

Package ‘ergm’

June 10, 2019

Version 3.10.4

Date 2019-06-10

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

Depends network (\geq 1.15)

Imports robustbase (\geq 0.93.5), coda (\geq 0.19.2), trust (\geq 0.1.7), Matrix (\geq 1.2.17), lpSolve (\geq 5.6.13), parallel, methods, MASS (\geq 7.3.51.4), statnet.common (\geq 4.3.0), purrr (\geq 0.3.2), rlang (\geq 0.3.4), tibble (\geq 2.1.1), dplyr (\geq 0.8.0.1)

Suggests lattice (\geq 0.20.38), latticeExtra (\geq 0.6.28), sna (\geq 2.4), latentnet (\geq 2.9.0), rmarkdown (\geq 1.12), knitr (\geq 1.22), Rmpi (\geq 0.6.9), testthat (\geq 2.1.1), tergm, ergm.count, ergm.userterms, networkDynamic

SystemRequirements OpenMPI

BugReports <https://github.com/statnet/ergm/issues>

Description

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

License GPL-3 + file LICENSE

URL <https://statnet.org>

VignetteBuilder rmarkdown, knitr

RoxygenNote 6.1.1

Encoding UTF-8

Collate 'InitErgm.bipartite.R' 'InitErgmConstraint.R'
'InitErgmProposal.R' 'InitErgmProposal.blockdiag.R'
'InitErgmReference.R' 'ergm-deprecated.R' 'InitErgmTerm.R'
'InitErgmTerm.bipartite.degree.R' 'InitErgmTerm.coincidence.R'
'InitErgmTerm.dgw_sp.R' 'InitErgmTerm.extra.R'
'InitErgmTerm.indices.R' 'InitErgmTerm.test.R'
'InitErgmTerm.transitiveties.R' 'InitWtErgmProposal.R'
'InitWtErgmTerm.R' 'anova.ergm.R' 'anova.ergmlist.R'

'approx.hotelling.diff.test.R' 'as.network.numeric.R'
 'build_term_index.R' 'check.ErgmTerm.R' 'control.ergm.R'
 'control.ergm.bridge.R' 'control.gof.R' 'control.logLik.ergm.R'
 'control.san.R' 'control.simulate.R' 'data.R' 'ergm-defunct.R'
 'ergm-disambiguation.R' 'ergm.CD.fixed.R' 'ergm.Cprepare.R'
 'ergm.MCMCse.R' 'ergm.MCMCse.lognormal.R' 'ergm.MCMLE.R'
 'ergm.R' 'ergm.allstats.R' 'ergm.bounddeg.R' 'ergm.bridge.R'
 'ergm.check.R' 'ergm.coefficient.degeneracy.R'
 'ergm.curved.statsmatrix.R' 'ergm.degeneracy.R' 'ergm.design.R'
 'ergm.errors.R' 'ergm.estimate.R' 'ergm.eta.R' 'ergm.etagrad.R'
 'ergm.etagradmult.R' 'ergm.etamap.R' 'ergm.geodistn.R'
 'ergm.getCDsample.R' 'ergm.getMCMCsample.R' 'ergm.getnetwork.R'
 'ergm.initialfit.R' 'ergm.llik.R' 'ergm.llik.obs.R'
 'ergm.logitreg.R' 'ergm.mapl.R' 'ergm.maple.R' 'ergm.mple.R'
 'ergm.pen.glm.R' 'ergm.phase12.R' 'ergm.pl.R'
 'ergm.reviseinit.R' 'ergm.robmon.R' 'ergm.san.R'
 'ergm.steps.R' 'ergm.stocapprox.R' 'ergm.sufftoprob.R'
 'ergm.utility.R' 'ergmMPLE.R' 'ergm_estfun.R' 'ergm_model.R'
 'ergm_model.utils.R' 'ergm_proposal.R' 'formula.utils.R'
 'get.node.attr.R' 'godfather.R' 'gof.ergm.R' 'is.curved.R'
 'is.durational.R' 'is.dyad.independent.R' 'is.inCH.R'
 'locator.R' 'logLik.ergm.R' 'mcmc.diagnostics.ergm.R'
 'network.list.R' 'network.update.R' 'nparam.R'
 'parallel.utils.R' 'param_names.R' 'pending_update_network.R'
 'print.ergm.R' 'print.network.list.R' 'print.summary.ergm.R'
 'rlebdm.R' 'simulate.ergm.R' 'summary.ergm.R'
 'summary.ergm_model.R' 'summary.network.list.R'
 'summary.statistics.network.R' 'to_ergm_Cdouble.R'
 'vcov.ergm.R' 'wtd.median.R' 'zzz.R'

NeedsCompilation yes

Author Mark S. Handcock [aut],
 David R. Hunter [aut],
 Carter T. Butts [aut],
 Steven M. Goodreau [aut],
 Pavel N. Krivitsky [aut, cre] (<<https://orcid.org/0000-0002-9101-3362>>),
 Martina Morris [aut],
 Li Wang [ctb],
 Kirk Li [ctb],
 Skye Bender-deMoll [ctb],
 Chad Klumb [ctb]

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

Repository CRAN

Date/Publication 2019-06-10 05:30:07 UTC

R topics documented:

ergm-package	4
anova.ergm	6
approx.hotelling.diff.test	8
as.edgelist	9
as.network.numeric	10
check.ErgmTerm	11
control.ergm	13
control.ergm.bridge	22
control.ergm.godfather	23
control.gof	24
control.logLik.ergm	26
control.san	27
control.simulate.ergm	29
degreedist	31
ecoli	32
edges	33
enformulate.curved	33
ergm	35
ergm-constraints	42
ergm-parallel	46
ergm-references	48
ergm-terms	49
ergm.allstats	78
ergm.bounddeg	79
ergm.bridge.llr	81
ergm.degeneracy	83
ergm.exact	84
ergm.geodistdist	85
ergm.getnetwork	86
ergm.godfather	87
ergmMPLE	88
ergm_MCMC_sample	90
ergm_plot.mcmc.list	92
eut-upgrade	92
faux.desert.high	93
faux.dixon.high	95
faux.magnolia.high	96
faux.mesa.high	98
fix.curved	99
florentine	101
g4	102
Getting.Started	103
geweke.diag.mv	105
gof	106
hamming	108
is.curved	109

is.durational	110
is.dyad.independent	111
is.inCH	112
kapferer	113
logLik.ergm	114
logLikNull	115
mcmc.diagnostics	116
molecule	118
network.list	118
node-attr	119
nparam	121
param_names	122
print.summary.ergm	122
samplk	124
sampson	125
san	127
search.ergmTerms	130
simulate.ergm	131
spectrum0.mvar	136
summary.formula	136
update.network	137
wtd.median	139

Index**140**

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: <https://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, <https://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). <https://www.jstatsoft.org/v24/i07/>.
- Besag, J., 1974, Spatial interaction and the statistical analysis of lattice systems (with discussion), *Journal of the Royal Statistical Society, B*, 36, 192-236.
- Boer P, Huisman M, Snijders T, Zeggelink E (2003). StOCNET: an open software system for the advanced statistical analysis of social networks. Groningen: ProGAMMA / ICS, version 1.4 edition.
- Butts CT (2007). **sna**: Tools for Social Network Analysis. R package version 2.3-2. <https://cran.r-project.org/package=sna>
- Butts CT (2008). **network**: A Package for Managing Relational Data in R. *Journal of Statistical Software*, 24(2). <https://www.jstatsoft.org/v24/i02/>.
- Butts C (2015). **network**: Classes for Relational Data. The Statnet Project (<https://statnet.org>). R package version 1.12.0, <https://cran.r-project.org/package=network>.
- Frank, O., and Strauss, D.(1986). Markov graphs. *Journal of the American Statistical Association*, 81, 832-842.

- Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008a). A **statnet** Tutorial. *Journal of Statistical Software*, 24(8). <https://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. <https://www.csss.washington.edu/Papers/wp39.pdf>
- Handcock MS (2003b). **degrenet**: Models for Skewed Count Distributions Relevant to Networks. Statnet Project, Seattle, WA. Version 1.0, <https://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, <https://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, <https://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). <https://www.jstatsoft.org/v24/i03/>.
- Krivitsky PN, Handcock MS (2007). **latentnet**: Latent position and cluster models for statistical networks. Seattle, WA. Version 2, <https://statnet.org>.
- Krivitsky PN (2012). Exponential-Family Random Graph Models for Valued Networks. *Electronic Journal of Statistics*, 2012, 6, 1100-1128. doi: [10.1214/12EJS696](https://doi.org/10.1214/12EJS696)
- Morris M, Handcock MS, Hunter DR (2008). Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 24(4). <https://www.jstatsoft.org/v24/i04/>.
- Strauss, D., and Ikeda, M.(1990). Pseudolikelihood estimation for social networks *Journal of the American Statistical Association*, 85, 204-212.

 anova.ergm

 ANOVA for ERGM Fits

Description

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

Usage

```
## S3 method for class 'ergm'
anova(object, ..., eval.loglik = FALSE)
```

```
## S3 method for class 'ergm'
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.
scale	numeric. An estimate of the noise variance σ^2 . If zero this will be estimated from the largest model considered.
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.

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 's default of `na.action = na.omit` is used, and [anova.ergm](#) 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)
fit0 <- ergm(molecule ~ edges)
anova(fit0)
fit1 <- ergm(molecule ~ edges + nodefactor("atomic type"))
anova(fit1)

fit2 <- ergm(molecule ~ edges + nodefactor("atomic type") + gwesp(0.5,
  fixed=TRUE), eval.loglik=TRUE) # Note the eval.loglik argument.
anova(fit0, fit1)
anova(fit0, fit1, fit2)

```

approx.hotelling.diff.test

Approximate Hotelling T^2 -Test for One Sample Means

Description

A multivariate hypothesis test for a single population mean or a difference between them. This version attempts to adjust for multivariate autocorrelation in the samples.

Usage

```

approx.hotelling.diff.test(x, y = NULL, mu0 = 0,
  assume.indep = FALSE, var.equal = FALSE)

```

Arguments

x	a numeric matrix of data values with cases in rows and variables in columns.
y	an optional matrix of data values with cases in rows and variables in columns for a 2-sample test.
mu0	an optional numeric vector: for a 1-sample test, the population mean under the null hypothesis; and for a 2-sample test, the difference between population means under the null hypothesis; defaults to a vector of 0s.
assume.indep	if TRUE, performs an ordinary Hotelling's test without attempting to account for autocorrelation.
var.equal	for a 2-sample test, perform the pooled test: assume population variance-covariance matrices of the two variables are equal.

Value

An object of class `htest` with the following information:

<code>statistic</code>	The T^2 statistic.
<code>parameter</code>	Degrees of freedom.
<code>p.value</code>	P-value.
<code>method</code>	Method specifics.
<code>null.value</code>	Null hypothesis mean or mean difference.
<code>alternative</code>	Always "two.sided".
<code>estimate</code>	Sample difference.
<code>covariance</code>	Estimated variance-covariance matrix of the estimate of the difference.

It has a print method `print.htest()`.

Note

For `mcmc.list` input, the variance for this test is estimated with unpooled means. This is not strictly correct.

References

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

See Also

`t.test()`

as.edgelist

Convert a network object into a numeric edgelist matrix

Description

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

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

Details

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

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

Note

The `as.edgelist` functions have been moved to the network package. See [as.edgelist](#)

See Also

See also [as.edgelist](#), [as.matrix.network.edgelist](#)

Examples

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

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

Description

[as.network.numeric](#) creates a random Bernoulli network of the given size as an object of class [network](#).

Usage

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

Arguments

<code>x</code>	count; the number of nodes in the network. If <code>bipartite=TRUE</code> , it is the number of events in the network.
<code>directed</code>	logical; should edges be interpreted as directed?
<code>hyper</code>	logical; are hyperedges allowed? Currently ignored.
<code>loops</code>	logical; should loops be allowed? Currently ignored.
<code>multiple</code>	logical; are multiplex edges allowed? Currently ignored.
<code>bipartite</code>	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.
<code>ignore.eval</code>	logical; ignore edge values? Currently ignored.
<code>names.eval</code>	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 <code>init</code> 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 not have vertex, edge or network attributes. These can be added with operators such as `%v%`, `%n%`, `%e%`.

Value

An object of class `network`

References

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

See Also

`network`

Examples

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

check.ErgmTerm	<i>Ensures an Ergm Term and its Arguments Meet Appropriate Conditions</i>
----------------	---

Description

Helper functions for implementing `ergm()` terms, to check whether the term can be used with the specified network. For information on ergm terms, see [ergm-terms](#). `ergm.checkargs`, `ergm.checkbipartite`, and `ergm.checkdirected` are helper functions for an old API and are deprecated. Use `check.ErgmTerm`.

Usage

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

Arguments

<code>nw</code>	the network that term X is being checked against
<code>arglist</code>	the list of arguments for term X
<code>directed</code>	logical, whether term X requires a directed network; default=NULL
<code>bipartite</code>	whether term X requires a bipartite network (T or F); default=NULL
<code>nonnegative</code>	whether term X requires a network with only nonnegative weights; default=FALSE
<code>varnames</code>	the vector of names of the possible arguments for term X; default=NULL
<code>vartypes</code>	the vector of types of the possible arguments for term X; default=NULL
<code>defaultvalues</code>	the list of default values for the possible arguments of term X; default=list()
<code>required</code>	the logical vector of whether each possible argument is required; default=NULL
<code>response</code>	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.
<code>dep.inform, dep.warn</code>	a list of length equal to the number of arguments the term can take; if the corresponding element of the list is not FALSE, a <code>message()</code> or a <code>warning()</code> respectively will be issued if the user tries to pass it; if the element is a character string, it will be used as a suggestion for replacement.

Details

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

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

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

Value

A list of the values for each possible argument of term X; user provided values are used when given, default values otherwise. The list also has an `attr(,"missing")` attribute containing a named logical vector indicating whether a particular argument had been set to its default.

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,
  checkpoint = NULL, resume = NULL, MPLE.max.dyad.types = 1e+06,
  MPLE.samplesize = 50000, MPLE.type = c("glm", "penalized"),
  MCMC.prop.weights = "default", MCMC.prop.args = list(),
  MCMC.interval = 1024, MCMC.burnin = MCMC.interval * 16,
  MCMC.samplesize = 1024, MCMC.effectiveSize = NULL,
  MCMC.effectiveSize.damp = 10, MCMC.effectiveSize.maxruns = 1000,
  MCMC.effectiveSize.base = 1/2, MCMC.effectiveSize.points = 5,
  MCMC.effectiveSize.order = 1, MCMC.return.stats = TRUE,
  MCMC.runtime.traceplot = FALSE, MCMC.init.maxedges = 20000,
  MCMC.max.maxedges = Inf, MCMC.addto.se = TRUE,
  MCMC.compress = FALSE, MCMC.packagenames = c(), SAN.maxit = 4,
  SAN.nsteps.times = 8, SAN.control = control.san(term.options =
  term.options, SAN.maxit = SAN.maxit, SAN.prop.weights =
  MCMC.prop.weights, SAN.prop.args = MCMC.prop.args, SAN.init.maxedges =
  MCMC.init.maxedges, SAN.max.maxedges = MCMC.max.maxedges, SAN.nsteps =
  MCMC.burnin * SAN.nsteps.times, SAN.samplesize = MCMC.samplesize,
  SAN.packagenames = MCMC.packagenames, parallel = parallel, parallel.type
  = parallel.type, parallel.version.check = parallel.version.check),
  MCMLE.termination = c("Hummel", "Hotelling", "precision", "none"),
  MCMLE.maxit = 20, MCMLE.conv.min.pval = 0.5, MCMLE.NR.maxit = 100,
  MCMLE.NR.reltol = sqrt(.Machine$double.eps),
  obs.MCMC.samplesize = MCMC.samplesize,
  obs.MCMC.interval = MCMC.interval, obs.MCMC.burnin = MCMC.burnin,
  obs.MCMC.burnin.min = obs.MCMC.burnin/10,
  obs.MCMC.prop.weights = MCMC.prop.weights,
  obs.MCMC.prop.args = MCMC.prop.args,
  obs.MCMC.impute.min_informative = function(nw) network.size(nw)/4,
  obs.MCMC.impute.default_density = function(nw) 2/network.size(nw),
  MCMLE.check.degeneracy = FALSE, MCMLE.MCMC.precision = 0.005,
  MCMLE.MCMC.max.ESS.frac = 0.1, MCMLE.metric = c("lognormal",
  "logtaylor", "Median.Likelihood", "EF.Likelihood", "naive"),
  MCMLE.method = c("BFGS", "Nelder-Mead"), MCMLE.trustregion = 20,
  MCMLE.dampening = FALSE, MCMLE.dampening.min.ess = 20,
  MCMLE.dampening.level = 0.1, MCMLE.steplength.margin = 0.05,
  MCMLE.steplength = NVL2(MCMLE.steplength.margin, 1, 0.5),
```

```

MCMLE.adaptive.trustregion = 3, MCMLE.sequential = TRUE,
MCMLE.density.guard.min = 10000, MCMLE.density.guard = exp(3),
MCMLE.effectiveSize = NULL, MCMLE.last.boost = 4,
MCMLE.Hummel.esteq = TRUE, MCMLE.Hummel.miss.sample = 100,
MCMLE.Hummel.maxit = 25, MCMLE.steplength.min = 1e-04,
MCMLE.effectiveSize.interval_drop = 2,
MCMLE.save_intermediates = NULL, SA.phase1_n = NULL,
SA.initial_gain = NULL, SA.nsubphases = 4, SA.niterations = NULL,
SA.phase3_n = NULL, SA.trustregion = 0.5, RM.phase1n_base = 7,
RM.phase2n_base = 100, RM.phase2sub = 7, RM.init_gain = 0.5,
RM.phase3n = 500, Step.MCMC.samplesize = 100, Step.maxit = 50,
Step.gridsize = 100, CD.nsteps = 8, CD.multiplicity = 1,
CD.nsteps.obs = 128, CD.multiplicity.obs = 1, CD.maxit = 60,
CD.conv.min.pval = 0.5, CD.NR.maxit = 100,
CD.NR.reltol = sqrt(.Machine$double.eps), CD.metric = c("naive",
"lognormal", "logtaylor", "Median.Likelihood", "EF.Likelihood"),
CD.method = c("BFGS", "Nelder-Mead"), CD.trustregion = 20,
CD.dampening = FALSE, CD.dampening.min.ess = 20,
CD.dampening.level = 0.1, CD.steplength.margin = 0.5,
CD.steplength = 1, CD.adaptive.trustregion = 3,
CD.adaptive.epsilon = 0.01, CD.Hummel.esteq = TRUE,
CD.Hummel.miss.sample = 100, CD.Hummel.maxit = 25,
CD.steplength.min = 1e-04, loglik.control = control.logLik.ergm(),
term.options = NULL, 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 <code>control\$init.method</code> . If a numeric vector is given, the elements of the vector are interpreted as follows: <ul style="list-style-type: none"> • Elements corresponding to terms enclosed in <code>offset()</code> are used as the fixed offset coefficients. Note that offset coefficients alone can be more conveniently specified using <code>ergm()</code> argument <code>offset.coef</code>. If both <code>offset.coef</code> and <code>init</code> arguments are given, values in <code>offset.coef</code> 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 <code>control\$init.method</code>. |

Passing `control.ergm(init=coef(prev.fit))` can be used to “resume” an uncovered `ergm()` run, but see `enformulate.curved`.

init.method	A character vector or NULL. The default method depends on the reference measure used. For the binary ("Bernoulli") ERGMs, with dyad-independent constraints, it's maximum pseudo-likelihood estimation (MPLE). Other valid values include "zeros" for a 0 vector of appropriate length and "CD" for contrastive divergence. If passed explicitly, this setting overrides the reference's limitations. Valid initial methods for a given reference are set by the <code>InitErgmReference.*</code> function.
main.method	One of "MCMLE" (default), "Robbins-Monro", "Stochastic-Approximation", or "Stepping". Chooses the estimation method used to find the MLE. MCMLE attempts to maximize an approximation to the log-likelihood function. Robbins-Monro and Stochastic-Approximation are both stochastic approximation algorithms that try to solve the method of moments equation that yields the MLE in the case of an exponential family model. Another alternative is a partial stepping algorithm (Stepping) as in Hummel et al. (2012). The direct use of the likelihood function has many theoretical advantages over stochastic approximation, but the choice will depend on the model and data being fit. See Handcock (2000) and Hunter and Handcock (2006) for details. Note that in recent versions of ERGM, the enhancements of Stepping have been folded into the default MCMLE, which is able to handle more modeling scenarios.
force.main	Logical: If TRUE, then force MCMC-based estimation method, even if the exact MLE can be computed via maximum pseudolikelihood estimation.
main.hessian	Logical: If TRUE, then an approximate Hessian matrix is used in the MCMC-based estimation method.
checkpoint	At the start of every iteration, save the state of the optimizer in a way that will allow it to be resumed. The name is passed through <code>sprintf()</code> with iteration number as the second argument. (For example, <code>checkpoint="step_%03d.RData"</code> will save to <code>step_001.RData</code> , <code>step_002.RData</code> , etc.)
resume	If given a file name of an RData file produced by checkpoint, the optimizer will attempt to resume after restoring the state. Control parameters from the saved state will be reused, except for those whose value passed via <code>control.ergm()</code> had change from the saved run. Note that if the network, the model, or some critical settings differ between runs, the results may be undefined.
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 <code>MPLE.max.dyad.types</code> .
MPLE.samplesize	Not currently documented; used in conditional-on-degree version of MPLE.
MPLE.type	One of "glm" or "penalized". Chooses method of calculating MPLE. "glm" is the usual formal logistic regression, whereas "penalized" uses the bias-reduced method of Firth (1993) as originally implemented by Meinhard Ploner, Daniela Dunkler, Harry Southworth, and Georg Heinze in the "logistf" package.
MCMC.prop.weights, obs.MCMC.prop.weights	Specifies the proposal distribution used in the MCMC Metropolis-Hastings algorithm. Possible choices depending on selected reference and constraints

arguments of the `ergm()` function, but often include "TNT" and "random", and the "default" is to use the one with the highest priority available.

