Package ‘tergm’

July 28, 2021

Version 4.0.2

Date 2021-07-28

Title Fit, Simulate and Diagnose Models for Network Evolution Based on Exponential-Family Random Graph Models

Depends ergm (>= 4.1.0),
network (>= 1.15),
networkDynamic (>= 0.10.0)

Imports methods,
utils,
robustbase (>= 0.93.5),
coda (>= 0.19.2),
nlme (>= 3.1.139),
MASS (>= 7.3.51.4),
statnet.common (>= 4.4.0),
purrr

LinkingTo ergm

Suggests lattice (>= 0.20.38),
parallel,
rmarkdown (>= 1.12),
knitr (>= 1.22),
tibble,
testthat,
covr

BugReports `https://github.com/statnet/tergm/issues`

Description An integrated set of extensions to the 'ergm' package to analyze and simulate network evolution based on exponential-family random graph models (ERGM). 'tergm' is a part of the 'statnet' suite of packages for network analysis. See Krivitsky and Handcock (2014) <doi:10.1111/rssb.12014> and Carnegie, Krivitsky, Hunter, and Goodreau (2015) <doi:10.1080/10618600.2014.903087>.

License GPL-3 + file LICENSE

URL `https://statnet.org`

VignetteBuilder rmarkdown, knitr

RoxygenNote 7.1.1

Roxygen list(markdown = TRUE)

Encoding UTF-8

R topics documented:

`tergm-package` *Fit, Simulate and Diagnose Dynamic Network Models derived from Exponential-Family Random Graph Models*

Description

`tergm` is a collection of extensions to the `ergm` package to fit, diagnose, and simulate models for dynamic networks — networks that evolve over time — based on exponential-family random graph models (ERGMs). For a list of functions type `help(package='tergm')`

Details

When publishing results obtained using this package, please cite the original authors as described in `citation(package="tergm")`.

All programs derived from this package must cite it.

An exponential-family random graph model (ERGM) postulates an exponential family over the sample space of networks of interest, and `ergm` package implements a suite of tools for modeling single networks using ERGMs.

There have been a number of extensions of ERGMs for modeling the evolution of networks, including the temporal ERGM (TERGM) of Hanneke et al. (2010) and the separable termporal ERGM (STERGM) of Krivitsky and Handcock (2014). The latter model allows familiar ERGM terms and statistics to be reused in a dynamic context, interpreted in terms of formation and dissolution (persistence) of ties. Krivitsky (2012) suggested a method for fitting dynamic models when only a cross-sectional network is available, provided some temporal information for it is available as well.

This package aims to implement these and other ERGM-based models for network evoluation. At this time, it implements, via the `tergm` function, a general framework for modeling tie dynamics in temporal networks with flexible model specification (including (S)TERGMs). Estimation options include a conditional MLE (CMLE) approach for fitting to a series of networks and an Equilibrium Generalized Method of Moments Estimation (EGMME) for fitting to a single network with temporal information. For further development, see the referenced papers.

If you previously used the `stergm()` function in this package, please note that `stergm()` has been superceded by the new `tergm` function, and is likely to be deprecated in future releases. The `dissolution` formula in `stergm()` maps to the new `Persist()` operator in the `tergm()` function, **not** the `Diss()` operator.

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

References

Hanneke S, Fu W and Xing EP (2010). Discrete Temporal Models of Social Networks. *Electronic Journal of Statistics*, 2010, 4, 585-605. doi: 10.1214/09EJS548

Krackhardt, D and Handcock, MS (2006) Heider vs Simmel: Emergent features in dynamic structures. ICML Workshop on Statistical Network Analysis. Springer, Berlin, Heidelberg, 2006.

Krivitsky PN & Handcock MS (2014) A Separable Model for Dynamic Networks. *Journal of the Royal Statistical Society, Series B*, 76(1): 29-46. doi: 10.1111/rssb.12014

Krivitsky, PN (2012). Modeling of Dynamic Networks based on Egocentric Data with Durational Information. *Pennsylvania State University Department of Statistics Technical Report*, 2012(2012-01). `https://web.archive.org/web/20170830053722/https://stat.psu.edu/research/technical-report-files` `2012-technical-reports/TR1201A.pdf`

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

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*, 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/`.

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/`.

`.extract.fd.formulae` *An Internal Function for Extracting (Some) Formation and Dissolution Formulas from a Combined Formula*

Description

This function is used in `tergm.EGMME.initialfit` and also when targets or monitoring formulas are specified by characters. It makes a basic attempt to identify the formation and dissolution formulas within a larger combined formula (which may also include non-separable terms). Instances of `Form` at the top level (which may occur inside `offset`) contribute to the formation formula; instances of `Persist` and `Diss` at the top level (which may also occur inside `offset`) contribute to the dissolution formula. All other terms are regarded as non-separable; this includes instances of `Form`, `Persist`, and `Diss` that occur inside other operator terms, including inside `Offset`, and also includes all interactions at the top level (for which the top level term is effectively the interaction operator `*` or `:`), whether or not they include `Form`, `Persist`, and/or `Diss`. The formation and dissolution formulas are obtained by adding the contributing terms, replacing `Form` and `Persist` with trivial operators that protect the environments of their formula arguments but have no effect on statistics or coefficient names (meaning the formulas effectively become cross-sectional), and replacing `Diss` by a similar operator that negates statistics. These are included in the return value as the `form` and `pers` elements of the list (the "dissolution" formula really being the persistence formula), which also includes the formula of non-separable terms as `nonsep`, and the formula of all terms after replacing `Form`, `Persist`, and `Diss` as described above as `all`.

If usage proves problematic, one may specify the monitoring and/or targets formulas explicitly (rather than by characters), and one may pass initial coefficient values for the EGMME to avoid running `tergm.EGMME.initialfit`.

Usage

```
.extract.fd.formulae(formula)
```

Arguments

`formula`	a `formula`.

Value

A `list` containing `form`, `pers`, `nonsep`, and `all` formulas as described above.

combine_networks — *A single block-diagonal network created by combining multiple networks*

Description

Given a list of compatible networks, the `combine_networks()` returns a single block-diagonal network, preserving attributes that can be preserved.

