRUE, print detailed information.
`...`	Further arguments to `ergm.bridge.llr` and `simulate.formula.ergm`.
`llronly`	Logical: If TRUE, only the estiamted log-ratio will be returned.
`control`	Control arguments. See `control.ergm.bridge` for details.
`coef`	A vector of coefficients for the configuration of interest.
`llkonly`	Whether only the estiamted log-likelihood should be returned. (Defaults to TRUE.)

Value

If `llronly=TRUE`, returns the scalar log-likelihood-ratio. Otherwise, returns a list with the following components:

`llr`	The estimated log-ratio.
`llrs`	The estimated log-ratios for each of the `nsteps` bridges.
`path`	A numeric matrix with nsteps rows, with each row being the respective bridge's parameter configuration.
`stats`	A numeric matrix with nsteps rows, with each row being the respective bridge's vector of simulated statistics.
`Dtheta.Du`	The gradient vector of the parameter values with respect to position of the bridge.

References

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

See Also

`simulate.formula.ergm`, `ergm.bridge.dindstart.llk`

`ergm.ConstraintImplications` *Set up the implied constraints from the current constraint*

Description

This is a low-level function not intended to be called directly by end users. For information on constraints, see the `ergm-constraints` page. This function set up the implied constraints from the current constraint. (It is defined in the scope of local environment)

Usage

```
ergm.ConstraintImplications(implier, implies)
```

Arguments

`implier`	The current constraint specified in the model. For the list of constraints, see `ergm-constraints`
`implies`	Implied constraints from the current constraint (based on the user's knowledge).

`ergm.Cprepare` *Internal Function to Prepare Data for ergm's C Interface*

Description

These are internal functions not intended to be called by end users. The `ergm.Cprepare` function builds an object called Clist that contains all the necessary ingredients to be passed to the C functions, other functions create edgelists and handle missing edge data.

Usage

```
ergm.Cprepare(nw, m, response = NULL)

ergm.Cprepare.el(x, attrname=NULL, prototype=NULL)

ergm.Cprepare.miss(nw)

ergm.design(nw, model, verbose = FALSE)
```

Arguments

`nw,x`	a network object
`m,model`	a model object, as returned by `ergm.getmodel`
`response,attrname`	character name of an edge attribute
`prototype`	A network whose relevant attributes (size, directedness, bipartitedness, and presence of loops) are imposed on the output edgelist if `x` is already an edgelist. (For example, if the prototype is undirected, `ergm.Cprepare.el` will ensure that $t < h$.)
`verbose`	logical, whether the design matrix should be printed; default=FALSE

Details

These low-level functions are used by other ergm-related packages, but should never need to be called directly by the user.

- `ergm.Cprepare` builds an object called Clist that contains all the necessary ingredients to be passed to the C functions
- `ergm.Cprepare.el` constructs and serializes a very simple static edgelist, with the vertex having the lesser index the tail and sorted by tails, then by heads.
- `ergm.Cprepare.miss` constructs an edgelist as `ergm.Cprepare.el`, but only includes 'missing' edges (marked as NA)
- `ergm.design` functions as `ergm.Cprepare` would, but acts on the network of missing edges

Value

`ergm.Cprepare` returns Clist: a list of parameters used by several of the fitting routines containing

- n : the size of the network
- dir : whether the network is directed (T or F)
- bipartite : whether the network is bipartite (T or F)
- ndyads : the number of dyads in the network
- nedges : the number of edges in this network
- tails : the vector of tail nodes; tail nodes are the 1st column of the implicit edgelist, so either the lower-numbered nodes in an undirected graph, or the out nodes of a directed graph, or the b1 nodes of a bi- partite graph
- heads : the vector of head nodes; head nodes are the 2nd column of the implicit edgelist, so either the higher-numbered nodes in an undirected graph, or the in nodes of a directed graph, or the b2 nodes of a bi- partite graph
- nterms : the number of model terms
- nstats : the total number of change statistics for all model terms
- inputs : the concatenated vector of 'input's from each model term as returned by <InitErgmTerm.X> or <InitErgm.X>
- fnamestring : the concatenated string of model term names

- snamestring : the concatenated string of package names that contain the C function 'd_fname'; default="ergm" for each fname in fnamestring

`ergm.design` returns: Clist.miss

- if 'nw' has missing edges, see the return list, 'Clist', from the `ergm.Cprepare` function header
- if 'nw' hasn't any missing edges, the list will only contain NULL values for the 'tails' and 'heads' components, a 0 for 'nedges' and 'dir' appropriately set

`ergm.Cprepare.miss` returns a vector of length 1+Nmissing*2. The first element is the number of missing edges, and the remainder a column-major edgelist

---

`ergm.degeneracy` *Checks an ergm Object for Degeneracy*

Description

The `ergm.degeneracy` function checks a given ergm object for degeneracy by computing and returning the instability value of the model and the value of the log-likelihood function at the maximized theta values

Usage

```
ergm.degeneracy(object, control = object$control,
                fast = TRUE, test.only = FALSE,
                verbose = FALSE)
```

Arguments

`object`	an `ergm` object
`control`	the list of control parameters as returned by `control.ergm`; default=control.ergm()
`fast`	whether the degeneracy check should be "fast", i.e to sample changeobs(?) when there are > 100, rather than use all changeobs; default=TRUE
`test.only`	whether to silence printing of the model instability calculation (T or F); this parameter is ignored if the instability > 1; default=FALSE
`verbose`	whether to print a notification when 'object' is deemed degenerate (T or F); default=FALSE

Value

returns the original ergm object with 2 additional components:

- degeneracy.value: the instability of the model
- degeneracy.type : a 2-element vector containing
  - loglikelihood: the value of the log-likelihood function corresponding to 'theta'; if degenerate, this is a vector of Inf
  - theta : the vector of theta values found through maximixing the log- likelihood; if degenerate, this is 'guess'

ergm.eta — *Operations with 'eta' vector of canonical parameter values from ergm model*

Description

The ergm.eta function calculates and returns eta, mapped from theta using the etamap object created by ergm.etamap.

The ergm.etagrad function caculates and returns the gradient of eta mapped from theta using the etamap object created by ergm.etamap. If the gradient is only intended to be a multiplier for some vector, the more efficient ergm.etagradmult is recommended.

The ergm.etagradmult function calculates and returns the product of the gradient of eta with a vector v

The ergm.etamap function takes a model object and creates a mapping from the model parameters, theta, to the canonical (linear) eta parameters; the mapping is carried out by ergm.eta

Usage

```
ergm.eta(theta, etamap)

ergm.etagrad(theta, etamap)

ergm.etagradmult(theta, v, etamap)

ergm.etamap(model)
```

Arguments

theta	the curved model parameters
etamap	the list of values that constitutes the theta-> eta mapping and is returned by ergm.etamap
v	a vector of the same length as the vector of mapped eta parameters
model	model object, as returned by ergm.getmodel

Details

This function is only important in the case of curved exponential family models, i.e., those in which the parameter of interest (theta) is not a linear function of the sufficient statistics (eta) in the exponential-family model. In non-curved models, we may assume without loss of generality that eta(theta)=theta.

A succinct description of how eta(theta) is incorporated into an ERGM is given by equation (5) of Hunter (2007). See Hunter and Handcock (2006) and Hunter (2007) for further details about how eta and its derivatives are used in the estimation process.

Value

- for `ergm.eta`: `eta` the canonical eta parameters as mapped from theta
- for `ergm.etagrad`: `etagrad` a matrix of the gradient of eta
- for `ergm.etagradmult`: `ans` the vector that is the product of the gradient of eta and `v`; infinite values are replaced by (+-)10000
- for `ergm.etamap` the theta -> eta mapping given by a list of the following:
  - canonical : a numeric vector whose ith entry specifies whether the ith component of theta is canonical (via non- negative integers) or curved (via zeroes)
  - offsetmap : a logical vector whose ith entry tells whether the ith coefficient of the canonical parameterization was "offset", i.e fixed
  - offset : a logical vector whose ith entry tells whether the ith model term was offset/fixed
  - offsettheta: a logical vector whose ith entry tells whether the ith curved theta coeffient was offset/fixed;
  - curved : a list with one component per curved EF term in the model containing
    - from : the indices of the curved theta parameter that are to be mapped from
    - to : the indices of the canonical eta parameters to be mapped to
    - map : the map provided by <InitErgmTerm>
    - gradient: the gradient function provided by InitErgmTerm
    - cov : the eta covariance ??, possibly always NULL (no <Init> function creates such an item)
  - etalength : the length of the eta vector

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.

See Also

`ergm-terms`

`ergm.exact` *Calculate the exact loglikelihood for an ERGM*

Description

`ergm.exact` calculates the exact loglikelihood, evaluated at `eta`, for the `statnet` exponential-family random graph model represented by `formula`.

Usage

```
ergm.exact (eta, formula, statmat=NULL, weights=NULL, ...)
```

Arguments

`eta`	vector of canonical parameter values at which the loglikelihood should be evaluated.
`formula`	an R `link{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.
`statmat`	if NULL, call `ergm.allstats` to generate all possible graph statistics for the networks in this model.
`weights`	In case `statmat` is not `NULL`, this should be the vector of counts corresponding to the rows of `statmat`. If `statmat` is `NULL`, this is generated by the call to `ergm.allstats`.
`...`	further arguments; not currently used.

Details

`ergm.exact` should only be used for small networks, since the number of possible networks grows extremely fast with the number of nodes. An error results if it is used on a directed network of more than 6 nodes or an undirected network of more than 8 nodes; use `force=TRUE` to override this error.

In case this function is to be called repeatedly, for instance by an optimization routine, it is preferable to call `ergm.allstats` first, then pass `statmat` and `weights` explicitly to avoid repeatedly calculating these objects.

Value

Returns the value of the exact loglikelihood, evaluated at `eta`, for the `statnet` exponential-family random graph model represented by `formula`.

See Also

`ergm.allstats`

Examples

```
# Count by brute force all the edge statistics possible for a 7-node
# undirected network
mynw <- network(matrix(0,7,7),dir=FALSE)
unix.time(a <- ergm.allstats(mynw~edges))

# Summarize results
rbind(t(a$statmat),a$weights)

# Each value of a$weights is equal to 21-choose-k,
# where k is the corresponding statistic (and 21 is
# the number of dyads in an 7-node undirected network).
# Here's a check of that fact:
as.vector(a$weights - choose(21, t(a$statmat)))
```

```
# Simple ergm.exact outpuf for this network.
# We know that the loglikelihood for my empty 7-node network
# should simply be -21*log(1+exp(eta)), so we may check that
# the following two values agree:
-21*log(1+exp(.1234))
ergm.exact(.1234, mynw~edges, statmat=a$statmat, weights=a$weights)
```

`ergm.formula.utils` *Internal Functions for Querying, Validating and Extracting from ERGM Formulas*

Description

These are all functions that are generally not called directly by users, but may be employed by other depending packages.