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

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

<code>MCMC.prop.args</code> , <code>obs.MCMC.prop.args</code>	An alternative, direct way of specifying additional arguments to <code>proposal</code> . <code>obs.MCMC.prop.args</code> , if given separately, specifies the weights to be used for the constrained MCMC when missing dyads are present, defaulting to the same as <code>MCMC.prop.args</code> .
<code>MCMC.interval</code>	Number of proposals between sampled statistics. Increasing interval will reduce the autocorrelation in the sample, and may increase the precision in estimates by reducing MCMC error, at the expense of time. Set the interval higher for larger networks.
<code>MCMC.burnin</code>	Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.
<code>MCMC.samplesize</code>	Number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm. Increasing sample size may increase the precision in the estimates by reducing MCMC error, at the expense of time. Set it higher for larger networks, or when using parallel functionality.
<code>MCMC.return.stats</code>	Logical: If TRUE, return the matrix of MCMC-sampled network statistics. This matrix should have <code>MCMC.samplesize</code> rows. This matrix can be used directly by the coda package to assess MCMC convergence.
<code>MCMC.runtime.traceplot</code>	Logical: If TRUE, plot traceplots of the MCMC sample after every MCMC MLE iteration.
<code>MCMC.init.maxedges</code> , <code>MCMC.max.maxedges</code>	These parameters control how much space is allocated for storing edgelist for return at the end of MCMC sampling. Allocating more than needed wastes memory, so <code>MCMC.init.maxedges</code> is the initial amount allocated, but it will be incremented by a factor of 10 if exceeded during the simulation, up to <code>MCMC.max.maxedges</code> , at which point the process will stop with an error.
<code>MCMC.addto.se</code>	Whether to add the standard errors induced by the MCMC algorithm to the estimates' standard errors.
<code>MCMC.compress</code>	Logical: If TRUE, the matrix of sample statistics returned is compressed to the set of unique statistics with a column of frequencies post-pended.
<code>MCMC.packagenames</code>	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 <code>target.stats</code> argument is passed to <code>ergm()</code> , the maximum number of attempts to use <code>san</code> to obtain a network with statistics close to those specified.
SAN.nsteps.times	Multiplier for <code>SAN.nsteps</code> relative to <code>MCMC.burnin</code> . This lets one control the amount of SAN burn-in (arguably, the most important of SAN parameters) without overriding the other <code>SAN.control</code> defaults.
SAN.control	Control arguments to <code>san</code> . See <code>control.san</code> for details.
MCMLE.termination	<p>The criterion used for terminating MCMLE estimation:</p> <ul style="list-style-type: none"> • "Hummel" Terminate when the Hummel step length is 1 for two consecutive iterations. For the last iteration, the sample size is boosted by a factor of <code>MCMLE.last.boost</code>. See Hummel et. al. (2012). <p>Note that this criterion is incompatible with <code>MCMLE.steplength</code> \neq 1 or <code>MCMLE.steplength.margin</code> = NULL.</p> <ul style="list-style-type: none"> • "Hotelling" After every MCMC sample, an autocorrelation-adjusted Hotelling's T^2 test for equality of MCMC-simulated network statistics to observed is conducted, and if its P-value exceeds <code>MCMLE.conv.min.pval</code>, the estimation is considered to have converged and finishes. This was the default option in <code>ergm</code> version 3.1. • "precision" Terminate when the estimated loss in estimating precision due to using MCMC standard errors is below the precision bound specified by <code>MCMLE.MCMC.precision</code>, and the Hummel step length is 1 for two consecutive iterations. See <code>MCMLE.MCMC.precision</code> for details. This feature is in experimental status until we verify the coverage of the standard errors. <p>Note that this criterion is incompatible with <code>MCMLE.steplength</code> \neq 1 or <code>MCMLE.steplength.margin</code> = NULL.</p> <ul style="list-style-type: none"> • "none" Stop after <code>MCMLE.maxit</code> iterations.
MCMLE.maxit	Maximum number of times the parameter for the MCMC should be updated by maximizing the MCMC likelihood. At each step the parameter is changed to the values that maximizes the MCMC likelihood based on the current sample.
MCMLE.conv.min.pval	The P-value used in the Hotelling test for early termination.
MCMLE.NR.maxit, MCMLE.NR.reltol	The method, maximum number of iterations and relative tolerance to use within the <code>optim</code> routine in the MLE optimization. Note that by default, <code>ergm</code> uses <code>trust</code> , and falls back to <code>optim</code> when <code>trust</code> fails.
obs.MCMC.samplesize, obs.MCMC.burnin, obs.MCMC.interval, obs.MCMC.burnin.min	Sample size, burnin, and interval parameters for the MCMC sampling used when unobserved data are present in the estimation routine.
obs.MCMC.impute.min_informative, obs.MCMC.impute.default_density	Controls for imputation of missing dyads for initializing MCMC sampling. If numeric, <code>obs.MCMC.impute.min_informative</code> specifies the minimum number dyads that need to be non-missing before sample network density is used as the

imputation density. It can also be specified as a function that returns this value. `obs.MCMC.impute.default_density` similarly controls the imputation density when number of non-missing dyads is too low.

`MCMLE.check.degeneracy`

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

`MCMLE.MCMC.precision`, `MCMLE.MCMC.max.ESS.frac`

`MCMLE.MCMC.precision` is a vector of upper bounds on the standard errors induced by the MCMC algorithm, expressed as a percentage of the total standard error. The MCMLE algorithm will terminate when the MCMC standard errors are below the precision bound, and the Hummel step length is 1 for two consecutive iterations. This is an experimental feature.

If effective sample size is used (see `MCMC.effectiveSize`), then `ergm` may increase the target ESS to reduce the MCMC standard error.

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

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

`MCMLE.trustregion`

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

Note that not all metrics abide by it.

`MCMLE.dampening`

(logical) Should likelihood dampening be used?

`MCMLE.dampening.min.ess`

The effective sample size below which dampening is used.

`MCMLE.dampening.level`

The proportional distance from boundary of the convex hull move.

`MCMLE.steplength.margin`

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

`MCMLE.steplength`

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

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

`MCMLE.adaptive.trustregion`

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

- `MCMLE.sequential`
 Logical: If TRUE, the next iteration of the fit uses the last network sampled as the starting network. If FALSE, always use the initially passed network. The results should be similar (stochastically), but the TRUE option may help if the `target.stats` in the `ergm()` function are far from the initial network.
- `MCMLE.density.guard.min`, `MCMLE.density.guard`
 A simple heuristic to stop optimization if it finds itself in an overly dense region, which usually indicates ERGM degeneracy: if the sampler encounters a network configuration that has more than `MCMLE.density.guard.min` edges and whose number of edges exceeds the observed network by more than `MCMLE.density.guard`, the optimization process will be stopped with an error.
- `MCMLE.effectiveSize`, `MCMLE.effectiveSize.interval_drop`, `MCMC.effectiveSize`, `MCMC.effectiveSize.damp`
 Set `MCMLE.effectiveSize` to non-NULL value to adaptively determine the burn-in and the MCMC length needed to get the specified effective size using the method of Sahlin (2011); 50 is a reasonable value. This feature is in experimental status until we verify the coverage of the standard errors.
- `MCMLE.last.boost`
 For the Hummel termination criterion, increase the MCMC sample size of the last iteration by this factor.
- `MCMLE.Hummel.esteq`
 For curved ERGMs, should the estimating function values be used to compute the Hummel step length? This allows the Hummel stepping algorithm converge when some sufficient statistics are at 0.
- `MCMLE.Hummel.miss.sample`
 In fitting the missing data MLE, the rules for step length become more complicated. In short, it is necessary for *all* points in the constrained sample to be in the convex hull of the unconstrained (though they may be on the border); and it is necessary for their centroid to be in its interior. This requires checking a large number of points against whether they are in the convex hull, so to speed up the procedure, a sample is taken of the points most likely to be outside it. This parameter specifies the sample size.
- `MCMLE.Hummel.maxit`
 Maximum number of iterations in searching for the best step length.
- `MCMLE.steplength.min`
 Stops MCMLE estimation when the step length gets stuck below this minimum value.
- `MCMLE.save_intermediates`
 Every iteration, after MCMC sampling, save the MCMC sample and some miscellaneous information to a file with this name. This is mainly useful for diagnostics and debugging. The name is passed through `sprintf()` with iteration number as the second argument. (For example, `MCMLE.save_intermediates="step_%03d.RData"` will save to `step_001.RData`, `step_002.RData`, etc.)
- `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 <code>MCMLE.maxit</code> . See Snijders (2002) for details.
SA.iterations	Number of MCMC samples to draw in Phase 2 of the stochastic approximation algorithm. Defaults to 7 plus the number of terms in the model. See Snijders (2002) for details.
SA.phase3_n	Sample size for the MCMC sample in Phase 3 of the stochastic approximation algorithm. See Snijders (2002) for details.
SA.trustregion	The trust region parameter for the likelihood functions, used in the stochastic approximation algorithm.
RM.phase1n_base, RM.phase2n_base, RM.phase2sub, RM.init_gain, RM.phase3n	The Robbins-Monro control parameters are not yet documented.
Step.MCMC.samplesize	MCMC sample size for the preliminary steps of the "Stepping" method of optimization. This is usually chosen to be smaller than the final MCMC sample size (which equals <code>MCMC.samplesize</code>). See Hummel et al. (2012) for details.
Step.maxit	Maximum number of iterations (steps) allowed by the "Stepping" method.
Step.gridsize	Integer N such that the "Stepping" style of optimization chooses a step length equal to the largest possible multiple of $1/N$. See Hummel et al. (2012) for details.
CD.nsteps, CD.multiplicity	<p>Main settings for contrastive divergence to obtain initial values for the estimation: respectively, the number of Metropolis–Hastings steps to take before reverting to the starting value and the number of tentative proposals per step. Computational experiments indicate that increasing <code>CD.multiplicity</code> improves the estimate faster than increasing <code>CD.nsteps</code> — up to a point — but it also samples from the wrong distribution, in the sense that while as $CD.nsteps \rightarrow \infty$, the CD estimate approaches the MLE, this is not the case for <code>CD.multiplicity</code>.</p> <p>In practice, MPLE, when available, usually outperforms CD for even a very high <code>CD.nsteps</code> (which is, in turn, not very stable), so CD is useful primarily when MPLE is not available. This feature is to be considered experimental and in flux. The default values have been set experimentally, providing a reasonably stable, if not great, starting values.</p>
CD.nsteps.obs, CD.multiplicity.obs	When there are missing dyads, <code>CD.nsteps</code> and <code>CD.multiplicity</code> must be set to a relatively high value, as the network passed is not necessarily a good start for CD. Therefore, these settings are in effect if there are missing dyads in the observed network, using a higher default number of steps.
CD.maxit, CD.conv.min.pval, CD.NR.maxit, CD.NR.reltol,	<p>Miscellaneous tuning parameters of the CD sampler and optimizer. These have the same meaning as their MCMC.* counterparts.</p> <p>Note that only the Hotelling's stopping criterion is implemented for CD.</p>
CD.metric, CD.method, CD.trustregion, CD.dampening, CD.dampening.min.ess,	<p>Miscellaneous tuning parameters of the CD sampler and optimizer. These have the same meaning as their MCMC.* counterparts.</p> <p>Note that only the Hotelling's stopping criterion is implemented for CD.</p>

CD.dampening.level, CD.steplength.margin, CD.steplength, CD.adaptive.trustregion,	Miscellaneous tuning parameters of the CD sampler and optimizer. These have the same meaning as their MCMC.* counterparts. Note that only the Hotelling's stopping criterion is implemented for CD.
CD.adaptive.epsilon, CD.Hummel.esteq, CD.Hummel.miss.sample,	Miscellaneous tuning parameters of the CD sampler and optimizer. These have the same meaning as their MCMC.* counterparts. Note that only the Hotelling's stopping criterion is implemented for CD.
CD.Hummel.maxit, CD.steplength.min	Miscellaneous tuning parameters of the CD sampler and optimizer. These have the same meaning as their MCMC.* counterparts. Note that only the Hotelling's stopping criterion is implemented for CD.
loglik.control	See control.ergm.bridge
term.options	A list of additional arguments to be passed to term initializers. It can also be set globally via <code>option(ergm.term=list(...))</code> .
seed	Seed value (integer) for the random number generator. See set.seed .
parallel	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
parallel.type	API to use for parallel processing. Supported values are "MPI" and "PSOCK". Defaults to using the parallel package with PSOCK clusters. See ergm-parallel
parallel.version.check	Logical: If TRUE, check that the version of ergm running on the slave nodes is the same as that running on the master node.
...	Additional arguments, passed to other functions This argument is helpful because it collects any control parameters that have been deprecated; a warning message is printed in case of deprecated arguments.

Details

This function is only used within a call to the [ergm\(\)](#) function. See the usage section in [ergm\(\)](#) for details.

Value

A list with arguments as components.

References

- Snijders, T.A.B. (2002), Markov Chain Monte Carlo Estimation of Exponential Random Graph Models. *Journal of Social Structure*. Available from <https://www.cmu.edu/joss/content/articles/volume3/Snijders.pdf>.
- Firth (1993), Bias Reduction in Maximum Likelihood Estimates. *Biometrika*, 80: 27-38.
- Hunter, D. R. and M. S. Handcock (2006), Inference in curved exponential family models for networks. *Journal of Computational and Graphical Statistics*, 15: 565-583.
- Hummel, R. M., Hunter, D. R., and Handcock, M. S. (2012), Improving Simulation-Based Algorithms for Fitting ERGMs, *Journal of Computational and Graphical Statistics*, 21: 920-939.

- Kristoffer Sahlin. Estimating convergence of Markov chain Monte Carlo simulations. Master's Thesis. Stockholm University, 2011. <https://www2.math.su.se/matstat/reports/master/2011/rep2/report.pdf>

See Also

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

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

Description

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

Usage

```
control.ergm.bridge(nsteps = 20, MCMC.burnin = 10000,
  MCMC.interval = 100, MCMC.samplesize = 10000,
  obs.MCMC.samplesize = MCMC.samplesize,
  obs.MCMC.interval = MCMC.interval, obs.MCMC.burnin = MCMC.burnin,
  MCMC.prop.weights = "default", MCMC.prop.args = list(),
  MCMC.init.maxedges = 20000, MCMC.packagenames = c(),
  term.options = list(), seed = NULL, parallel = 0,
  parallel.type = NULL, parallel.version.check = TRUE)
```

Arguments

<code>nsteps</code>	Number of geometric bridges to use.
<code>MCMC.burnin</code>	Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.
<code>MCMC.interval</code>	Number of proposals between sampled statistics.
<code>MCMC.samplesize</code>	Number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm.
<code>obs.MCMC.burnin</code> , <code>obs.MCMC.interval</code> , <code>obs.MCMC.samplesize</code>	The obs versions of these arguments are for the unobserved data simulation algorithm.
<code>MCMC.prop.weights</code>	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 <code>constraints</code> argument of the <code>ergm</code> 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.
term.options	A list of additional arguments to be passed to term initializers. It can also be set globally via <code>option(ergm.term=list(...))</code> .
seed	Seed value (integer) for the random number generator. See set.seed .
parallel	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
parallel.type	API to use for parallel processing. Supported values are "MPI" and "PSOCK". Defaults to using the parallel package with PSOCK clusters. See ergm-parallel
parallel.version.check	Logical: If TRUE, check that the version of ergm running on the slave nodes is the same as that running on the master node.

Details

This function is only used within a call to the [ergm.bridge.llr](#) or [ergm.bridge.dindstart.llk](#) functions.

Value

A list with arguments as components.

See Also

[ergm.bridge.llr](#), [ergm.bridge.dindstart.llk](#)

control.ergm.godfather

Control parameters for [ergm.godfather\(\)](#).

Description

Returns a list of its arguments.

Usage

```
control.ergm.godfather(GF.init.maxedges.mul = 5, term.options = NULL)
```

Arguments

GF.init.maxedges.mul	How much space is allocated for the edgelist of the final network. It is used adaptively, so should not be greater than 10.
term.options	A list of additional arguments to be passed to term initializers. It can also be set globally via <code>option(ergm.term=list(...))</code> .

 control.gof

Auxiliary for Controlling ERGM Goodness-of-Fit Evaluation

Description

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

The `control.gof.ergm` version is intended to be used with `gof.ergm()` specifically and will "inherit" as many control parameters from `ergm` fit as possible().

Usage

```
control.gof.formula(nsim = 100, MCMC.burnin = 10000,
  MCMC.interval = 1000, MCMC.prop.weights = "default",
  MCMC.prop.args = list(), MCMC.init.maxedges = 20000,
  MCMC.packagenames = c(), MCMC.runtime.traceplot = FALSE,
  network.output = "network", seed = NULL, parallel = 0,
  parallel.type = NULL, parallel.version.check = TRUE)
```

```
control.gof.ergm(nsim = 100, MCMC.burnin = NULL,
  MCMC.interval = NULL, MCMC.prop.weights = NULL,
  MCMC.prop.args = NULL, MCMC.init.maxedges = NULL,
  MCMC.packagenames = NULL, MCMC.runtime.traceplot = FALSE,
  network.output = "network", seed = NULL, parallel = 0,
  parallel.type = NULL, parallel.version.check = TRUE)
```

Arguments

nsim	Number of networks to be randomly drawn using Markov chain Monte Carlo. This sample of networks provides the basis for comparing the model to the observed network.
MCMC.burnin	Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number.
MCMC.interval	Number of proposals between sampled statistics.

<code>MCMC.prop.weights</code>	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 <code>constraints</code> argument of the <code>ergm</code> 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.
<code>MCMC.prop.args</code>	An alternative, direct way of specifying additional arguments to proposal.
<code>MCMC.init.maxedges</code>	Maximum number of edges expected in network.
<code>MCMC.packagenames</code>	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.
<code>MCMC.runtime.traceplot</code>	Logical: If TRUE, plot traceplots of the MCMC sample after every MCMC MLE iteration.
<code>network.output</code>	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)
<code>seed</code>	Seed value (integer) for the random number generator. See set.seed .
<code>parallel</code>	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
<code>parallel.type</code>	API to use for parallel processing. Supported values are "MPI" and "PSOCK". Defaults to using the <code>parallel</code> package with PSOCK clusters. See ergm-parallel
<code>parallel.version.check</code>	Logical: If TRUE, check that the version of <code>ergm</code> 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 = NULL,
  MCMC.init.maxedges = NULL, MCMC.packagenames = NULL,
  term.options = NULL, seed = NULL, parallel = NULL,
  parallel.type = NULL, parallel.version.check = TRUE)
```

Arguments

- | | |
|---|---|
| nsteps | Number of geometric bridges to use. |
| MCMC.burnin | Number of proposals before any MCMC sampling is done. It typically is set to a fairly large number. |
| MCMC.interval | Number of proposals between sampled statistics. |
| MCMC.samplesize | Number of network statistics, randomly drawn from a given distribution on the set of all networks, returned by the Metropolis-Hastings algorithm. |
| obs.MCMC.burnin, obs.MCMC.interval, obs.MCMC.samplesize | The obs versions of these arguments are for the unobserved data simulation algorithm. |
| MCMC.prop.weights | Specifies the proposal distribution used in the MCMC Metropolis-Hastings algorithm. Possible choices are "TNT" or "random"; the "default" is one of these two, depending on the constraints in place (as defined by the constraints argument of the ergm function), though not all weights may be used with all constraints. The TNT (tie / no tie) option puts roughly equal weight on selecting a dyad with or without a tie as a candidate for toggling, whereas the random option puts equal weight on all possible dyads, though the interpretation of random may change according to the constraints in place. When no constraints are in place, the default is TNT, which appears to improve Markov chain mixing particularly for networks with a low edge density, as is typical of many realistic social networks. |
| MCMC.prop.args | An alternative, direct way of specifying additional arguments to proposal. |

warn.dyads	Whether or not a warning should be issued when sample space constraints render the observed number of dyads ill-defined. Now defunct: use <code>options(ergm.logLik.warn_dyads=...)</code> instead.
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.
term.options	A list of additional arguments to be passed to term initializers. It can also be set globally via <code>option(ergm.term=list(...))</code> .
seed	Seed value (integer) for the random number generator. See set.seed .
parallel	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
parallel.type	API to use for parallel processing. Supported values are "MPI" and "PSOCK". Defaults to using the parallel package with PSOCK clusters. See ergm-parallel
parallel.version.check	Logical: If TRUE, check that the version of ergm running on the slave nodes is the same as that running on the master node.

Details

This function is only used within a call to the [logLik.ergm](#) function.

Value

A list with arguments as components.

See Also

[logLik.ergm](#)

control.san	<i>Auxiliary for Controlling SAN</i>
-------------	--------------------------------------

Description

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

Usage

```
control.san(SAN.maxit = 4, SAN.tau = 1, SAN.invcov = NULL,
  SAN.invcov.diag = FALSE, SAN.nsteps.alloc = function(nsim)
  2^seq_len(nsim), SAN.nsteps = 2^19, SAN.samplesize = 2^12,
  SAN.init.maxedges = 20000, SAN.max.maxedges = 2^26,
  SAN.prop.weights = "default", SAN.prop.args = list(),
  SAN.packagenames = c(), term.options = list(), seed = NULL,
  parallel = 0, parallel.type = NULL, parallel.version.check = TRUE)
```

Arguments

<code>SAN.maxit</code>	Number of temperature levels to use.
<code>SAN.tau</code>	Tuning parameter, specifying the temperature of the process during the <i>penultimate</i> iteration. (During the last iteration, the temperature is set to 0, resulting in a greedy search, and during the previous iterations, the temperature is set to <code>SAN.tau*(iterations left after this one)</code>).
<code>SAN.invcov</code>	Initial inverse covariance matrix used to calculate Mahalanobis distance in determining how far a proposed MCMC move is from the <code>target.stats</code> vector. If NULL, initially set to the identity matrix, then during subsequent runs estimated empirically.
<code>SAN.invcov.diag</code>	Whether to only use the diagonal of the covariance matrix. It seems to work better in practice.
<code>SAN.nsteps.alloc</code>	Either a numeric vector or a function of the number of runs giving a sequence of relative lengths of simulated annealing runs.
<code>SAN.nsteps</code>	Number of MCMC proposals for all the annealing runs combined.
<code>SAN.samplesize</code>	Number of realisations' statistics to obtain for tuning purposes.
<code>SAN.init.maxedges</code>	Maximum number of edges expected in network.
<code>SAN.max.maxedges</code>	Hard upper bound on the number of edges in the network.
<code>SAN.prop.weights</code>	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.
<code>SAN.prop.args</code>	An alternative, direct way of specifying additional arguments to proposal.
<code>SAN.packagenames</code>	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.
<code>term.options</code>	A list of additional arguments to be passed to term initializers. It can also be set globally via <code>option(ergm.term=list(...))</code> .
<code>seed</code>	Seed value (integer) for the random number generator. See set.seed .
<code>parallel</code>	Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on parallel processing for details and troubleshooting.
<code>parallel.type</code>	API to use for parallel processing. Supported values are "MPI" and "PSOCK". Defaults to using the <code>parallel</code> package with PSOCK clusters. See ergm-parallel
<code>parallel.version.check</code>	Logical: If TRUE, check that the version of <code>ergm</code> 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.ergm *Auxiliary for Controlling ERGM Simulation*

Description

Auxiliary function as user interface for fine-tuning ERGM simulation. `control.simulate`, `control.simulate.formula`, and `control.simulate.formula.ergm` are all aliases for the same function.

While the others supply a full set of simulation settings, `control.simulate.ergm` when passed as a control parameter to `simulate.ergm()` allows some settings to be inherited from the ERGM stimulation while overriding others.

Usage

```
control.simulate.formula.ergm(MCMC.burnin = 10000,
  MCMC.interval = 1000, MCMC.prop.weights = "default",
  MCMC.prop.args = list(), MCMC.init.maxedges = 20000,
  MCMC.packagenames = c(), MCMC.runtime.traceplot = FALSE,
  network.output = "network", term.options = NULL, parallel = 0,
  parallel.type = NULL, parallel.version.check = TRUE, ...)
```

```
control.simulate(MCMC.burnin = 10000, MCMC.interval = 1000,
  MCMC.prop.weights = "default", MCMC.prop.args = list(),
  MCMC.init.maxedges = 20000, MCMC.packagenames = c(),
  MCMC.runtime.traceplot = FALSE, network.output = "network",
  term.options = NULL, parallel = 0, parallel.type = NULL,
  parallel.version.check = TRUE, ...)
```

```
control.simulate.formula(MCMC.burnin = 10000, MCMC.interval = 1000,
  MCMC.prop.weights = "default", MCMC.prop.args = list(),
  MCMC.init.maxedges = 20000, MCMC.packagenames = c(),
  MCMC.runtime.traceplot = FALSE, network.output = "network",
  term.options = NULL, 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",
term.options = NULL, parallel = 0, parallel.type = NULL,
parallel.version.check = TRUE, ...)
```

Arguments

- 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)
- term.options** A list of additional arguments to be passed to term initializers. It can also be set globally via `option(ergm.term=list(...))`.
- parallel** Number of threads in which to run the sampling. Defaults to 0 (no parallelism). See the entry on [parallel processing](#) for details and troubleshooting.
- parallel.type** API to use for parallel processing. Supported values are "MPI" and "PSOCK". Defaults to using the `parallel` package with PSOCK clusters. See [ergm-parallel](#)
- parallel.version.check**
Logical: If TRUE, check that the version of `ergm` running on the slave nodes is the same as that running on the master node.

... Additional arguments, passed to other functions This argument is helpful because it collects any control parameters that have been deprecated; a warning message is printed in case of deprecated arguments.

Details

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

Value

A list with arguments as components.

See Also

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

degreedist	<i>Computes and Returns the Degree Distribution Information for a Given Network</i>
------------	---

Description

The `degreedist` generic computes and returns the degree distribution (number of vertices in the network with each degree value) for a given network. This help page documents the function. For help about [the ERGM sample space constraint with that name](#), try `help("degreedist-constraint")`.

Usage

```
degreedist(object, ...)