Usage

```
combine_networks(
  nwl,
  ignore.nattr = c("mnext"),
  ignore.vattr = c(),
  ignore.eattr = c(),
  blockID.vattr = ".NetworkID",
  blockName.vattr = NULL,
  detect.edgecov = FALSE,
  keep.unshared.attr = FALSE,
  subnet.cache = FALSE
)

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

## S3 method for class 'combined_networks'
summary(object, ...)

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

Arguments

`nwl`	a list of `network::network`s to be combined; they must have similar fundamental properties: directedness and bipartedness, though their sizes (and the size of each bipartite group) can vary.
`ignore.nattr, ignore.vattr, ignore.eattr`	network, vertex, and edge attributes not to be processed as described below.
`blockID.vattr`	name of the vertex attribute into which to store the index of the network to which that vertex originally belonged.
`blockName.vattr`	if not `NULL`, the name of the vertex attribute into which to store the name of the network to which that vertex originally belonged.
`detect.edgecov`	if `TRUE`, combine network attributes that look like dyadic covariate (`ergm::edgecov`) matrices into a block-diagonal matrix.

`keep.unshared.attr`	whether to keep those network, vertex, and edge attributes not shared by all networks in the list; if `TRUE`, positions corresponding to networks lacking the attribute are replaced with `NA`, `NULL`, or some other placeholder; incompatible with `detect.edgecov==TRUE`.
`subnet.cache`	whether to save the input network list as an attribute of the combined network, so that if the network is resplit using on the same attribute (e.g. using `uncombine_network()`), an expensive call to `split.network()` can be avoided, at the cost of storage.
`x, object`	a combined network.
`...`	additional arguments to methods.

Value

an object of class `combined_networks` inheriting from `network::network` with a block-diagonal structure (or its bipartite equivalent) comprising the networks passed in `nwl`. In particular,

- the returned network's size is the sum of the input networks';
- its basic properties (directedness and bipartednes) are the same;
- the input networks' sociomatrices (both edge presence and edge attributes) are the blocks in the sociomatrix of the returned network;
- vertex attributes are concatenated;
- edge attributes are assigned to their respective edges in the returned network;
- network attributes are stored in a list; but if `detect.edgecov==TRUE`, those network attributes that have the same dimension as the sociomatrices of the constituent networks, they are combined into a single block-diagonal matrix that is then stored as that attribute.

In addition, a two new vertex attibutes, specified by `blockID.vattr` and (optionally) `blockName.vattr` contain, respectively, the index in `nwl` of the network from which that vertex came and its name, determined as follows:

1. If `nwl` is a named list, names from the list are used.
2. If not 1, but the network has an attribute `title`, it is used.
3. Otherwise, a numerical index is used.

If `blockID.vattr` already exists on the constituent networks, the index is *prepended* to the attribute.

Methods (by generic)

- `print`: A wrapper around `network::print.network()` to print constituent network information and omit some internal variables.
- `summary`: A wrapper around `network::summary.network()` to print constituent network information and omit some internal variables.
- `print`: A wrapper around `network::print.summary.network()` to print constituent network information and omit some internal variables.

Examples

```
data(samplk)

o1 <- combine_networks(list(samplk1, samplk2, samplk3))
image(as.matrix(o1))
head(get.vertex.attribute(o1, ".NetworkID"))
o2 <- combine_networks(list(o1, o1))
image(as.matrix(o2))
head(get.vertex.attribute(o2, ".NetworkID", unlist=FALSE))

data(florentine)
f1 <- combine_networks(list(business=flobusiness, marriage=flomarriage),
                       blockName.vattr=".NetworkName")
image(as.matrix(f1))
head(get.vertex.attribute(f1, ".NetworkID"))
head(get.vertex.attribute(f1, ".NetworkName"))
```

`control.simulate.network` *Auxiliary for Controlling Separable Temporal ERGM Simulation*

Description

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

Usage

```
control.simulate.network(
  MCMC.burnin.min = 1000,
  MCMC.burnin.max = 1e+05,
  MCMC.burnin.pval = 0.5,
  MCMC.burnin.add = 1,
  MCMC.prop.form = ~discord + sparse,
  MCMC.prop.diss = ~discord + sparse,
  MCMC.prop.weights.form = "default",
  MCMC.prop.weights.diss = "default",
  MCMC.prop.args.form = NULL,
  MCMC.prop.args.diss = NULL,
  MCMC.maxedges = Inf,
  MCMC.maxchanges = 1e+06,
  term.options = NULL,
  MCMC.packagenames = c()
)

control.simulate.stergm(
  MCMC.burnin.min = NULL,
  MCMC.burnin.max = NULL,
  MCMC.burnin.pval = NULL,
  MCMC.burnin.add = NULL,
  MCMC.prop.form = NULL,
  MCMC.prop.diss = NULL,
  MCMC.prop.weights.form = NULL,
  MCMC.prop.weights.diss = NULL,
  MCMC.prop.args.form = NULL,
  MCMC.prop.args.diss = NULL,
  MCMC.maxedges = NULL,
  MCMC.maxchanges = NULL,
  term.options = NULL,
  MCMC.packagenames = NULL
)
```

Arguments

`MCMC.burnin.min`, `MCMC.burnin.max`, `MCMC.burnin.pval`, `MCMC.burnin.add`
: Number of Metropolis-Hastings steps per time step used in simulation. By default, this is determined adaptively by keeping track of increments in the Hamming distance between the transitioned-from network and the network being sampled. Once `MCMC.burnin.min` steps have elapsed, the increments are tested against 0, and when their average number becomes statistically indistinguishable from 0 (with the p-value being greater than `MCMC.burnin.pval`), or `MCMC.burnin.max` steps are proposed, whichever comes first, the simulation is stopped after an additional `MCMC.burnin.add` times the number of elapsed steps have been taken. (Stopping immediately would bias the sampling.)

  To use a fixed number of steps, set `MCMC.burnin.min` and `MCMC.burnin.max` to the same value.

`MCMC.prop.form`
: Hints and/or constraints for selecting and initializing the proposal.

`MCMC.prop.weights.form`
: Specifies the proposal weighting scheme to be used in the MCMC Metropolis-Hastings algorithm. Possible choices may be determined by calling `ergm_proposal_table`.

`MCMC.prop.weights.diss`, `MCMC.prop.args.diss`, `MCMC.prop.diss`
: Ignored. These are included for backwards compatibility of calls to `control` functions only; they have no effect on `simulate` behavior.

`MCMC.prop.args.form`
: An alternative, direct way of specifying additional arguments to proposals.

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

`MCMC.maxchanges`
: Maximum number of changes for which to allocate space.

`term.options`
: A list of additional arguments to be passed to term initializers. It can also be set globally via `options(ergm.term=list(...))`.

`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.

Details

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

These functions are included for backwards compatibility, and users are encouraged to use `control.simulate.tergm` or `control.simulate.formula.tergm` with the `simulate.tergm` family of functions instead. When a `control.simulate.stergm` or `control.simulate.network` object is passed to one of the `simulate.stergm` functions, the corresponding `simulate.tergm` function is invoked, and uses the formation proposal control arguments, ignoring the dissolution proposal control arguments.

Note: The old `dissolution` formula in `stergm` represents tie persistence. As a result it maps to the new `Persist()` operator in `tergm`, NOT the `Diss()` operator

Value

A list with arguments as components.

See Also

`simulate.stergm`, `simulate.formula`. `control.stergm` performs a similar function for `stergm`.

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

Description

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

Usage

