Package 'ergm'

July 19, 2014

Version 3.1.3

Date 2014-07-19

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

Depends statnet.common (>= 3.1-0), network (>= 1.7-1)

Imports robustbase (>= 0.9-10), coda, trust, Matrix

Suggests lattice, latticeExtra, sna, Rglpk, snow, latentnet

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

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]

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

NeedsCompilation yes

Repository CRAN

Date/Publication 2014-07-19 16:43:04

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 (2006). **netperm**: Permutation Models for Relational Data. Version 0.2, http://erzuli.ss.uci.edu/R.stuff.

Butts CT (2007). **sna**: Tools for Social Network Analysis. Version 1.5, http://erzuli.ss.uci.edu/R.stuff.

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 CT, with help~from David~Hunter, Handcock MS (2007). **network**: Classes for Relational Data. Version 1.2, http://erzuli.ss.uci.edu/R.stuff.

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 Linear Model Fits*

Description

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

Usage

```
## S3 method for class 'ergm'
anova(object, ..., 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(1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,3,3,3,3)
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)
```

---

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)
```

---

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)
```

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.burnin=10000,
             MCMC.interval=100,
             MCMC.samplesize=10000,
             MCMC.return.stats=TRUE,
             MCMC.burnin.retries=0,
             MCMC.burnin.check.last=1/2,
             MCMC.burnin.check.alpha=0.01,
             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.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*10,
                         SAN.interval=MCMC.interval,
```

```
            SAN.packagenames=MCMC.packagenames,
            parallel=parallel,
            parallel.type=parallel.type,
            parallel.version.check=parallel.version.check),

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,
MCMLE.check.degeneracy=FALSE,
MCMLE.MCMC.precision=0.05,
MCMLE.metric=c("lognormal", "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=0.5,
MCMLE.adaptive.trustregion=3,
MCMLE.adaptive.epsilon=0.01,
MCMLE.sequential=TRUE,
MCMLE.density.guard.min=10000,
MCMLE.density.guard=exp(3),


SA.phase1_n=NULL,
SA.initial_gain=NULL,
SA.nsubphases=MCMLE.maxit,
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,

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 for finding the starting coefficient values, if init is not specified, is maximum pseudo-likelihood estimation (MPLE). Another valid value is "zeros" for a vector of 0 of appropriate length.
main.method	One of "MCMLE","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.
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`
: 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.samplesize`
: Number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm.

`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.burnin.retries`
: Maximum number of times to rerun the burn-in phase if a failure to converge is detected. Defaults to 0 for no checks.

`MCMC.burnin.check.last`
: What fraction at the end of the burn-in phase to use for detecting non-convergence. Defaults to one half.

`MCMC.burnin.check.alpha`
: Maximum Bonferroni-adjusted P-value under the Geweke test for a statistic to be considered unconverged. Defaults to 0.01.

`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`
: Not yet documented.

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

`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`
: After every MCMC sample, a Hotelling's T^2 test for equality of MCMC-simulated network statistics to observed is conducted, and if its P-value exceeds this setting, the estimation is considered to have converged and finishes. To turn this off and perform all `control$MCMLE.maxit` iterations, set `MCMLE.conv.min.pval=1`.

`MCMLE.NR.maxit`
: Maximum number of iterations in the Newton-Raphson optimization.

`MCMLE.NR.reltol`
: Not yet documented.

`obs.MCMC.samplesize,obs.MCMC.burnin,obs.MCMC.interval`
: 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`
: Vector of upper bounds on the precisions of the standard errors induced by the MCMC algorithm.

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

`MCMLE.method`
: Name of the optimization method to use. See `optim` for the options. The default method `"BFGS"` is a quasi-Newton method (also known as a variable metric algorithm). It is attributed to Broyden, Fletcher, Goldfarb and Shanno. This uses function values and gradients to build up a picture of the surface to be optimized.

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

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

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

`MCMLE.adaptive.epsilon`
: convergence tolerance: If the change in loglikelihood value when `MCMLE.steplength="adaptive"` is smaller than this tolerance (and the adaptive steplength equals 1), stop even if we haven't yet reached `MCMLE.maxit` iterations.

`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.

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

`SA.trustregion` Not yet documented.

`RM.phase1n_base`
: The Robbins-Monro control parameters are not yet documented.

`RM.phase2n_base`
: Not yet documented.

`RM.phase2sub` Not yet documented.

RM.init_gain	Not yet documented.
RM.phase3n	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.
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 "SOCK". Defaults to using the snow package default.
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, to appear.

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)
```

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`

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=1000,
                    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 `"SOCK"`. Defaults to using the `snow` package default.

`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)
```

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`

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.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.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 `"SOCK"`. Defaults to using the `snow` package default.