Usage

```
ergm.getmodel(formula, nw, response = NULL, silent = FALSE, role = "static", ...)

ergm.getnetwork(form, loopswarning = TRUE)

ergm.getterms(formula)

offset.info.formula(object, response = NULL)

remove.offset.formula(object, response = NULL)

# Deprecated. Use nonsimp.update.formula() from statnet.common package.
ergm.update.formula(object, new, ..., from.new = FALSE)
```

Arguments

`formula`	a formula of the form `network ~ model.term(s)`
`nw`	the network of interest
`response`	charcter, name of edge attribute containing edge weights
`silent`	logical, whether to print the warning messages from the initialization of each model term; default=FALSE
`role`	A hint about how the model will be used. Used primarily for dynamic network models.
`...`	additional parameters for model formulation
`form`	same as formula, a formula of the form `'network ~ model.term(s)'`
`loopswarning`	whether warnings about loops should be printed (T or F);default=TRUE

`object`	formula object to be updated
`new`	new formula to be used in updating
`from.new`	logical or character vector of variable names. controls how environment of formula gets updated.

Details

- The `ergm.getmodel` function parses the given formula, and initiliazes each ergm term via the `InitErgmTerm` functions to create a `model.ergm` object for the given network
- The `ergm.getnetwork` function ensures that the network in a given formula is valid; if so, the network is returned; if not, execution is halted with warnings
- The `ergm.getterms` function returns the terms of a given formula and ensures that the formula is indeed a formula with the necessary ~ operator
- `ergm.update.formula` (DEPRECATED: use `nonsimp.update.formula` instead) is a reimplementation of `update.formula` that does not simplify. Note that the resulting formula's environment is set as follows. If `from.new==FALSE`, it is set to that of object. Otherwise, a new sub-environment of object, containing, in addition, variables in new listed in from.new (if a character vector) or all of new (if TRUE).
- `offset.info.formula` returns the offset vectors associated with a formula.
- `remove.offset.formula` deletes all `offset` terms in an ERGM formula.

Value

`ergm.getmodel` returns a 'model.ergm' object as a list containing:

- formula : the formula inputted to `ergm.getmodel`
- coef.names : a vector of coefficient names
- offset : a logical vector of whether each term was "offset", i.e. fixed
- terms : a list of terms and 'term components' initialized by the appropriate `InitErgmTerm.X` function.
- network.stats0: NULL always??
- etamap : the theta -> eta mapping as a list returned from <ergm.etamap>
- class : the character string "model.ergm"

`ergm.getnetwork` returns:

- the network from the formula IF (i) the formula was correctly structured and (ii) the network is found within the formula's enviornment

`ergm.getterms` returns:

- the terms object associated with the formula and returned by the native R function `terms`>. see `terms.object` for details about the components

`terms.list.formula` returns a list of formula terms, each of witch having an additional attribute `"sign"`.

`ergm.update.formula`, `remove.offset.formula` and

ergm.geodistdist — *calculate geodesic distance distribution for a network or edgelist*

Description

`ergm.geodistdist` calculates geodesic distance distribution for a given `network` and returns it as a vector.

`ergm.geodistn` calculates geodesic deistance distribution based on an input edgelist, and has very little error checking so should not normally be called by users. The C code requires the edgelist to be directed and sorted correctly.

Usage

```
ergm.geodistdist(nw, directed = is.directed(nw))

ergm.geodistn(edgelist, n = max(edgelist), directed = FALSE)
```

Arguments

`nw`	`network` object over which distances should be calculated
`directed`	logical, should the network be treated as directed
`edgelist`	an edgelist representation of a network as an mx2 matrix
`n`	integer, size of the network

Details

`ergm.geodistdist` is a network wrapper for `ergm.geodistn`, which calculates and returns the geodesic distance distribution for a given network via full_geodesic_distribution.C

Value

a vector `ans` with length equal to the size of the network where

- ans[i], i=1, ..., n-1 is the number of pairs of geodesic length i
- ans[n] is the number of pairs of geodesic length infinity.

See Also

See also the sna package `geodist` function

Examples

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

ergm.getglobalstats *internal function to return global statistics for a given network*

Description

The `ergm.getglobalstats` function is a low-level function not normally called by the user. It calculates and returns the global statistics for a given network and model.

Usage

```
ergm.getglobalstats(nw, m, response = NULL)
```

Arguments

`nw`	a `network` object
`m`	the model in use with network `nw`, as returned by `ergm.getmodel`
`response`	character name of an edge attribute to be used (for weighted ergm models)

Details

Calculates and returns the global statistics for a given network via `ergm.Cprepare` and `network_stats_wrapper.C` or `wt_network_stats_wrapper.C` if the model is weighted. It is called by `summary.statistics.network` which is generally the better way to access the functionality.

Value

returns a vector of the global statistics

See Also

`summary.statistics.network`

ergm.getMCMCsample *Internal Function to Sample Networks Using C Wrapper*

Description

This is an internal function, not normally called directly by the user. The `ergm.getMCMCsample` function samples networks using an MCMC algorithm via `MCMC_wrapper.C` and is caple of running in multiple threads using `ergm.mcmcslave`.

The `ergm.mcmcslave` function is that which the slave nodes in a parallel process will call to perform a validation on the mcmc equal to their slave number. It also returns an MCMC sample.

Usage

```
ergm.getMCMCsample(nw, model, MHproposal, eta0, control, verbose, response = NULL, ...)

ergm.mcmcslave(Clist, MHproposal, eta0, control, verbose, ..., prev.run = NULL,
    burnin = NULL, samplesize = NULL, interval = NULL, maxedges = NULL)
```

Arguments

`nw`	a network object
`model`	a model for the given 'nw' as returned by <ergm.getmodel>
`MHproposal`	a list of the parameters needed for Metropolis-Hastings proposals and the result of calling <MHproposal>
`eta0`	the initial eta coefficients
`control`	list of MCMC tuning parameters; (see control.ergm)
`verbose`	whether the C functions should be verbose; default=FALSE
`response`	characher, name of an edge attribute
`...`	additional arugments?
`Clist`	the list of parameters returned by `ergm.Cprepare`
`prev.run`	output stats from previous run ???
`burnin`	number of proposals before any MCMC sampling is done. see control.ergm MCMC.burnin
`samplesize`	number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm. see control.ergm MCMC.samplesize
`interval`	number of proposals between sampled statistics. see control.ergm MCMC.interval
`maxedges`	maximum number of edges expected in network. see control.ergm MCMC.init.maxedges

Details

Note that the returned stats will be relative to the original network, i.e., the calling function must shift the statistics if required. The calling function must also attach column names to the statistics matrix if required.

Value

for `ergm.getMCMCsample`,the sample as a list containing:

- statsmatrix: the stats matrix for the sampled networks, RELATIVE TO THE ORIGINAL NETWORK!
- newnetwork : the edgelist of the final sampled network
- nedges : the number of edges in the 'newnetwork'

for `ergm.mcmcslave` the MCMC sample as a list of the following:

- s : the statsmatrix

- newnwtails: the vector of tails for the new network- is this the final network sampled? - is this the original nw if 'maxedges' is 0
- newnwheads: the vector of heads for the new network - same q's

---

`ergm.init.methods` *Set up the initial fitting methods for reference measure and query available methods for that reference measure*

Description

This is a low-level function not intended to be called directly by end users. This function sets up the available initial fitting methods for each reference measure and queries them.

Usage

```
ergm.init.methods(reference, new.methods)
```

Arguments

`reference`	The reference measure used in the model.
`new.methods`	If passed, prepends the new initial fitting methods to the list for that reference measure.

Value

A character vector listing initial methods for the reference measure specified. (If `new.methods` is passed, does so invisibly.)

`ergm.MHP.table` *Table mapping reference,constraints, etc. to Metropolis Hastings Proposals (MHP)*

Description

This is a low-level function not intended to be called directly by end users. For information on Metropolis-Hastings proposal methods, ergm_MH_proposals. This function sets up the table mapping constraints, references, etc. to MHproposals. (It is defined in the scope of local environment)

Usage

```
ergm.MHP.table(Class, Reference, Constraints, Priority, Weights, MHP)
```

Arguments

`Class`	default to "c"
`Reference`	The reference measure used in the model. For the list of reference measures, see `ergm-references`
`Constraints`	The constraints used in the model. For the list of constraints, see `ergm-constraints`
`Priority`	On existence of multiple qualifying MHPs, specifies the priority (-1,0,1) of MHPs to be used.
`Weights`	The sampling weights on selecting toggles (random, TNT, etc).
`MHP`	The matching MHP from the previous arguments.

`ergm.mple` *Find a maximizer to the psuedolikelihood function*

Description

The `ergm.mple` function finds a maximizer to the psuedolikelihood function (MPLE). It is the default method for finding the ERGM starting coefficient values. It is normally called internally the ergm process and not directly by the user. Generally `ergmMPLE` would be called by users instead.

`ergm.pl` is an even more internal workhorse function that prepares many of the components needed by `ergm.mple` for the regression rountines that are used to find the MPLE estimated ergm. It should not be called directly by the user.