```
control.simulate.tergm(
  MCMC.burnin.min = NULL,
  MCMC.burnin.max = NULL,
  MCMC.burnin.pval = NULL,
  MCMC.burnin.add = NULL,
  MCMC.prop = NULL,
  MCMC.prop.weights = NULL,
  MCMC.prop.args = NULL,
  MCMC.maxedges = NULL,
  MCMC.maxchanges = NULL,
  term.options = NULL,
  MCMC.packagenames = NULL
)

control.simulate.formula.tergm(
  MCMC.burnin.min = 1000,
  MCMC.burnin.max = 1e+05,
  MCMC.burnin.pval = 0.5,
  MCMC.burnin.add = 1,
  MCMC.prop = ~discord + sparse,
  MCMC.prop.weights = "default",
  MCMC.prop.args = NULL,
  MCMC.maxedges = Inf,
  MCMC.maxchanges = 1e+06,
  term.options = NULL,
  MCMC.packagenames = c()
)
```

Arguments

`MCMC.burnin.min`, `MCMC.burnin.max`, `MCMC.burnin.pval`, `MCMC.burnin.add`
: Number of Metropolis-Hastings steps per time step used in simulation. By default, this is determined adaptively by keeping track of increments in the Hamming distance between the transitioned-from network and the network being sampled. Once `MCMC.burnin.min` steps have elapsed, the increments are tested against 0, and when their average number becomes statistically indistinguishable from 0 (with the p-value being greater than `MCMC.burnin.pval`), or `MCMC.burnin.max` steps are proposed, whichever comes first, the simulation is stopped after an additional `MCMC.burnin.add` times the number of elapsed steps have been taken. (Stopping immediately would bias the sampling.)

  To use a fixed number of steps, set `MCMC.burnin.min` and `MCMC.burnin.max` to the same value.

`MCMC.prop`
: Hints and/or constraints for selecting and initializing the proposal.

`MCMC.prop.weights`
: Specifies the proposal weighting scheme to be used in the MCMC Metropolis-Hastings algorithm. Possible choices may be determined by calling `ergm_proposal_table`.

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

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

`MCMC.maxchanges`
: Maximum number of changes for which to allocate space.

`term.options`
: A list of additional arguments to be passed to term initializers. It can also be set globally via `options(ergm.term=list(...))`.

`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.

Details

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

Value

A list with arguments as components.

See Also

simulate.tergm, simulate.formula. control.tergm performs a similar function for tergm.

---

control.stergm *Auxiliary for Controlling Separable Temporal ERGM Fitting*

Description

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

Usage