## S3 method for class 'network'
degreedist(object, print = TRUE, ...)
```

Arguments

<code>object</code>	a network object or some other object for which degree distribution is meaningful.
...	Additional arguments to functions.
<code>print</code>	logical, whether to print the degree distribution.

Value

If directed, a matrix of the distributions of in and out degrees; this is row bound and only contains degrees for which one of the in or out distributions has a positive count. If bipartite, a list containing the degree distributions of `b1` and `b2`. Otherwise, a vector of the positive values in the degree distribution

Methods (by class)

- network: Method for `network` objects.

Examples

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

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

Description

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

Usage

```
data(ecoli)
```

Details

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

Licenses and Citation

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

Source

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

References

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

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

edges	edges (<i>disambiguation</i>)
-------	---------------------------------

Description

edges may refer to:

- [An ERGM statistic](#) (`help("edges-term")`)
- [An ERGM sample space constraint](#) (`help("edges-constraint")`)

<code>enformulate.curved</code>	<i>Convert a curved ERGM into a form suitable as initial values for the same ergm.</i>
---------------------------------	--

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

```
enformulate.curved(object, ...)

## S3 method for class 'ergm'
enformulate.curved(object, ...)

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

Arguments

<code>object</code>	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.
<code>...</code>	Unused at this time.
<code>theta</code>	Curved model parameter configuration.
<code>response</code>	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.

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:

<code>formula</code>	The formula with curved parameter estimates incorporated.
<code>theta</code>	The coefficient vector with curved parameter estimates removed.

See Also

[ergm](#), [simulate.ergm](#)

Examples

```
data(sampson)
gest<-ergm(samplike~edges+gwesp(decay=.5, fixed=FALSE),
  control=control.ergm(MCMC.burnin=1024, MCMC.interval=8, MCMLL.maxit=1))
# Error:
gest2<-try(ergm(gest$formula,
  control=control.ergm(init=coef(gest), MCMC.burnin=1024,
    MCMC.interval=8, MCMLL.maxit=1)))

print(gest2)

# Works:
tmp<-enformulate.curved(gest)
tmp
gest2<-try(ergm(tmp$formula,
  control=control.ergm(init=tmp$theta, MCMC.burnin=1024,
    MCMC.interval=8, MCMLL.maxit=1)))

summary(gest2)
```

Description

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

Usage

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

is.ergm(object)

## S3 method for class 'ergm'
nobs(object, ...)

## S3 method for class 'ergm'
print(x, digits = max(3, getOption("digits") - 3), ...)

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

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

## S3 method for class 'ergm'
vcov(object, sources = c("all", "model", "estimation"),
      ...)
```

Arguments

`formula` An R [formula](#) object, of the form $y \sim \langle \text{model terms} \rangle$, where y is a [network](#) object or a matrix that can be coerced to a [network](#) object. For the details on the possible $\langle \text{model terms} \rangle$, 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 <code>offset()</code> fixes its value to one specified in <code>offset.coef</code> .
<code>response</code>	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.
<code>reference</code>	A one-sided formula specifying the reference measure ($h(y)$) to be used. (Defaults to <code>~Bernoulli</code> .) See help for ERGM reference measures implemented in the <code>ergm</code> package.
<code>constraints</code>	<p>A formula specifying one or more constraints on the support of the distribution of the networks being modeled, using syntax similar to the <code>formula</code> argument, on the right-hand side. Multiple constraints may be given, separated by “+” and “-” operators. (See ERGM constraints for the explanation of their semantics.) Together with the model terms in the formula and the reference measure, the constraints define the distribution of networks being modeled.</p> <p>It is also possible to specify a proposal function directly either by passing a string with the function’s name (in which case, arguments to the proposal should be specified through the <code>prop.args</code> argument to <code>control.ergm</code>) or by giving it on the LHS of the constraints formula, in which case it will override the one chosen automatically.</p> <p>The default is <code>~.</code>, for an unconstrained model.</p> <p>See the ERGM constraints documentation for the constraints implemented in the <code>ergm</code> package. Other packages may add their own constraints.</p> <p>Note that not all possible combinations of constraints and reference measures are supported. However, for relatively simple constraints (i.e., those that simply permit or forbid specific dyads or sets of dyads from changing), arbitrary combinations should be possible.</p>
<code>offset.coef</code>	A vector of coefficients for the offset terms.
<code>target.stats</code>	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.
<code>eval.loglik</code>	Logical: For dyad-dependent models, if TRUE, use bridge sampling to evaluate the log-likelihood 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. Can be set globally via <code>option(ergm.eval.loglik=...)</code> , which is set to TRUE when the package is loaded.
<code>estimate</code>	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 <code>force.main</code> argument of <code>control.ergm</code> . If "CD" (<i>EXPERIMENTAL</i>), the Monte-Carlo contrastive divergence estimate is returned.)

control	A list of control parameters for algorithm tuning. Constructed using <code>control.ergm</code> .
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.
object	an ergm object.
x, digits	See <code>print()</code> . Automatically called when an object of class <code>ergm</code> is printed. Currently, summarizes the size of the MCMC sample, the θ vector governing the selection of the sample, and the Monte Carlo MLE. The optional <code>digits</code> argument specifies the significant digits for coefficients
sources	For the <code>vcov</code> method, specify whether to return the covariance matrix from the ERGM model, the estimation process, or both combined.

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, generated by the last iteration of the MCMC-based likelihood maximization routine. These statistics are centered with respect to the observed statistics or <code>target.stats</code> , unless missing data MLE is used.
sample.obs	As <code>sample</code> , 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, <code>sample</code> is roughly a random sample from the distribution of network statistics specified by the model with the parameter equal to <code>MCMCtheta</code> . If <code>estimate="MPLE"</code> then <code>MCMCtheta</code> equals the MPLE.
loglikelihood	The approximate change in log-likelihood in the last iteration. The value is only approximate because it is estimated based on the MCMC random sample.
gradient	The value of the gradient vector of the approximated loglikelihood function, evaluated at the maximizer. This vector should be very close to zero.
covar	Approximate covariance matrix for the MLE, based on the inverse Hessian of the approximated loglikelihood evaluated at the maximizer.
failure	Logical: Did the MCMC estimation fail?
network	Original network
newnetworks	A list of the final networks at the end of the MCMC simulation, one for each thread.
newnetwork	The first (possibly only) element of <code>newnetworks</code> .
coef.init	The initial value of θ .

<code>est.cov</code>	The covariance matrix of the model statistics in the final MCMC sample.
<code>coef.hist</code> , <code>step.len.hist</code> , <code>stats.hist</code> , <code>stats.obs.hist</code>	For the MCMLE method, the history of coefficients, Hummel step lengths, and average model statistics for each iteration..
<code>control</code>	The control list passed to the call.
<code>etamap</code>	The set of functions mapping the true parameter θ to the canonical parameter η (irrelevant except in a curved exponential family model)
<code>formula</code>	The original <code>formula</code> entered into the <code>ergm</code> function.
<code>target.stats</code>	The <code>target.stats</code> used during estimation (passed through from the Arguments)
<code>target.esteq</code>	Used for curved models to preserve the target mean values of the curved terms. It is identical to <code>target.stats</code> for non-curved models.
<code>constrained</code>	The list of constraints implied by the constraints used by original <code>ergm</code> call
<code>constraints</code>	Constraints used during estimation (passed through from the Arguments)
<code>reference</code>	The reference measure used during estimation (passed through from the Arguments)
<code>estimate</code>	The estimation method used (passed through from the Arguments).
<code>offset</code>	vector of logical telling which model parameters are to be set at a fixed value (i.e., not estimated).
<code>drop</code>	If <code>control\$drop=TRUE</code> , a numeric vector indicating which terms were dropped due to extreme values of the corresponding statistics on the observed network, and how: \emptyset The term was not dropped. -1 The term was at its minimum and the coefficient was fixed at $-\text{Inf}$. +1 The term was at its maximum and the coefficient was fixed at $+\text{Inf}$.
<code>estimable</code>	A logical vector indicating which terms could not be estimated due to a constraints constraint fixing that term at a constant value.
<code>null.lik</code>	Log-likelihood of the null model. Valid only for unconstrained models.
<code>mle.lik</code>	The approximate log-likelihood for the MLE. The value is only approximate because it is estimated based on the MCMC random sample.
<code>degeneracy.value</code>	Score calculated to assess the degree of degeneracy in the model. Only shows when <code>MCMLE.check.degeneracy</code> is <code>TRUE</code> in <code>control.ergm</code> .
<code>degeneracy.type</code>	Supporting output for <code>degeneracy.value</code> . Only shows when <code>MCMLE.check.degeneracy</code> is <code>TRUE</code> in <code>control.ergm</code> . Mainly for internal use.

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

Methods (by generic)

- `nobs`: Return the number of informative dyads of a model fit.
- `print`:
- `coef`: extracts estimated model coefficients.
- `coefficients`: An *alias* for `ergm`.
- `vcov`: extracts the variance-covariance matrix of parameter estimates.

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. <https://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). <https://www.jstatsoft.org/v24/i07/>.
- Butts CT (2007). **sna**: Tools for Social Network Analysis. R package version 2.3-2. <https://cran.r-project.org/package=sna>.
- Butts CT (2008). **network**: A Package for Managing Relational Data in R. *Journal of Statistical Software*, 24(2). <https://www.jstatsoft.org/v24/i02/>.
- Butts C (2015). **network**: The Statnet Project (<https://statnet.org>). R package version 1.12.0, <https://cran.r-project.org/package=network>.
- Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008a). A **statnet** Tutorial. *Journal of Statistical Software*, 24(8). <https://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. <https://www.csss.washington.edu/Papers/wp39.pdf>

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

Handcock MS and Gile KJ (2010). Modeling Social Networks from Sampled Data. *Annals of Applied Statistics*, 4(1), 5-25. doi: [10.1214/08AOAS221](https://doi.org/10.1214/08AOAS221)

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, <https://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, <https://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). <https://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: [10.1214/12EJS696](https://doi.org/10.1214/12EJS696)

Morris M, Handcock MS, Hunter DR (2008). Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 24(4). <https://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 <https://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(1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,3,3,3,3)
#
# 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)

# Extract parameter estimates as a numeric vector:
coef(gest)

# Sources of variation in parameter estimates:
vcov(gest, sources="model")
vcov(gest, sources="estimation")
vcov(gest, sources="all") # the default

```

ergm-constraints

Sample Space Constraints for Exponential-Family Random Graph Models

Description

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

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

Constraints formula

A constraints formula is a one- or two-sided formula whose left-hand side is an optional direct selection of the `InitErgmProposal` function and whose right-hand side is a series of one or more terms separated by “+” and “-” operators, specifying the constraint.

The sample space (over and above the reference distribution) is determined by iterating over the constraints terms from left to right, each term updating it as follows:

- If the constraint introduces complex dependence structure (e.g., constrains degree or number of edges in the network), then this constraint always restricts the sample space. It may only have a “+” sign.

- If the constraint only restricts the set of dyads that may vary in the sample space (e.g., block-diagonal structure or fixing specific dyads at specific values) and has a “+” sign, the set of dyads that may vary is restricted to those that may vary according to this constraint *and* all the constraints to date.
- If the constraint only restricts the set of dyads that may vary in the sample space but has a “-” sign, the set of dyads that may vary is expanded to those that may vary according to this constraint *or* all the constraints up to date.

For example, a constraints formula $\sim a - b + c - d$ with all constraints dyadic will allow dyads permitted by either ‘a’ or ‘b’ but only if they are also permitted by ‘c’; as well as all dyads permitted by ‘d’. If ‘A’, ‘B’, ‘C’, and ‘D’ were logical matrices, the matrix of variable dyads would be equal to ‘((A|B)&C)|D’.

Terms with a positive sign can be viewed as “adding” a constraint while those with a negative sign can be viewed as “relaxing” a constraint.

Constraints implemented in the `ergm` package

- . **or NULL (dyad-independent)** A placeholder for no constraints: all networks of a particular size and type have non-zero probability. Cannot be combined with other constraints.
- `bd(attrs,maxout,maxin,minout,minin)` Constrain maximum and minimum vertex degree. See “Placing Bounds on Degrees” section for more information.
- `blockdiag(attrname)` **(dyad-independent)** Force a block-diagonal structure (and its bipartite analogue) on the network. Only dyads (i, j) for which `attrname(i)==attrname(j)` can have edges.
Note that the current implementation requires that blocks be contiguous for “unipartite” graphs, and for bipartite graphs, they must be contiguous within a partition and must have the same ordering in both partitions. (They do not, however, require that all blocks be represented in both partitions, but those that overlap must have the same order.)
- `degrees` **and** `nodedegrees` Preserve the degree of each vertex of the given network: only networks whose vertex degrees are the same as those in the network passed in the model formula have non-zero probability. If the network is directed, both indegree and outdegree are preserved.
- `odegrees`, `idegrees`, `b1degrees`, `b2degrees` For directed networks, `odegrees` preserves the out-degree 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` **(dyad-independent)** Preserve the observed dyads of the given network.
- `fixedas(present, absent)` **(dyad-independent)** Preserve the edges in ‘present’ and preclude the edges in ‘absent’. Both ‘present’ and ‘absent’ can take input object as edgelist and network, the latter will convert to the corresponding edgelist.

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

Not all combinations of the above are supported.

Placing Bounds on Degrees:

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

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

```
constraints=~bd(maxout=3)
```

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

You can also set ego specific limits. For example:

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

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

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

```
bd(attrs=attrs,
    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, `attrs` is a matrix of the attributes on which we are conditioning. The dimensions of `attrs` 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 `attrs[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 `attrs` 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). <https://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). <https://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: [10.1214/12EJS696](https://doi.org/10.1214/12EJS696)
- Morris M, Handcock MS, Hunter DR (2008). Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects. *Journal of Statistical Software*, 24(4). <https://www.jstatsoft.org/v24/i04/>.

Description

For estimation that require MCMC, `ergm` can take advantage of multiple CPUs or CPU cores on the system on which it runs, as well as computing clusters. It uses package `parallel` and `snow` to facilitate this, and supports all cluster types that they does. The number of nodes used and the parallel API are controlled using the `parallel` and `parallel.type` arguments passed to the control functions, such as `control.ergm`.

The `ergm.getCluster` function is usually called internally by the `ergm` process (in `ergm_MCMC_sample`) and will attempt to start the appropriate type of cluster indicated by the `control.ergm` settings. It will also check that the same version of `ergm` is installed on each node.

The `ergm.stopCluster` shuts down a cluster, but only if `ergm.getCluster` was responsible for starting it.

The `ergm.restartCluster` restarts and returns a cluster, but only if `ergm.getCluster` was responsible for starting it.

`nthreads` is a simple generic to obtain the number of parallel processes represented by its argument, keeping in mind that having no cluster (e.g., `NULL`) represents one thread.

Usage

```
ergm.getCluster(control = NULL, verbose = FALSE,
               stop_on_exit = parent.frame())

ergm.stopCluster(..., verbose = FALSE)

ergm.restartCluster(control = NULL, verbose = FALSE)

nthreads(clinfo = NULL, ...)