Usage

```
ergm.mple(Clist, Clist.miss, m, init = NULL, MPLEtype = "glm", family = "binomial",
        maxMPLEsamplesize = 1e+06, save.glm = TRUE, theta1 = NULL, conddeg = NULL,
          control = NULL, MHproposal = NULL, verbose = FALSE, ...)

ergm.pl(Clist, Clist.miss, m, theta.offset=NULL,
                      maxMPLEsamplesize=1e+6,
                      conddeg=NULL, control, MHproposal,
                      verbose=FALSE)
```

Arguments

`Clist`	a list of parameters used for fitting and returned by `ergm.Cprepare`
`Clist.miss`	the corresponding 'Clist' for the network of missing edges returned by `ergm.design`
`m`	the model, as returned by `ergm.getmodel`
`init`	a vector a vector of initial theta coefficients
`MPLEtype`	the method for MPL estimation as "penalized", "glm" or "logitreg"; default="glm"
`family`	the family to use in the R native routine `glm`; only applicable if "glm" is the 'MPLEtype'; default="binomial"

`maxMPLEsamplesize`	the sample size to use for endogenous sampling in the psuedo-likelihood computation; default=1e6
`save.glm`	whether the mple fit and the null mple fit should be returned (T or F); if false, NULL is returned for both; default==TRUE
`theta1`	the independence theta; if specified and non-NULL, this is ignored except to return its value in the returned ergm; default=NULL, in which case 'theta1' is computed
`conddeg`	an indicator of whether the MPLE should be conditional on degree; non-NULL values indicate yes, NULL no; default=NULL.
`control`	a list of MCMC related parameters; recognized components include: samplesize : the number of networks to sample Clist.miss : see 'Clist.miss' above; some of the code uses this Clist.miss,
`MHproposal`	an MHproposal object, as returned by `MHproposal`
`verbose`	whether this and the C routines should be verbose (T or F); default=FALSE
`theta.offset`	a logical vector specifying which of the model coefficients are offset, i.e. fixed
`...`	additional parameters passed from within; all will be ignored

Details

According to Hunter et al. (2008): "The maximizer of the pseudolikelihood may thus easily be found (at least in principle) by using logistic regression as a computational device." In order for this to work, the predictors of the logistic regression model must be calculated. These are the change statistics as described in Section 3.2 of Hunter et al. (2008), put into matrix form so that each pair of nodes is one row whose values are the vector of change statistics for that node pair. The ergm.pl function computes these change statistics and the ergm.mple function implements the logistic regression using R's glm function. Generally, neither ergm.mple nor ergm.pl should be called by users if the logistic regression output is desired; instead, use the `ergmMPLE` function.

In the case where the ERGM is a dyadic independence model, the MPLE is the same as the MLE. However, in general this is not the case and, as van Duijn et al. (2009) warn, the statistical properties of MPLEs in general are somewhat mysterious.

MPLE values are used even in the case of dyadic dependence models as starting points for the MCMC algorithm.

Value

`ergm.mple` returns an ergm object as a list containing several items; for details see the return list in the `ergm`

`ergm.pl` returns a list containing:

- xmat : the compressed and possibly sampled matrix of change statistics
- zy : the corresponding vector of responses, i.e. tie values
- foffset : ??
- wend : the vector of weights for 'xmat' and 'zy'
- numobs : the number of dyads

- xmat.full: the 'xmat' before sampling; if no sampling is needed, this is NULL
- zy.full : the 'zy' before sampling; if no sampling is needed, this is NULL
- foffset.full : ??
- theta.offset : a numeric vector whose ith entry tells whether the the ith curved coefficient?? was offset/fixed; -Inf implies the coefficient was fixed, 0 otherwise; if the model hasn't any curved terms, the first entry of this vector is one of log(Clist$nedges/(Clist$ndyads-Clist$nedges)) log(1/(Clist$ndyads-1)) depending on 'Clist$nedges'
- maxMPLEsamplesize: the 'maxMPLEsamplesize' inputted to `ergm.pl`

References

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris and Martina (2008). "ergm: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks." _Journal of Statistical Software_, *24*(3), pp. 1-29. http://www.jstatsoft.org/article/view/v024i03

van Duijn MAJ, Gile K, Handcock MS (2009). "Comparison of Maximum Pseudo Likelihood and Maximum Likelihood Estimation of Exponential Family Random Graph Models." _Social Networks_, *31*, pp. 52-62.

See Also

`ergmMPLE`, `ergm`,`control.ergm`

`ergmMPLE` *ERGM Predictors and response for logistic regression calculation of MPLE*

Description

Return the predictor matrix, response vector, and vector of weights that can be used to calculate the MPLE for an ERGM.

Usage

```
ergmMPLE(formula, fitmodel=FALSE, output=c("matrix","array", "fit"),
         as.initialfit = TRUE, control=control.ergm(),
         verbose=FALSE, ...)
```

Arguments

`formula`	An ERGM formula. See `ergm`.
`fitmodel`	Deprecated. Use `output="fit"` instead.
`output`	Character, partially matched. See Value.
`as.initialfit`	Logical. Specifies whether terms are initialized with argument `initialfit==TRUE` (the default). Generally, if `TRUE`, all curved ERGM terms will be treated as having their curved parameters fixed. See Example.

`control`	A list of control parameters for tuning the fitting of an ERGM. Most of these parameters are irrelevant in this context. See `control.ergm` for details about all of the control parameters.
`verbose`	Logical; if `TRUE`, the program will print out some additional information.
`...`	Additional arguments, to be passed to lower-level functions.

Details

The MPLE for an ERGM is calculated by first finding the matrix of change statistics. Each row of this matrix is associated with a particular pair (ordered or unordered, depending on whether the network is directed or undirected) of nodes, and the row equals the change in the vector of network statistics (as defined in `formula`) when that pair is toggled from a 0 (no edge) to a 1 (edge), holding all the rest of the network fixed. The MPLE results if we perform a logistic regression in which the predictor matrix is the matrix of change statistics and the response vector is the observed network (i.e., each entry is either 0 or 1, depending on whether the corresponding edge exists or not).

Using `output="matrix"`, note that the result of the fit may be obtained from the `glm` function, as shown in the examples below.

When `output="array"`, the `MPLE.max.dyad.types` control parameter must be greater than `network.dyadcount(.)` of the response network, or not all elements of the array that ought to be filled in will be.

Value

If `output=="matrix"` (the default), then only the response, predictor, and weights are returned; thus, the MPLE may be found by hand or the vector of change statistics may be used in some other way. To save space, the algorithm will automatically search for any duplicated rows in the predictor matrix (and corresponding response values). `ergmMPLE` function will return a list with three elements, `response`, `predictor`, and `weights`, respectively the response vector, the predictor matrix, and a vector of weights, which are really counts that tell how many times each corresponding response, predictor pair is repeated.

If `output=="array"`, a list with similarly named three elements is returned, but `response` is formatted into a sociomatrix; `predictor` is a 3-dimensional array of with cell `predictor[t,h,k]` containing the change score of term `k` for dyad (`t`,`h`); and `weights` is also formatted into a sociomatrix, with an element being 1 if it is to be added into the pseudolikelihood and 0 if it is not.

In particular, for a unipartite network, cells corresponding to self-loops, i.e., `predictor[i,i,k]` will be `NA` and `weights[i,i]` will be 0; and for a unipartite undirected network, lower triangle of each `predictor[,,k]` matrix will be set to `NA`, with the lower triangle of `weights` being set to 0.

If `output=="fit"`, then `ergmMPLE` simply calls the `ergm` function with the `estimate="MPLE"` option set, returning an object of class `ergm` that gives the fitted pseudolikelihood model.

See Also

`ergm`, `glm`

Examples

```
data(faux.mesa.high)
formula <- faux.mesa.high ~ edges + nodematch("Sex") + nodefactor("Grade")
```

```
mplesetup <- ergmMPLE(formula)

# Obtain MPLE coefficients "by hand":
glm(mplesetup$response ~ . - 1, data = data.frame(mplesetup$predictor),
    weights = mplesetup$weights, family="binomial")$coefficients

# Check that the coefficients agree with the output of the ergm function:
ergmMPLE(formula, output="fit")$coef

# We can also format the predictor matrix into an array:
mplearray <- ergmMPLE(formula, output="array")

# The resulting matrices are big, so only print the first 5 actors:
mplearray$response[1:5,1:5]
mplearray$predictor[1:5,1:5,]
mplearray$weights[1:5,1:5]

formula2 <- faux.mesa.high ~ gwesp(0.5,fix=FALSE)

# The term is treated as fixed: only the gwesp term is returned:
colnames(ergmMPLE(formula2, as.initialfit=TRUE)$predictor)

# The term is treated as curved: individual esp# terms are returned:
colnames(ergmMPLE(formula2, as.initialfit=FALSE)$predictor)
```

ergm_deprecated *Functions that will no longer be supported in future releases of the package*

Description

Functions that have been superceed, were never documented, or will be removed from the package for other reasons

- delete.isolates
- central.network
- largest.components
- degreedistfactor
- sociality.default
- sociality.ergm
- sociality.formula
- sociality.network
- rspartnerdist
- espartnerdist
- dspartnerdist

- twopathdist
- ostar2deg
- drawpie
- invert.network
- is.invertible
- mvmodel
- mvmodel.default
- mvmodel.ergm
- mvmodel.formula
- ergm.mahalanobis
- robust.inverse

`ergm_MH_proposals` *Metropolis-Hastings Proposal Methods for ERGM MCMC*

Description

`ergm` uses a Metropolis-Hastings (MH) algorithm to control the behavior of the Markov Chain Monte Carlo (MCMC) for sampling networks. The MCMC chain is intended to step around the sample space of possible networks, selecting a network at regular intervals to evaluate the statistics in the model. For each MCMC step, n ($n = 1$ in the simple case) toggles are proposed to change the dyad(s) to the opposite value. The probability of accepting the proposed change is determined by the MH acceptance ratio. The role of the different MH methods implemented in `ergm` is to vary how the sets of dyads are selected for toggle proposals. This is used in some cases to improve the performance (speed and mixing) of the algorithm, and in other cases to constrain the sample space.

MH proposal methods implemented in the `ergm` package

MH proposals for non-constrained ergm models

InitMHP.randomtoggle Propose a randomly selected dyad to toggle.

InitMHP.TNT Default MH algorithm. Stratifies the population of dyads by edge status: those having ties and those having no ties (hence T/NT). This is useful for improving performance in sparse networks, because it gives at least 50% chance of proposing a toggle of an existing edge.

MH proposals for constrained ergm models

InitMHP.blockdiag MHp for $constraints = \tilde{}blockdiag$. Select a diagonal block according to the weight, then randomly select a dayd within the block for the toggle proposal.

InitMHP.blockdiagNonObserved MHp for $constraints = \tilde{}blockdiag + observed$. Similar to InitMHP.blockdiag, but applied only to missing dyads.

InitMHP.blockdiagNonObservedTNT Similar to InitMHP.blockdiagNonObserved, except that it selects ties and non-ties for proposed toggles (in the block by construction) with equal probability. Like the unconstrained TNT proposal, this is useful for improving performance in sparse networks.

InitMHP.blockdiagTNT MHp for $constraints = \tilde{}blockdiag$. Similar to InitMHP.blockdiag, except that it selects ties and non-ties for proposed toggles (in the block by construction) with equal probability. Like the unconstrained TNT proposal, this is useful for improving performance in sparse networks.

InitMHP.CondB1Degree MHp for $constraints = \tilde{}b1degrees$. For bipartite networks, randomly select an edge B1i, B2j and an empty dyad with the same node B1i, B1i, B2k, and propose to toggle both B1i, B2j and B1i, B2k. This ensures that the degrees of individual nodes in mode 1 are preserved.