```
control.stergm(
  init.form = NULL,
  init.diss = NULL,
  init.method = NULL,
  force.main = FALSE,
  MCMC.prop.form = ~discord + sparse,
  MCMC.prop.diss = ~discord + sparse,
  MCMC.prop.weights.form = "default",
  MCMC.prop.args.form = NULL,
  MCMC.prop.weights.diss = "default",
  MCMC.prop.args.diss = NULL,
  MCMC.maxedges = Inf,
  MCMC.maxchanges = 1e+06,
  MCMC.packagenames = c(),
  CMLE.MCMC.burnin = 1024 * 16,
  CMLE.MCMC.interval = 1024,
  CMLE.ergm = NULL,
  CMLE.form.ergm = control.ergm(init = init.form, MCMC.burnin = CMLE.MCMC.burnin,
   MCMC.interval = CMLE.MCMC.interval, MCMC.prop = MCMC.prop.form, MCMC.prop.weights =
   MCMC.prop.weights.form, MCMC.prop.args = MCMC.prop.args.form, MCMC.maxedges =
    MCMC.maxedges, MCMC.packagenames = MCMC.packagenames, parallel = parallel,
   parallel.type = parallel.type, parallel.version.check = parallel.version.check,
    force.main = force.main),
  CMLE.diss.ergm = control.ergm(init = init.diss, MCMC.burnin = CMLE.MCMC.burnin,
   MCMC.interval = CMLE.MCMC.interval, MCMC.prop = MCMC.prop.diss, MCMC.prop.weights =
   MCMC.prop.weights.diss, MCMC.prop.args = MCMC.prop.args.diss, MCMC.maxedges =
    MCMC.maxedges, MCMC.packagenames = MCMC.packagenames, parallel = parallel,
   parallel.type = parallel.type, parallel.version.check = parallel.version.check,
    force.main = force.main),
  CMLE.NA.impute = c(),
  CMLE.term.check.override = FALSE,
  EGMME.main.method = c("Gradient-Descent"),
  EGMME.initialfit.control = control.ergm(),
  EGMME.MCMC.burnin.min = 1000,
  EGMME.MCMC.burnin.max = 1e+05,
  EGMME.MCMC.burnin.pval = 0.5,
  EGMME.MCMC.burnin.add = 1,
  MCMC.burnin = NULL,
  MCMC.burnin.mul = NULL,
  SAN.maxit = 4,
  SAN.nsteps.times = 8,
 SAN = control.san(term.options = term.options, SAN.maxit = SAN.maxit, SAN.prop =
    MCMC.prop.form, SAN.prop.weights = MCMC.prop.weights.form, SAN.prop.args =
    MCMC.prop.args.form, SAN.nsteps = round(sqrt(EGMME.MCMC.burnin.min *
   EGMME.MCMC.burnin.max)) * SAN.nsteps.times, SAN.packagenames = MCMC.packagenames,
    parallel = parallel, parallel.type = parallel.type, parallel.version.check =
    parallel.version.check),
  SA.restarts = 10,
  SA.burnin = 1000,
  SA.plot.progress = FALSE,
  SA.max.plot.points = 400,
  SA.plot.stats = FALSE,
  SA.init.gain = 0.1,
  SA.gain.decay = 0.5,
  SA.runlength = 25,
  SA.interval.mul = 2,
  SA.init.interval = 500,
  SA.min.interval = 20,
  SA.max.interval = 500,
  SA.phase1.minruns = 4,
  SA.phase1.tries = 20,
  SA.phase1.jitter = 0.1,
  SA.phase1.max.q = 0.1,
  SA.phase1.backoff.rat = 1.05,
  SA.phase2.levels.max = 40,
  SA.phase2.levels.min = 4,
  SA.phase2.max.mc.se = 0.001,
  SA.phase2.repeats = 400,
  SA.stepdown.maxn = 200,
  SA.stepdown.p = 0.05,
  SA.stop.p = 0.1,
  SA.stepdown.ct = 5,
  SA.phase2.backoff.rat = 1.1,
  SA.keep.oh = 0.5,
  SA.keep.min.runs = 8,
  SA.keep.min = 0,
  SA.phase2.jitter.mul = 0.2,
  SA.phase2.maxreljump = 4,
  SA.guard.mul = 4,
  SA.par.eff.pow = 1,
  SA.robust = FALSE,
  SA.oh.memory = 1e+05,
  SA.refine = c("mean", "linear", "none"),
  SA.se = TRUE,
  SA.phase3.samplesize.runs = 10,
  SA.restart.on.err = TRUE,
  term.options = NULL,
  seed = NULL,
  parallel = 0,
  parallel.type = NULL,
  parallel.version.check = TRUE,
  ...
)
```

Arguments

`init.form`, `init.diss`
: numeric or `NA` vector equal in length to the number of parameters in the formation/dissolution model or `NULL` (the default); the initial values for the estimation and coefficient offset terms. If `NULL` is passed, all of the initial values are computed using the method specified by `control$init.method`. If a numeric vector is given, the elements of the vector are interpreted as follows:

  - Elements corresponding to terms enclosed in `offset()` are used as the fixed offset coefficients. These should match the offset values given in `offset.coef.form` and `offset.coef.diss`.
  - Elements that do not correspond to offset terms and are not `NA` are used as starting values in the estimation.
  - Initial values for the elements that are `NA` are fit using the method specified by `control$init.method`.

  Passing coefficients from a previous run can be used to "resume" an uncoverged `stergm` run.

`init.method`
: Estimation method used to acquire initial values for estimation. If `NULL` (the default), the initial values are computed using the edges dissolution approximation (Carnegie et al.) when appropriate; note that this relies on `.extract.fd.formulae` to identify the formation and dissolution parts of the formula; the user should be aware of its behavior and limitations. If `init.method` is set to "zeros", the initial values are set to zeros.

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

`MCMC.prop.form`
: Hints and/or constraints for selecting and initializing the proposal.

`MCMC.prop.weights.form`
: Specifies the proposal weighting to use.

`MCMC.prop.args.form`
: A direct way of specifying arguments to the proposal.

`MCMC.prop.weights.diss`, `MCMC.prop.args.diss`, `MCMC.prop.diss`
: Ignored.

`MCMC.maxedges`
: Maximum number of edges for which to allocate space.

`MCMC.maxchanges`
: Maximum number of changes in dynamic network simulation for which to allocate space.

`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.

`CMLE.MCMC.burnin`
: Burnin used in CMLE fitting.

`CMLE.MCMC.interval`
: Number of Metropolis-Hastings steps between successive draws when running MCMC MLE.

`CMLE.ergm`
: A convenience argument for specifying both `CMLE.form.ergm` and `CMLE.diss.ergm` at once. See `control.ergm`.

`CMLE.form.ergm`
: Control parameters used to fit the CMLE. See `control.ergm`.

`CMLE.diss.ergm`
: Ignored, with the exception of initial parameter values.

`CMLE.NA.impute`
: In STERGM CMLE, missing dyads in transitioned-to networks are accommodated using methods of Handcock and Gile (2009), but a similar approach to transitioned-from networks requires much more complex methods that are not, currently, implemented. `CMLE.NA.impute` controls how missing dyads in transitioned-from networks are be imputed. See argument `imputers` of `impute.network.list` for details.

  By default, no imputation is performed, and the fitting stops with an error if any transitioned-from networks have missing dyads.

`CMLE.term.check.override`
: The method `stergm{stergm}` uses at this time to fit a series of more than two networks requires certain assumptions to be made about the ERGM terms being used, which are tested before a fit is attempted. This test sometimes fails despite the model being amenable to fitting, so setting this option to `TRUE` overrides the tests.

`EGMME.main.method`
: Estimation method used to find the Equilibrium Generalized Method of Moments estimator. Currently only "Gradient-Descent" is implemented.

`EGMME.initialfit.control`
: Control object for the ergm fit in tergm.EGMME.initialfit

`EGMME.MCMC.burnin.min, EGMME.MCMC.burnin.max,`
: Number of Metropolis-Hastings steps per time step used in EGMME fitting. By default, this is determined adaptively by keeping track of increments in the Hamming distance between the transitioned-from network and the network being sampled. Once `EGMME.MCMC.burnin.min` steps have elapsed, the increments are tested against 0, and when their average number becomes statistically indistinguishable from 0 (with the p-value being greater than `EGMME.MCMC.burnin.pval`), or `EGMME.MCMC.burnin.max` steps are proposed, whichever comes first, the simulation is stopped after an additional `EGMME.MCMC.burnin.add` times the number of elapsed steps had been taken. (Stopping immediately would bias the sampling.)

To use a fixed number of steps, set `EGMME.MCMC.burnin.min` and `EGMME.MCMC.burnin.max` to the same value.

`EGMME.MCMC.burnin.pval`, `EGMME.MCMC.burnin.add`
: Number of Metropolis-Hastings steps per time step used in EGMME fitting. By default, this is determined adaptively by keeping track of increments in the Hamming distance between the transitioned-from network and the network being sampled. Once `EGMME.MCMC.burnin.min` steps have elapsed, the increments are tested against 0, and when their average number becomes statistically indistinguishable from 0 (with the p-value being greater than `EGMME.MCMC.burnin.pval`), or `EGMME.MCMC.burnin.max` steps are proposed, whichever comes first, the simulation is stopped after an additional `EGMME.MCMC.burnin.add` times the number of elapsed steps had been taken. (Stopping immediately would bias the sampling.)

  To use a fixed number of steps, set `EGMME.MCMC.burnin.min` and `EGMME.MCMC.burnin.max` to the same value.

`MCMC.burnin`, `MCMC.burnin.mul`
: No longer used. See `EGMME.MCMC.burnin.min`, `EGMME.MCMC.burnin.max`, `EGMME.MCMC.burnin.pval`, `EGMME.MCMC.burnin.pval`, `EGMME.MCMC.burnin.add` and `CMLE.MCMC.burnin` and `CMLE.MCMC.interval`.

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

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