`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=1000,
                 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=1000,
                         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=1000,
                              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 `"SOCK"`. Defaults to using the `snow` package default.

`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`.

---

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 linear exponential 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 either a maximum pseudo-likelihood estimate or an approximate maximum likelihood estimator based on a Monte Carlo scheme.

Usage

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

`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?

`mc.se` MCMC standard error estimates

`newnetwork` The final network at the end of the MCMC simulation

`network` Original network

`coef.init` The initial value of θ.

`initial` The MPLE fit as a `glm` object.

`null.deviance` Deviance 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.

`etamap` The set of functions mapping the true parameter theta to the canonical parameter eta (irrelevant except in a curved exponential family model)

`degeneracy.value`
Score calculated to assess the degree of degeneracy in the model.

`degeneracy.type`
Supporting output for `degeneracy.value`. Mainly for internal use.

`formula` The original `formula` entered into the `ergm` function.

`constraints` Constraints used by original `ergm` call

`control` The control list passed to the call.

`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`.

`coef.hist, stats.hist, stats.obs.hist`
For the MCMLE method, the history of coefficients and resulting average statistics.

`estimable` A logical vector indicating which terms could not be estimated due to a `constraints` constraint fixing that term at a constant value.

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 (2006). **netperm**: Permutation Models for Relational Data. Version 0.2, http://erzuli.ss.uci.edu/R.stuff.

Butts CT (2007). **sna**: Tools for Social Network Analysis. Version 1.5, http://erzuli.ss.uci.edu/R.stuff.

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 CT, with help~from David~Hunter, Handcock MS (2007). **network**: Classes for Relational Data. Version 1.2, http://erzuli.ss.uci.edu/R.stuff.

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 linear exponential 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 on the network. Only dyads (i, j) for which `attrname(i)==attrname(j)` can have edges.

`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.

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

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 snow to facilitate this, and supports all cluster types that it does.

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

This entry describes common problems and workarounds associated with particular parallel processing APIs.

MPI

To use MPI to accellerate ERGM sampling, pass the control parameter parallel.type="MPI". ergm and snow use Rmpi package to communicate with an MPI cluster. ergm will check if an MPI cluster already exists and will create one if one doesn't.

On some installations, the function stopCluster does not work properly for MPI clusters. Because ergm creates a cluster every time it needs an MCMC sample and disbands it once it finishes, using

MPI on these installations may fail. A workaround is to start the cluster outside of ergm (e.g., `dummy <- makeCluster(nnodes, type="MPI")`). ergm will notice the preexisting cluster and make use of it, but it will not stop it.

Examples

```
# See help(ergm) for a description of this model.
data(florentine)
gest <- ergm(flomarriage ~ kstar(1:2) + absdiff("wealth") + triangles,
             eval.loglik=FALSE,
             control=control.ergm(parallel=2, parallel.type="SOCK"))
summary(gest)
# Note the combined MCMC diagnostics:
mcmc.diagnostics(gest)
```

---

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 $\mathsf{a}, \mathsf{a}+1, \ldots, \mathsf{b}-1, \mathsf{b}$. 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 $[\mathsf{a}, \mathsf{b}]$. At this time, both a and b must be finite.

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.)

Terms to represent network statistics included in the ergm pacakge

`absdiff(attrname, pow=1)` **(binary),** `absdiff(attrname, pow=1, form ="sum")` **(valued)** *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),** `absdiffcat(attrname, base=NULL, form="sum")` **(valued)** *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)** *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)** *Asymmetric dyads:* This term adds one network statistic to the model equal to the number of pairs of actors for which exactly one of $(i \to j)$ or $(j \to i)$ 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)** *Number of ties with values greater than or equal to a threshold* Adds one statistic equaling to the number of ties whose values equal or exceed `threshold`.

`b1concurrent(by=NULL)` **(binary)** *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. This term can only be used with undirected bipartite networks.

`b1degrange(from, to=+Inf, by=NULL, homophily=FALSE)` **(binary)** *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)** *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)** *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.

`b1star(k, attrname=NULL)` **(binary)** *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)** *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)** *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)** *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. This term can only be used with undirected bipartite networks.

`b2degrange(from, to=+Inf, by=NULL, homophily=FALSE)` **(binary)** *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)** *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)** *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.

`b2star(k, attrname=NULL)` **(binary)** *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 $\{O_1, \ldots, O_k\}$ 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 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)** *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)** *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)** *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)** *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)** *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)** *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), a.k.a.** `ctriad` *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 $\{(i \to j), (j \to k), (k \to i)\}$. 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)** *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),** `cyclicalties(threshold=0)` **(valued)** *Cyclical ties:* This term adds one statistic, equal to the number of ties $i \to j$ 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 $i \to j$ 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)** *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.