InitMHP.CondB2Degree MHp for $constraints = \tilde{}b2degrees$. For bipartite network, randomly select an edge B1j, B2i and an empty dyad with the same node B2i, B1k, B2i, and propose to toggle both B1j, B2i and B1k, B2i. This ensures that the degrees of individual nodes in mode 2 are preserved.

InitMHP.CondDegree MHp for $constraints = \tilde{}degree$. Propose either 4 toggles (MH_CondDegreeTetrad) or 6 toggles (MH_CondDegreeHexad) at once. For undirected networks, propose 4 toggles (MH_CondDegreeTetrad). MH_CondDegreeTetrad selects two edges with no nodes in common, A1-A2 and B1-B2, s.t. A1-B2 and B1-A2 are not edges, and propose to replace the former two by the latter two. MH_CondDegreeHexad selects three edges A1->A2, B1->B2, C1->C2 at random and rotate them to A1->B2, B1->C2, and C1->A2.

InitMHP.CondDegreeDist MHp for $constraints = \tilde{}degreedist$. Randomly select a node (T) and its edge (E). If the head node of the edge (H) has 1 degree more than another randomly select node (A), and A is disconnected to both T and H, then propose to toggle E and the dyad between T and A.

InitMHP.CondDegreeMix MHp for $constraints = \tilde{}degreesmix$. Similar to `InitMHP.CondDegree`, except that the toggle is proposed only if the mixing matrix of degrees is preserved before and after the toggle.

InitMHP.ConstantEdges MHp for $constraints = \tilde{}edges$. Propose pairs of toggles that keep number of edges the same. This is done by (a) choosing an existing edge at random; (b) repeatedly choosing dyads at random until one is found that does not have an edge; and (c) proposing toggling both these dyads. Note that step (b) will be very inefficient if the network is nearly complete, so this proposal is NOT recommended for such networks. However, most network datasets are sparse, so this is not likely to be an issue.

InitMHP.CondInDegreeDist MHp for $constraints = \tilde{}idegreedist$. For directed networks, similar to InitMHP.CondDegreeDist, except for indegree case

InitMHP.CondOutDegreeDist MHp for $constraints = \tilde{}odegreedist$. For directed networks, similar to InitMHP.CondDegreeDist, except for outdegree case

InitMHP.fixedas MHp for $constraints = \tilde{}fixedas(present, absent)$. Select a random dyad that is not in either 'present' edgelist or 'absent' edgelist to toggle. Edges in 'present' and empty dyads in 'absent' are remained fixed.

InitMHP.fixedasTNT Similar to InitMHP.fixedas, except that it selects ties and non-ties for proposed toggles with equal probability. Like the unconstrained TNT proposal, this is useful for improving performance in sparse networks.

InitMHP.fixallbut MHp for $constraints = \tilde{}fixallbut(free.dyads)$. Select a random dyad that is in free.dyads edgelist to toggle.

InitMHP.fixallbutTNT Similar to InitMHP.fixallbut, except that it selects ties and non-ties for proposed toggles with equal probability. Like the unconstrained TNT proposal, this is useful for improving performance in sparse networks.

InitMHP.randomtoggleNonObserved MHp for $constraints = \tilde{}observed$. Randomly select a missing/non-observed dyad and propose a toggle.

InitMHP.NonObservedTNT Similar to InitMHP.randomtoggleNonObserved, except that it selects ties and non-ties for proposed toggles with equal probability. Like the unconstrained TNT proposal, this is useful for improving performance in sparse networks.

InitMHP.CondInDegree MHp for $constraints = \tilde{}idegrees$. For directed networks, randomly select two dyads with a common head node, one having an edge one not, and propose to swap the tie from one tail to the other.

InitMHP.CondOutDegree MHp for $constraints = \tilde{}odegrees$. For directed networks, randomly select two dyads with a common tail node, one having an edge and one not, and propose to swap the tie from one head to the other.

References

Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008a). A **statnet** Tutorial. *Journal of Statistical Software*, 24(8). `http://www.jstatsoft.org/v24/i08/`.

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

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008b). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software*, 24(3). `http://www.jstatsoft.org/v24/i03/`.

Krivitsky PN (2012). Exponential-Family Random Graph Models for Valued Networks. *Electronic Journal of Statistics*, 2012, 6, 1100-1128. `doi:10.1214/12-EJS696`

Morris M, Handcock MS, Hunter DR (2008). Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 24(4). `http://www.jstatsoft.org/v24/i04/`.

See Also

`ergm` package, `ergm`, `ergm-constraints`, `MHproposal`

`eut-upgrade` *Updating* `ergm.userterms` *prior to 3.1*

Description

Explanation and instructions for updating custom ERGM terms developed prior to the release of `ergm` version 3.1 (including 3.0–999 preview release) to be used with versions 3.1 or later.

Explanation

ergm.userterms — Statnet's mechanism enabling users to write their own ERGM terms — comes in a form of an R package containing files for the user to put their own statistics into (i.e., changestats.user.h, changestats.user.c, and InitErgmTerm.user.R), as well as some boilerplate to support them (e.g., edgetree.h, edgetree.c, changestat.h, changestat.c, etc.).

Although the ergm.userterms API is stable, recent developments in ergm have necessitated the boilerplate files in ergm.userterms to be updated. To reiterate, the user-written statistic source code (changestats.user.h, changestats.user.c, and InitErgmTerm.user.R) can be used without modification, but other files that came with the package need to be changed.

To make things easier in the future, we have implemented a mechanism (using R's LinkingTo API, in case you are wondering) that will keep things in sync in releases after the upcoming one. However, for the upcoming release, we need to transition to this new mechanism.

Instructions

The transition entails the following steps. They only need to be done once for a package. Future releases will keep up to date automatically.

1. Download the up-to-date ergm.userterms source from CRAN using, e.g., download.packages and unpack it.
2. Copy the R and C files defining the user-written terms (usually changestats.user.h, changestats.user.c, and InitErgmTerm.user.R) and *only* those files from the old ergm.userterms source code to the new. Do *not* copy the boilerplate files that you did not modify.
3. If you have customized the package DESCRIPTION file (e.g., to change the package name) or zzz.R (e.g., to change the startup message), modify them as needed in the updated ergm.userterms, but do *not* simply overwrite them with their old versions.
4. Make sure that your ergm installation is up to date, and rebuild ergm.userterms.

---

faux.desert.high *Faux desert High School as a network object*

Description

This data set represents a simulation of a directed in-school friendship network. The network is named faux.desert.high.

Usage

```
data(faux.desert.high)
```

Format

faux.desert.high is a network object with 107 vertices (students, in this case) and 439 directed edges (friendship nominations). To obtain additional summary information about it, type summary(faux.desert.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 Race attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes six possible values: White (non-Hisp.), Black (non-Hisp.), Hispanic, Asian (non-Hisp.), Native American, and 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: Software tools for the Statistical Modeling of Network Data*
statnet.org.

Source

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

The school in question (a single school with 7th through 12th grades) was selected from the Add Health "structure files." Documentation on these files can be found here: http://www.cpc.unc.edu/projects/addhealth/codebooks/wave1/structur.zip.

The stucture file contains directed out-ties representing each instance of a student who named another student as a friend. Students could nominate up to 5 male and 5 female friends. Note that registered students who did not take the AddHealth survey or who were not listed by name on the schools' student roster are not included in the stucture files. In addition, we removed any students with missing values for race, grade or sex.

The following ergm model was fit to the original data:

```
desert.fit <- ergm(original.net ~ edges + mutual + absdiff("grade") +
nodefactor("race", base=5) + nodefactor("grade", base=3) +
nodefactor("sex") + nodematch("race", diff = TRUE) +
nodematch("grade", diff = TRUE) + nodematch("sex", diff = FALSE) +
idegree(0:1) + odegree(0:1) + gwesp(0.1,fixed=T),
  constraints = ~bd(maxout=10),
  control =  control.ergm(MCMLE.steplength = .25, MCMC.burnin = 100000,
   MCMC.interval = 10000, MCMC.samplesize = 2500,
MCMLE.maxit = 100),
  verbose=T)
```

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

```
faux.desert.high <- simulate(desert.fit, nsim=1, burnin=1e+8,
 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.desert.high`, `faux.mesa.high`, `faux.magnolia.high`

`faux.dixon.high` *Faux dixon High School as a network object*

Description

This data set represents a simulation of a directed in-school friendship network. The network is named faux.dixon.high.

Usage

```
data(faux.dixon.high)
```

Format

`faux.dixon.high` is a `network` object with 248 vertices (students, in this case) and 1197 directed edges (friendship nominations). To obtain additional summary information about it, type `summary(faux.dixon.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 `Race` attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes six possible values: White (non-Hisp.), Black (non-Hisp.), Hispanic, Asian (non-Hisp.), Native American, and 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: Software tools for the Statistical Modeling of Network Data*
`statnet.org`.

Source

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

The school in question (a single school with 7th through 12th grades) was selected from the Add Health "structure files." Documentation on these files can be found here: http://www.cpc.unc.edu/projects/addhealth/codebooks/wave1/structur.zip.

The stucture file contains directed out-ties representing each instance of a student who named another student as a friend. Students could nominate up to 5 male and 5 female friends. Note that registered students who did not take the AddHealth survey or who were not listed by name on the schools' student roster are not included in the stucture files. In addition, we removed any students with missing values for race, grade or sex.

The following ergm model was fit to the original data:

```
dixon.fit <- ergm(original.net ~ edges + mutual + absdiff("grade") +
  nodefactor("race", base=5) + nodefactor("grade", base=3) +
  nodefactor("sex") + nodematch("race", diff = TRUE) +
  nodematch("grade", diff = TRUE) + nodematch("sex", diff = FALSE) +
  idegree(0:1) + odegree(0:1) + gwesp(0.1,fixed=T),
 constraints = ~bd(maxout=10),
 control =  control.ergm(MCMLE.steplength = .25, MCMC.burnin = 100000,
  MCMC.interval = 10000, MCMC.samplesize = 2500,
  MCMLE.maxit = 100),
 verbose=T)
```

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

```
faux.dixon.high <- simulate(dixon.fit, nsim=1, burnin=1e+8,
 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.desert.high, faux.mesa.high, faux.magnolia.high `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 `Race` attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes six possible values: White (non-Hisp.), Black (non-Hisp.), Hispanic, Asian (non-Hisp.), Native American, and 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: Software tools for the Statistical Modeling of Network Data*
`statnet.org`.

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=control.ergm(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`

`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 `Race` attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes six possible values: White (non-Hisp.), Black (non-Hisp.), Hispanic, Asian (non-Hisp.), Native American, and 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: Software tools for the Statistical Modeling of Network Data*
`statnet.org`.