`SAN`
: SAN control parameters. See `control.san`

`SA.restarts`
: Maximum number of times to restart a failed optimization process.

`SA.burnin`
: Number of time steps to advance the starting network before beginning the optimization.

`SA.plot.progress`, `SA.plot.stats`
: Logical: Plot information about the fit as it proceeds. If `SA.plot.progress==TRUE`, plot the trajectories of the parameters and target statistics as the optimization progresses. If `SA.plot.stats==TRUE`, plot a heatmap representing correlations of target statistics and a heatmap representing the estimated gradient.

  Do NOT use these with non-interactive plotting devices like `pdf`. (In fact, it will refuse to do that with a warning.)

`SA.max.plot.points`
: If `SA.plot.progress==TRUE`, the maximum number of time points to be plotted. Defaults to 400. If more iterations elapse, they will be thinned to at most 400 before plotting.

`SA.init.gain`
: Initial gain, the multiplier for the parameter update size. If the process initially goes crazy beyond recovery, lower this value.

`SA.gain.decay`
: Gain decay factor.

`SA.runlength`
: Number of parameter trials and updates per C run.

`SA.interval.mul`
: The number of time steps between updates of the parameters is set to be this times the mean duration of extant ties.

`SA.init.interval`
: Initial number of time steps between updates of the parameters.

`SA.min.interval`, `SA.max.interval`
: Upper and lower bounds on the number of time steps between updates of the parameters.

`SA.phase1.minruns`
: Number of runs during Phase 1 for estimating the gradient, before every gradient update.

`SA.phase1.tries`
: Number of runs trying to find a reasonable parameter and network configuration.

`SA.phase1.jitter`
: Initial jitter standard deviation of each parameter.

`SA.phase1.max.q`
: Q-value (false discovery rate) that a gradient estimate must obtain before it is accepted (since sign is what is important).

`SA.phase1.backoff.rat`, `SA.phase2.backoff.rat`
: If the run produces this relative increase in the approximate objective function, it will be backed off.

`SA.phase2.levels.min`, `SA.phase2.levels.max`
: Range of gain levels (subphases) to go through.

`SA.phase2.max.mc.se`
: Approximate precision of the estimates that must be attained before stopping.

`SA.phase2.repeats`, `SA.stepdown.maxn`,
: A gain level may be repeated multiple times (up to `SA.phase2.repeats`) if the optimizer detects that the objective function is improving or the estimating equations are not centered around 0, so slowing down the parameters at that point is counterproductive. To detect this it looks at the the window controlled by `SA.keep.oh`, thinning objective function values to get `SA.stepdown.maxn`, and 1) fitting a GLS model for a linear trend, with AR(2) autocorrelation and 2) conductiong an approximate Hotelling's T^2 test for equality of estimating equation values to 0. If there is no significance for either at `SA.stepdown.p` `SA.stepdown.ct` runs in a row, the gain level (subphase) is allowed to end. Otherwise, the process continues at the same gain level.

`SA.stepdown.p`, `SA.stepdown.ct`
: A gain level may be repeated multiple times (up to `SA.phase2.repeats`) if the optimizer detects that the objective function is improving or the estimating equations are not centered around 0, so slowing down the parameters at that point is counterproductive. To detect this it looks at the the window controlled by `SA.keep.oh`, thinning objective function values to get `SA.stepdown.maxn`, and 1) fitting a GLS model for a linear trend, with AR(2) autocorrelation and 2) conductiong an approximate Hotelling's T^2 test for equality of estimating equation values to 0. If there is no significance for either at `SA.stepdown.p` `SA.stepdown.ct` runs in a row, the gain level (subphase) is allowed to end. Otherwise, the process continues at the same gain level.

`SA.stop.p` At the end of each gain level after the minimum, if the precision is sufficiently high, the relationship between the parameters and the targets is tested for evidence of local nonlinearity. This is the p-value used.

If that test fails to reject, a Phase 3 run is made with the new parameter values, and the estimating equations are tested for difference from 0. If this test fails to reject, the optimization is finished.

If either of these tests rejects, at `SA.stop.p`, optimization is continued for another gain level.

`SA.keep.oh`, `SA.keep.min`, `SA.keep.min.runs`

Parameters controlling how much of optimization history to keep for gradient and covariance estimation.

A history record will be kept if it's at least one of the following:

- Among the last `SA.keep.oh` (a fraction) of all runs.
- Among the last `SA.keep.min` (a count) records.
- From the last `SA.keep.min.runs` (a count) optimization runs.

`SA.phase2.jitter.mul`

Jitter standard deviation of each parameter is this value times its standard deviation without jitter.

`SA.phase2.maxreljump`

To keep the optimization from "running away" due to, say, a poor gradient estimate building on itself, if a magnitude of change (Mahalanobis distance) in parameters over the course of a run divided by average magnitude of change for recent runs exceeds this, the change is truncated to this amount times the average for recent runs.

`SA.guard.mul` The multiplier for the range of parameter and statistics values to compute the guard width.

`SA.par.eff.pow` Because some parameters have much, much greater effects than others, it improves numerical conditioning and makes estimation more stable to rescale the kth estimating function by $s_k = (\sum_{i=1}^{q} G_{i,k}^2 / V_{i,i})^{-p/2}$, where $G_{i,k}$ is the estimated gradient of the ith target statistics with respect to kth parameter. This parameter sets the value of p: `0` for no rescaling, `1` (default) for scaling by root-mean-square normalized gradient, and greater values for greater penalty.

`SA.robust` Whether to use robust linear regression (for gradients) and covariance estimation.

`SA.oh.memory` Absolute maximum number of data points per thread to store in the full optimization history.

`SA.refine` Method, if any, used to refine the point estimate at the end: "linear" for linear interpolation, "mean" for average, and "none" to use the last value.

`SA.se` Logical: If TRUE (the default), get an MCMC sample of statistics at the final estimate and compute the covariance matrix (and hence standard errors) of the parameters. This sample is stored and can also be used by `mcmc.diagnostics()` to assess convergence.

`SA.phase3.samplesize.runs`

This many optimization runs will be used to determine whether the optimization has converged and to estimate the standard errors.

`SA.restart.on.err`	Logical: if `TRUE` (the default) an error somewhere in the optimization process will cause it to restart with a smaller gain value. Otherwise, the process will stop. This is mainly used for debugging
`term.options`	A list of additional arguments to be passed to term initializers. It can also be set globally via `option(ergm.term=list(...))`.
`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 default.
`parallel.version.check`	Logical: If TRUE, check that the version of `ergm` running on the slave nodes is the same as that running on the master node.
`...`	Additional arguments, passed to other functions This argument is helpful because it collects any control parameters that have been deprecated; a warning message is printed in case of deprecated arguments.

Details

This function is only used within a call to the `stergm` function. See the usage section in `stergm` for details. Generally speaking, `control.stergm` is remapped to `control.tergm`, with dissolution controls ignored and formation controls used as controls for the overall `tergm` process. An exception to this rule is the initial parameter values specified via `init.form`, `init.diss`, `CMLE.form.ergm$init`, and `CMLE.diss.ergm$init`, which will be remapped jointly with the `stergm()` arguments `offset.coef.form` and `offset.coef.diss` to determine the initial parameter values passed to `tergm`.

It is recommended that new code make use of `tergm` and `control.tergm` directly; `stergm` wrappers are included only for backwards compatibility.

Value

A list with arguments as components.

References

Boer, P., Huisman, M., Snijders, T.A.B., and Zeggelink, E.P.H. (2003), StOCNET User\'s Manual. Version 1.4.

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. (2010), A Steplength Algorithm for Fitting ERGMs, Penn State Department of Statistics Technical Report.

See Also

stergm,tergm,control.tergm. The control.simulate.stergm function performs a similar function for simulate.tergm.

---

control.tergm *Auxiliary for Controlling Temporal ERGM Fitting*

Description

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

Usage