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

`dsp(d)` **(binary)** *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)** *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),** `edgecov(x, attrname=NULL, form="sum")` **(valued)** *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 or valued), a.k.a** `nonzero` **(valued)** *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)** *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).

`greaterthan(threshold=0)` **(valued)** *Number of dyads with values strictly greater than a threshold:* Adds one statistic equaling to the number of ties whose values exceed `threshold`.

`gwb1degree(decay, fixed=FALSE, cutoff=30)` **(binary)** *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)** *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)** *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)** *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)** *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)** *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)** *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)** *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)** *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)** *Hamming distance within mixing:* This term adds one statistic to the model for every possible pairing of attribute values of the network. 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)** *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)** *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)** *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)** *Number of ties whose values are in an interval* Adds one statistic equaling to the number of ties 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)** *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)** *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)** *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 $\{O_1, \ldots, O_k\}$ such that the ties $(O_j \rightarrow N)$ exist for $j = 1, \ldots, k$. The optional argument attrname is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is 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)** *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 $\{O_1, \ldots, O_k\}$ 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 where all nodes have the same value of the attribute. This term can only be used for undirected networks; for directed networks, see istar, ostar, twopath and m2star. Note that kstar(1) is equal to edges.

`localtriangle(x)` **(binary)** *Triangles within neighborhoods:* This term adds one statistic to the model equal to the number of triangles in the network between nodes "close to" each other. For an undirected network, a local triangle is defined to be any set of three edges between nodal pairs $\{(i,j),(j,k),(k,i)\}$ that are in the same neighborhood. For a directed network, a triangle is defined as any set of three edges $(i{\rightarrow}j),(j{\rightarrow}k)$ and either $(k{\rightarrow}i)$ or $(k{\leftarrow}i)$ where again all nodes are within the same neighborhood. The argument `x` is 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)** *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 $(i{\rightarrow}j),(j{\rightarrow}k)$. A mixed 2-star is sometimes called a 2-path because it is a directed path of length 2 from i to k via j. 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)** *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),** `mutual(form="min",threshold=0)` **(valued)** *Mutuality:* In binary ERGMs, equal to the number of pairs of actors i and j for which $(i{\rightarrow}j)$ and $(j{\rightarrow}i)$ 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)** *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)` **(binary),** `nodecov(attrname, form="sum")` **(valued), a.k.a.** `nodemain` *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)** *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),** `nodefactor(attrname, base=1, form="sum")` **(valued)** *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)` **(binary),** `nodeicov(attrname, form="sum")` **(valued)** *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)** *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),** `nodeifactor(attrname, base=1, form="sum")` **(valued)** *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)** *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),** `nodematch(attrname, diff=FALSE, keep=NULL, form="`
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),** `nodemix(attrname, base=NULL, form="sum")` **(valued)**
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)` **(binary),** `nodeocov(attrname, form="sum")` **(valued)** *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)** *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),** `nodeofactor(attrname, base=1, form="sum")` **(valued)**
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)** *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)** *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)** *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)** *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)** *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)** *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)** *"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)** *k-Outstars:* The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The ith such statistic counts the number of distinct `k[i]`-outstars in the network, where a k-outstar is defined to be a node N and a set of k different nodes $\{O_1, \ldots, O_k\}$ such that the ties $(N \rightarrow O_j)$ exist for $j = 1, \ldots, k$. The optional argument `attrname` is a character string giving the name of an attribute in the network's vertex attribute list. If this is specified then the count is the number of k-outstars where all nodes have the same value of the attribute. This term can only be used with directed networks; for undirected networks see `kstar`. Note that `ostar(1)` is equal to both `istar(1)` and `edges`.

`receiver(base=1)` **(binary)** *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)** *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)** *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)** *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)** *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 degree:* This term adds one network statistic for each node equal to the number of ties of that node. The optional `attrname` argu-

ment 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)** *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.

`threepath(keep=1:4)` **(binary)** *Three-paths:* For an undirected network, this term adds one statistic equal to the number of threepaths, where a threepath is defined as a path of length three that traverses three distinct edges. Note that a threepath need not include four distinct nodes; in particular, a triangle counts as three threepaths. 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 threepath is of the form $i \to j \to k \to l$, and RRL threepath is of the form $i \to j \to k \leftarrow l$, etc. Like in the undirected case, there is no requirement that the nodes be distinct in a directed threepath. However, the three edges must all be distinct. Thus, a mutual tie $i \leftrightarrow j$ does not count as a threepath of the form $i \to j \to i \leftarrow j$; however, in the subnetwork $i \leftrightarrow j \to k$, there are two directed threepaths, one LRR ($k \leftarrow j \to i \leftarrow j$) and one RRR ($j \to i \to j \leftarrow k$).