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=TRUE) + nodematch("Race",diff=TRUE)
 + nodematch("Sex",diff=FALSE) + gwdegree(1.0,fixed=TRUE)
 + gwesp(1.0,fixed=TRUE) + gwdsp(1.0,fixed=TRUE)
```

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 (2008).

References

Hunter D.R., Goodreau S.M. and Handcock M.S. (2008). *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`, `faux.magnolia.high` `fix.curved` *Convert a curved ERGM into a corresponding “fixed” ERGM.*

Description

The generic `fix.curved` converts an `ergm` object or formula of a model with curved terms to the variant in which the curved parameters are fixed. Note that each term has to be treated as a special case.

Usage

```
## S3 method for class 'ergm'
fix.curved(object, ...)
## S3 method for class 'formula'
fix.curved(object, theta, response = NULL, ...)
```

Arguments

`object`	An `ergm` object or an ERGM formula. The curved terms of the given formula (or the formula used in the fit) must have all of their arguments passed by name.
`theta`	Curved model parameter configuration.
`response`	For valued ERGM, an edge attribute used as the response variable.
`...`	Unused at this time.

Details

Some ERGM terms such as `gwesp` and `gwdegree` have two forms: a curved form, for which their decay or similar parameters are to be estimated, and whose canonical statistics is a vector of the term’s components (`esp(1)`, `esp(2)`, . . . and `degree(1)`, `degree(2)`, . . . , respectively) and a "fixed" form where the decay or similar parameters are fixed, and whose canonical statistic is just the term itself. It is often desirable to fit a model estimating the curved parameters but simulate the "fixed" statistic.

This function thus takes in a fit or a formula and performs this mapping, returning a “fixed” model and parameter specification. It only works for curved ERGM terms included with the `ergm` package. It does not work with curved terms not included in ergm.

Value

A list with the following components:

`formula`	The “fixed” formula.
`theta`	The “fixed” parameter vector.

See Also

`ergm`, `simulate.ergm`

Examples

```
data(sampson)
gest<-ergm(samplike~edges+gwesp(alpha=.5,fixed=FALSE),
           control=control.ergm(MCMLE.maxit=3))
summary(gest)
# A statistic for esp(1),...,esp(16)
simulate(gest,statsonly=TRUE)

tmp<-fix.curved(gest)
tmp
# A gwesp() statistic only
simulate(tmp$formula, coef=tmp$theta, statsonly=TRUE)
```

`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), estimate="MPLE"))
summary(ergm(g4 ~ odegree(3), control=control.ergm(init=0)))
```

get.free.dyads *Create a network containing only edges meeting a specific criteria*

Description

`get.free.dyads` will create a network object with only edges that are not targeted by ergm-constraints. `get.miss.dyads` creates a network object with only edges that are missing/not present (not missing in the NA sense) in a network subject to constraints

Usage

```
get.free.dyads(constraints)

get.miss.dyads(constraints, constraints.obs)
```

Arguments

`constraints, constraints.obs`	A list of initialized constraints produced by `InitConstraint.*` functions for the model of interest and the same list with `+observed` constraint appended if missing dyads are present.

Value

A network object containing the specified set of edges

See Also

ergm-constraints

`get.node.attr` *Retrieve and check assumptions about vertex attributes (nodal covariates) in a network*

Description

The `get.node.attr` function returns the vector of nodal covariates for the given network and specified attribute if the attribute exists - execution will halt if the attribute is not correctly given as a single string or is not found in the vertex attribute list; optionally `get.node.attr` will also check that return vector is numeric, halting execution if not. The purpose is to validate assumptions before passing attribute data into an ergm term.

Usage

```
get.node.attr(nw, attrname, functionname = NULL, numeric = FALSE)
```

Arguments

`nw`	a `network` object
`attrname`	the name of a nodal attribute, as a character string
`functionname`	the name of the calling function a character string; this is only used for the warning messages that accompany a halt
`numeric`	logical, whether to halt execution if the return vector is not numeric; default=FALSE

Value

returns the vector of 'attrname' covariates for the vertices in the network

See Also

`get.vertex.attribute` for a version without the checking functionality

Examples

```
data(faux.mesa.high)
get.node.attr(faux.mesa.high,'Grade')
```

`Getting.Started` *Getting Started with "ergm": Fit, simulate and diagnose exponential-family models for networks*

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 given in `citation("ergm")`:

Mark S. Handcock, David R. Hunter, Carter T. Butts, Steven M. Goodreau, and Martina Morris. 2003 *ergm: Fit, simulate and diagnose exponential-family models for networks* `statnet.org`.

All published work 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: `statnet.org`. A tutorial, support newsgroup, references and links to further resources are provided there.

Author(s)

Mark S. Handcock <handcock@stat.ucla.edu>,
David R. Hunter <dhunter@stat.psu.edu>,
Carter T. Butts <buttsc@uci.edu>,
Steven M. Goodreau <goodreau@u.washington.edu>,
Pavel N. Krivitsky <krivitsky@stat.psu.edu>, and
Martina Morris <morrism@u.washington.edu>

Maintainer: David R. Hunter <dhunter@stat.psu.edu>

References

Admiraal R, Handcock MS (2007). **networksis**: Simulate bipartite graphs with fixed marginals through sequential importance sampling. Statnet Project, Seattle, WA. Version 1, statnet.org.

Bender-deMoll S, Morris M, Moody J (2008). Prototype Packages for Managing and Animating Longitudinal Network Data: **dynamicnetwork** and **rSoNIA**. *Journal of Statistical Software*, 24(7). http://www.jstatsoft.org/v24/i07/.

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

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

Butts CT (2007). **sna**: Tools for Social Network Analysis. R package version 2.3-2. http://CRAN.R-project.org/package=sna.

Butts CT (2008). **network**: A Package for Managing Relational Data in R. *Journal of Statistical Software*, 24(2). http://www.jstatsoft.org/v24/i02/.

Butts C (2015). **network**: The Statnet Project (http://www.statnet.org). R package version 1.12.0, http://CRAN.R-project.org/package=network.

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

Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008a). A **statnet** Tutorial. *Journal of Statistical Software*, 24(8). http://www.jstatsoft.org/v24/i08/.

Goodreau SM, Kitts J, Morris M (2008b). Birds of a Feather, or Friend of a Friend? Using Exponential Random Graph Models to Investigate Adolescent Social Networks. *Demography*, 45, in press.

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

Handcock MS (2003b). **degreenet**: Models for Skewed Count Distributions Relevant to Networks. Statnet Project, Seattle, WA. Version 1.0, statnet.org.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003a). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. Statnet Project, Seattle, WA. Version 2, statnet.org.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003b). **statnet**: Software Tools for the Statistical Modeling of Network Data. Statnet Project, Seattle, WA. Version 2, statnet.org.

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

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008b). **ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software*, 24(3). http://www.jstatsoft.org/v24/i03/.

Krivitsky PN, Handcock MS (2007). **latentnet**: Latent position and cluster models for statistical networks. Seattle, WA. Version 2, statnet.org.

Morris M, Handcock MS, Hunter DR (2008). Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 24(4). http://www.jstatsoft.org/v24/i04/.

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

---

gof *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(object,
                      ...,
                      coef=NULL,
                      GOF=NULL,
                      constraints=~.,
                      control=control.gof.formula(),
      unconditional=TRUE,
                      verbose=FALSE)
## S3 method for class 'ergm'
gof(object,
                   ...,
                   coef=NULL,
                   GOF=NULL,
                   constraints=NULL,
                   control=control.gof.ergm(),
                   verbose=FALSE)
```

Arguments

`object`	an R object. Either a formula or an `ergm` object. See documentation for `ergm`.
`...`	Additional arguments, to be passed to lower-level functions in the future.
`coef`	When given either a formula or an object of class ergm, `coef` are the parameters from which the sample is drawn. By default set to a vector of 0.
`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. Currently supported terms are the degree distribution ("degree" for undirected graphs, or "idegree" and/or "odegree" for directed graphs), geodesic distances ("distance"), shared partner distributions ("espartners" and "dspartners"), the triad census ("triadcensus"), and the terms of the original model ("model"). The default formula for undirected networks is `~ degree + espartners + distance`, and the default formula for directed networks is `~ idegree + odegree + espartners + distance`.
`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 `control.gof.formula` or `control.gof.ergm` (which have different defaults).
`unconditional`	logical; if `TRUE`, the simulation is unconditional on the observed dyads. if not `TRUE`, the simulation is conditional on the observed dyads. This is primarily used internally when the network has missing data and a conditional GoF is produced.
`verbose`	Provide verbose information on the progress of the simulation.

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`.

For `GOF = ~model`, the model's observed sufficient statistics are plotted as quantiles of the simulated sample. In a good fit, the observed statistics should be near the sample median (0.5).

For `gof.ergm` and `gof.formula`, default behavior depends on the directedness of the network involved; if undirected then degree, espartners, and distance are used as default properties to examine. If the network in question is directed, "degree" in the above is replaced by idegree and odegree.

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

# test the gof.ergm function
gofflo <- gof(gest)
gofflo
summary(gofflo)

# Plot 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 log-odds
plot(gofflo, plotlogodds=TRUE)

# Use the formula version of gof
gofflo2 <-gof(flomarriage ~ edges + kstar(2), coef=c(-1.6339, 0.0049))
plot(gofflo2)
```

`is.curved` *Testing for curved exponential family*

Description

These functions test whether an ERGM fit or formula is curved.

Usage

```
## S3 method for class 'ergm'
is.curved(object, ...)
## S3 method for class 'formula'
is.curved(object,
                    response=NULL,
                    basis=NULL,
                    ...)
## S3 method for class 'ergm.model'
is.curved(object, ...)
## S3 method for class 'NULL'
is.curved(object, ...)
```

Arguments

`object`	An `ergm` object or an ERGM formula.
`response, basis`	Name of the edge attribute whose value is to be modeled. See `ergm`.
`...`	Unused at this time.

Details

Curvature is checked by testing if all model parameters are canonical.

Value

`TRUE` if the model fit or one implied by the formula is curved; `FALSE` otherwise.

`is.durational` *Testing for durational dependent models*

Description

These functions test whether an ERGM model or formula is durational dependent or not. If the formula or model does not include any terms that need information about the duration of existing ties, the ergm proceass can use more efficient internal data structures.

Usage

```
## S3 method for class 'character'
is.durational(object, ...)
## S3 method for class 'ergm.model'
is.durational(object, ...)
## S3 method for class 'formula'
is.durational(object,
                                        response=NULL,
                                        basis=NULL,
                                        ...)
```

Arguments

`object`	An `ergm` object or an ERGM formula, or some characters, e.g., object="all" for monitoring purpose.
`response, basis`	See `ergm`.
`...`	Unused at this time.