```
control.tergm(
  init = NULL,
  init.method = NULL,
  force.main = FALSE,
  MCMC.prop = ~discord + sparse,
  MCMC.prop.weights = "default",
  MCMC.prop.args = NULL,
  MCMC.maxedges = Inf,
  MCMC.maxchanges = 1e+06,
  MCMC.packagenames = c(),
  CMLE.MCMC.burnin = 1024 * 16,
  CMLE.MCMC.interval = 1024,
 CMLE.ergm = control.ergm(init = init, MCMC.burnin = CMLE.MCMC.burnin, MCMC.interval =
   CMLE.MCMC.interval, MCMC.prop = MCMC.prop, MCMC.prop.weights = MCMC.prop.weights,
   MCMC.prop.args = MCMC.prop.args, MCMC.maxedges = MCMC.maxedges, MCMC.packagenames =
    MCMC.packagenames, parallel = parallel, parallel.type = parallel.type,
    parallel.version.check = parallel.version.check, force.main = force.main),
  CMLE.NA.impute = c(),
  CMLE.term.check.override = FALSE,
  EGMME.main.method = c("Gradient-Descent"),
  EGMME.initialfit.control = control.ergm(),
  EGMME.MCMC.burnin.min = 1000,
  EGMME.MCMC.burnin.max = 1e+05,
  EGMME.MCMC.burnin.pval = 0.5,
  EGMME.MCMC.burnin.add = 1,
  MCMC.burnin = NULL,
  MCMC.burnin.mul = NULL,
  SAN.maxit = 4,
  SAN.nsteps.times = 8,
 SAN = control.san(term.options = term.options, SAN.maxit = SAN.maxit, SAN.prop =
   MCMC.prop, SAN.prop.weights = MCMC.prop.weights, SAN.prop.args = MCMC.prop.args,
    SAN.nsteps = round(sqrt(EGMME.MCMC.burnin.min * EGMME.MCMC.burnin.max)) *
    SAN.nsteps.times, SAN.packagenames = MCMC.packagenames, parallel = parallel,
   parallel.type = parallel.type, parallel.version.check = parallel.version.check,
     parallel.inherit.MT = parallel.inherit.MT),
  SA.restarts = 10,
  SA.burnin = 1000,
  SA.plot.progress = FALSE,
  SA.max.plot.points = 400,
  SA.plot.stats = FALSE,
  SA.init.gain = 0.1,
  SA.gain.decay = 0.5,
  SA.runlength = 25,
  SA.interval.mul = 2,
  SA.init.interval = 500,
  SA.min.interval = 20,
  SA.max.interval = 500,
  SA.phase1.minruns = 4,
  SA.phase1.tries = 20,
  SA.phase1.jitter = 0.1,
  SA.phase1.max.q = 0.1,
  SA.phase1.backoff.rat = 1.05,
  SA.phase2.levels.max = 40,
  SA.phase2.levels.min = 4,
  SA.phase2.max.mc.se = 0.001,
  SA.phase2.repeats = 400,
  SA.stepdown.maxn = 200,
  SA.stepdown.p = 0.05,
  SA.stop.p = 0.1,
  SA.stepdown.ct = 5,
  SA.phase2.backoff.rat = 1.1,
  SA.keep.oh = 0.5,
  SA.keep.min.runs = 8,
  SA.keep.min = 0,
  SA.phase2.jitter.mul = 0.2,
  SA.phase2.maxreljump = 4,
  SA.guard.mul = 4,
  SA.par.eff.pow = 1,
  SA.robust = FALSE,
  SA.oh.memory = 1e+05,
  SA.refine = c("mean", "linear", "none"),
  SA.se = TRUE,
  SA.phase3.samplesize.runs = 10,
  SA.restart.on.err = TRUE,
  term.options = NULL,
  seed = NULL,
  parallel = 0,
  parallel.type = NULL,
  parallel.version.check = TRUE,
  parallel.inherit.MT = FALSE
)
```

Arguments

`init` numeric or `NA` vector equal in length to the number of parameters in the model or `NULL` (the default); the initial values for the estimation and coefficient offset terms. If `NULL` is passed, all of the initial values are computed using the method specified by `control$init.method`. If a numeric vector is given, the elements of the vector are interpreted as follows:

- Elements corresponding to terms enclosed in `offset()` are used as the fixed offset coefficients. These should match the offset values given in `offset.coef`.
- Elements that do not correspond to offset terms and are not `NA` are used as starting values in the estimation.
- Initial values for the elements that are `NA` are fit using the method specified by `control$init.method`.

Passing coefficients from a previous run can be used to "resume" an uncoverged `tergm` run.

`init.method` Estimation method used to acquire initial values for estimation. If `NULL` (the default), the initial values are computed using the edges dissolution approximation (Carnegie et al.) when appropriate; note that this relies on `.extract.fd.formulae` to identify the formation and dissolution parts of the formula; the user should be aware of its behavior and limitations. If `init.method` is set to "zeros", the initial values are set to zeros.

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

`MCMC.prop` Hints and/or constraints for selecting and initializing the proposal.

`MCMC.prop.weights`
Specifies the proposal weighting to use.

`MCMC.prop.args` A direct way of specifying arguments to the proposal.

`MCMC.maxedges` Maximum number of edges permitted to occur during the simulation.

`MCMC.maxchanges`
Maximum number of changes permitted to occur during the simulation.

`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.

`CMLE.MCMC.burnin`
Burnin used in CMLE fitting.

`CMLE.MCMC.interval`
Number of Metropolis-Hastings steps between successive draws when running MCMC MLE.

`CMLE.ergm` Control parameters used to fit the CMLE. See `control.ergm`.

`CMLE.NA.impute` In TERGM CMLE, missing dyads in transitioned-to networks are accommodated using methods of Handcock and Gile (2009), but a similar approach to transitioned-from networks requires much more complex methods that are not, currently, implemented. `CMLE.NA.impute` controls how missing dyads in transitioned-from networks are be imputed. See argument `imputers` of `impute.network.list` for details.

By default, no imputation is performed, and the fitting stops with an error if any transitioned-from networks have missing dyads.

`CMLE.term.check.override`
: The method `tergm{tergm}` uses at this time to fit a series of more than two networks requires certain assumptions to be made about the ERGM terms being used, which are tested before a fit is attempted. This test sometimes fails despite the model being amenable to fitting, so setting this option to `TRUE` overrides the tests.

`EGMME.main.method`
: Estimation method used to find the Equilibrium Generalized Method of Moments estimator. Currently only "Gradient-Descent" is implemented.

`EGMME.initialfit.control`
: Control object for the ergm fit in tergm.EGMME.initialfit

`EGMME.MCMC.burnin.min, EGMME.MCMC.burnin.max,`
: Number of Metropolis-Hastings steps per time step used in EGMME fitting. By default, this is determined adaptively by keeping track of increments in the Hamming distance between the transitioned-from network and the network being sampled. Once `EGMME.MCMC.burnin.min` steps have elapsed, the increments are tested against 0, and when their average number becomes statistically indistinguishable from 0 (with the p-value being greater than `EGMME.MCMC.burnin.pval`), or `EGMME.MCMC.burnin.max` steps are proposed, whichever comes first, the simulation is stopped after an additional `EGMME.MCMC.burnin.add` times the number of elapsed steps had been taken. (Stopping immediately would bias the sampling.)

  To use a fixed number of steps, set `EGMME.MCMC.burnin.min` and `EGMME.MCMC.burnin.max` to the same value.

`EGMME.MCMC.burnin.pval, EGMME.MCMC.burnin.add`
: Number of Metropolis-Hastings steps per time step used in EGMME fitting. By default, this is determined adaptively by keeping track of increments in the Hamming distance between the transitioned-from network and the network being sampled. Once `EGMME.MCMC.burnin.min` steps have elapsed, the increments are tested against 0, and when their average number becomes statistically indistinguishable from 0 (with the p-value being greater than `EGMME.MCMC.burnin.pval`), or `EGMME.MCMC.burnin.max` steps are proposed, whichever comes first, the simulation is stopped after an additional `EGMME.MCMC.burnin.add` times the number of elapsed steps had been taken. (Stopping immediately would bias the sampling.)

  To use a fixed number of steps, set `EGMME.MCMC.burnin.min` and `EGMME.MCMC.burnin.max` to the same value.

`MCMC.burnin, MCMC.burnin.mul`
: No longer used. See `EGMME.MCMC.burnin.min`, `EGMME.MCMC.burnin.max`, `EGMME.MCMC.burnin.pval`, `EGMME.MCMC.burnin.pval`, `EGMME.MCMC.burnin.add` and `CMLE.MCMC.burnin` and `CMLE.MCMC.interval`.

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

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

`SAN`