`transitive` **(binary)** *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),** `transitiveties(threshold=0)` **(valued)** *Transitive ties:* This term adds one statistic, equal to the number of ties $i \to j$ 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 $i \to j$ 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)** *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 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, 030T, 030C, 201, 120D, 120U, 120C,` 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)** *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)** *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), a.k.a.** `ttriad` *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)** *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

- 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
- 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, 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.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

This function 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.

Usage

```
ergm.bridge.llr(object,
                response=NULL,
                constraints=~.,
                from,
                to,
                basis=NULL,
                verbose=FALSE,
                ...,
                llronly=FALSE,
                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 simulate.formula.ergm.
llronly	Logical: If TRUE, only the estiamted log-ratio will be returned.)
control	Control arguments. See control.ergm.bridge for details.

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.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)
```

`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)
```

---

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.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

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

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

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

`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 (2006). **netperm**: Permutation Models for Relational Data. Version 0.2, http://erzuli.ss.uci.edu/R.stuff.

Butts CT (2007). **sna**: Tools for Social Network Analysis. Version 1.5, http://erzuli.ss.uci.edu/R.stuff.

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 CT, with help~from David~Hunter, Handcock MS (2007). **network**: Classes for Relational Data. Version 1.2, http://erzuli.ss.uci.edu/R.stuff.

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(),
                      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).
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.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.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,
                                    ...)
```

Arguments

`object`	An `ergm` object or an ERGM formula.
`response, basis`	See `ergm`.
`...`	Unused at this time.

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`.

Usage

```
is.inCH(p, M)
```

Arguments

`p`	A d-dimensional vector
`M`	An r by d matrix. Each row of `M` is a d-dimensional vector.

Details

This function depends on the package `Rglpk` to solve a constrained linear optimization problem in order to determine an answer. The question of whether `p` is in a closed convex set `S` may be formulated as the question of whether there exists a separating hyperplane between `p` and `S`, which may in turn be formulated as the question of whether the maximum possible value of a linear function, subject to constraints, has a strictly positive solution.

Note that the answer given could be incorrect simply due to rounding error if the true maximum is close to zero. For this reason, the package `rcdd`, which produces exact rational-number solutions to linear programs, could be used instead of `Rglpk`. However, this approach would require more computing and would therefore be slower.

Value

Logical, telling whether `p` is in the closed convex hull of the points in `M`.

References

http://www.cs.mcgill.ca/~fukuda/soft/polyfaq/node22.html

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)

## End(Not run)
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.
## Not run:
# The default number of bridges is currently 20. We'll only use 3 here
# for speed's sake.
gest <- logLik(gest, add=TRUE)

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

---

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,
                                  curved=TRUE,
                                  vars.per.page=3,
                                  ...)
```

Arguments

object	An ergm object. See documentation for ergm.
center	Logical: If TRUE, ; center the samples on the observed statistics.
curved	Logical: If TRUE, summarize the curved statistics (evaluated at the MLE of any non-linear parameters), rather than the raw components of the curved statistics.
vars.per.page	Number of rows (one variable per row) per plotting page. Ignored if latticeExtra package is not installed.
...	Additional arguments, to be passed to plotting functions.

Details

The plots produced are a trace of the sampled output and a density estimate for each variable in the chain. The diagnostics printed include correlations and convergence diagnostics.

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

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

Value

mcmc.diagnostics.ergm returns 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

```
#
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()
#
```

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

Description

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

Usage

```
data(molecule)
```

See Also

florentine, sampson, network, plot.network, ergm

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

Description

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

Usage

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

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.

Value

network.update returns a network object.

See Also

ergm, network

Examples

```
#
data(florentine)
#
# test the network.update function
#
# Create a Bernoulli network
rand.net <- network(network.size(flomarriage))
# store the sociomatrix
rand.mat <- rand.net[,]
# Update the network
network.update(flomarriage, rand.mat, 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")
```

---

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

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 "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

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

The data were gathered at three times to capture changes in group sentiment over time. They represent three time points in the period during which a new cohort entered the monastery near the end of the study but before the major conflict began.

Each member ranked only his top three choices on "liking."

(Some subjects offered tied ranks for their top four choices). A tie from monk A to monk B exists if A nominated B as one of his three best friends at that that time point.

`samplike` is the time-aggregated graph.

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

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.

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.

---

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,
                         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,
                       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.
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$parallel>0 the returned networks are "interleaved", in the sense that for y[i,j] is the jth network from MCMC chain i, the sequence returned if control$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,
                eps = 1e-04)
```

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.
eps	number; indicates the smallest p-value. See printCoefmat.
...	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
```

---

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