Value

`TRUE` if the ERGM terms in the formula or model are durational dependent ; `FALSE` otherwise.

is.dyad.independent *Testing for dyad-independence*

Description

These functions test whether an ERGM fit or formula is dyad-independent.

Usage

```
## S3 method for class 'ergm'
is.dyad.independent(object, ...)
## S3 method for class 'formula'
is.dyad.independent(object,
                    response=NULL,
                    basis=NULL,
                    ...)
## S3 method for class 'conlist'
is.dyad.independent(object,
                    object.obs = NULL,
                    ...)
## S3 method for class 'ergm.model'
is.dyad.independent(object, ...)
## S3 method for class 'NULL'
is.dyad.independent(object, ...)
```

Arguments

`object`	An `ergm` object or an ERGM formula.
`response, basis`	Name of the edge attribute whose value is to be modeled. See `ergm`.
`object.obs`	
`...`	Unused at this time.

Details

Dyad independence is determined by checking if all of the constituent parts of the object (formula, ergm terms, etc) are flagged as dyad-independent.

Value

`TRUE` if the model fit or one implied by the formula is dyad-independent; `FALSE` otherwise.

is.inCH *Determine whether a vector is in the closure of the convex hull of some sample of vectors*

Description

`is.inCH` returns `TRUE` if and only if `p` is contained in the convex hull of the points given as the rows of `M`. If `p` is a matrix, each row is tested individually, and `TRUE` is returned if all rows are in the convex hull.

Usage

```
is.inCH(p, M, verbose=FALSE, ...)
```

Arguments

`p`	A d-dimensional vector or a matrix with d columns
`M`	An r by d matrix. Each row of `M` is a d-dimensional vector.
`verbose`	A logical vector indicating whether to print progress
`...`	arguments passed directly to linear program solver

Details

The d-vector `p` is in the convex hull of the d-vectors forming the rows of `M` if and only if there exists no separating hyperplane between `p` and the rows of `M`. This condition may be reworded as follows:

Letting $q = (1p')'$ and $L = (1M)$, if the maximum value of $z'q$ for all z such that $z'L \leq 0$ equals zero (the maximum must be at least zero since z=0 gives zero), then there is no separating hyperplane and so `p` is contained in the convex hull of the rows of `M`. So the question of interest becomes a constrained optimization problem.

Solving this problem relies on the package `lpSolve` to solve a linear program. We may put the program in "standard form" by writing $z = a - b$, where a and b are nonnegative vectors. If we write $x = (a'b')'$, we obtain the linear program given by:

Minimize $(-q'q')x$ subject to $x'(L - L) \leq 0$ and $x \geq 0$. One additional constraint arises because whenever any strictly negative value of $(-q'q')x$ may be achieved, doubling x arbitrarily many times makes this value arbitrarily large in the negative direction, so no minimizer exists. Therefore, we add the constraint $(q' - q')x \leq 1$.

This function is used in the "stepping" algorithm of Hummel et al (2012).

Value

Logical, telling whether `p` is (or all rows of `p` are) in the closed convex hull of the points in `M`.

References

- `http://www.cs.mcgill.ca/~fukuda/soft/polyfaq/node22.html`
- Hummel, R. M., Hunter, D. R., and Handcock, M. S. (2012), Improving Simulation-Based Algorithms for Fitting ERGMs, Journal of Computational and Graphical Statistics, 21: 920-939.

`kapferer` *Kapferer's tailor shop data*

Description

This well-known social network dataset, collected by Bruce Kapferer in Zambia from June 1965 to August 1965, involves interactions among workers in a tailor shop as observed by Kapferer himself. Here, an interaction is defined by Kapferer as "continuous uninterrupted social activity involving the participation of at least two persons"; only transactions that were relatively frequent are recorded. All of the interactions in this particular dataset are "sociational", as opposed to "instrumental". Kapferer explains the difference (p. 164) as follows:

"I have classed as transactions which were sociational in content those where the activity was markedly convivial such as general conversation, the sharing of gossip and the enjoyment of a drink together. Examples of instrumental transactions are the lending or giving of money, assistance at times of personal crisis and help at work."

Kapferer also observed and recorded instrumental transactions, many of which are unilateral (directed) rather than reciprocal (undirected), though those transactions are not recorded here. In addition, there was a second period of data collection, from September 1965 to January 1966, but these data are also not recorded here. All data are given in Kapferer's 1972 book on pp. 176-179.

During the first time period, there were 43 individuals working in this particular tailor shop; however, the better-known dataset includes only those 39 individuals who were present during both time collection periods. (Missing are the workers named Lenard, Peter, Lazarus, and Laurent.) Thus, we give two separate network datasets here: `kapferer` is the well-known 39-individual dataset, whereas `kapferer2` is the full 43-individual dataset.

Usage

```
data(kapferer)
```

Format

Two `network` objects, `kapferer` and `kapferer2`. The `kapferer` dataset contains only the 39 individuals who were present at both data-collection time periods. However, these data only reflect data collected during the first period. The individuals' names are included as a nodal covariate called `names`.

Source

Original source: Kapferer, Bruce (1972), Strategy and Transaction in an African Factory, Manchester University Press.

`lasttoggle` *Storing last toggle information in a network*

Description

An informal extension to `network` objects allowing some limited temporal information to be stored.

Details

WARNING: THIS DOCUMENTATION IS PROVIDED AS A COURTESY, AND THE API DESCRIBED IS SUBJECT TO CHANGE WITHOUT NOTICE, DOWN TO COMPLETE REMOVAL. NOT ALL FUNCTIONS THAT COULD SUPPORT IT DO. USE AT YOUR OWN RISK.

While `networkDynamic` provides a flexible, consistent method for storing dynamic networks, the `C` routines of `ergm` and `tergm` required a simpler and more lightweight representation.

This representation consisted of a single integer representing the time stamp and an integer vector of length to `network.dyadcount(nw)` — the number of potential ties in the network, giving the last time point during which each of the dyads in the network had changed.

Though this is an API intended for internal use, some functions, like `stergm` (for EGMME), `simulate`, and `summary` can be passed networks with this information using the following `network` (i.e., `%n%`) attributes:

`time` the time stamp associated with the network

`lasttoggle` a vector of length `network.dyadcount(nw)`, giving the last change time associated with each dyad. See the source code of `ergm` internal functions `to.matrix.lasttoggle`, `ergm.el.lasttoggle`, and `to.lasttoggle.matrix` for how they are serialized.

For technical reasons, the `tergm` routines treat the `lasttoggle` time points as shifted by -1.

Again, this API is subject to change without notice.

`logLik.ergm` *A `logLik` method for `ergm`.*

Description

A function to return the log-likelihood associated with an `ergm` fit, evaluating it if necessary. `logLikNull` computes, when possible (see Value), the log-probability of observing the observed, unconstrained dyads of the network observed under the null model.

Usage

```
## S3 method for class 'ergm'
logLik(object,
                      add=FALSE,
                      force.reeval=FALSE,
                      eval.loglik=add || force.reeval,
                      control=control.logLik.ergm(),
                      ...)

logLikNull(object, ...)

## S3 method for class 'ergm'
logLikNull(object,
                          control=control.logLik.ergm(),
                          ...)
```

Arguments

`object`	An `ergm` fit, returned by `ergm`.
`add`	Logical: If TRUE, instead of returning the log-likelihood, return `object` with log-likelihood value set.
`force.reeval`	Logical: If TRUE, reestimate the log-likelihood even if `object` already has an estiamte.
`eval.loglik`	Logical: If TRUE, evaluate the log-likelihood if not set on `object`.
`control`	A list of control parameters for algorithm tuning. Constructed using `control.logLik.ergm`.
`...`	Other arguments to the likelihood functions.

Details

If the log-likelihood was not computed for `object`, produces an error unless `eval.loglik=TRUE`

Value

The form of the output of `logLik.ergm` depends on `add`: `add=FALSE` (the default), a `logLik` object. If `add=TRUE` (the default), an `ergm` object with the log-likelihood set.

`logLikNull` returns an object of type `logLik` if it is able to compute the null model probability, and `NA` otherwise.

As of version 3.1, all likelihoods for which `logLikNull` is not implemented are computed relative to the reference measure. (I.e., a null model, with no terms, is defined to have likelihood of 0, and all other models are defined relative to that.)

References

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

See Also

logLik, ergm.bridge.llr, ergm.bridge.dindstart.llk

Examples

```
# See help(ergm) for a description of this model. The likelihood will
# not be evaluated.
data(florentine)
## Not run:
# The default maximum number of iterations is currently 20. We'll only
# use 2 here for speed's sake.
gest <- ergm(flomarriage ~ kstar(1:2) + absdiff("wealth") + triangle, eval.loglik=FALSE)

gest <- ergm(flomarriage ~ kstar(1:2) + absdiff("wealth") + triangle, eval.loglik=FALSE,
             control=control.ergm(MCMLE.maxit=2))
# Log-likelihood is not evaluated, so no deviance, AIC, or BIC:
summary(gest)
# Evaluate the log-likelihood and attach it to the object.

# The default number of bridges is currently 20. We'll only use 3 here
# for speed's sake.
gest.logLik <- logLik(gest, add=TRUE)

gest.logLik <- logLik(gest, add=TRUE, control=control.logLik.ergm(nsteps=3))
# Deviances, AIC, and BIC are now shown:
summary(gest.logLik)
# Null model likelihood can also be evaluated, but not for all constraints:
logLikNull(gest) # == network.dyadcount(flomarriage)*log(1/2)

## End(Not run)
```

---

mcmc.diagnostics *Conduct MCMC diagnostics on an ergm fit*

Description

This function prints diagnistic information and creates simple diagnostic plots for the MCMC sampled statistics produced from a fit.

Usage

```
## S3 method for class 'ergm'
 mcmc.diagnostics(object,
               center=TRUE,
               esteq=TRUE,
               vars.per.page=3,
               ...)
## S3 method for class 'mcmc.list.ergm'
plot(x,
                    main = NULL,
                    vars.per.page = 3,
                    ...)
```

Arguments

`object`	An ergm object. See documentation for `ergm`.
`center`	Logical: If TRUE, ; center the samples on the observed statistics.
`esteq`	Logical: If TRUE, for statistics corresponding to curved ERGM terms, summarize the curved statistics by their estimating equation values (evaluated at the MLE of any curved parameters) (i.e., $\eta'_I(\hat{\theta}) \cdot g_I(y)$ for I being indices of the canonical parameters in question), rather than the canonical (sufficient) vectors of the curved statistics ($g_I(y)$).
`vars.per.page`	Number of rows (one variable per row) per plotting page. Ignored if `latticeExtra` package is not installed.
`x`	an mcmc.list object containing the mcmc diagnostic samples
`main`	character, main plot heading title
`...`	Additional arguments, to be passed to plotting functions.