## S3 method for class 'cluster'
nthreads(clinfo = NULL, ...)

## S3 method for class 'NULL'
nthreads(clinfo = NULL, ...)

## S3 method for class 'control.list'
nthreads(clinfo = NULL, ...)
```

Arguments

`control` a `control.ergm` (or similar) list of parameter values from which the parallel settings should be read; can also be `NULL`, in which case an existing cluster is used if started, or no cluster otherwise.

verbose	logical, should detailed status info be printed to console?
stop_on_exit	An environment or NULL. If an environment, defaulting to that of the calling function, the cluster will be stopped when the calling the frame in question exits.
...	not currently used
clinfo	a cluster or another object.

Details

Further details on the various cluster types are included below.

PSOCK clusters

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

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

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

MPI clusters

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

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

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

User initiated clusters

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

Examples

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

Description

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

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

Possible reference measures to represent baseline distributions

Reference measures currently available are:

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

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

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

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

References

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008b). `ergm`: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software*, 24(3). <https://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: [10.1214/12EJS696](https://doi.org/10.1214/12EJS696)

See Also

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

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.

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

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 $y \sim \langle \text{term 1} \rangle + \langle \text{term 2} \rangle \dots$, where y is a network object or a matrix that can be coerced to a network object, and $\langle \text{term 1} \rangle$, $\langle \text{term 2} \rangle$, 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 independence — 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 considered to have a tie. (Value less than or equal to `threshold` is considered a nontie.)

Nodal attribute levels and indices:

Terms taking a categorical nodal covariate also take the `levels` argument. (There are analogous `b1levels` and `b2levels` arguments for some terms that apply to bipartite networks, and the `levels2` argument for mixing terms.) The `levels` argument can be used to control the set and the ordering of attribute levels.

Terms that allow the selection of nodes do so with the `nodes` argument, which is interpreted in the same way as the `levels` argument, where the categories are the relevant nodal indices themselves. Both `levels` and `nodes` use the new level selection UI. (See [Specifying Vertex Attributes and Levels](#) for details.)

Legacy arguments:

The legacy `base` and `keep` arguments are deprecated as of version 3.10, and replaced by the `levels` UI. The `levels` argument provides consistent and flexible mechanisms for specifying which attribute levels to exclude (previously handled by `base`) and include (previously handled by `keep`). If `levels` or `nodes` argument is given, then `base` and `keep` arguments are ignored. The legacy arguments will most likely be removed in a future version.

Note that this exact behavior is new in version 3.10, and it differs slightly from older versions: previously if both `levels` and `base/keep` were given, `levels` argument was applied first and then applied the `base/keep` argument. Since version 3.10, `base/keep` would be ignored, even if old term behavior is invoked (as described in the next section).

Term versioning:

When a term's behavior has changed from prior version, it is often possible to invoke the old behavior by setting and/or passing a version term option, giving the version (constructed by `as.package_version`) desired.

Custom ergm terms:

Users and other packages may build custom 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.

Terms included in the `ergm` package

As noted above, a cross-referenced HTML version of the term documentation is available via `vignette('ergm-term-crossRef')` and terms can also be searched via `search.ergmTerms`.

`absdiff(attr, pow=1)` **(binary) (dyad-independent) (frequently-used) (directed) (undirected) (quantitative nodal attribute)**

Absolute difference: The `attr` argument specifies a quantitative attribute (see [Specifying Vertex Attributes and Levels](#) for details). This term adds one network statistic to the model equaling the sum of $\text{abs}(\text{attr}[i] - \text{attr}[j])^{\text{pow}}$ for all edges (i, j) in the network.

Note that `ergm` versions 3.9.4 and earlier used different arguments for this term. See the above section on versioning for invoking the old behavior.

`absdiffcat(attr, base=NULL, levels=NULL)` **(binary) (dyad-independent) (directed) (undirected) (categorical nodal attribute)**

Categorical absolute difference: The `attr` argument specifies a quantitative attribute (see [Specifying Vertex Attributes and Levels](#) for details). This term adds one statistic for every possible nonzero distinct value of $\text{abs}(\text{attr}[i] - \text{attr}[j])$ in the network; the value of each such statistic is the number of edges in the network with the corresponding absolute difference. The optional argument `levels` specifies which nonzero differences to include in or exclude from the model (see [Specifying Vertex Attributes and Levels](#) for details). For example, if the possible values of $\text{abs}(\text{attr}[i] - \text{attr}[j])$ are 0, 0.5, 3, 3.5, and 10, then to omit 0.5 and 10 one could set `levels=2:3` (we wish to retain the second and third nonzero difference, when differences are sorted in increasing order). 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.

The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels` are passed, `levels` overrides `base`.

`altkstar(lambda, fixed=FALSE)` **(binary) (undirected) (curved) (categorical nodal attribute)**

Alternating k-star: This term adds one network statistic to the model equal to a weighted alternating sequence of k-star statistics with weight parameter `lambda`. This is the version given in Snijders et al. (2006). The `gwdegree` and `altkstar` produce mathematically equivalent models, as long as they are used together with the `edges` (or `kstar(1)`) term, yet the interpretation of the `gwdegree` parameters is slightly more straightforward than the interpretation of the `altkstar` parameters. For this reason, we recommend the use of the `gwdegree` instead of `altkstar`. See Section 3 and especially equation (13) of Hunter (2007) for details. The optional argument `fixed` indicates whether the decay parameter is fixed at the given value, or 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(attr=NULL, diff=FALSE, keep=NULL, levels=NULL)` **(binary) (directed) (dyad-independent) (triad-rel)**

Asymmetric dyads: This term adds one network statistic to the model equal to the number of pairs of actors for which exactly one of $(i \rightarrow j)$ or $(j \rightarrow i)$ exists. This term can only be used with directed networks. The optional argument `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). If `attr` is specified, only asymmetric pairs that match on the vertex attribute `attr` are counted. The optional modifiers `diff` and `levels` are used in the same way as for the `nodematch` term; refer to this term for details and an example. The argument `keep` is retained for backwards compatibility and may be removed in a future version. When both `keep` and `levels` are passed, `levels` overrides `keep`.

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

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

`b1concurrent(by=NULL, levels=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**

Concurrent node count for the first mode in a bipartite (aka two-mode) network: This term adds one network statistic to the model, equal to the number of nodes in the first mode of the network with degree 2 or higher. The first mode of a bipartite network object is sometimes known as the "actor" mode. The optional argument `by` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details); it functions just like the `by` argument of the `b1degree` term. Without the optional argument, this statistic is equivalent to `b1mindegree(2)`. This term can only be used with undirected bipartite networks.

`b1cov(attr)` **(binary) (undirected) (bipartite) (dyad-independent) (quantitative nodalattribute) (frequently-used), b1**

Main effect of a covariate for the first mode in a bipartite (aka two-mode) network: The `attr` argument specifies one or more quantitative attributes (see [Specifying Vertex Attributes and Levels](#) for details). This term adds a single network statistic for each quantitative attribute or matrix column to the model equaling the total value of `attr(i)` for all edges (i, j) in the network. This term may only be used with bipartite networks. For categorical attributes, see `b1factor`.

Note that ergm versions 3.9.4 and earlier used different arguments for this term. See the above section on versioning for invoking the old behavior.

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

Degree range for the first mode in a bipartite (a.k.a. two-mode) network: The `from` and `to` arguments are vectors of distinct integers (or `+Inf`, for `to` (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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, levels=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute) (frequently-used)**

Degree for the first mode in a bipartite (aka two-mode) network: The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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(attr, base=1, levels=-1)` **(binary) (bipartite) (undirected) (dyad-independent) (frequently-used) (categorical)**

Factor attribute effect for the first mode in a bipartite (aka two-mode) network: The `attr` argument specifies a categorical vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attr` 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.

The optional `levels` argument controls which levels of the attribute should be included and which should be excluded. (See [Specifying Vertex Attributes and Levels](#) for details.) For example, if the "fruit" attribute has levels "orange", "apple", "banana", and "pear", then to include just two levels, one for "apple" and one for "pear", use any of `b1factor("fruit", levels=-(2:3))`, `b1factor("fruit", levels=c(1,4))`, and `b1factor("fruit", levels=c("apple", "pear"))`.

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. The default, `levels=-1`, is therefore to omit the first (in lexicographic order) attribute level. To include all levels, pass either `levels=TRUE` (i.e., keep all levels) or `levels=NULL` (i.e., do not filter levels).

The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels` are passed, `levels` overrides `base`.

This term can only be used with undirected bipartite networks.

`b1mindegree(d)` **(binary) (bipartite) (undirected)** *Minimum degree for the first mode in a bipartite (aka two-mode) network:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the i th such statistic equals the number of nodes in the first mode of a bipartite network with at least degree `d[i]`. The first mode of a bipartite network object is sometimes known as the "actor" mode.

This term can only be used with undirected bipartite networks.

`b1nodematch(attr, diff=FALSE, keep=NULL, alpha=1, beta=1, byb2attr=NULL, levels=NULL)` **(binary) (bipartite)**

Nodal attribute-based homophily effect for the first mode in a bipartite (aka two-mode) network: This term is introduced in Bomiriyá et al (2014). The `attr` argument specifies a categorical vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). Out of the two arguments (discount parameters) `alpha` and `beta`, both of which take values from $[0,1]$, only one should be set at a time. If none is set to a value other than 1, this term will simply be a homophily based two-star statistic. This term adds one statistic to the model unless `diff` is set to `TRUE`, in which case the term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attr` attribute. To include only the attribute values you wish, use the `levels` arguments.

The argument `keep` is retained for backwards compatibility and may be removed in a future version. When both `keep` and `levels` are passed, `levels` overrides `keep`.

If an `alpha` discount parameter is used, each of these statistics gives the sum of the number of common second-mode nodes raised to the power `alpha` for each pair of first-mode nodes with that attribute. If a `beta` discount parameter is used, each of these statistics gives half the sum of the number of two-paths with two first-mode nodes with that attribute as the two ends of the two path raised to the power `beta` for each edge in the network. The `byb2attr` argument specifies a second mode categorical attribute. Setting this argument will separate the original statistics based on the values of the set second mode attribute— i.e. for example, if `diff` is `FALSE`, then the sum of all the statistics for each level of this second-mode attribute will be equal to the original `b1nodematch` statistic where `byb2attr` set to `NULL`. This term can only be used with undirected bipartite networks.

`b1sociality(nodes=-1)` **(binary) (bipartite) (undirected) (dyad-independent)**, `b1sociality(nodes=-1, form="sum")`

Degree: This term adds one network statistic for each node in the first bipartition, equal to the number of ties of that node. By default, `nodes=-1` means that the statistic for the first node will be omitted, but this argument may be changed to control which statistics are included. The `nodes` argument is interpreted using the new UI for level specification (see [Specifying Vertex Attributes and Levels](#) for details), where both the attribute and the sorted unique values are the vector of vertex indices $1:nb1$, where `nb1` is the size of the first bipartition. This term can only be used with bipartite networks. For directed networks, see `sender` and `receiver`. For unipartite networks, see `sociality`.

`b1star(k, attr=NULL, levels=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**

k-Stars for the first mode in a bipartite (aka two-mode) network: The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The i th such statistic counts the number of distinct `k[i]`-stars whose center node is in the first mode of the network. The first mode of a bipartite network object is sometimes known as the "actor" mode. A k -star is defined to be a center node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $\{N, O_i\}$ exist for $i = 1, \dots, k$. The optional argument `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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, levels=NULL)` **(binary) (bipartite) (undirected) (categorical node)**

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 b_2 nodes (called events in some contexts) are homophilous in the sense that they all share the same value of `attrname`. However, the b_1 node (in some contexts, the actor) at the center of the k -star does NOT have to have the same value as the b_2 nodes; indeed, the values taken by the b_1 nodes may be completely distinct from those of the b_2 nodes, which allows for the use of this term in cases where there are two separate nodal attributes, one for the b_1 nodes and another for the b_2 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 b_1 node, even if no k -stars are observed with this value. Whether a different statistic is created for each value seen in a b_2 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 b_1 node.

The optional `levels` argument controls which levels of the attribute should be included and which should be excluded. (See [Specifying Vertex Attributes and Levels](#) for details.) For example, if the “fruit” attribute has levels “orange”, “apple”, “banana”, and “pear”, then to include just two levels, one for “apple” and one for “pear”, use any of `b1starmix("fruit", levels=-(2:3))`, `b1starmix("fruit", levels=c(1,4))`, and `b1starmix("fruit", levels=c("apple", "pear"))`. By default, all levels are included.

The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels` are passed, `levels` overrides `base`.

`b1twostar(b1attr, b2attr, base=NULL, b1levels=NULL, b2levels=NULL, levels2=NULL)` **(binary) (bipartite) (undirected)**

Two-star census for central nodes centered on the first mode of a bipartite network: This term takes two nodal attributes (see [Specifying Vertex Attributes and Levels](#) for details), one for b_1 nodes (actors in some contexts) and one for b_2 nodes (events in some contexts). Only `b1attr` is required; if `b2attr` is not passed, it is assumed to be the same as `b1attr`. Assuming that there are n_1 values of `b1attr` among the b_1 nodes and n_2 values of `b2attr` among the b_2 nodes, then the total number of distinct categories of two stars according to these two attributes is $n_1(n_2)(n_2 + 1)/2$. By default, this model term creates a distinct statistic counting each of these categories. The `b1levels`, `b2levels`, `base`, and `levels2` arguments may be used to leave some of these categories out (see [Specifying Vertex Attributes and Levels](#) for details).

The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels` are passed, `levels` overrides `base`. The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels2` are passed, `levels2` overrides `base`.

`b2concurrent(by=NULL)` **(binary) (bipartite) (undirected) (frequently-used)** *Concurrent node*

count for the second mode in a bipartite (aka two-mode) network: This term adds one network statistic to the model, equal to the number of nodes in the second mode of the network with degree 2 or higher. The second mode of a bipartite network object is sometimes known as the “event” mode. The optional argument `by` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details); it functions just like the `by` argument of the `b2degree` term. Without the optional argument, this statistic is equivalent to `b2mindegree(2)`.

This term can only be used with undirected bipartite networks.

b2cov(attr) (**binary**) (**undirected**) (**bipartite**) (**dyad-independent**) (**quantitative nodal attribute**) (**frequently-used**), b2

Main effect of a covariate for the second mode in a bipartite (aka two-mode) network: The `attr` argument specifies one or more quantitative attributes (see [Specifying Vertex Attributes and Levels](#) for details). This term adds a single network statistic for each quantitative attribute or matrix column to the model equaling the total value of `attr(j)` for all edges (i, j) in the network. This term may only be used with bipartite networks. For categorical attributes, see `b2factor`.

Note that `ergm` versions 3.9.4 and earlier used different arguments for this term. See the above section on versioning for invoking the old behavior.

b2degrange(from, to=+Inf, by=NULL, homophily=FALSE, levels=NULL) (**binary**) (**bipartite**) (**undirected**)

Degree range for the second mode in a bipartite (a.k.a. two-mode) network: The `from` and `to` arguments are vectors of distinct integers (or `+Inf`, for `to` (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). If this is specified and `homophily` is `TRUE`, then degrees are calculated using the subnetwork consisting of only edges whose endpoints have the same value of the `by` attribute. If `by` is specified and `homophily` is `FALSE` (the default), then separate degree range statistics are calculated for nodes having each separate value of the attribute.

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

b2degree(d, by=NULL) (**binary**) (**bipartite**) (**undirected**) (**categorical nodal attribute**) (**frequently-used**)

Degree for the second mode in a bipartite (aka two-mode) network: The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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(attr, base=1, levels=-1) (**binary**) (**bipartite**) (**undirected**) (**dyad-independent**) (**categorical nodal attrib**)

Factor attribute effect for the second mode in a bipartite (aka two-mode) network : The `attr` argument specifies a categorical vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attr` 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.

The optional `levels` argument controls which levels of the attribute should be included and which should be excluded. (See [Specifying Vertex Attributes and Levels](#) for details.) For example, if the "fruit" attribute has levels "orange", "apple", "banana", and "pear", then to include just two levels, one for "apple" and one for "pear", use any of `b2factor("fruit", levels=-(2:3))`, `b2factor("fruit", levels=c(1,4))`, and `b2factor("fruit", levels=c("apple", "pear"))`.

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. The default, `levels=-1`, is therefore to omit the first (in lexicographic order) attribute level. To include all levels, pass either `levels=TRUE` (i.e., keep all levels) or `levels=NULL` (i.e., do not filter levels).

The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels` are passed, `levels` overrides `base`.

This term can only be used with undirected bipartite networks.

`b2mindegree(d)` **(binary) (bipartite) (undirected)** *Minimum degree for the second mode in a bipartite (aka two-mode) network:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the i th such statistic equals the number of nodes in the second mode of a bipartite network with at least degree `d[i]`. The second mode of a bipartite network object is sometimes known as the "event" mode.

This term can only be used with undirected bipartite networks.

`b2nodematch(attr, diff=FALSE, keep=NULL, alpha=1, beta=1, byb1attr=NULL, levels=NULL)` **(binary) (bipartite) (undirected)** *Nodal attribute-based homophily effect for the second mode in a bipartite (aka two-mode) network:* This term is introduced in Bomirriya et al (2014). The `attr` argument specifies a categorical vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). Out of the two arguments (discount parameters) `alpha` and `beta`, both which takes values from $[0,1]$, only one should be set at a time. If none is set to a value other than 1, this term will simply be a homophily based two-star statistic. This term adds one statistic to the model unless `diff` is set to `TRUE`, in which case the term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attr` attribute. To include only the attribute values you wish, use the `levels` argument.

The argument `keep` is retained for backwards compatibility and may be removed in a future version. When both `keep` and `levels` are passed, `levels` overrides `keep`.

If an `alpha` discount parameter is used, each of these statistics gives the sum of the number of common first-mode nodes raised to the power `alpha` for each pair of second-mode nodes with that attribute. If a `beta` discount parameter is used, each of these statistics gives half the sum of the number of two-paths with two second-mode nodes with that attribute as the two ends of the two path raised to the power `beta` for each edge in the network. The `byb1attr` argument specifies a first mode categorical attribute. Setting this argument will separate the original statistics based on the values of the set first mode attribute— i.e. for example, if `diff` is `FALSE`, then the sum of all the statistics for each level of this first-mode attribute will be equal to the original `b2nodematch` statistic where `byb1attr` set to `NULL`.

This term can only be used with undirected bipartite networks.

`b2sociality(nodes=-1)` **(binary) (bipartite) (undirected) (dyad-independent)**, `b2sociality(nodes=-1, form="sum")` *Degree:* This term adds one network statistic for each node in the second bipartition, equal to the number of ties of that node. By default, `nodes=-1` means that the statistic for the first node (in the second bipartition) will be omitted, but this argument may be changed to control which statistics are included. The `nodes` argument is interpreted using the new UI for level specification (see [Specifying Vertex Attributes and Levels](#) for details), where both the attribute and the sorted unique values are the vector of vertex indices $(nb1 + 1):n$, where `nb1` is the size of the first bipartition and `n` is the total number of nodes in the network. Thus `nodes=120` will include only the statistic for the 120th node in the second bipartition, while `nodes=I(120)` will include only the statistic for the 120th node in the entire network. This term can only be used with undirected bipartite networks. For directed networks, see `sender` and `receiver`. For unipartite networks, see `sociality`.

`b2star(k, attr=NULL, levels=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**

k-Stars for the second mode in a bipartite (aka two-mode) network: The *k* argument is a vector of distinct integers. This term adds one network statistic to the model for each element in *k*. The *i*th such statistic counts the number of distinct *k*[*i*]-stars whose center node is in the second mode of the network. The second mode of a bipartite network object is sometimes known as the "event" mode. A *k*-star is defined to be a center node *N* and a set of *k* different nodes $\{O_1, \dots, O_k\}$ such that the ties $\{N, O_i\}$ exist for $i = 1, \dots, k$. The optional argument *attr* specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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, levels=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**

Mixing matrix for *k*-stars centered on the second mode of a bipartite network: This term is exactly the same as `b1starmix` except that the roles of *b1* and *b2* are reversed.

`b2twostar(b1attr, b2attr, base=NULL, b1levels=NULL, b2levels=NULL, levels2=NULL)` **(binary) (bipartite) (undirected) (categorical nodal attribute)**

Two-star census for central nodes centered on the second mode of a bipartite network: This term is exactly the same as `b1twostar` except that the roles of *b1* and *b2* are reversed.

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

`coincidence(levels=NULL, active=0)` **(binary) (bipartite) (undirected)** *Coincident node count*

for the second mode in a bipartite (aka two-mode) network: By default this term adds one network statistic to the model for each pair of nodes of mode two. It is equal to the number of (first mode) mutual partners of that pair. The first mode of a bipartite network object is sometimes known as the "actor" mode and the second as the "event" mode. So this is the number of actors going to both events in the pair. The optional argument *levels* specifies which pairs of nodes in mode two to include (see [Specifying Vertex Attributes and Levels](#) for details). The second optional argument, *active*, selects pairs for which the observed count is at least *active*. If both *levels* and *active* are specified, then *active* is ignored. (Thus, indices passed as *levels* should correspond to indices when *levels* = `NULL` and *active* = 0.) This term can only be used with undirected bipartite networks.

Note that `ergm` versions 3.9.4 and earlier used different arguments for this term. See the above section on versioning for invoking the old behavior.

`concurrent(by=NULL, levels=NULL)` **(binary) (undirected) (categorical nodal attribute)** *Concurrent node count*

: This term adds one network statistic to the model, equal to the number of nodes in the network with degree 2 or higher. The optional argument *by* specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details); it functions just like the *by* argument of the degree term. This term can only be used with undirected networks.

`concurrentties(by=NULL, levels=NULL)` **(binary) (undirected) (categorical nodal attribute)**

Concurrent tie count: This term adds one network statistic to the model, equal to the number of ties incident on each actor beyond the first. The optional argument *by* specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details); it functions just like the *by* argument of the degree term. This term can only be used with undirected networks.

`ctriple(attr=NULL, diff=FALSE, levels=NULL)` **(binary) (directed) (triad-related) (categorical nodal attribute)**, a

Cyclic triples: By default, this term adds one statistic to the model, equal to the number of cyclic triples in the network, defined as a set of edges of the form $\{(i \rightarrow j), (j \rightarrow k), (k \rightarrow i)\}$. Note that for all directed networks, `triangle` is equal to `ttriple+ctriple`, so at most two of these three terms can be in a model. The optional argument `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). If `attr` is specified and `diff` is `FALSE`, then the statistic is the number of cyclic triples where all three nodes have the same value of the attribute. If `attr` is specified and `diff` is `TRUE`, then one statistic is added to the model for each value of `attr` (or each value of `attr` specified by `levels` if that argument is passed), equal to the number of cyclic triples where all three nodes have that value of the attribute. This term can only be used with directed networks.

`cycle(k)` **(binary) (directed) (undirected)** *Cycles:* The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`; the i th 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.

`cyclicalities(attr=NULL, levels=NULL)` **(binary) (directed)**, `cyclicalities(threshold=0)` **(valued) (directed) (undirected)**

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

`cyclicalweights(twopath="min", combine="max", affect="min")` **(valued) (directed) (undirected)**

Cyclical weights: This statistic implements the cyclical weights statistic, like that defined by Krivitsky (2012), Equation 13, but with the focus dyad being $y_{j,i}$ rather than $y_{i,j}$. The currently implemented options for `twopath` is the minimum of the constituent dyads ("min") or their geometric mean ("geomean"); for `combine`, the maximum of the 2-path strengths ("max") or their sum ("sum"); and for `affect`, the minimum of the focus dyad and the combined strength of the two paths ("min") or their geometric mean ("geomean"). For each of these options, the first (and the default) is more stable but also more conservative, while the second is more sensitive but more likely to induce a multimodal distribution of networks.

`ddsp(d, type="OTP")` **(binary) (directed)** *Directed dyadwise shared partners:* This term adds one network statistic to the model for each element in `d` where the i th such statistic equals the number of *dyads* in the network with exactly `d[i]` shared partners. This term can only be used with directed networks.

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

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

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

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

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

By default, outgoing two-paths ("OTP") are calculated. Note that Robins et al. (2009) define closely related statistics to several of the above, using slightly different terminology.

`degrange(from, to=+Inf, by=NULL, homophily=FALSE, levels=NULL)` **(binary) (undirected) (categorical nodal attribute)**

Degree range: The `from` and `to` arguments are vectors of distinct integers (or `+Inf`, for `to` (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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 of the second mode ("events"), use `b2degrange`.

`degree(d, by=NULL, homophily=FALSE, levels=NULL)` **(binary) (undirected) (categorical nodal attribute) (frequency)**

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 i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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`.

`degree1.5` **(binary) (undirected)** *Degree to the 3/2 power:* 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.

`degreepopularity` **(binary) (undirected) (deprecated)** *Degree popularity (deprecated):* see `degree1.5`.

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

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

`density` **(binary) (dyad-independent) (directed) (undirected)** *Density:* This term adds one network statistic equal to the density of the network. For undirected networks, `density` equals

kstar(1) or edges divided by $n(n-1)/2$; for directed networks, density equals edges or istar(1) or ostar(1) divided by $n(n-1)$.

`diff(attr, pow=1, dir="t-h", sign.action="identity")` **(binary) (dyad-independent) (frequently-used) (directed)**

Difference: The `attr` argument specifies a quantitative vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). For values of `pow` other than 0, this term adds one network statistic to the model, equaling the sum, over directed edges (i, j) , of `sign.action(attr[i]-attr[j])^pow` if `dir` is "t-h" (the default), "tail-head", or "b1-b2" and of `sign.action(attr[j]-attr[i])^pow` if "h-t", "head-tail", or "b2-b1". That is, the argument `dir` determines which vertex's attribute is subtracted from which, with tail being the origin of a directed edge and head being its destination, and bipartite networks' edges being treated as going from the first part (b1) to the second (b2).

If `pow==0`, the exponentiation is replaced by the signum function: +1 if the difference is positive, 0 if there is no difference, and -1 if the difference is negative. Note that this function is applied *after* the `sign.action`. The comparison is exact, so when using calculated values of `attr`, ensure that values that you want to be considered equal are, in fact, equal.

The following `sign.action`s are possible:

"identity" **(the default)** no transformation of the difference regardless of sign

"abs" absolute value of the difference: equivalent to the `absdiff` term

"posonly" positive differences are kept, negative differences are replaced by 0

"negonly" negative differences are kept, positive differences are replaced by 0

Note that this term may not be meaningful for unipartite undirected networks unless `sign.action=="abs"`.

When used on such a network, it behaves as if all edges were directed, going from the lower-indexed vertex to the higher-indexed vertex.

`desp(d, type="OTP")` **(binary) (directed)** *Directed edgewise shared partners:* This term adds one network statistic to the model for each element in `d` where the i th such statistic equals the number of *edges* in the network with exactly `d[i]` shared partners. This term can only be used with directed networks.

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

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

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

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

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

By default, outgoing two-paths ("OTP") are calculated. Note that Robins et al. (2009) define closely related statistics to several of the above, using slightly different terminology.

`dgdwsp(decay=0, fixed=FALSE, cutoff=30, type="OTP")` **(binary) (directed)** *Geometrically weighted dyadwise shared partner distribution:* This term adds one network statistic to the model equal to the geometrically weighted dyadwise shared partner distribution with decay parameter `decay`, which should be non-negative. (this parameter was called α)

prior to `ergm` 3.7). The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used instead as the starting value for the estimation of decay in a curved exponential family model (when `fixed=FALSE`, the default) (see Hunter and Handcock, 2006). Note that the `GWESP` statistic is equal to the sum of `GWNSP` plus `GWESP`.

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

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

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

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

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

By default, outgoing two-paths ("OTP") are calculated. Note that Robins et al. (2009) define closely related statistics to several of the above, using slightly different terminology.

The optional argument `cutoff` sets the number of underlying DSP terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm](#).)

`dgwesp(decay=0, fixed=FALSE, cutoff=30, type="OTP")` **(binary) (directed) Geometrically weighted edgewise shared partner distribution:** This term adds a statistic equal to the geometrically weighted *edgewise* (not *dyadwise*) shared partner distribution with decay parameter `decay`, which should be non-negative. (this parameter was called `alpha` prior to `ergm` 3.7). The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used instead as the starting value for the estimation of decay in a curved exponential family model (when `fixed=FALSE`, the default) (see Hunter and Handcock, 2006).

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

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

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

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

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

By default, outgoing two-paths ("OTP") are calculated. Note that Robins et al. (2009) define closely related statistics to several of the above, using slightly different terminology.

The optional argument `cutoff` sets the number of underlying ESP terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm](#).)

`dgwnsp(decay=0, fixed=FALSE, cutoff=30, type="OTP")` **(binary) (directed)** *Geometrically weighted non-edgewise shared partner distribution*: This term is just like `gwesp` and `gwdsp` except it adds a statistic equal to the geometrically weighted nonedgewise (that is, over dyads that do not have an edge) shared partner distribution with decay parameter `decay`, which should be non-negative. (this parameter was called `alpha` prior to `ergm` 3.7). The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used instead as the starting value for the estimation of decay in a curved exponential family model (when `fixed=FALSE`, the default) (see Hunter and Handcock, 2006).