: SAN control parameters. See `control.san`

`SA.restarts`
: Maximum number of times to restart a failed optimization process.

`SA.burnin`
: Number of time steps to advance the starting network before beginning the optimization.

`SA.plot.progress`, `SA.plot.stats`
: Logical: Plot information about the fit as it proceeds. If `SA.plot.progress==TRUE`, plot the trajectories of the parameters and target statistics as the optimization progresses. If `SA.plot.stats==TRUE`, plot a heatmap representing correlations of target statistics and a heatmap representing the estimated gradient.

  Do NOT use these with non-interactive plotting devices like `pdf`. (In fact, it will refuse to do that with a warning.)

`SA.max.plot.points`
: If `SA.plot.progress==TRUE`, the maximum number of time points to be plotted. Defaults to 400. If more iterations elapse, they will be thinned to at most 400 before plotting.

`SA.init.gain`
: Initial gain, the multiplier for the parameter update size. If the process initially goes crazy beyond recovery, lower this value.

`SA.gain.decay`
: Gain decay factor.

`SA.runlength`
: Number of parameter trials and updates per C run.

`SA.interval.mul`
: The number of time steps between updates of the parameters is set to be this times the mean duration of extant ties.

`SA.init.interval`
: Initial number of time steps between updates of the parameters.

`SA.min.interval`, `SA.max.interval`
: Upper and lower bounds on the number of time steps between updates of the parameters.

`SA.phase1.minruns`
: Number of runs during Phase 1 for estimating the gradient, before every gradient update.

`SA.phase1.tries`
: Number of runs trying to find a reasonable parameter and network configuration.

`SA.phase1.jitter`
: Initial jitter standard deviation of each parameter.

`SA.phase1.max.q`
: Q-value (false discovery rate) that a gradient estimate must obtain before it is accepted (since sign is what is important).

`SA.phase1.backoff.rat`, `SA.phase2.backoff.rat`
: If the run produces this relative increase in the approximate objective function, it will be backed off.

`SA.phase2.levels.min`, `SA.phase2.levels.max`
: Range of gain levels (subphases) to go through.

`SA.phase2.max.mc.se`
: Approximate precision of the estimates that must be attained before stopping.

`SA.phase2.repeats`, `SA.stepdown.maxn`,
: A gain level may be repeated multiple times (up to `SA.phase2.repeats`) if the optimizer detects that the objective function is improving or the estimating equations are not centered around 0, so slowing down the parameters at that point is counterproductive. To detect this it looks at the the window controlled by `SA.keep.oh`, thinning objective function values to get `SA.stepdown.maxn`, and 1) fitting a GLS model for a linear trend, with AR(2) autocorrelation and 2) conductiong an approximate Hotelling's T^2 test for equality of estimating equation values to 0. If there is no significance for either at `SA.stepdown.p` `SA.stepdown.ct` runs in a row, the gain level (subphase) is allowed to end. Otherwise, the process continues at the same gain level.

`SA.stepdown.p`, `SA.stepdown.ct`
: A gain level may be repeated multiple times (up to `SA.phase2.repeats`) if the optimizer detects that the objective function is improving or the estimating equations are not centered around 0, so slowing down the parameters at that point is counterproductive. To detect this it looks at the the window controlled by `SA.keep.oh`, thinning objective function values to get `SA.stepdown.maxn`, and 1) fitting a GLS model for a linear trend, with AR(2) autocorrelation and 2) conductiong an approximate Hotelling's T^2 test for equality of estimating equation values to 0. If there is no significance for either at `SA.stepdown.p` `SA.stepdown.ct` runs in a row, the gain level (subphase) is allowed to end. Otherwise, the process continues at the same gain level.

`SA.stop.p`
: At the end of each gain level after the minimum, if the precision is sufficiently high, the relationship between the parameters and the targets is tested for evidence of local nonlinearity. This is the p-value used.

  If that test fails to reject, a Phase 3 run is made with the new parameter values, and the estimating equations are tested for difference from 0. If this test fails to reject, the optimization is finished.

  If either of these tests rejects, at `SA.stop.p`, optimization is continued for another gain level.

`SA.keep.oh`, `SA.keep.min`, `SA.keep.min.runs`
: Parameters controlling how much of optimization history to keep for gradient and covariance estimation.

  A history record will be kept if it's at least one of the following:

  - Among the last `SA.keep.oh` (a fraction) of all runs.
  - Among the last `SA.keep.min` (a count) records.
  - From the last `SA.keep.min.runs` (a count) optimization runs.

`SA.phase2.jitter.mul`
: Jitter standard deviation of each parameter is this value times its standard deviation without jitter.

`SA.phase2.maxreljump`
: To keep the optimization from "running away" due to, say, a poor gradient estimate building on itself, if a magnitude of change (Mahalanobis distance) in

parameters over the course of a run divided by average magnitude of change for recent runs exceeds this, the change is truncated to this amount times the average for recent runs.

`SA.guard.mul` The multiplier for the range of parameter and statistics values to compute the guard width.

`SA.par.eff.pow` Because some parameters have much, much greater effects than others, it improves numerical conditioning and makes estimation more stable to rescale the kth estimating function by $s_k = (\sum_{i=1}^{q} G_{i,k}^2 / V_{i,i})^{-p/2}$, where $G_{i,k}$ is the estimated gradient of the ith target statistics with respect to kth parameter. This parameter sets the value of p: `0` for no rescaling, `1` (default) for scaling by root-mean-square normalized gradient, and greater values for greater penalty.

`SA.robust` Whether to use robust linear regression (for gradients) and covariance estimation.

`SA.oh.memory` Absolute maximum number of data points per thread to store in the full optimization history.

`SA.refine` Method, if any, used to refine the point estimate at the end: "linear" for linear interpolation, "mean" for average, and "none" to use the last value.

`SA.se` Logical: If TRUE (the default), get an MCMC sample of statistics at the final estimate and compute the covariance matrix (and hence standard errors) of the parameters. This sample is stored and can also be used by `mcmc.diagnostics()` to assess convergence.

`SA.phase3.samplesize.runs`
This many optimization runs will be used to determine whether the optimization has converged and to estimate the standard errors.

`SA.restart.on.err`
Logical: if `TRUE` (the default) an error somewhere in the optimization process will cause it to restart with a smaller gain value. Otherwise, the process will stop. This is mainly used for debugging

`term.options` A list of additional arguments to be passed to term initializers. It can also be set globally via `option(ergm.term=list(...))`.

`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 default.

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

`parallel.inherit.MT`
Logical: If TRUE, slave nodes and processes inherit the `set.MT_terms()` setting.

Details

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

Value

A list with arguments as components.

References

Boer, P., Huisman, M., Snijders, T.A.B., and Zeggelink, E.P.H. (2003), StOCNET User\'s Manual. Version 1.4.

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. (2010), A Steplength Algorithm for Fitting ERGMs, Penn State Department of Statistics Technical Report.

See Also

`tergm`. The `control.simulate.tergm` function performs a similar function for `simulate.tergm`.

---

control.tergm.godfather

Control parameters for `tergm.godfather()`.

Description

Returns a list of its arguments.

Usage