Details

A pair of plots are produced for each statistic:a trace of the sampled output statistic values on the left and density estimate for each variable in the MCMC chain on the right. Diagnostics printed to the console include correlations and convergence diagnostics.

Recent changes in the ergm estimation algorithm mean that these plots can no longer be used to ensure that the mean statistics from the model match the observed network statistics. For that functionality, please use the GOF command: `gof(object, GOF=~model)`.

In fact, an ergm output `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/software/bugs/the-bugs-project-winbugs/coda-readme/`.

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

Value

`mcmc.diagnostics.ergm` returns some degeneracy information, if it is included in the original object. The function is mainly used for its side effect, which is to produce plots and summary output based on those plots.

References

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.

This function is based on the coda package 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.

See Also

`ergm`, `network` package, coda package, `summary.ergm`

Examples

```
## Not run:
#
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)
#
# A full range of diagnostics is available
# using codamenu()
#

## End(Not run)
```

`MHproposal` *Functions to initialize the MHproposal object*

Description

S3 Functions that initialize the Metropolis-Hastings Proposal (MHproposal) object using the InitMHP.* function that corresponds to the name given in 'object'. These functions are not generally called directly by the user. See ergm_MH_proposals for general explanation and lists of available Metropolis-Hastings proposal types.

Usage

```
## S3 method for class 'character'
MHproposal(object,
                   arguments,
                   nw, ...,
                   response=NULL,
                   reference=reference
                   )


## S3 method for class 'formula'
MHproposal(object,
                   arguments,
                   nw,
                   weights="default",
                   class="c",
                   reference=~Bernoulli,
                   response=NULL,
                  ...)

## S3 method for class 'ergm'
MHproposal(object, ...,
                   constraints=NULL,
                   arguments=NULL,
                   nw=NULL,
                   weights=NULL,
                   class="c",
                   reference=NULL,
                   response=NULL)
```

Arguments

`object`	Either a character, 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.
`nw`	The network object originally given to `ergm` via 'formula'
`weights`	Specifies the method used to allocate probabilities of being proposed to dyads; options are "TNT", "TNT10", "random", "nonobserved" and "default"; default="default"
`arguments`	A list of parameters used by the Init.MHP routines
`response`	*EXPERIMENTAL.* Name of the edge attribute whose value is to be modeled. Defaults to `NULL` for simple presence or absence.
`reference`	*EXPERIMENTAL.* One-sided formula whose RHS gives the reference measure to be used. (Defaults to `~Bernoulli`.)
`class`	The class of the proposal; choices include "c", "f", and "d" default="c".

`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` and see list of implemented constraints for more information.
`...`	Further arguments passed to other functions.

Value

Returns an MHproposal object: a list with class `'MHProposal'` containing the following named elements:

- name : the C name of the proposal
- inputs : NULL (I think - the only non-null value returned by the InitMH is for <nobetweengroupties>, but this isn't included in the look-up table
- package: shared library name where the proposal can be found (usually "ergm")
- arguments: list of arguments passed to the InitMHP function; in particular,
  - constraints: list of constraints
  - constraints$bd: the list of parameters to bound degree in the fitting process and returned by `ergm.bounddeg`

See Also

`InitMHP`

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

Usage

```
network.update(nw, newmatrix, matrix.type=NULL, output="network",
               ignore.nattr = c("bipartite", "directed", "hyper",
               "loops", "mnext", "multiple", "n"), ignore.vattr = c())
```

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.
`output`	Currently unused.
`ignore.nattr`	character vector of the names of network-level attributes to ignore when updating network objects (defaults to standard network properties)
`ignore.vattr`	character vector of the names of vertex-level attributes to ignore when updating network objects

Value

`network.update` returns a new `network` object with the edges specified by `newmatrix` and network and vertex attributes copied from the input network `nw`. Input network is not modified.

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, matrix.type="adjacency")
# Try this with an edgelist
rand.mat <- as.matrix.network.edgelist(flomarriage)[1:5,]
network.update(flomarriage, rand.mat, matrix.type="edgelist")
```

newnw.extract — *Internal function to create a new network from the ergm MCMC sample output*

Description

An internal function to generate a new network object using the output (lists of toggled heads and tail vertices) from an ergm MCMC or SAN process.

Usage

```
newnw.extract(oldnw, z, output = "network", response = NULL)
```

Arguments

oldnw	a network object (presumably input to the ergm process) from which the network- and vertex-level attributes will be copied
z	a list having either a component named newedgelist or two components newtails and newheads containing the ids of the head and tails vertices of the edges. Optionall newweights containing edgewights.
output	passed to network.update, which claims not to use it
response	optional character string giving the name of the edge attribute where the edge values (weight/count) should be stored.

Value

a network object with properties copied from oldnw and edges corresponding to the lists of tails and head vertex ids in z

Note

This is an internal ergm function, it most cases with edgelists to be converted to networks it will probably be simpler to use network.edgelist

See Also

network.edgelist, network.update

nvattr.copy.network *Copy network- and vertex-level attributes between two network objects*

Description

An internal ergm utility function to copy the network-level attributes and vertex-level attributes from one `network` object to another, ignoring some standard properties by default.

Usage

```
nvattr.copy.network(to, from, ignore = c("bipartite", "directed",
                                         "hyper", "loops", "mnext",
                                         "multiple", "n"))
```

Arguments

`to`	the `network` that attributes should be copied to
`from`	the `network` that attributes should be copied to
`ignore`	vector of charcter names of network attributes that should not be copied. Default is the standard list of network properties created by `network.initialize`

Value

returns the `to` network, with attributes copied from `from`

Note

does not check that networks are of the same size, etc

See Also

`set.vertex.attribute`, `set.network.attribute`

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

```
## Not run:
#
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), coef=c(-1.6339, 0.0049))

## End(Not run)
```

plot.network.ergm *Two-Dimensional Visualization of Networks*

Description

plot.network.ergm 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 'ergm'
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"),
    seed=NULL,
    latent.control=list(maxit=500,
                        trace=0,
                        dyadsample=10000,
                        penalty.sigma=c(5,0.5),
                        nsubsample=200),
    colornames="rainbow",
    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.
`seed`	Integer for seeding random number generator. See `set.seed`.
`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` package. 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

```
## S3 method for class 'ergm'
print(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 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

NOTE: It appears that as of `ergm` 3.6.0, the vertex labels are been permutted. This will be fixed in the next release.

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. They have three vertex attributes:

group Groups of novices as classified by Sampson: "Loyal", "Outcasts", and "Young Turks". There is also an interstitial group not represented here.

cloisterville An indicator if attendance the minor seminary of "Cloisterville" before coming to the monastery.

vertex.names The given names of the novices.

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.

This dataset was updated for version 2.5 (March 2012) to add the `cloisterville` variable and refine the names. This information is from de Nooy, Mrvar, and Batagelj (2005). The original vertex names were: Romul_10, Bonaven_5, Ambrose_9, Berth_6, Peter_4, Louis_11, Victor_8, Winf_12, John_1, Greg_2, Hugh_14, Boni_15, Mark_7, Albert_16, Amand_13, Basil_3, Elias_17, Simp_18.

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.

http://vlado.fmf.uni-lj.si/pub/networks/data/esna/sampson.htm

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.

Wouter de Nooy, Andrej Mrvar, Vladimir Batagelj (2005) *Exploratory Social Network Analysis with Pajek*, Cambridge: Cambridge University Press

See Also

sampson, florentine, network, plot.network, ergm

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

Description

NOTE: It appears that as of `ergm` 3.6.0, the vertex labels are been permutted. This will be fixed in the next release.

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 network. 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.

The graph is stored as an `network` objects. It has three vertex attributes:

group Groups of novices as classified by Sampson: “loyal”, “outcasts”, and “Turks”. There is also an interstitial group not represented here.

cloisterville An indicator of attendance the minor seminary of “Cloisterville” before coming to the monastery.

vertex.names The given names of the novices and their IDs in the original dataset.

In addition, it has an edge attribute, `nominations`, giving the number of times (out of 3) that monk A nominated monk B.

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.

This dataset was updated for version 2.5 (March 2012) to add the `cloisterville` variable and refine the names. This information is from de Nooy, Mrvar, and Batagelj (2005). The original vertex names were: Romul_10, Bonaven_5, Ambrose_9, Berth_6, Peter_4, Louis_11, Victor_8, Winf_12, John_1, Greg_2, Hugh_14, Boni_15, Mark_7, Albert_16, Amand_13, Basil_3, Elias_17, Simp_18.

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.

http://vlado.fmf.uni-lj.si/pub/networks/data/esna/sampson.htm

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.

Wouter de Nooy, Andrej Mrvar, Vladimir Batagelj (2005) *Exploratory Social Network Analysis with Pajek*, Cambridge: Cambridge University Press

See Also

florentine, network, plot.network, ergm

san — *Use Simulated Annealing to attempt to match a network to a vector of mean statistics*

Description

This function attempts to find a network or networks whose statistics match those passed in via the `target.stats` vector.

Usage

```
## S3 method for class 'formula'
san(object,
                      response=NULL,
                      reference=~Bernoulli,
                      constraints=~.,
                      target.stats=NULL,
                      nsim=1,
                      basis=NULL,
                      sequential=TRUE,
                      control=control.san(),
                      verbose=FALSE,
                      ...)
## S3 method for class 'ergm'
san(object,
                   formula=object$formula,
                   constraints=object$constraints,
                   target.stats=object$target.stats,
                   nsim=1,
                   basis=NULL,
                   sequential=TRUE,
                   control=object$control$SAN.control,
                   verbose=FALSE,
                   ...)
```

Arguments

`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.
`response`	*EXPERIMENTAL.* Name of the edge attribute whose value is to be modeled. Defaults to `NULL` for simple presence or absence.
`reference`	*EXPERIMENTAL.* One-sided formula whose RHS gives the reference measure to be used. (Defaults to `~Bernoulli`.)