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

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

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

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

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

By default, outgoing two-paths ("OTP") are calculated. Note that Robins et al. (2009) define closely related statistics to several of the above, using slightly different terminology.

The optional argument `cutoff` sets the number of underlying NSP terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm.](#))

`dnsp(d, type="OTP")` **(binary) (directed)** *Directed non-edgewise shared partners*: This term adds one network statistic to the model for each element in `d` where the i th such statistic equals the number of *non-edges* in the network with exactly `d[i]` shared partners. This term can only be used with directed networks.

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

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

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

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

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

By default, outgoing two-paths ("OTP") are calculated. Note that Robins et al. (2009) define closely related statistics to several of the above, using slightly different terminology.

`dsp(d)` **(binary) (directed) (undirected)** *Dyadwise shared partners*: The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in

d ; the i th such statistic equals the number of dyads in the network with exactly $d[i]$ shared partners. This term can be used with directed and undirected networks. For directed networks the count is over homogeneous shared partners only (i.e., only partners on a directed two-path connecting the nodes in the dyad).

`dyadcov(x, attrname=NULL)` **(binary) (dyad-independent) (directed) (undirected) (categorical nodal attribute)**

Dyadic covariate: 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 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 edge-wise 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 `edgescov` and `dyadcov` terms are equivalent for undirected networks.

`edgescov(x, attrname=NULL)` **(binary) (dyad-independent) (directed) (undirected) (frequently-used)**, `edgescov(x, attrname=)`

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 `edgescov` term applies to both directed and undirected networks. For undirected networks the covariates are also assumed to be undirected. The `edgescov` and `dyadcov` terms are equivalent for undirected networks.

`edges` **(binary) (valued) (dyad-independent) (directed) (undirected) (frequently-used)**, a.k.a **nonzero (valued) (directed)**

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

`esp(d)` **(binary) (directed) (undirected)** *Edgewise shared partners:* This is just like the `dsp` term, except this term adds one network statistic to the model for each element in d where the i th such statistic equals the number of *edges* (rather than dyads) in the network with exactly $d[i]$ shared partners. This term can be used with directed and undirected networks. For directed networks the count is over homogeneous shared partners only (i.e., only partners on a directed two-path connecting the nodes in the edge and in the same direction).

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

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

`gwb1degree(decay, fixed=FALSE, attr=NULL, cutoff=30, levels=NULL)` **(binary) (bipartite) (undirected) (curved)**

Geometrically weighted degree distribution for the first mode in a bipartite (aka two-mode) network: This term adds one network statistic to the model equal to the weighted degree distribution with decay controlled by the decay parameter, which should be non-negative, 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` sets the number of underlying degree terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm](#).)

If `attr` is specified (see [Specifying Vertex Attributes and Levels](#) for details) then separate degree statistics are calculated for nodes having each separate value of the attribute. This term can only be used with undirected bipartite networks.

`gwb2degree(decay, fixed=FALSE, attr=NULL, cutoff=30, levels=NULL)` **(binary) (bipartite) (undirected) (curved)**

Geometrically weighted degree distribution for the second mode in a bipartite (aka two-mode) network: This term adds one network statistic to the model equal to the weighted degree distribution with decay controlled by the which should be non-negative, 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` sets the number of underlying degree terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm](#).)

If `attr` is specified (see [Specifying Vertex Attributes and Levels](#) for details) then separate degree statistics are calculated for nodes having each separate value of the attribute. This term can only be used with undirected bipartite networks.

`gwdegree(decay, fixed=FALSE, attr=NULL, cutoff=30, levels=NULL)` **(binary) (undirected) (curved) (frequently)**

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 instead as the starting value for the estimation of decay in a curved exponential family model (when `fixed=FALSE`, the default) (see Hunter and Handcock, 2006).

The optional argument `cutoff` sets the number of underlying degree terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm](#).)

If `attr` is specified (see [Specifying Vertex Attributes and Levels](#) for details) then separate degree statistics are calculated for nodes having each separate value of the attribute. This term can only be used with undirected networks.

`gwdsp(decay=0, fixed=FALSE, cutoff=30)` **(binary) (directed) (undirected) (curved)** *Geometrically*

weighted dyadwise shared partner distribution: This term adds one network statistic to the model equal to the geometrically weighted dyadwise shared partner distribution with decay parameter decay parameter, which should be non-negative. The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used instead as the starting value for the

estimation of decay in a curved exponential family model (when `fixed=FALSE`, the default) (see Hunter and Handcock, 2006). 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` sets the number of underlying DSP terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm.](#))

`gwesp(decay=0, fixed=FALSE, cutoff=30)` **(binary) (frequently-used) (directed) (undirected) (curved)**

Geometrically weighted edgewise shared partner distribution: This term is just like `gwdsp` except it adds a statistic equal to the geometrically weighted *edgewise* (not *dyadwise*) shared partner distribution with decay parameter `decay`, which should be non-negative. The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used instead as the starting value for the estimation of decay in a curved exponential family model (when `fixed=FALSE`, the default) (see Hunter and Handcock, 2006). 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`

The optional argument `cutoff` sets the number of underlying ESP terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm.](#))

`gwidegree(decay, fixed=FALSE, attr=NULL, cutoff=30, levels=NULL)` **(binary) (directed) (curved)**

Geometrically weighted in-degree distribution: This term adds one network statistic to the model equal to the weighted in-degree distribution with decay parameter `decay`, which should be non-negative. (this parameter was called `alpha` prior to `ergm 3.7`). The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used instead as the starting value for the estimation of decay in a curved exponential family model (when `fixed=FALSE`, the default) (see Hunter and Handcock, 2006). This term can only be used with directed networks.

The optional argument `cutoff` sets the number of underlying degree terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm.](#))

If `attr` is specified (see [Specifying Vertex Attributes and Levels](#) for details) then separate degree statistics are calculated for nodes having each separate value of the attribute.

`gwensp(decay=0, fixed=FALSE, cutoff=30)` **(binary) (directed) (undirected) (curved)** *Geometrically*

weighted nonedgewise shared partner distribution: This term is just like `gwesp` and `gwdsp` except it adds a statistic equal to the geometrically weighted *nonedgewise* (that is, over dyads that do not have an edge) shared partner distribution with weight parameter `decay`, which should be non-negative. (this parameter was called `alpha` prior to `ergm 3.7`). The optional argument `fixed` indicates whether the decay parameter is fixed at the given value, or 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` sets the number of underlying NSP terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm](#).)

`gwodegree(decay, fixed=FALSE, attr=NULL, cutoff=30, levels=NULL)` **(binary) (directed) (curved)**

Geometrically weighted out-degree distribution: This term adds one network statistic to the model equal to the weighted out-degree distribution with decay parameter `decay`, which should be non-negative. (this parameter was called `alpha` prior to `ergm 3.7`). The value supplied for this parameter may be fixed (if `fixed=TRUE`), or it may be used instead as the starting value for the estimation of decay in a curved exponential family model (when `fixed=FALSE`, the default) (see Hunter and Handcock, 2006). This term can only be used with directed networks.

The optional argument `cutoff` sets the number of underlying degree terms to use in computing the statistics when `fixed=FALSE`, in order to reduce the computational burden. Its default value can also be controlled by the `gw.cutoff` term option control parameter. (See [control.ergm](#).)

If `attr` is specified (see [Specifying Vertex Attributes and Levels](#) for details) then separate degree statistics are calculated for nodes having each separate value of the attribute.

`hamming(x, cov, attrname=NULL)` **(binary) (dyad-independent) (directed) (undirected)** *Hamming*

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

`hammingmix(attr, x, base=NULL, levels=NULL, levels2=NULL)` **(binary) (directed) (dyad-independent)**

Hamming distance within mixing: By default, this term adds one statistic to the model for every possible pairing of attribute values of the network for the vertex attribute specified by `attr` (see [Specifying Vertex Attributes and Levels](#)). 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 by default, but this may be overridden using the `levels` arguments. The optional arguments `levels` and `levels2` allow the user to control what pairings are included (see [Specifying Vertex Attributes and Levels](#)), and the order in which they appear. For example `levels2=-2` will omit the second statistic, making it the de facto reference category.

The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels2` are passed, `levels2` overrides `base`.

This term can only be used with directed networks.

`idegrange(from, to=+Inf, by=NULL, homophily=FALSE, levels=NULL)` **(binary) (directed) (categorical nodal attri**

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 i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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, levels=NULL)` **(binary) (directed) (categorical nodal attribute) (frequently-used)**

In-degree: The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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`.

`idegree1.5` **(binary) (directed)** *In-degree to the 3/2 power:* This term adds one network statistic to the model equaling the sum over the actors of each actor's indegree 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.

`idegreepopularity` **(binary) (directed) (deprecated)** *In-degree popularity (deprecated):* see `idegree1.5`.

`ininterval(lower=-Inf, upper=+Inf, open=c(TRUE, TRUE))` **(valued) (directed) (undirected) (dyad-independent)**

Number of dyads whose values are in an interval Adds one statistic equaling to the number of dyads whose values are between `lower` and `upper`. Argument `open` is a logical vector of length 2 that controls whether the interval is open (exclusive) on the lower and on the upper end, respectively. `open` can also be specified as one of `"[]"`, `"(]"`, `"[""`, and `"())"`.

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

`isolates` **(binary) (directed) (undirected) (frequently-used)** *Isolates:* This term adds one statistic to the model equal to the number of isolates in the network. For an undirected network, an isolate is defined to be any node with degree zero. For a directed network, an isolate is any node with both in-degree and out-degree equal to zero.

`istar(k, attr=NULL, levels=NULL)` **(binary) (directed) (categorical nodal attribute)** *In-stars:*

The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The i th such statistic counts the number of distinct `k[i]`-instars in the network, where a k -instar is defined to be a node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $(O_j \rightarrow N)$ exist for $j = 1, \dots, k$. The optional argument `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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, attr=NULL, levels=NULL)` **(binary) (undirected) (categorical nodal attribute)** *k-Stars*:

The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The i th such statistic counts the number of distinct `k[i]`-stars in the network, where a k -star is defined to be a node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $\{N, O_i\}$ exist for $i = 1, \dots, k$. The optional argument `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). If this is specified then the count is over the number of k -stars where all nodes have the same value of the attribute. This term can only be used for undirected networks; for directed networks, see `istar`, `ostar`, `twopath` and `m2star`. Note that `kstar(1)` is equal to `edges`.

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

`localtriangle(x)` **(binary) (triad-related) (directed) (undirected)** *Triangles within neighborhoods*:

This term adds one statistic to the model equal to the number of triangles in the network between nodes “close to” each other. For an undirected network, a local triangle is defined to be any set of three edges between nodal pairs $\{(i, j), (j, k), (k, i)\}$ that are in the same neighborhood. For a directed network, a triangle is defined as any set of three edges $(i \rightarrow j), (j \rightarrow k)$ 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) (directed)** *Mixed 2-stars, a.k.a 2-paths*: This term adds one statistic to the model, equal to the number of mixed 2-stars in the network, where a mixed 2-star is a pair of distinct edges $(i \rightarrow j), (j \rightarrow k)$. 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) (dyad-independent) (directed) (undirected)** *Mean vertex degree*: This term adds one network statistic to the model equal to the average degree of a node. Note that this term is a constant multiple of both `edges` and `density`.

`mm(attrs, levels=NULL, levels2=NULL)` **(binary) (dyad-independent) (frequently-used) (directed) (undirected) (categorical)**

Mixing matrix cells and margins: `attrs` is a two-sided formula whose LHS gives the attribute or attribute function (see [Specifying Vertex Attributes and Levels](#)) for the rows of the mixing matrix and whose RHS gives that for its columns. A one-sided formula (e.g., $\sim A$) is symmetrized (e.g., $A \sim A$). `levels` similarly specifies the subset of rows and columns to be used. `levels2` can then be used to filter which specific cells of the matrix to include. A two-sided formula with a dot on one side calculates the margins of the mixing matrix, analogously to `nodefactor`, with $A \sim \cdot$ calculating the row/sender/b1 margins and $\cdot \sim A$ calculating the column/receiver/b2 margins.

`mutual(same=NULL, by=NULL, diff=FALSE, keep=NULL, levels=NULL)` **(binary) (directed) (frequently-used)**, *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 (see [Specifying Vertex Attributes and Levels](#) for details), 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 (again, see [Specifying Vertex Attributes and Levels](#)), 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 `levels` is passed, this tells which statistics should be kept whenever the mutual term would ordinarily result in multiple statistics (see [Specifying Vertex Attributes and Levels](#)).

The argument `keep` is retained for backwards compatibility and may be removed in a future version. When both `keep` and `levels` are passed, `levels` overrides `keep`.

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

nodecov(attr) (binary) (dyad-independent) (frequently-used) (directed) (undirected) (quantitative nodal attribute),

Main effect of a covariate: The `attr` argument specifies one or more quantitative attributes (see [Specifying Vertex Attributes and Levels](#) for details). This term adds a single network statistic for each quantitative attribute or matrix column to the model equaling the sum of `attr(i)` and `attr(j)` for all edges (i, j) in the network. For categorical attributes, see `nodefactor`. Note that for directed networks, `nodecov` equals `nodeicov` plus `nodeocov`.

Note that `ergm` versions 3.9.4 and earlier used different arguments for this term. See the above section on versioning for invoking the old behavior.

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

nodefactor(attr, base=1, levels=-1) (binary) (dyad-independent) (directed) (undirected) (categorical nodal attr)

Factor attribute effect: The `attr` argument specifies one or more categorical attributes (see [Specifying Vertex Attributes and Levels](#) for details). This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attr` 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.

The optional `levels` argument controls which levels of the attribute should be included and which should be excluded. (See [Specifying Vertex Attributes and Levels](#) for details.) For example, if the "fruit" attribute has levels "orange", "apple", "banana", and "pear", then to include just two levels, one for "apple" and one for "pear", use any of `nodefactor("fruit", levels=-(2:3))`, `nodefactor("fruit", levels=c(1,4))`, and `nodefactor("fruit", levels=c("apple", "pear"))`.

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. The default, `levels=-1`, is therefore to omit the first (in lexicographic order) attribute level. To include all levels, pass either `levels=TRUE` (i.e., keep all levels) or `levels=NULL` (i.e., do not filter levels).

The argument base is retained for backwards compatibility and may be removed in a future version. When both base and levels are passed, levels overrides base.

`nodeicov(attr)` **(binary) (directed) (quantitative nodal attribute) (frequently-used)**, `nodeicov(attr, form="sum")`

Main effect of a covariate for in-edges: The `attr` argument specifies one or more quantitative attributes (see [Specifying Vertex Attributes and Levels](#) for details). This term adds a single network statistic for each quantitative attribute or matrix column to the model equaling the total value of `attr(j)` for all edges (i, j) in the network. This term may only be used with directed networks. For categorical attributes, see `nodeifactor`.

Note that `ergm` versions 3.9.4 and earlier used different arguments for this term. See the above section on versioning for invoking the old behavior.

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

`nodeifactor(attr, base=1, levels=-1)` **(binary) (dyad-independent) (directed) (categorical nodal attribute) (frequently-used)**

Factor attribute effect for in-edges: The `attr` argument specifies one or more categorical attributes (see [Specifying Vertex Attributes and Levels](#) for details). This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attr` 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.

The optional `levels` argument controls which levels of the attribute should be included and which should be excluded. (See [Specifying Vertex Attributes and Levels](#) for details.) For example, if the “fruit” attribute has levels “orange”, “apple”, “banana”, and “pear”, then to include just two levels, one for “apple” and one for “pear”, use any of `nodeifactor("fruit", levels=-(2:3))`, `nodeifactor("fruit", levels=c(1,4))`, and `nodeifactor("fruit", levels=c("apple", "pear"))`.

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. The default, `levels=-1`, is therefore to omit the first (in lexicographic order) attribute level. To include all levels, pass either `levels=TRUE` (i.e., keep all levels) or `levels=NULL` (i.e., do not filter levels).

The argument base is retained for backwards compatibility and may be removed in a future version. When both base and levels are passed, levels overrides base.

For an analogous term for quantitative vertex attributes, see `nodeicov`.

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

`nodematch(attr, diff=FALSE, keep=NULL, levels=NULL)` **(binary) (dyad-independent) (frequently-used) (directed)**

Uniform homophily and differential homophily: The `attr` argument specifies one or more attributes (see [Specifying Vertex Attributes and Levels](#) for details). When `diff=FALSE`, this term adds one network statistic to the model, which counts the number of edges (i, j) for which `attr(i)==attr(j)`. This is also called “uniform homophily,” because each group is assumed to have the same propensity for within-group ties. When multiple attribute names are given, the statistic counts only ties for which all of the attributes match. When `diff=TRUE`, p network statistics are added to the model, where p is the number of unique values of the `attr` attribute. The k th such statistic counts the number of edges (i, j) for which `attr(i) ==`

`attr(j) == value(k)`, where `value(k)` is the k th smallest unique value of the `attr` attribute. This is also called "differential homophily," because each group is allowed to have a unique propensity for within-group ties. Note that a statistical test of uniform vs. differential homophily should be conducted using the ANOVA function.

By default, matches on all levels k are counted. The optional `levels` argument controls which levels of the attribute should be included and which should be excluded. (See [Specifying Vertex Attributes and Levels](#) for details.) For example, if the "fruit" attribute has levels "orange", "apple", "banana", and "pear", then to include just two levels, one for "apple" and one for "pear", use any of `nodematch("fruit", levels=-(2:3))`, `nodematch("fruit", levels=c(1,4))`, and `nodematch("fruit", levels=c("apple", "pear"))`. This works for both `diff=TRUE` and `diff=FALSE`.

The argument `keep` is retained for backwards compatibility and may be removed in a future version. When both `keep` and `levels` are passed, `levels` overrides `keep`.

`nodemix(attr, base=NULL, b1levels=NULL, b2levels=NULL, levels=NULL, levels2=NULL)` **(binary) (dyad-independent)**

Nodal attribute mixing: The `attr` argument specifies one or more categorical vertex attributes (see [Specifying Vertex Attributes and Levels](#) for details). 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 attributes are specified, a statistic is added for each combination of attribute values for those attributes.) In other words, this term produces one statistic for every entry in the mixing matrix for the attribute(s). By default, the ordering of the attribute values is lexicographic: alphabetical (for nominal categories) or numerical (for ordered categories), but this can be overridden using the `levels` arguments. The optional arguments `levels`, `levels2`, `b1levels`, and `b2levels` control what statistics are included in the model, and the order in which they appear. `levels2` apply to all networks; `levels` applies to unipartite networks; `b1levels` and `b2levels` apply to bipartite networks (see [Specifying Vertex Attributes and Levels](#)).

The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels2` are passed, `levels2` overrides `base`.

`nodecov(attr)` **(binary) (directed) (dyad-independent)(quantitative nodal attribute)**, `nodecov(attr, form="sum"`

Main effect of a covariate for out-edges: The `attr` argument specifies one or more quantitative attributes (see [Specifying Vertex Attributes and Levels](#) for details). This term adds a single network statistic for each quantitative attribute or matrix column to the model equaling the total value of `attr(i)` for all edges (i, j) in the network. This term may only be used with directed networks. For categorical attributes, see `nodeofactor`.

Note that `ergm` versions 3.9.4 and earlier used different arguments for this term. See the above section on versioning for invoking the old behavior.

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

`nodeofactor(attr, base=1, levels=-1)` **(binary) (dyad-independent) (directed) (categorical nodal attribute)**, `nodeofactor`

Factor attribute effect for out-edges: The `attr` argument specifies one or more categorical attributes (see [Specifying Vertex Attributes and Levels](#) for details). This term adds multiple network statistics to the model, one for each of (a subset of) the unique values of the `attr` 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.

The optional `levels` argument controls which levels of the attribute should be included and which should be excluded. (See [Specifying Vertex Attributes and Levels](#) for details.) For example, if the “fruit” attribute has levels “orange”, “apple”, “banana”, and “pear”, then to include just two levels, one for “apple” and one for “pear”, use any of `nodeofactor("fruit", levels=(2:3))`, `nodeofactor("fruit", levels=c(1,4))`, and `nodeofactor("fruit", levels=c("apple", "pear"))`.

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. The default, `levels=-1`, is therefore to omit the first (in lexicographic order) attribute level. To include all levels, pass either `levels=TRUE` (i.e., keep all levels) or `levels=NULL` (i.e., do not filter levels).

The argument `base` is retained for backwards compatibility and may be removed in a future version. When both `base` and `levels` are passed, `levels` overrides `base`.

This term can only be used with directed networks.

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

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

`nsp(d)` (**binary**) (**directed**) (**undirected**) *Nonedgewise shared partners*: This is just like the `dsp` and `esp` terms, except this term adds one network statistic to the model for each element in `d` where the i th 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, `levels`=NULL) (**binary**) (**directed**) (**categorical nodal attribute**) *Out-degree range*: The `from` and `to` arguments are vectors of distinct integers (or +Inf, for to (its default)). If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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, `levels`=NULL) (**binary**) (**directed**) (**categorical nodal attribute**) (**frequent**) *Out-degree*: The `d` argument is a vector of distinct integers. This term adds one network statis-

tic to the model for each element in d ; the i th 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` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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`.

`odegree1.5 (binary) (directed)` *Out-degree to the 3/2 power*: 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.

`odegreepopularity (binary) (directed) (deprecated)` *Out-degree popularity (deprecated)*: see `odegree1.5`.

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

`ostar(k, attr=NULL, levels=NULL) (binary) (directed) (categorical nodal attribute)` *k-Outstars*: The `k` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `k`. The i th such statistic counts the number of distinct $k[i]$ -outstars in the network, where a k -outstar is defined to be a node N and a set of k different nodes $\{O_1, \dots, O_k\}$ such that the ties $(N \rightarrow O_j)$ exist for $j = 1, \dots, k$. The optional argument `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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, nodes=-1) (binary) (directed) (dyad-independent)`, `receiver(base=1, nodes=-1, form="sum")` *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` and `nodes` arguments allow the user to determine which nodes' statistics should be included or excluded (see [Specifying Vertex Attributes and Levels](#) for details). The argument `nodes` is preferred to `base`, although `base` carries a default value of 1 for backwards compatibility. (If both `base` and `nodes` are supplied, then `nodes` overrides `base`.) This term can only be used with directed networks. For undirected networks, see `sociality`.

`sender(base=1, nodes=-1) (binary) (directed) (dyad-independent)`, `sender(base=1, nodes=-1, form="sum")` *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 `nodes` arguments allow the user to determine which nodes' statistics should be included or excluded (see [Specifying Vertex Attributes and Levels](#) for details).

The argument base is retained for backwards compatibility and may be removed in a future version. When both base and nodes are passed, nodes overrides base.

This term can only be used with directed networks. For undirected networks, see `sociality`.

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

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

`smalldiff`(attr, cutoff) (**binary**) (**dyad-independent**) (**directed**) (**undirected**) (**quantitative nodal attribute**)

Number of ties between actors with similar (but not necessarily identical) attribute values: The attr argument specifies a quantitative vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). 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 cutoff; that is, number of edges between i to j such that $\text{abs}(\text{attr}[i]-\text{attr}[j]) < \text{cutoff}$.

`sociality`(attr=NULL, base=1, levels=NULL, nodes=-1) (**binary**) (**undirected**) (**dyad-independent**) (**categorical nodal attribute**)

Undirected degree: This term adds one network statistic for each node equal to the number of ties of that node. The optional attr argument specifies a categorical vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). If provided, this term only counts ties between nodes with the same value of the attribute (an actor-specific version of the nodematch term), restricted to be one of the values specified by levels if levels is not NULL. This term can only be used with undirected networks. For directed networks, see `sender` and `receiver`. By default, nodes=-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 nodes argument of `sender` and `receiver` terms.

The argument base is retained for backwards compatibility and may be removed in a future version. When both base and nodes are passed, nodes overrides base.

`sum`(pow=1) (**valued**) (**directed**) (**undirected**) *Sum of dyad values (optionally taken to a power)*:

This term adds one statistic equal to the sum of dyad values taken to the power pow, which defaults to 1.

`threetrail`(keep=NULL, levels=NULL) (**binary**) (**directed**) (**undirected**) (**triad-related**), *Three-*

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

as a 3-trail of the form $i \rightarrow j \rightarrow i \leftarrow j$; however, in the subnetwork $i \leftrightarrow j \rightarrow k$, there are two directed 3-trails, one LRR ($k \leftarrow j \rightarrow i \leftarrow j$) and one RRR ($j \rightarrow i \rightarrow j \leftarrow k$).

The argument `keep` is retained for backwards compatibility and may be removed in a future version. When both `keep` and `levels` are passed, `levels` overrides `keep`. This term used to be (inaccurately) called `threepath`. That name has been deprecated and may be removed in a future version.

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

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

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

`triadcensus(levels) (binary) (triad-related) (directed) (undirected)` *Triad census*: For a directed network, this term adds one network statistic for each of an arbitrary subset of the 16 possible types of triads categorized by Davis and Leinhardt (1972) as 003, 012, 102, 021D, 021U, 021C, 111D, 111U, 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 using the `levels` argument (see [Specifying Vertex Attributes and Levels](#) for details), 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 `levels` argument.

`triangle(attr=NULL, diff=FALSE, levels=NULL) (binary) (frequently-used) (triad-related) (directed) (undirected)` *Triangles*: By default, this term adds one statistic to the model equal to the number of triangles in the network. For an undirected network, a triangle is defined to be any set $\{(i, j), (j, k), (k, i)\}$ of three edges. For a directed network, a triangle is defined as any set of three edges $(i \rightarrow j)$ and $(j \rightarrow k)$ and either $(k \rightarrow i)$ or $(k \leftarrow i)$. The former case is called a "transitive triple" and the latter is called a "cyclic triple", so in the case of a directed net-

work, `triangle` equals `ttriple` plus `ctriple` — thus at most two of these three terms can be in a model. The optional argument `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). If `attr` is specified and `diff` is `FALSE`, then the count is restricted to those triples of nodes with equal values of the vertex attribute specified by `attr`. If `attr` is specified and `diff` is `TRUE`, then one statistic is added for each value of `attr` (or each value specified by `levels` if that argument is passed), equal to the number of triangles where all three nodes have that value of the attribute.

`trippercent(attr=NULL, diff=FALSE, levels=NULL)` **(binary) (undirected) (triad-related) (categorical nodal attribute)**

Triangle percentage: By default, 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 `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). If `attr` is specified and `diff` is `FALSE`, the counts (both numerator and denominator) are restricted to those triples of nodes with equal values of the vertex attribute specified by `attr`. If `attr` is specified and `diff` is `TRUE`, then one statistic is added for each value of `attr` (or each value specified by `levels` if that argument is passed), where the counts (both numerator and denominator) are restricted to those triples of nodes with that value of the vertex attribute specified by `attr`. 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(attr=NULL, diff=FALSE, levels=NULL)` **(binary) (directed) (triad-related) (categorical nodal attribute)**, a

Transitive triples: By default, this term adds one statistic to the model, equal to the number of transitive triples in the network, defined as a set of edges $\{(i \rightarrow j), (j \rightarrow k), (i \rightarrow k)\}$. Note that `triangle` equals `ttriple`+`ctriple` for a directed network, so at most two of the three terms can be in a model. The optional argument `attr` specifies a vertex attribute (see [Specifying Vertex Attributes and Levels](#) for details). If `attr` is specified and `diff` is `FALSE`, then the count is over the number of transitive triples where all three nodes have the same value of the attribute. If `attr` is specified and `diff` is `TRUE`, then one statistic is added for each value of `attr` (or each value of `attr` specified by `levels` if that argument is passed), equal to the number of transitive triples where all three nodes have that value of `attr`. This term can only be used with directed networks.

`twopath` **(binary) (directed) (undirected) 2-Paths:** This term adds one statistic to the model, equal to the number of 2-paths in the network. For a directed network this is defined as a pair of edges $(i \rightarrow j), (j \rightarrow k)$, 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

- Bomirya, R. P, Bansal, S., and Hunter, D. R. (2014). Modeling Homophily in ERGMs for Bipartite Networks. Submitted.
- Butts, CT. (2008). “A Relational Event Framework for Social Action.” *Sociological Methodology*, 38(1).

- Davis, J.A. and Leinhardt, S. (1972). The Structure of Positive Interpersonal Relations in Small Groups. In J. Berger (Ed.), *Sociological Theories in Progress, Volume 2*, 218–251. Boston: Houghton Mifflin.
- Holland, P. W. and S. Leinhardt (1981). An exponential family of probability distributions for directed graphs. *Journal of the American Statistical Association*, 76: 33–50.
- Hunter, D. R. and M. S. Handcock (2006). Inference in curved exponential family models for networks. *Journal of Computational and Graphical Statistics*, 15: 565–583.
- Hunter, D. R. (2007). Curved exponential family models for social networks. *Social Networks*, 29: 216–230.
- Krackhardt, D. and Handcock, M. S. (2007). Heider versus Simmel: Emergent Features in Dynamic Structures. *Lecture Notes in Computer Science*, 4503, 14–27.
- Krivitsky P. N. (2012). Exponential-Family Random Graph Models for Valued Networks. *Electronic Journal of Statistics*, 2012, 6, 1100-1128. doi: [10.1214/12EJS696](https://doi.org/10.1214/12EJS696)
- Robins, G; Pattison, P; and Wang, P. (2009). “Closure, Connectivity, and Degree Distributions: Exponential Random Graph (p*) Models for Directed Social Networks.” *Social Networks*, 31:105-117.
- Snijders T. A. B., G. G. van de Bunt, and C. E. G. Steglich. Introduction to Stochastic Actor-Based Models for Network Dynamics. *Social Networks*, 2010, 32(1), 44-60. doi: [10.1016/j.socnet.2009.02.004](https://doi.org/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. <https://www.jstatsoft.org/v24/i04>
- Snijders, T. A. B., P. E. Pattison, G. L. Robins, and M. S. Handcock (2006). New specifications for exponential random graph models, *Sociological Methodology*, 36(1): 99-153.

See Also

[ergm](#) package, [search.ergmTerms](#), [ergm](#), [network](#), [%v%](#), [%n%](#)

Examples

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

ergm(molecule ~ edges + kstar(2:3) + triangle
      + nodematch("atomic type",diff=TRUE)
      + triangle + absdiff("atomic type"))

## End(Not run)
```

 ergm.allstats

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

Description

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

Usage

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

Arguments

formula	an formula object of the form $y \sim \langle \text{model terms} \rangle$, where y is a network object or a matrix that can be coerced to a network object. For the details on the possible $\langle \text{model terms} \rangle$, see ergm-terms . To create a network object in <code>R</code> , use the <code>network()</code> function, then add nodal attributes to it using the <code>%v%</code> 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)
system.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 output for this network.
# We know that the loglikelihood for my empty 7-node network
# should simply be -21*log(1+exp(eta)), so we may check that
# the following two values agree:
-21*log(1+exp(.1234))
ergm.exact(.1234, mynw~edges, statmat=a$statmat, weights=a$weights)
```

ergm.bounddeg

Initializes the parameters to bound degree during sampling

Description

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

Usage

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

Arguments

bounddeg	<p>a list of parameters which may contain the following for a network of size n nodes:</p> <ul style="list-style-type: none"> • <code>attribs</code>: an $n \times p$ matrix, where entry ij is TRUE if node i has attribute j, and FALSE otherwise; default=an $n \times 1$ matrix of 1's • <code>maxout</code> : an $n \times p$ matrix, where entry ij is the maximum number of out degrees for node i to nodes with attribute j; default=an $n \times p$ matrix of the value $(n-1)$ • <code>maxin</code> : defined similarly to <code>maxout</code>, but ignored for undirected networks; default=an $n \times p$ matrix of the value $(n-1)$ • <code>minout</code> : defined similarly to <code>maxout</code>; default=an $n \times p$ matrix of 0's • <code>minin</code> : defined similarly to <code>maxout</code>, but ignored for undirected networks; default=an $n \times p$ matrix of 0's
nw	the original network specified to <code>ergm</code> in 'formula'

Details

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

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

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

Value

a list of parameters used to bound degree during sampling

<code>condAllDegExact</code>	always FALSE
<code>attribs</code>	as defined above
<code>maxout</code>	as defined above
<code>maxin</code>	as defined above
<code>minout</code>	as defined above
<code>minin</code>	as defined above

See Also

[ergm-proposals](#)

ergm.bridge.llr	<i>Bridge sampling to evaluate ERGM log-likelihoods and log-likelihood ratios</i>
-----------------	---

Description

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

`ergm.bridge.0.llk` is a convenience wrapper that returns the log-likelihood of configuration θ relative to the reference measure. That is, the configuration with $\theta = 0$ is defined as having log-likelihood of 0.

`ergm.bridge.dindstart.llk` is a wrapper 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. Note that it only handles binary ERGMs (`response=NULL`) and with constraints (`constraints=`) that do not induce dyadic dependence.

Usage

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

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

```
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 <code>ergm</code> for details.
response	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to <code>NULL</code> for simple presence or absence, modeled via a binary ERGM.
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 <code>ergm</code> for more information.
from, to	The initial and final parameter vectors.
basis	An optional <code>network</code> object to start the Markov chain. If omitted, the default is the left-hand-side of the object.
verbose	Logical: If <code>TRUE</code> , print detailed information.
...	Further arguments to <code>ergm.bridge.llr</code> and <code>simulate.formula.ergm</code> .
llronly	Logical: If <code>TRUE</code> , only the estimated log-ratio will be returned by <code>ergm.bridge.llr</code> .

control	Control arguments. See control.ergm.bridge for details.
coef	A vector of coefficients for the configuration of interest.
llkonly	Whether only the estimated log-likelihood should be returned by the <code>ergm.bridge.0.llk</code> and <code>ergm.bridge.dindstart.llk</code> . (Defaults to TRUE.)
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 overall density term (edges) added if not redundant.
coef.dind	Parameter configuration for the dyad-independent starting point. Defaults to the MLE of dind.

Value

If `llronly=TRUE` or `llkonly=TRUE`, these functions return the scalar log-likelihood-ratio or the log-likelihood. Otherwise, they return a list with the following components:

llr	The estimated log-ratio.
llrs	The estimated log-ratios for each of the <code>nsteps</code> bridges.
path	A numeric matrix with <code>nsteps</code> rows, with each row being the respective bridge's parameter configuration.
stats	A numeric matrix with <code>nsteps</code> 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.

`ergm.bridge.0.llk` result list also includes an `llk` element, with the log-likelihood itself (with the reference distribution assumed to have likelihood 0).

`ergm.bridge.dindstart.llk` result list also includes an `llk` element, with the log-likelihood itself and an `llk.dind` element, with the log-likelihood of the nearest dyad-independent model.

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.degeneracy	<i>Checks an ergm Object for Degeneracy</i>
-----------------	---

Description

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

Usage

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

Arguments

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

Value

returns the original `ergm` object with 2 additional components:

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

 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 link{formula} object of the form $y \sim \langle \text{model terms} \rangle$, where y is a network object or a matrix that can be coerced to a network object. For the details on the possible $\langle \text{model terms} \rangle$, 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)
system.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 output 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.geodistdist

Calculate geodesic distance distribution for a network or edgelist

Description

ergm.geodistdist calculates geodesic distance distribution for a given [network](#) and returns it as a vector.

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

Usage

```

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

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

```

Arguments

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

Details

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

Value

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

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

See Also

See also the sna package [geodist](#) function

Examples

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

ergm.getnetwork	<i>Acquire and verify the network from the LHS of an ergm formula and verify that it is a valid network.</i>
-----------------	--

Description

The function function ensures that the network in a given formula is valid; if so, the network is returned; if not, execution is halted with warnings.

Usage

```
ergm.getnetwork(formula, loopswarning = TRUE)
```

Arguments

formula	a two-sided formula whose LHS is a network , an object that can be coerced to a network , or an expression that evaluates to one.
loopswarning	whether warnings about loops should be printed (TRUE or FALSE); defaults to TRUE.

Value

A [network](#) object constructed by evaluating the LHS of the model formula in the formula's environment.

ergm.godfather	<i>A function to apply a given series of changes to a network.</i>
----------------	--

Description

Gives the network a series of proposals it can't refuse. Returns the statistics of the network, and, optionally, the final network.

Usage

```
ergm.godfather(formula, changes = NULL, response = NULL,
  end.network = FALSE, stats.start = FALSE, changes.only = FALSE,
  verbose = FALSE, control = control.ergm.godfather())
```

Arguments

formula	An ergm -style formula, with a network on its LHS.
changes	Either a matrix with three columns: tail, head, and new value, describing the changes to be made; or a list of such matrices to apply these changes in a sequence. For binary network models, the third column may be omitted. In that case, the changes are treated as toggles. Note that if a list is passed, it must either be all of changes or all of toggles.
response	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.
end.network	Whether to return a network that results. Defaults to FALSE.
stats.start	Whether to return the network statistics at start (before any changes are applied) as the first row of the statistics matrix. Defaults to FALSE, to produce output similar to that of simulate for ERGMs when output="stats", where initial network's statistics are not returned.
changes.only	Whether to return network statistics or only their changes relative to the initial network.
verbose	Whether to print progress messages.
control	A control list generated by control.ergm.godfather .

Value

If `end.network==FALSE` (the default), an [mcmc](#) object with the requested network statistics associated with the network series produced by applying the specified changes. Its [mcmc](#) attributes encode the timing information: so `start(out)` gives the time point associated with the first row returned, and `end(out)` out the last. The "thinning interval" is always 1.

If `end.network==TRUE`, return a [network](#) object, representing the final network, with a matrix of statistics described in the previous paragraph attached to it as an `attr`-style attribute "stats".

See Also

`tergm::tergm.godfather()`, `simulate.ergm()`, `simulate.formula()`

Examples

```
data(florentine)
ergm.godfather(flomarriage~edges+absdiff("wealth")+triangles,
              changes=list(cbind(1:2,2:3),
                           cbind(3,5),
                           cbind(3,5),
                           cbind(1:2,2:3)),
              stats.start=TRUE)
```

ergmMPLE

ERG M 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

<code>formula</code>	An ERGM formula. See ergm .
<code>fitmodel</code>	Deprecated. Use <code>output="fit"</code> instead.
<code>output</code>	Character, partially matched. See Value .
<code>as.initialfit</code>	Logical. Specifies whether terms are initialized with argument <code>initialfit==TRUE</code> (the default). Generally, if <code>TRUE</code> , all curved ERGM terms will be treated as having their curved parameters fixed. See Example .
<code>control</code>	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.
<code>verbose</code>	Logical; if <code>TRUE</code> , the program will print out some additional information.
<code>...</code>	Additional arguments, to be passed to lower-level functions.

Details

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

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

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

Value

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

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

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

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

See Also

[ergm](#), [glm](#)

Examples

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

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

# Check that the coefficients agree with the output of the ergm function:
```

```

ergmMPLE(formula, output="fit")$coef

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

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

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

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

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

```

ergm_MCMC_sample

Internal Function to Sample Networks and Network Statistics

Description

This is an internal function, not normally called directly by the user. The `ergm_MCMC_sample` function samples networks and network statistics using an MCMC algorithm via `MCMC_wrapper` and is capable of running in multiple threads using `ergm_MCMC_slave`.

The `ergm_MCMC_slave` function calls the actual C routine and does minimal preprocessing.

Usage

```

ergm_MCMC_sample(nw, model, proposal, control, theta = NULL,
  response = NULL, update.nws = TRUE, verbose = FALSE, ...,
  eta = ergm.eta(theta, model$etamap))

```

```

ergm_MCMC_slave(Clist, proposal, eta, control, verbose, ...,
  prev.run = NULL, burnin = NULL, samplesize = NULL,
  interval = NULL, maxedges = NULL)

```

Arguments

<code>nw</code>	a network (or pending_update_network) object representing the sampler state.
<code>model</code>	an ergm_model to be sampled from, as returned by ergm_model() .
<code>proposal</code>	a list of the parameters needed for Metropolis-Hastings proposals and the result of calling ergm_proposal() .
<code>control</code>	list of MCMC tuning parameters; see control.ergm() .
<code>theta</code>	the (possibly curved) parameters of the model.

response	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.
update.nws	whether to actually update the network state or to return an object "promising" to update the network.
verbose	verbosity level.
...	additional arguments.
eta	the natural parameters of the model; by default constructed from theta.
Clist	the list of parameters returned by ergm.Cprepare
prev.run	a summary of the state of the sampler allowing a run to be resumed quickly by <code>ergm_MCMC_slave</code> .
burnin, samplesize, interval, maxedges	MCMC paramters that can be used to temporarily override those in the control list.

Details

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

Value

`ergm_MCMC_sample` returns a list containing:

stats	an <code>mcmc.list</code> with sampled statistics.
networks	a list of final sampled networks, one for each thread.
status	status code, propagated from <code>ergm.mcmcslave</code> .
final.interval	adaptively determined MCMC interval.

If `update.nws==FALSE`, rather than returning the updated networks, the function will return a [pending_update_network](#).

`ergm_MCMC_slave` returns the MCMC sample as a list of the following:

s	the matrix of statistics.
newntails	the vector of tails for the new network.
newnheads	the vector of heads for the new network.
newnweights	the vector of weights for the new network (if applicable)
status	success or failure code: 0 is success, 1 for too many edges, and 2 for a Metropolis-Hastings proposal failing.
maxedges	maximum allowed edges at the time of return.

Note

`ergm_MCMC_sample` and `ergm_MCMC_slave` replace `ergm.getMCMCsample` and `ergm.mcmcslave` respectively. They differ slightly in their argument names and in their return formats. For example, `ergm_MCMC_sample` expects `proposal` rather than `MHproposal` and `theta` or `eta` rather than `eta0`; and it does not return `statsmatrix` or `newnetwork` elements. Rather, if parallel processing is not in effect, `stats` is an `mcmc.list` with one chain and `networks` is a list with one element.

`ergm_plot.mcmc.list` *Plot MCMC list using lattice package graphics*

Description

Plot MCMC list using lattice package graphics

Usage

```
ergm_plot.mcmc.list(x, main = NULL, vars.per.page = 3, ...)
```

Arguments

<code>x</code>	an <code>mcmc.list</code> object containing the mcmc diagnostic samples.
<code>main</code>	character, main plot heading title.
<code>vars.per.page</code>	Number of rows (one variable per row) per plotting page. Ignored if <code>latticeExtra</code> package is not installed.
<code>...</code>	additional arguments, currently unused.

Note

This is not a method at this time.

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

Description

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

Explanation

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

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

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

Instructions

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

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

`faux.desert.high`

Faux desert High School as a network object

Description

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

Usage

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

Format

faux.desert.high is a `network` object with 107 vertices (students, in this case) and 439 directed edges (friendship nominations). To obtain additional summary information about it, type `summary(faux.desert.high)`.

The vertex attributes are Grade, Sex, and Race. The Grade attribute has values 7 through 12, indicating each student's grade in school. The Race attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes six possible values: White (non-Hisp.), Black (non-Hisp.), Hispanic, Asian (non-Hisp.), Native American, and Other (non-Hisp.)

Licenses and Citation

If the source of the data set does not specified otherwise, this data set is protected by the Creative Commons License <https://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* <https://statnet.org>.

Source

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

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

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

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

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

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

```
faux.desert.high <- simulate(desert.fit, nsim=1, burnin=1e+8,
constraint = "edges")
```

References

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

See Also

[network](#), [plot.network](#), [ergm](#), [faux.desert.high](#), [faux.mesa.high](#), [faux.magnolia.high](#)

faux.dixon.high

Faux dixon High School as a network object

Description

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

Usage

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

Format

faux.dixon.high is a [network](#) object with 248 vertices (students, in this case) and 1197 directed edges (friendship nominations). To obtain additional summary information about it, type `summary(faux.dixon.high)`.

The vertex attributes are Grade, Sex, and Race. The Grade attribute has values 7 through 12, indicating each student's grade in school. The Race attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes six possible values: White (non-Hisp.), Black (non-Hisp.), Hispanic, Asian (non-Hisp.), Native American, and Other (non-Hisp.)

Licenses and Citation

If the source of the data set does not specified otherwise, this data set is protected by the Creative Commons License <https://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* <https://statnet.org>.

Source

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

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

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

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

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

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

```
faux.dixon.high <- simulate(dixon.fit, nsim=1, burnin=1e+8,
constraint = "edges")
```

References

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

See Also

[network](#), [plot.network](#), [ergm](#), [faux.desert.high](#), [faux.mesa.high](#), [faux.magnolia.high](#)

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

Description

This data set represents a simulation of an in-school friendship network. The network is named `faux.magnolia.high` because the school communities 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 <https://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* <https://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 community on which it is based is in the rural western US, with a student body that is largely Hispanic and Native American.

Usage

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

Format

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

The vertex attributes are Grade, Sex, and Race. The Grade attribute has values 7 through 12, indicating each student's grade in school. The Race attribute is based on the answers to two questions, one on Hispanic identity and one on race, and takes six possible values: White (non-Hisp.), Black (non-Hisp.), Hispanic, Asian (non-Hisp.), Native American, and Other (non-Hisp.)

Licenses and Citation

If the source of the data set does not specified otherwise, this data set is protected by the Creative Commons License <https://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* <https://statnet.org>.

Source

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

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

The following [ergm](#) formula was used to fit a model to the original data:

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

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

The processes for handling missing data and defining the race attribute are described in Hunter, Goodreau & Handcock (2008).

References

Hunter D.R., Goodreau S.M. and Handcock M.S. (2008). *Goodness of Fit of Social Network Models*, *Journal of the American Statistical Association*.

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

See Also

[network](#), [plot.network](#), [ergm](#), [faux.magnolia.high](#)

fix.curved

Convert a curved ERGM into a corresponding "fixed" ERGM.

Description

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

Usage

```
fix.curved(object, ...)

## 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.
...	Unused at this time.
theta	Curved model parameter configuration.
response	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.

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(decay=.5, fixed=FALSE),
```

```

control=control.ergm(MCMLE.maxit=3, MCMC.burnin=1024, MCMC.interval=128))
summary(gest)
# A statistic for esp(1),...,esp(16)
simulate(gest,output="stats")

tmp<-fix.curved(gest)
tmp
# A gwesp() statistic only
simulate(tmp$formula, coef=tmp$theta, output="stats")

```

florentine	<i>Florentine Family Marriage and Business Ties Data as a "network" object</i>
------------	--

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.

Usage

```
data(florentine)
```

Details

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 Strozzi (15).

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.

Usage

```
data(g4)
```

Details

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

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	<i>Getting Started with "ergm": Fit, simulate and diagnose exponential-family models for networks</i>
-----------------	---

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*
<https://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: <https://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 <morris@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, <https://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). <https://www.jstatsoft.org/v24/i07/>.
- Besag, J., 1974, Spatial interaction and the statistical analysis of lattice systems (with discussion), *Journal of the Royal Statistical Society, B*, 36, 192-236.
- Boer P, Huisman M, Snijders T, Zeggelink E (2003). StOCNET: an open software system for the advanced statistical analysis of social networks. Groningen: ProGAMMA / ICS, version 1.4 edition.
- Butts CT (2007). **sna**: Tools for Social Network Analysis. R package version 2.3-2. <https://cran.r-project.org/package=sna>.
- Butts CT (2008). **network**: A Package for Managing Relational Data in R. *Journal of Statistical Software*, 24(2). <https://www.jstatsoft.org/v24/i02/>.
- Butts C (2015). **network**: The Statnet Project (<https://statnet.org>). R package version 1.12.0, <https://cran.r-project.org/package=network>.
- Frank, O., and Strauss, D.(1986). Markov graphs. *Journal of the American Statistical Association*, 81, 832-842.
- Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008a). A **statnet** Tutorial. *Journal of Statistical Software*, 24(8). <https://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. <https://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, <https://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, <https://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, <https://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). <https://www.jstatsoft.org/v24/i03/>.

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

geweke.diag.mv *Multivariate version of coda's* `coda::geweke.diag()`.

Description

Rather than comparing each mean independently, compares them jointly. Note that it returns an `htest` object, not a `geweke.diag` object.

Usage

```
geweke.diag.mv(x, frac1 = 0.1, frac2 = 0.5, split.mcmc.list = FALSE)
```

Arguments

`x` an `mcmc`, `mcmc.list`, or just a matrix with observations in rows and variables in columns.

`frac1`, `frac2` the fraction at the start and, respectively, at the end of the sample to compare.

`split.mcmc.list` when given an `mcmc.list`, whether to test each chain individually.

Value

An object of class `htest`, inheriting from that returned by `approx.hotelling.diff.test()`, but with p-value considered to be 0 on insufficient sample size.

Note

If `approx.hotelling.diff.test()` returns an error, then assume that burn-in is insufficient.

See Also

`coda::geweke.diag()`, `approx.hotelling.diff.test()`

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

```
gof(object, ...)
```

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

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

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

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

Arguments

object	Either a formula or an <code>ergm</code> object. See documentation for <code>ergm</code> .
...	Additional arguments, to be passed to lower-level functions.
coef	When given either a formula or an object of class <code>ergm</code> , <code>coef</code> are the parameters from which the sample is drawn. By default set to a vector of 0.
GOF	formula; an formula object, of the form <code>~ <model terms></code> 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 <code>formula</code> , 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 “dpartners”), the triad census (“triadcensus”), and the terms of the original model (“model”). The default formula for undirected networks is <code>~ degree + espartners + distance + model</code> , and the default formula for directed networks is <code>~ idegree + odegree + espartners + distance +</code>

	model. By default a “model” term is added to the formula. It is a very useful overall validity check and a reminder of the statistical variation in the estimates of the mean value parameters. To omit the “model” term, add “- model” to the formula.
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 <code>gof.formula</code> , defaults to unconstrained. For <code>gof.ergm</code> , defaults to the constraints with which object was fitted.
control	A list to control parameters, constructed using <code>control.gof.formula</code> or <code>control.gof.ergm</code> (which have different defaults).
verbose	Provide verbose information on the progress of the simulation.
unconditional	logical; if TRUE, the simulation is unconditional on the observed dyads. if not TRUE, the simulation is conditional on the observed dyads. This is primarily used internally when the network has missing data and a conditional GoF is produced.
x	an object of class <code>gof</code> for printing or plotting.
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.

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.

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

Value

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

Methods (by class)

- `ergm`: Perform simulation to evaluate goodness-of-fit for a specific `ergm()` fit.
- `formula`: Perform simulation to evaluate goodness-of-fit for a model configuration specified by a `formula`, coefficient, constraints, and other settings.
- `gof`: `print.gof` summarizes 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. (`summary.gof` is a deprecated alias that may be repurposed in the future.)

- `gof`: [plot.gof](#) 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.

Note

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`.

See Also

[ergm\(\)](#), [network\(\)](#), [simulate.ergm\(\)](#), [summary.ergm\(\)](#)

Examples

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

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

hamming

hamming (*disambiguation*)

Description

hamming may refer to:

- [An ERGM statistic](#) (`help("hamming-term")`)
- [A ERGM sample space constraint](#) (`help("hamming-constraint")`)

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

Description

These functions test whether an ERGM fit, formula, or some other object represents a curved exponential family.

The method for NULL always returns FALSE by convention.

Usage

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

Arguments

<code>object</code>	An ergm object or an ERGM formula.
<code>...</code>	Arguments passed on to lower-level functions.
<code>response</code>	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.
<code>basis</code>	See ergm() .

Details

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

Value

TRUE if the object represents a curved exponential family; FALSE otherwise.

is.durational

Testing for durational dependent models

Description

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

The method for NULL always returns FALSE by convention.

The method for character always returns FALSE by convention.

Usage

```
is.durational(object, ...)

## S3 method for class 'NULL'
is.durational(object, ...)

## S3 method for class 'character'
is.durational(object, ...)

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

Arguments

object	An ergm object or an ERGM formula, or some characters, e.g., object="all" for monitoring purpose.
...	Unused at this time.
response	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.
basis	See ergm() .

Value

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

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

Description

These functions test whether an ERGM fit, a formula, or some other object represents a dyad-independent model.

The method for NULL always returns FALSE by convention.

Usage

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

Arguments

object	The object to be tested for dyadic independence.
...	Unused at this time.
response	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to NULL for simple presence or absence, modeled via a binary ERGM.
basis	See ergm .
object.obs	For the ergm_conlist method, the observed data constraint.