`formula`	(By default, the `formula` is taken from the `ergm` object. If a different `formula` object is wanted, specify it here.
`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` and see list of implemented constraints 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.
`target.stats`	A vector of the same length as the number of terms implied by the formula, which is either `object` itself in the case of `san.formula` or `object$formula` in the case of `san.ergm`.
`nsim`	Number of desired networks.
`basis`	If not NULL, a `network` object used to start the Markov chain. If NULL, this is taken to be the network named in the formula.
`sequential`	Logical: If TRUE, the returned draws always use the prior draw as the starting network; if FALSE, they always use the original network.
`control`	A list of control parameters for algorithm tuning; see `control.san`.
`verbose`	Logical: If TRUE, print out more detailed information as the simulation runs.
`...`	Further arguments passed to other functions.

Value

A network or list of networks that hopefully have network statistics close to the `target.stats` vector.

`search.ergmTerms` *Search the ergm-terms documentation for appropriate terms*

Description

Searches through the `ergm.terms` help page and prints out a list of terms appropriate for the specified network's structural constraints, optionally restricting by additional categories and keyword matches.

Usage

```
search.ergmTerms(keyword, net,  categories, name)
```

Arguments

`keyword`	optional character keyword to search for in the text of the term descriptions. Only matching terms will be returned. Matching is case insensitive.
`net`	a network object that the term would be applied to, used as template to determine directedness, bipartite, etc
`categories`	optional character vector of category tags to use to restrict the results (i.e. 'curved', 'triad-related')
`name`	optional character name of a specific term to return

Details

Uses grep internally to match keywords against the term description, so keywords is currently matched as a single phrase. Category tags will only return a match if all of the specified tags are included in the term.

Value

prints out the name and short description of matching terms, and invisibly returns them as a list. If name is specified, prints out the full definition for the named term.

Author(s)

skyebend@uw.edu

See Also

See also ergm.terms for the complete documentation

Examples

```
# find all of the terms that mention triangles
search.ergmTerms('triangle')

# two ways to search for bipartite terms:

# search using a bipartite net as a template
myNet<-network.initialize(5,bipartite=3)
search.ergmTerms(net=myNet)

# or request the bipartite category
search.ergmTerms(categories='bipartite')

# search on multiple categories
search.ergmTerms(categories=c('bipartite','dyad-independent'))

# print out the content for a specific term
search.ergmTerms(name='b2factor')
```

---

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,
                            coef,
                            response=NULL, reference=~Bernoulli,
                            constraints=~.,
                            monitor=NULL,
                            basis=NULL,
                            statsonly=FALSE,
                            esteq=FALSE,
                            sequential=TRUE,
                            control=control.simulate.formula(),
                            verbose=FALSE,
                            ...)
## S3 method for class 'ergm'
simulate(object, nsim=1, seed=NULL,
                         coef=object$coef,
                         response=object$response, reference=object$reference,
                         constraints=object$constraints,
                         monitor=NULL,
                         statsonly=FALSE,
                         esteq=FALSE,
                         sequential=TRUE,
                         control=control.simulate.ergm(),
                         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. See `set.seed`.
`coef`	Vector of parameter values for the model from which the sample is to be drawn. If `object` is of class `ergm`, the default value is the vector of estimated coefficients.
`response`	*EXPERIMENTAL.* Name of the edge attribute whose value is to be modeled. Defaults to `NULL` for simple presence or absence, modeled via binary ERGM terms. Passing anything but `NULL` uses valued ERGM terms.
`reference`	*EXPERIMENTAL.* A one-sided formula specifying the reference measure ($h(y)$) to be used. (Defaults to `~Bernoulli`.) See help for ERGM reference measures implemented in the `ergm` package.

`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` and see list of implemented constraints 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.
`monitor`	A one-sided formula specifying one or more terms whose value is to be monitored. These terms are appeneded to the model, along with a coefficient of 0, so their statistics are returned.
`basis`	An optional `network` object to start the Markov chain. If omitted, the default is the left-hand-side of the `formula`. If neither a left-hand-side nor a `basis` is present, an error results because the characteristics of the network (e.g., size and directedness) must be specified.
`statsonly`	Logical: If TRUE, return only the network statistics, not the network(s) themselves.
`esteq`	Logical: If TRUE, compute the sample estimating equations of an ERGM: if the model is non-curved, all non-offset statistics are returned either way, but if the model is curved, the score estimating function values (3.1) by Hunter and Handcock (2006) are returned instead.
`sequential`	Logical: If FALSE, each of the `nsim` simulated Markov chains begins at the initial network. If TRUE, the end of one simulation is used as the start of the next. Irrelevant when `nsim=1`.
`control`	A list of control parameters for algorithm tuning. Constructed using `control.simulate.ergm` or `control.simulate.formula`, which have different defaults.
`verbose`	Logical: If TRUE, extra information is printed as the Markov chain progresses.
`...`	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 `coef` 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 `coef`.

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

If `statsonly==TRUE` a matrix containing the simulated network statistics. If `control$parallel>0`, the statistics from each Markov chain are stacked.

Otherwise, if `nsim==1`, an object of class `network`. If `nsim>1`, it returns an object of class `network.list`: a list of networks with the following `attr`-style attributes on the list:

`formula`	The `formula` used to generate the sample.

`stats`	The `nsim` $\times p$ matrix of network statistics, where p is the number of network statistics specified in the model.
`control`	Control parameters used to generate the sample.
`constraints`	Constraints used to generate the sample.
`reference`	The reference measure for the sample.
`monitor`	The monitoring formula.
`response`	The edge attribute used as a response.

If `statsonly==FALSE && control$parallel>0` the returned networks are "interleaved", in the sense that for `y[i,j]` is the `j`th network from MCMC chain `i`, the sequence returned if `control$parallel==2` is `list(y[1,1], y[2,1], y[1,2], y[2,2], y[1,3], y[2,3], ...)`. This is different from the behavior when `statsonly==TRUE`. This detail may change in the future.

This object has summary and print methods.

See Also

`ergm`, `network`

Examples

```
#
# Let's draw from a Bernoulli model with 16 nodes
# and density 0.5 (i.e., coef = c(0,0))
#
g.sim <- simulate(network(16) ~ edges + mutual, coef=c(0, 0))
#
# What are the statistics like?
#
summary(g.sim ~ edges + mutual)
#
# Now simulate a network with higher mutuality
#
g.sim <- simulate(network(16) ~ edges + mutual, coef=c(0,2))
#
# How do the statistics look?
#
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 3 realizations
# of a edges and 2-star network
#
g.sim <- simulate(~edges+kstar(2), nsim=3, coef=c(-1.8,0.03),
               basis=g.use, control=control.simulate(
                 MCMC.burnin=1000,
                 MCMC.interval=100))
```

```
g.sim
summary(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 (satatistics only), and observe the number
# of triangles as well.
#
g.sim <- simulate(gest, nsim=10,
            monitor=~triangles, statsonly=TRUE,
            control=control.simulate.ergm(MCMC.burnin=1000, MCMC.interval=100))
g.sim
```

`summary.ergm` *Summarizing ERGM Model Fits*

Description

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

Usage

```
## S3 method for class 'ergm'
summary(object, ...,
                 digits = max(3, getOption("digits") - 3),
                 correlation = FALSE, covariance = FALSE,
                 total.variation=TRUE)
```

Arguments

`object`	an object of class `"ergm"`, usually, a result of a call to `ergm`.
`digits`	Significant digits for coefficients
`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.
`total.variation`	logical; if `TRUE`, the standard errors reported in the `Std. Error` column are based on the sum of the likelihood variation and the MCMC variation. If `FALSE` only the likelihood varuation is used. The p-values are based on this source of variation.

`...`	Arguments to `logLik.ergm`

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

```
## Not run:
#
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)

## End(Not run)
```

---

summary.network.list *Summarizing network.list objects*

Description

summary and print methods for class network.list.

Usage

```
  ## S3 method for class 'network.list'
summary(object,
                                stats.print=TRUE,
                                net.print=FALSE,
                                net.summary=FALSE,
                                ...)

  ## S3 method for class 'network.list'
print(x, stats.print=FALSE, ...)
```

Arguments

`object, x`	an object of class `network.list`, such as the output from `simulate.ergm`
`stats.print`	Logical: If TRUE, print network statistics.
`net.print`	Logical: If TRUE, print network overviews.
`net.summary`	Logical: If TRUE, print network summaries.
`...`	Additional arguments to be passed to lower-level functions.

Value

The `summary.network.list` function returns a `summary.network` object. The `print.summary.list` function calls the `summary.network.list` function but returns the `network.list` object.

See Also

`simulate.ergm`

Examples

```
# 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 3 realizations
# of a model with edges and 2-star terms
#
g.sim <- simulate(~edges+kstar(2), nsim=3, coef=c(-1.8, 0.03),
               basis=g.use, control=control.simulate(
                 MCMC.burnin=100000,
                 MCMC.interval=1000))
print(g.sim)
summary(g.sim)
```

`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

```
## Default S3 method:
summary.statistics(object, response=NULL, ..., basis=NULL)
## S3 method for class 'matrix'
summary.statistics(object, response=NULL, ..., basis=NULL)
## S3 method for class 'network'
summary.statistics(object, response=NULL, ..., basis=NULL)
## S3 method for class 'network.list'
summary.statistics(object, response=NULL, ..., basis=NULL)
## S3 method for class 'formula'
summary.statistics(object, ..., basis=NULL)
## S3 method for class 'ergm'
summary.statistics(object, ..., basis=NULL)
```

Arguments

`object`	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.
`response`	Name of the edge attribute whose value is to be modeled. Defaults to `NULL` for simple presence or absence, modeled via binary ERGM terms. Passing anything but `NULL` uses valued ERGM terms.
`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.

`summary.formula` for networks understands the `lasttoggle` "API".

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
```

`vcov.ergm` *Extract Model Covariance Matrix*

Description

`vcov` is a method which extracts the covariance matrix from the output object returned by the `ergm` estimation.

Usage

```
## S3 method for class 'ergm'
vcov(object, sources=c("all","model","estimation"), ...)
```

Arguments

`object`	The `ergm` output object.
`sources`	Specify whether to return the covariance matrix from the ERGM model, the estimation process, or both combined.
`...`	other arguments.

Value

Coefficients extracted from the model object `object`.

See Also

`coef.ergm`

Examples

```
data(florentine)
fit <- ergm(flomarriage ~ edges + concurrent)
vcov(fit, sources="model")
vcov(fit, sources="estimation")
vcov(fit, sources="all") # the default
```

`wtd.median` *Weighted Median*

Description

Compute weighted median.

Usage

```
wtd.median (x, na.rm = FALSE, weight=FALSE)
```

Arguments

`x`	Vector of data, same length as `weight`
`na.rm`	Logical: Should NAs be stripped before computation proceeds?
`weight`	Vector of weights

Details

Uses a simple algorithm based on sorting.

Value

Returns an empirical .5 quantile from a weighted sample.

Index