Details

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

Value

TRUE if the model implied by the object is dyad-independent; FALSE otherwise.

is.inCH	<i>Determine whether a vector is in the closure of the convex hull of some sample of vectors</i>
---------	--

Description

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

Usage

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

Arguments

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

Details

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

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

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

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

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

Value

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

References

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

kapferer

Kapferer's tailor shop data

Description

This well-known social network dataset, collected by Bruce Kapferer in Zambia from June 1965 to August 1965, involves interactions among workers in a tailor shop as observed by Kapferer himself.

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`.

Details

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.

Source

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

logLik.ergm	A logLik method for ergm fits.
-------------	--

Description

A function to return the log-likelihood associated with an [ergm](#) fit, evaluating it if necessary. If the log-likelihood was not computed for object, produces an error unless `eval.loglik=TRUE`.

Usage

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

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.

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](#), [logLikNull](#), [ergm.bridge.llr](#), [ergm.bridge.dindstart.llk](#)

Examples

```

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

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

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

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

## End(Not run)

```

logLikNull

Calculate the null model likelihood

Description

Calculate the null model likelihood

Usage

```
logLikNull(object, ...)
```

```
## S3 method for class 'ergm'
```

```
logLikNull(object, control = control.logLik.ergm(), ...)
```

Arguments

object a fitted model.

... further arguments to lower-level functions.
logLikNull computes, when possible the log-probability of the data under the null model (reference distribution).

control A list of control parameters for algorithm tuning. Constructed using [control.logLik.ergm](#).

Value

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

Methods (by class)

- `ergm`: A method for `ergm` fits; currently only implemented for binary ERGMs with dyad-independent sample-space constraints.

<code>mcmc.diagnostics</code>	<i>Conduct MCMC diagnostics on a model fit</i>
-------------------------------	--

Description

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

Usage

```
mcmc.diagnostics(object, ...)

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

Arguments

<code>object</code>	A model fit object to be diagnosed.
<code>...</code>	Additional arguments, to be passed to plotting functions.
<code>center</code>	Logical: If TRUE, center the samples on the observed statistics.
<code>esteq</code>	Logical: If TRUE, for statistics corresponding to curved ERGM terms, summarize the curved statistics by their estimating equation values (evaluated at the MLE of any curved parameters) (i.e., $\eta'_I(\hat{\theta}) \cdot g_I(y)$ for I being indices of the canonical parameters in question), rather than the canonical (sufficient) vectors of the curved statistics ($g_I(y)$).
<code>vars.per.page</code>	Number of rows (one variable per row) per plotting page. Ignored if <code>latticeExtra</code> package is not installed.

Details

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

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

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

More information can be found by looking at the documentation of [ergm](#).

Value

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

Methods (by class)

- `ergm`:

References

Markov University of Washington, run with diagnostics: Implementation strategies for Markov chain Monte Carlo. *Statistical Science*, 7, 493-497.

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

This function is based on the `coda` package. It is based on the the R function `raftery.diag` in `coda`. `raftery.diag`, in turn, is based on the FORTRAN program `gibbsit` written by Steven Lewis which is available from the Statlib archive.

See Also

[ergm](#), network package, `coda` package, [summary.ergm](#)

Examples

```
## Not run:
#
data(florentine)
#
# test the mcmc.diagnostics function
#
gest <- ergm(flomarriage ~ edges + kstar(2))
summary(gest)

#
# Plot the probabilities first
#
mcmc.diagnostics(gest)
#
# Use coda directly
```

```

#
library(coda)
#
plot(gest$sample, ask=FALSE)
#
# A full range of diagnostics is available
# using codamenu()
#

## End(Not run)

```

molecule	<i>Synthetic network with 20 nodes and 28 edges</i>
----------	---

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

- reminiscent of a chemical molecule. It is stored as a [network](#) object.

Usage

```
data(molecule)
```

See Also

florentine, sampson, network, plot.network, ergm

network.list	<i>A convenience container for a list of network objects, output by simulate.ergm among others.</i>
--------------	---

Description

A convenience container for a list of [network](#) objects, output by [simulate.ergm](#) among others.

Usage

```

network.list(object, ...)

## S3 method for class 'network.list'
print(x, stats.print = FALSE, ...)

## S3 method for class 'network.list'
summary(object, stats.print = TRUE,
         net.print = FALSE, net.summary = FALSE, ...)

```

Arguments

object, x	a list of networks or a <code>network.list</code> object.
...	for <code>network.list</code> , additional attributes to be set on the network list; for others, arguments passed down to lower-level functions.
stats.print	Logical: If TRUE, print network statistics.
net.print	Logical: If TRUE, print network overviews.
net.summary	Logical: If TRUE, print network summaries.

Methods (by generic)

- print: A `print()` method for network lists.
- summary: A `summary()` method for network lists.

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)
```

node-attr

Specifying nodal attributes and their levels

Description

This document describes the ways to specify nodal attributes or functions of nodal attributes and which levels for categorical factors to include. For the helper functions to facilitate this, see [node-attr-api](#).

Details

Term nodal attribute arguments, typically called `attr`, `attrs`, `by`, or `on` are interpreted as follows:

a character string Extract the vertex attribute with this name.

a character vector of length > 1 Extract the vertex attributes and paste them together, separated by dots if the term expects categorical attributes and (typically) combine into a covariate matrix if it expects quantitative attributes.

a function The function is called on the LHS network, expected to return a vector or matrix of appropriate dimension. (Shorter vectors and matrix columns will be recycled as needed.)

a formula The expression on the RHS of the formula is evaluated in an environment of the vertex attributes of the network, expected to return a vector or matrix of appropriate dimension. (Shorter vectors and matrix columns will be recycled as needed.) Within this expression, the network itself accessible as either `.` or `.nw`. For example, `nodecov(~abs(Grade-mean(Grade))/network.size(.))` would return the absolute difference of each actor's "Grade" attribute from its network-wide mean, divided by the network size.

For categorical attributes, to select which levels are of interest and their ordering, use the argument `levels`. Selection of nodes (from the appropriate vector of nodal indices) is likewise handled as the selection of levels, using the argument `nodes`. These arguments are interpreted as follows:

an expression wrapped in `I()` Use the given list of levels as is.

a numeric or logical vector Used for indexing of a list of all possible levels (typically, unique values of the attribute) in default order (typically lexicographic), i.e., `sort(unique(attr))[levels]`. In particular, `levels=TRUE` will retain all levels. Negative values exclude. To specify numeric or logical levels literally, wrap in `I()`.

NULL Retain all possible levels; usually equivalent to passing `TRUE`.

a character vector Use as is.

a function The function is called on the list of unique values of the attribute, the values of the attribute themselves, and the network itself, depending on its arity. Its return value is interpreted as above.

a formula The expression on the RHS of the formula is evaluated in an environment in which the network itself is accessible as `.nw`, the list of unique values of the attribute as `.` or as `.levels`, and the attribute vector itself as `.attr`. Its return value is interpreted as above.

Note that `levels` or `nodes` often has a default that is sensible for the term in question.

Examples

```
data(faux.mesa.high)

# Activity by grade with a baseline grade excluded:
summary(faux.mesa.high~nodefactor(~Grade))
# Retain all levels:
summary(faux.mesa.high~nodefactor(~Grade, levels=TRUE)) # or levels=NULL

# Mixing between lower and upper grades:
summary(faux.mesa.high~mm(~Grade>=10))
```

```
# Mixing between grades 7 and 8 only:
summary(faux.mesa.high~mm("Grade", levels=I(c(7,8))))
# or
summary(faux.mesa.high~mm("Grade", levels=1:2))
# or using levels2 (see ? mm) to filter the combinations of levels,
summary(faux.mesa.high~mm("Grade",
  levels2=~sapply(.levels,
    function(l)
      l[[1]]%in%c(7,8) && l[[2]]%in%c(7,8))))
```

nparam	<i>Length of the parameter vector associated with an object or with its terms.</i>
--------	--

Description

This is a generic that returns the number of parameters associated with a model or a model fit.

Usage

```
nparam(object, ...)
```

Default S3 method:

```
nparam(object, ...)
```

S3 method for class 'ergm'

```
nparam(object, offset = NA, ...)
```

Arguments

object	An object for which number of parameters is defined.
...	Additional arguments to methods.
offset	If NA (the default), all model terms are counted; if TRUE, only offset terms are counted; and if FALSE, offset terms are skipped.

Methods (by class)

- default: By default, the length of the `coef()` vector is returned.
- ergm: A method to return the number of parameters of an `ergm` fit.

param_names	<i>Names of the parameters associated with an object.</i>
-------------	---

Description

This is a generic that returns a vector giving the names of the parameters associated with a model or a model fit.

Usage

```
param_names(object, ...)

## Default S3 method:
param_names(object, ...)
```

Arguments

object	An object for which parameter names are defined.
...	Additional arguments to methods.

Methods (by class)

- default: By default, the names of the `coef()` vector is returned.

print.summary.ergm	<i>Summarizing ERGM Model Fits</i>
--------------------	------------------------------------

Description

`summary` method for `ergm` fits.

Usage

```
## S3 method for class 'summary.ergm'
print(x, digits = max(3, getOption("digits") - 3),
      correlation = FALSE, covariance = FALSE,
      signif.stars = getOption("show.signif.stars"), eps.Pvalue = 1e-04,
      print.header = TRUE, print.formula = TRUE, print.fitinfo = TRUE,
      print.coefmat = TRUE, print.message = TRUE, print.deviiances = TRUE,
      print.drop = TRUE, print.offset = TRUE, print.degeneracy = TRUE,
      ...)

## S3 method for class 'ergm'
summary(object, ..., correlation = FALSE,
        covariance = FALSE, total.variation = TRUE)
```

Arguments

<code>x</code>	object of class <code>summary.ergm</code> returned by <code>summary.ergm()</code> .
<code>digits</code>	Significant digits for coefficients
<code>correlation</code>	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
<code>covariance</code>	logical; if TRUE, the covariance matrix of the estimated parameters is returned and printed.
<code>signif.stars</code>	whether to print dots and stars to signify statistical significance. See <code>print.summary.lm()</code> .
<code>eps.Pvalue</code>	p -values below this level will be printed as " <code><eps.Pvalue</code> ".
<code>print.header</code> , <code>print.formula</code> , <code>print.fitinfo</code> , <code>print.coefmat</code> , <code>print.message</code> , <code>print.deviations</code> , <code>print.d</code>	which components of the fit summary to print.
<code>...</code>	Arguments to <code>logLik.ergm</code>
<code>object</code>	an object of class "ergm", usually, a result of a call to <code>ergm</code> .
<code>total.variation</code>	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 variation is used. The p -values are based on this source of variation.

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`. Note that for backwards compatibility, it returns two coefficient tables: `$coefs` which does not contain the z-statistics and `$coefficients` which does (and is therefore more similar to those returned by `summary.lm()`).

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)
```

 samplk

Longitudinal networks of positive affection within a monastery as a "network" object

Description

Three [network](#) objects containing the "liking" nominations of Sampson's (1969) monks at the three time points.

Usage

```
data(samplk)
```

Details

Sampson (1969) recorded the social interactions among a group of monks while he was a resident as an experimenter at the cloister. During his stay, a political "crisis in the cloister" resulted in the expulsion of four monks—namely, the three "outcasts," Brothers Elias, Simplicius, Basil, and the leader of the "young Turks," Brother Gregory. Not long after Brother Gregory departed, all but one of the "young Turks" left voluntarily: Brothers John Bosco, Albert, Boniface, Hugh, and Mark. Then, all three of the "waverers" also left: First, Brothers Amand and Victor, then later Brother Romuald. Eventually, Brother Peter and Brother Winfrid also left, leaving only four of the original group.

Of particular interest are the data on positive affect relations ("liking," using the terminology later adopted by White et al. (1976)), in which each monk was asked if he had positive relations to each of the other monks. Each monk ranked only his top three choices (or four, in the case of ties) on "liking". Here, we consider a directed edge from monk A to monk B to exist if A nominated B among these top choices.

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 had entered the monastery near the end of the study but before the major conflict began. These three time points are labeled T2, T3, and T4 in Tables D5 through D16 in the appendices of Sampson's 1969 dissertation. and the corresponding network data sets are named `samplk1`, `samplk2`, and `samplk3`, respectively.

See also the data set [sampion](#) containing the time-aggregated graph `samplike`.

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

The data sets are stored as [network](#) objects with three vertex attributes:

group Groups of novices as classified by Sampson, that is, "Loyal", "Outcasts", and "Turks", but with a fourth group called the "Waverers" by White et al. (1975) that comprises two of the original Loyal opposition and one of the original Outcasts. See the [samplike](#) data set for the original classifications of these three waverers.

cloisterville An indicator of attendance in the minor seminary of "Cloisterville" before coming to the monastery.

vertex.names The given names of the novices. NB: These names have been corrected as of `ergm` version 3.6.1.

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), Fienberg, Meyer, and Wasserman (1981), and Hoff, Raftery, and Handcock (2002), among others. This is only a small piece of the data collected by Sampson.

This data set 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. The numbers indicate the ordering used in the original dissertation of Sampson (1969).

Mislabeling in Versions Prior to 3.6.1

In `ergm` versions 3.6.0 and earlier, The adjacency matrices of the `samplike`, `samplk1`, `samplk2`, and `samplk3` networks reflected the original Sampson (1969) ordering of the names even though the vertex labels used the name order of de Nooy, Mrvar, and Batagelj (2005). That is, in `ergm` version 3.6.0 and earlier, the vertices were mislabeled. The correct order is the same one given in Tables D5, D9, and D13 of Sampson (1969): John Bosco, Gregory, Basil, Peter, Bonaventure, Berthold, Mark, Victor, Ambrose, Romauld (Sampson uses both spellings "Romauld" and "Ramauld" in the dissertation), Louis, Winfrid, Amand, Hugh, Boniface, Albert, Elias, Simplicius. By contrast, the order given in `ergm` version 3.6.0 and earlier is: Ramuald, Bonaventure, Ambrose, Berthold, Peter, Louis, Victor, Winfrid, John Bosco, Gregory, Hugh, Boniface, Mark, Albert, Amand, Basil, Elias, Simplicius.

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

A [network](#) object containing the cumulative "liking" nominations of Sampson's (1969) monks over the three time points.

Usage

```
data(sampson)
```

Details

Sampson (1969) recorded the social interactions among a group of monks while he was a resident as an experimenter at the cloister. During his stay, a political "crisis in the cloister" resulted in the expulsion of four monks—namely, the three "outcasts," Brothers Elias, Simplicius, Basil, and the leader of the "young Turks," Brother Gregory. Not long after Brother Gregory departed, all but one of the "young Turks" left voluntarily: Brothers John Bosco, Albert, Boniface, Hugh, and Mark. Then, all three of the "waverers" also left: First, Brothers Amand and Victor, then later Brother Romuald. Eventually, Brother Peter and Brother Winfrid also left, leaving only four of the original group.

Of particular interest are the data on positive affect relations ("liking," using the terminology later adopted by White et al. (1976)), in which each monk was asked if he had positive relations to each of the other monks. Each monk ranked only his top three choices (or four, in the case of ties) on "liking". Here, we consider a directed edge from monk A to monk B to exist if A nominated B among these top choices.

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 had entered the monastery near the end of the study but before the major conflict began. These three time points are labeled T2, T3, and T4 in Tables D5 through D16 in the appendices of Sampson's 1969 dissertation. The `samplike` data set is the time-aggregated network. Thus, a tie from monk A to monk B exists if A nominated B as one of his three (or four, in case of ties) best friends at any of the three time points.

See also the data sets `samplk1`, `samplk2`, and `samplk3`, containing the networks at each of the three individual time points.

The data set is stored as a [network](#) object with three vertex attributes:

group Groups of novices as classified by Sampson: "Loyal", "Outcasts", and "Turks".

cloisterville An indicator of attendance in the minor seminary of "Cloisterville" before coming to the monastery.

vertex.names The given names of the novices. NB: These names have been corrected as of `ergm` version 3.6.1; see details below.

In addition, the data set has an edge attribute, `nominations`, giving the number of times (out of 3) that monk A nominated monk B.

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

This data set 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. The numbers indicate the ordering used in the original dissertation of Sampson (1969).

Mislabeling in Versions Prior to 3.6.1

In `ergm` version 3.6.0 and earlier, The adjacency matrices of the `samplike`, `samplk1`, `samplk2`, and `samplk3` networks reflected the original Sampson (1969) ordering of the names even though the vertex labels used the name order of de Nooy, Mrvar, and Batagelj (2005). That is, in `ergm` version 3.6.0 and earlier, the vertices were mislabeled. The correct order is the same one given in Tables D5, D9, and D13 of Sampson (1969): John Bosco, Gregory, Basil, Peter, Bonaventure, Berthold, Mark, Victor, Ambrose, Romauld (Sampson uses both spellings "Romauld" and "Ramauld" in the dissertation), Louis, Winfrid, Amand, Hugh, Boniface, Albert, Elias, Simplicius. By contrast, the order given in `ergm` version 3.6.0 and earlier is: Ramuald, Bonaventure, Ambrose, Berthold, Peter, Louis, Victor, Winfrid, John Bosco, Gregory, Hugh, Boniface, Mark, Albert, Amand, Basil, Elias, Simplicius.

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

```

san(object, ...)

## S3 method for class 'formula'
san(object, response = NULL, reference = ~Bernoulli,
     constraints = ~., target.stats = NULL, nsim = NULL, basis = NULL,
     output = c("network", "edgelist", "pending_update_network"),
     only.last = TRUE, control = control.san(), verbose = FALSE, ...)

## S3 method for class 'ergm_model'
san(object, response = NULL,
     reference = ~Bernoulli, constraints = ~., target.stats = NULL,
     nsim = NULL, basis = NULL, output = c("network", "edgelist",
     "pending_update_network"), only.last = TRUE, control = control.san(),
     verbose = FALSE, ...)

## S3 method for class 'ergm'
san(object, formula = object$formula,
     constraints = object$constraints, target.stats = object$target.stats,
     nsim = NULL, basis = NULL, output = c("network", "edgelist",
     "pending_update_network"), only.last = TRUE,
     control = object$control$SAN.control, verbose = FALSE, ...)

```

Arguments

object	Either a formula or an ergm object. The formula should be of the form $y \sim \langle \text{model terms} \rangle$, where y is a network object or a matrix that can be coerced to a network object. For the details on the possible $\langle \text{model terms} \rangle$, see ergm-terms . To create a network object in <code>R</code> , use the <code>network()</code> function, then add nodal attributes to it using the <code>%%</code> operator if necessary.
...	Further arguments passed to other functions.
response	Name of the edge attribute whose value is to be modeled. Defaults to NULL for simple presence or absence.
reference	One-sided formula whose RHS gives the reference measure to be used. (Defaults to <code>~Bernoulli</code> .)
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 <code>simulate.formula</code> , defaults to no constraints. For <code>simulate.ergm</code> , defaults to using the same constraints as those with which <code>object</code> was fitted.
target.stats	A vector of the same length as the number of terms implied by the formula, which is either <code>object</code> itself in the case of <code>san.formula</code> or <code>object\$formula</code> in the case of <code>san.ergm</code> .
nsim	Number of networks to generate. Deprecated: just use replicate() .
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.

output	Character, one of "network" (default), "edgelist", or "pending_update_network": determines the output format. Partial matching is performed.
only.last	if TRUE, only return the last network generated; otherwise, return a network.list with <code>nsim</code> networks.
control	A list of control parameters for algorithm tuning; see control.san .
verbose	Logical or numeric giving the level of verbosity. Higher values produce more verbose output.
formula	(By default, the formula is taken from the <code>ergm</code> object. If a different formula object is wanted, specify it here.

Value

A network or list of networks that hopefully have network statistics close to the `target.stats` vector.

Methods (by class)

- `formula`: Sufficient statistics are specified by a [formula](#).
- `ergm_model`: A lower-level function that expects a pre-initialized [ergm_model](#).
- `ergm`: Sufficient statistics and other settings are inherited from the [ergm](#) fit unless overridden.

Examples

```
# initialize x to a random undirected network with 50 nodes and a density of 0.1
x <- network(50, density = 0.05, directed = FALSE)

# try to find a network on 50 nodes with 300 edges, 150 triangles,
# and 1250 4-cycles, starting from the network x
y <- san(x ~ edges + triangles + cycle(4), target.stats = c(300, 150, 1250))

# check results
summary(y ~ edges + triangles + cycle(4))

# initialize x to a random directed network with 50 nodes
x <- network(50)

# add vertex attributes
x %v% 'give' <- runif(50, 0, 1)
x %v% 'take' <- runif(50, 0, 1)

# try to find a set of 100 directed edges making the outward sum of
# 'give' and the inward sum of 'take' both equal to 62.5, so in
# edges (i,j) the node i tends to have above average 'give' and j
# tends to have above average 'take'
y <- san(x ~ edges + nodecov('give') + nodecov('take'), target.stats = c(100, 62.5, 62.5))

# check results
summary(y ~ edges + nodecov('give') + nodecov('take'))
```

```

# initialize x to a random undirected network with 50 nodes
x <- network(50, directed = FALSE)

# add a vertex attribute
x %v% 'popularity' <- runif(50, 0, 1)

# try to find a set of 100 edges making the total sum of
# popularity(i) and popularity(j) over all edges (i,j) equal to
# 125, so nodes with higher popularity are more likely to be
# connected to other nodes
y <- san(x ~ edges + nodecov('popularity'), target.stats = c(100, 125))

# check results
summary(y ~ edges + nodecov('popularity'))

# creates a network with denser "core" spreading out to sparser
# "periphery"
plot(y)

```

search.ergmTerms

Search the ergm-terms documentation for appropriate terms

Description

Searches through the [ergm.terms](#) help page and prints out a list of terms appropriate for the specified network's structural constraints, optionally restricting by additional categories and keyword matches.

Usage

```
search.ergmTerms(keyword, net, categories, name)
```

Arguments

keyword	optional character keyword to search for in the text of the term descriptions. Only matching terms will be returned. Matching is case insensitive.
net	a network object that the term would be applied to, used as template to determine directedness, bipartite, etc
categories	optional character vector of category tags to use to restrict the results (i.e. 'curved', 'triad-related')
name	optional character name of a specific term to return

Details

Uses [grep](#) internally to match keywords against the term description, so keywords is currently matched as a single phrase. Category tags will only return a match if all of the specified tags are included in the term.

Value

prints out the name and short description of matching terms, and invisibly returns them as a list. If name is specified, prints out the full definition for the named term.

Author(s)

skyebend@uw.edu

See Also

See also [ergm.terms](#) for the complete documentation

Examples

```
# find all of the terms that mention triangles
search.ergmTerms('triangle')

# two ways to search for bipartite terms:

# search using a bipartite net as a template
myNet<-network.initialize(5,bipartite=3)
search.ergmTerms(net=myNet)

# or request the bipartite category
search.ergmTerms(categories='bipartite')

# search on multiple categories
search.ergmTerms(categories=c('bipartite','dyad-independent'))

# print out the content for a specific term
search.ergmTerms(name='b2factor')
```

simulate.ergm

Draw from the distribution of an Exponential Family Random Graph Model

Description

[simulate](#) is used to draw from exponential family random network models. See [ergm](#) for more information on these models.

The method for [ergm](#) objects inherits the model, the coefficients, the response attribute, the reference, the constraints, and most simulation parameters from the model fit, unless overridden by passing them explicitly. Unless overridden, the simulation is initialized with a random draw from the fitted model, saved by [ergm\(\)](#).

Usage

```
## S3 method for class 'formula'
simulate(object, nsim = 1, seed = NULL, coef,
  response = NULL, reference = ~Bernoulli, constraints = ~.,
  monitor = NULL, basis = NULL, statonly = FALSE, esteq = FALSE,
  output = c("network", "stats", "edgelist", "pending_update_network"),
  simplify = TRUE, sequential = TRUE,
  control = control.simulate.formula(), verbose = FALSE, ...,
  do.sim = TRUE)

## S3 method for class 'ergm_model'
simulate(object, nsim = 1, seed = NULL, coef,
  response = NULL, reference = ~Bernoulli, constraints = ~.,
  monitor = NULL, basis = NULL, esteq = FALSE,
  output = c("network", "stats", "edgelist", "pending_update_network"),
  simplify = TRUE, 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, basis = object$newnetwork, statonly = FALSE,
  esteq = FALSE, output = c("network", "stats", "edgelist",
  "pending_update_network"), simplify = TRUE, sequential = TRUE,
  control = control.simulate.ergm(), verbose = FALSE, ...)
```

Arguments

object	Either a formula or an ergm object. The formula should be of the form $y \sim \langle \text{model terms} \rangle$, where y is a network object or a matrix that can be coerced to a network object. For the details on the possible $\langle \text{model terms} \rangle$, see ergm-terms . To create a network object in <code>R</code> , use the <code>network()</code> function, then add nodal attributes to it using the <code>%v%</code> 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	Seed value (integer) for the random number generator. See set.seed .
coef	Vector of parameter values for the model from which the sample is to be drawn. If object is of class <code>ergm</code> , the default value is the vector of estimated coefficients.
response	Name of the edge attribute whose value is to be modeled in the valued ERGM framework. Defaults to <code>NULL</code> for simple presence or absence, modeled via a binary ERGM.
reference	A one-sided formula specifying the reference measure ($h(y)$) to be used. (Defaults to <code>~Bernoulli</code> .) 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 <code>ergm</code> and see list of implemented constraints for more information. For <code>simulate.formula</code> , defaults to no constraints. For <code>simulate.ergm</code> , 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 appended to the model, along with a coefficient of 0, so their statistics are returned. An <code>ergm_model</code> object can be passed as well.
basis	An optional <code>network</code> 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. Deprecated in favor of <code>output=</code> .
esteq	Logical: If TRUE, compute the sample estimating equations of an ERGM: if the model is non-curved, all non-offset statistics are returned either way, but if the model is curved, the score estimating function values (3.1) by Hunter and Handcock (2006) are returned instead.
output	Character, one of "network" (default), "stats", "edgelist", or "pending_update_network": determines the output format. Partial matching is performed.
simplify	Logical: If TRUE the output is "simplified": sampled networks are returned in a single list, statistics from multiple parallel chains are stacked, etc.. This makes it consistent with behavior prior to <code>ergm</code> 3.10.
sequential	Logical: If FALSE, each of the <code>nsim</code> 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 <code>nsim=1</code> .
control	A list of control parameters for algorithm tuning. Constructed using <code>control.simulate.ergm</code> or <code>control.simulate.formula</code> , 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.
do.sim	Logical: If FALSE, do not proceed to the simulation but rather return a list of arguments that would have been passed to <code>simulate.ergm_model()</code> . This can be useful if, for example, one wants to run several simulations with varying coefficients and did not want to reinitialize the model and the proposal every time.

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` 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 `output=="stats"` an [mcmc](#) object containing the simulated network statistics. If `control$parallel>0`, an [mcmc.list](#) object. If `simplify=TRUE` (the default), these would then be "stacked" and converted to a standard [matrix](#). A logical vector indicating whether or not the term had come from the `monitor=` formula is stored in `attr()`-style attribute "monitored".

Otherwise, a representation of the simulated network is returned, in the form specified by `output`. In addition to a network representation or a list thereof, they have the following `attr`-style attributes:

`formula` The [formula](#) used to generate the sample.

`stats` An [mcmc](#) or [mcmc.list](#) object as above.

`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.

The following are the permitted network formats:

"network" 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 above-listed additional attributes.

"edgelist" An [edgelist](#) representation of the network, or a list thereof, depending on `nsim`.

"pending_update_network" A semi-internal representation of a network consisting of a [network](#) object emptied of edges, with an attached `edgelist` matrix, or a list thereof, depending on `nsim`.

If `simplify==FALSE`, the networks are returned as a nested list, with outer list being the parallel chain (including 1 for no parallelism) and inner list being the samples within that chains (including 1, if one network per chain). If `TRUE`, they are concatenated, and if a total of one network had been simulated, the network itself will be returned.

Note

`simulate.ergm_model()` is a lower-level interface, providing a `simulate()` method for [ergm_model](#) class. The `basis` argument is required; `monitor`, if passed, must be an [ergm_model](#) as well; and `constraints` can be an [ergm_proposal](#) object instead.

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(florentine ~ edges + kstar(2))
summary(gest)
#
# Draw from the fitted model (statistics only), and observe the number
# of triangles as well.
#
g.sim <- simulate(gest, nsim=10,
                 monitor=~triangles, output="stats",
                 control=control.simulate.ergm(MCMC.burnin=1000, MCMC.interval=100))
g.sim

```

spectrum0.mvar	<i>Multivariate version of coda's spectrum0.ar().</i>
----------------	---

Description

Its return value, divided by `nrow(cbind(x))`, is the estimated variance-covariance matrix of the sampling distribution of the mean of `x` if `x` is a multivariate time series with $AR(p)$ structure, with p determined by AIC.

Usage

```
spectrum0.mvar(x, order.max = NULL, aic = is.null(order.max),
  tol = .Machine$double.eps^0.5, ...)
```

Arguments

<code>x</code>	a matrix with observations in rows and variables in columns.
<code>order.max</code>	maximum (or fixed) order for the AR model.
<code>aic</code>	use AIC to select the order (up to <code>order.max</code>).
<code>tol</code>	drop components until the reciprocal condition number of the transformed variance-covariance matrix is greater than this.
<code>...</code>	additional arguments to ar() .

Note

[ar\(\)](#) fails if `crossprod(x)` is singular, which is remedied by mapping the variables onto the principal components of `x`, dropping redundant dimensions.

summary.formula	<i>Calculation of network or graph statistics or other attributes specified on a formula</i>
-----------------	--

Description

Most generally, this function computes those summaries of the object on the LHS of the formula that are specified by its RHS. In particular, if given a network as its LHS and [ergm-terms](#) on its RHS, it computes the sufficient statistics associated with those terms.

Usage

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

Arguments

object A formula having as its LHS a [network](#) object or a matrix that can be coerced to a [network](#) object, a [network.list](#), or other types to be summarized using a formula. (See 'methods('summary_formula')' for the possible LHS types.)

... further arguments passed to or used by methods.

Details

In practice, `summary_formula()` is a thin wrapper around the `summary_formula()` generic, which dispatches methods based on the class of the LHS of the formula.

`summary_formula` for networks understands the [lasttoggle](#) "API".

Value

A vector of statistics specified in RHS of the formula.

See Also

[ergm\(\)](#), [network\(\)](#), [ergm-terms](#)

Examples

```
#
# Lets look at the Florentine marriage data
#
data(florentine)
#
# test the summary_formula 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
```

update.network

Update the edges in a network based on a matrix

Description

Replaces the edges in a [network](#) object with the edges corresponding to the sociomatrix or edge list specified by new.

Usage

```
## S3 method for class 'network'
update(object, new, matrix.type = NULL,
       attrname = NULL, ..., ignore.nattr = c("bipartite", "directed",
       "hyper", "loops", "mnext", "multiple", "n"), ignore.vattr = c())
```

Arguments

object	a network object.
new	Either an adjacency matrix (a matrix of values indicating the presence and/or the value of a tie from i to j) or an edge list (a two-column matrix listing origin and destination node numbers for each edge, with an optional third column for the value of the edge).
matrix.type	One of "adjacency" or "edgelist" telling which type of matrix new is. Default is to use the which.matrix.type function.
attrname	For a network with edge weights gives the name of the edge attribute whose names to set.
...	Additional arguments; currently unused.
ignore.nattr	Character vector of the names of network-level attributes to ignore when updating network objects (defaults to standard network properties).
ignore.vattr	Character vector of the names of vertex-level attributes to ignore when updating network objects.

Value

A new [network](#) object with the edges specified by new and network and vertex attributes copied from the input network object. Input network is not modified.

See Also

[ergm\(\)](#), [network](#)

Examples

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

```
update(flomarriage, rand.mat, matrix.type="edgelist")
```

wtd.median	<i>Weighted Median</i>
------------	------------------------

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

- *Topic **classes**
 - as.network.numeric, 10
- *Topic **datasets**
 - ecoli, 32
 - faux.desert.high, 93
 - faux.dixon.high, 95
 - faux.magnolia.high, 96
 - faux.mesa.high, 98
 - florentine, 101
 - g4, 102
 - kapferer, 113
 - molecule, 118
 - samplk, 124
 - sampson, 125
- *Topic **graphs**
 - as.network.numeric, 10
 - gof, 106
- *Topic **models**
 - anova.ergm, 6
 - control.ergm, 13
 - control.ergm.bridge, 22
 - control.logLik.ergm, 26
 - control.san, 27
 - control.simulate.ergm, 29
 - ergm, 35
 - ergm-constraints, 42
 - ergm-package, 4
 - ergm-references, 48
 - ergm-terms, 49
 - ergm.allstats, 78
 - ergm.exact, 84
 - ergmMPLE, 88
 - Getting.Started, 103
 - gof, 106
 - logLik.ergm, 114
 - mcmc.diagnostics, 116
 - print.summary.ergm, 122
 - san, 127
 - simulate.ergm, 131
 - summary.formula, 136
 - update.network, 137
- *Topic **model**
 - enformulate.curved, 33
 - ergm.bridge.llr, 81
 - fix.curved, 99
 - is.curved, 109
 - is.durational, 110
 - is.dyad.independent, 111
- *Topic **package**
 - ergm-package, 4
 - Getting.Started, 103
- *Topic **regression**
 - anova.ergm, 6
 - ergmMPLE, 88
 - print.summary.ergm, 122
- *Topic **robust**
 - wtd.median, 139
- %n%, 77
- %v%, 77
- A ERGM sample space constraint, 108
- absdiff (ergm-terms), 49
- absdiffcat (ergm-terms), 49
- altkstar (ergm-terms), 49
- An ERGM sample space constraint, 33
- An ERGM statistic, 33, 108
- anova, 7
- anova.ergm, 6
- anova.ergm.list, 7
- anova.ergm.list (anova.ergm), 6
- approx.hotelling.diff.test, 8
- approx.hotelling.diff.test(), 105
- ar(), 136
- as.edgelist, 9, 9, 10
- as.matrix.network, 9
- as.matrix.network.edgelist, 10
- as.network.numeric, 10, 10
- as.package_version, 50
- asymmetric (ergm-terms), 49

- atleast (ergm-terms), 49
- atmost (ergm-terms), 49
- attr, 134
- attr(), 134
- attrname (node-attr), 119
- attrs (node-attr), 119

- b1concurrent (ergm-terms), 49
- b1cov (ergm-terms), 49
- b1degrange (ergm-terms), 49
- b1degree (ergm-terms), 49
- b1degrees (ergm-constraints), 42
- b1factor (ergm-terms), 49
- b1mindegree (ergm-terms), 49
- b1nodematch (ergm-terms), 49
- b1sociality (ergm-terms), 49
- b1star (ergm-terms), 49
- b1starmix (ergm-terms), 49
- b1twestar (ergm-terms), 49
- b2concurrent (ergm-terms), 49
- b2cov (ergm-terms), 49
- b2degrange (ergm-terms), 49
- b2degree (ergm-terms), 49
- b2degrees (ergm-constraints), 42
- b2factor (ergm-terms), 49
- b2mindegree (ergm-terms), 49
- b2nodematch (ergm-terms), 49
- b2sociality (ergm-terms), 49
- b2star (ergm-terms), 49
- b2starmix (ergm-terms), 49
- b2twestar (ergm-terms), 49
- balance (ergm-terms), 49
- bd (ergm-constraints), 42
- Bernoulli (ergm-references), 48
- blockdiag (ergm-constraints), 42
- by (node-attr), 119

- check.ErgmTerm, 11
- cluster, 47
- coda::geweke.diag(), 105
- coef, 123
- coef(), 121, 122
- coef.ergm (ergm), 35
- coefficients.ergm (ergm), 35
- coincidence (ergm-terms), 49
- concurrent (ergm-terms), 49
- concurrentties (ergm-terms), 49
- constraints-ergm (ergm-constraints), 42
- constraints.ergm (ergm-constraints), 42

- control.ergm, 13, 25, 31, 36, 37, 39, 46, 61, 62, 64–66, 88
- control.ergm(), 90
- control.ergm.bridge, 21, 22, 82
- control.ergm.godfather, 23, 87
- control.gof, 22, 24, 31
- control.gof.ergm, 107
- control.gof.formula, 107
- control.logLik.ergm, 26, 114, 115
- control.san, 17, 27, 129
- control.simulate, 22, 25
- control.simulate
(control.simulate.ergm), 29
- control.simulate.ergm, 29, 133
- control.simulate.formula, 133
- control\$drop, 38
- control\$init.method, 14
- ctriad (ergm-terms), 49
- ctriple (ergm-terms), 49
- cycle (ergm-terms), 49
- cyclicalities (ergm-terms), 49
- cyclicalweights (ergm-terms), 49

- ddsp (ergm-terms), 49
- degcor (ergm-terms), 49
- degcrossprod (ergm-terms), 49
- degrange (ergm-terms), 49
- degree, 100
- degree (ergm-terms), 49
- degree1.5 (ergm-terms), 49
- degreedist, 31
- degreedist-constraint
(ergm-constraints), 42
- degreepopularity (ergm-terms), 49
- degrees (ergm-constraints), 42
- density (ergm-terms), 49
- desp (ergm-terms), 49
- dgwdsp (ergm-terms), 49
- dgwesp (ergm-terms), 49
- dgwnsp (ergm-terms), 49
- diff (ergm-terms), 49
- DiscUnif (ergm-references), 48
- dnsp (ergm-terms), 49
- download.packages, 93
- dsp (ergm-terms), 49
- dyadcov (ergm-terms), 49

- ecoli, 32
- ecoli1 (ecoli), 32

- ecoli2(ecoli), 32
- edgecov(ergm-terms), 49
- edgelist, 134
- edges, 33
- edges-constraint(ergm-constraints), 42
- edges-term(ergm-terms), 49
- end, 87
- enformulate.curved, 14, 33
- environment, 47
- equalto(ergm-terms), 49
- ergm, 4, 5, 7, 21–28, 30, 31, 33–35, 35, 36–39, 42, 43, 46–50, 77, 81, 83, 87–89, 92–100, 103, 106–111, 114, 116–118, 121–123, 128, 129, 131–134
- ERGM constraints, 36
- ERGM reference measures, 36, 132
- ergm(), 11, 14, 16, 17, 19, 21, 22, 107–110, 116, 131, 137, 138
- ergm-constraints, 42
- ergm-package, 4
- ergm-parallel, 46
- ergm-references, 48
- ergm-terms, 11, 49, 137
- ergm.allstats, 78, 84
- ergm.bounddeg, 79
- ergm.bridge.0.llk(ergm.bridge.llr), 81
- ergm.bridge.dindstart.llk, 23, 114
- ergm.bridge.dindstart.llk(ergm.bridge.llr), 81
- ergm.bridge.llr, 23, 81, 114
- ergm.constraints(ergm-constraints), 42
- ergm.count, 5
- ergm.Cprepare, 91
- ergm.degeneracy, 83
- ergm.exact, 78, 79, 84
- ergm.geodesicmatrix(ergm.geodistdist), 85
- ergm.geodistdist, 85
- ergm.geodistn(ergm.geodistdist), 85
- ergm.getCluster(ergm-parallel), 46
- ergm.getnetwork, 86
- ergm.godfather, 87
- ergm.godfather(), 23
- ergm.nodegeodesics(ergm.geodistdist), 85
- ergm.pairgeodesic(ergm.geodistdist), 85
- ergm.parallel(ergm-parallel), 46
- ergm.references(ergm-references), 48
- ergm.restartCluster(ergm-parallel), 46
- ergm.stopCluster(ergm-parallel), 46
- ergm.terms, 130, 131
- ergm.terms(ergm-terms), 49
- ergm.userterms, 5, 50, 92, 93
- ergm_conlist, 111
- ergm_MCMC_sample, 46, 90
- ergm_MCMC_slave(ergm_MCMC_sample), 90
- ergm_model, 90, 129, 133, 134
- ergm_model(), 90
- ergm_plot.mcmc.list, 92
- ergm_proposal, 134
- ergm_proposal(), 90
- ergmMPL, 40, 88
- esp, 100
- esp(ergm-terms), 49
- eut-upgrade, 92
- faux.desert.high, 93, 95, 96
- faux.dixon.high, 95
- faux.magnolia.high, 51, 95, 96, 96, 99
- faux.mesa.high, 51, 95, 96, 98, 98
- fauxhigh(faux.mesa.high), 98
- fix.curved, 99
- fixallbut(ergm-constraints), 42
- fixedas(ergm-constraints), 42
- flobusiness(florentine), 101
- flomarriage(florentine), 101
- florentine, 101
- formula, 35, 38, 78, 107, 128, 129, 132–134
- g4, 102
- geodist, 86
- Getting.Started, 103
- geweke.diag.mv, 105
- glm, 89
- gof, 22, 25, 31, 106, 106, 107
- gof.ergm, 5, 107
- gof.ergm(), 24
- gof.formula, 107
- greaterthan(ergm-terms), 49
- grep, 130
- gwb1degree(ergm-terms), 49
- gwb2degree(ergm-terms), 49
- gwdegree, 100
- gwdegree(ergm-terms), 49
- gwdsp(ergm-terms), 49
- gwesp, 100

- gwesp (ergm-terms), 49
- gwidegree (ergm-terms), 49
- gwnsp (ergm-terms), 49
- gwodegree (ergm-terms), 49
- hamming, 108
- hamming-constraint (ergm-constraints), 42
- hamming-term (ergm-terms), 49
- hammingmix (ergm-terms), 49
- I(), 120
- idegrange (ergm-terms), 49
- idegree (ergm-terms), 49
- idegree1.5 (ergm-terms), 49
- idegreedist (ergm-constraints), 42
- idegreepopularity (ergm-terms), 49
- idegrees (ergm-constraints), 42
- ininterval (ergm-terms), 49
- InitErgmTerm, 12
- InitErgmTerm (ergm-terms), 49
- intransitive (ergm-terms), 49
- is.curved, 109
- is.durational, 110
- is.dyad.independent, 111
- is.ergm (ergm), 35
- is.inCH, 112
- isolates (ergm-terms), 49
- istar (ergm-terms), 49
- istar(2), 70
- kapferer, 113
- kapferer2 (kapferer), 113
- kstar (ergm-terms), 49
- kstar(2), 69, 72
- lasttoggle, 137
- list of implemented constraints, 128, 133
- localtriangle (ergm-terms), 49
- logLik, 114, 116
- logLik.ergm, 7, 27, 114, 123
- logLikNull, 114, 115
- m2star (ergm-terms), 49
- match (ergm-terms), 49
- matrix, 134
- mcmc, 87, 105, 134
- mcmc.diagnostics, 5, 116
- mcmc.diagnostics.ergm, 117
- mcmc.list, 9, 91, 92, 105, 134
- meandeg (ergm-terms), 49
- message(), 12
- mm (ergm-terms), 49
- molecule, 118
- mutual (ergm-terms), 49
- nearsimmelian (ergm-terms), 49
- network, 5, 10, 11, 32, 35, 49, 77, 78, 81, 84–87, 90, 94–99, 101–103, 118, 124, 126, 128, 132–134, 137, 138
- network(), 108, 137
- network.list, 118, 129, 134, 137
- nobs.ergm (ergm), 35
- node-attr, 119
- nodecov (ergm-terms), 49
- nodecovar (ergm-terms), 49
- nodedegrees (ergm-constraints), 42
- nodefactor (ergm-terms), 49
- nodeicov (ergm-terms), 49
- nodeicovar (ergm-terms), 49
- nodeifactor (ergm-terms), 49
- nodeisqrtcovar (ergm-terms), 49
- nodemain (ergm-terms), 49
- nodematch (ergm-terms), 49
- nodemix (ergm-terms), 49
- nodeocov (ergm-terms), 49
- nodeocovar (ergm-terms), 49
- nodeofactor (ergm-terms), 49
- nodeosqrtcovar (ergm-terms), 49
- nodesqrtcovar (ergm-terms), 49
- nonzero (ergm-terms), 49
- nparam, 121
- nsp (ergm-terms), 49
- nthreads (ergm-parallel), 46
- NULL, 46, 120
- observed (ergm-constraints), 42
- odegrange (ergm-terms), 49
- odegree (ergm-terms), 49
- odegree1.5 (ergm-terms), 49
- odegreedist (ergm-constraints), 42
- odegreepopularity (ergm-terms), 49
- odegrees (ergm-constraints), 42
- on (node-attr), 119
- opentriad (ergm-terms), 49
- ostar (ergm-terms), 49
- ostar(2), 71, 72

- parallel (ergm-parallel), 46
- parallel processing, 21, 23, 25, 27, 28, 30
- parallel-ergm (ergm-parallel), 46
- parallel.ergm (ergm-parallel), 46
- param_names, 122
- pending_update_network, 90, 91
- plot.gof, 107, 108
- plot.gof (gof), 106
- plot.network, 95, 96, 98, 99
- print(), 37, 119
- print.ergm, 38, 40
- print.ergm (ergm), 35
- print.gof, 107
- print.gof (gof), 106
- print.htest(), 9
- print.network.list (network.list), 118
- print.summary.ergm, 122
- print.summary.lm(), 123

- receiver (ergm-terms), 49
- references-ergm (ergm-references), 48
- references.ergm (ergm-references), 48
- replicate(), 128

- samplike, 124, 125, 127
- samplike (sampson), 125
- samplk, 124
- samplk1, 125–127
- samplk1 (samplk), 124
- samplk2, 125–127
- samplk2 (samplk), 124
- samplk3, 125–127
- samplk3 (samplk), 124
- sampson, 124, 125
- san, 17, 29, 127
- search.ergmTerms, 49, 50, 77, 130
- sender (ergm-terms), 49
- set.seed, 21, 23, 25, 27, 28, 132
- simmelian (ergm-terms), 49
- simmelianities (ergm-terms), 49
- simulate, 31, 49, 87, 131
- simulate(), 134
- simulate.ergm, 5, 22, 25, 31, 34, 100, 118, 119, 131
- simulate.ergm(), 29, 88, 108
- simulate.ergm_model (simulate.ergm), 131
- simulate.ergm_model(), 133, 134
- simulate.formula, 31
- simulate.formula (simulate.ergm), 131

- simulate.formula(), 88
- simulate.formula.ergm, 81, 82
- smalldiff (ergm-terms), 49
- smallerthan (ergm-terms), 49
- sna, 67, 75
- sociality (ergm-terms), 49
- Specifying Vertex Attributes and Levels, 50–60, 64–76
- spectrum0.ar(), 136
- spectrum0.mvar, 136
- sprintf(), 15, 19
- start, 87
- StdNormal (ergm-references), 48
- sum (ergm-terms), 49
- summary, 122, 123
- summary (summary.formula), 136
- summary(), 119
- summary.ergm, 38, 40, 117, 123
- summary.ergm (print.summary.ergm), 122
- summary.ergm(), 108, 123
- summary.formula, 136
- summary.formula(), 137
- summary.lm(), 123
- summary.network.list (network.list), 118
- summary_formula(), 137

- t.test(), 9
- tailor (kapferer), 113
- tergm, 5
- tergm::tergm.godfather(), 88
- terms-ergm (ergm-terms), 49
- terms.ergm (ergm-terms), 49
- the ERGM sample space constraint with that name, 31
- threepath (ergm-terms), 49
- threetrail (ergm-terms), 49
- transitive (ergm-terms), 49
- transitivities (ergm-terms), 49
- transitiveweights (ergm-terms), 49
- triad.classify, 67, 75
- triadcensus (ergm-terms), 49
- triangle (ergm-terms), 49
- triangles (ergm-terms), 49
- tripercnt (ergm-terms), 49
- ttriad (ergm-terms), 49
- ttriple (ergm-terms), 49
- twopath (ergm-terms), 49

- Unif (ergm-references), 48

`update.network`, [137](#)

`vcov.ergm (ergm)`, [35](#)

`warning()`, [12](#)

`which.matrix.type`, [138](#)

`wtd.median`, [139](#)