```
control.tergm.godfather(term.options = NULL)
```

Arguments

`term.options`	A list of optional settings such as calculation tuning options to be passed to the `InitErgmTerm` functions.

`ergm-hints` *Hints for Temporal Exponential Family Random Graph Models*

Description

This page describes the hints and network sample space constraints that are included with the `tergm` package. For more information, and instructions for using hints and constraints, see `ergm-hints` and `ergm`.

Hints implemented in the `tergm` package

`discord` *The Dynamic Discordance Hint:* Propose toggling discordant dyads with greater frequency (typically about 50 percent). May be used in dynamic fitting and simulation.

References

Krivitsky PN and Handcock MS (2014) A Separable Model for Dynamic Networks. *Journal of the Royal Statistical Society, Series B*, 76(1): 29-46. doi: 10.1111/rssb.12014

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

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/`.

`ergm-terms` *Temporally-Sensitive Operator and Durational Terms used in Exponential Family Random Graph Models*

Description

The terms described here are unique to temporal networks: each summarizes some type of change or durational information.

The operator terms include: `Form()`, `Persist()`, `Diss()`, `Cross()` and `Change()`. These are used to specify how the `ergm-terms` in a formula are evaluated across a network time-series. Note, you cannot use an operator within another operator, so `Cross(~Form(~edges))` is not a valid specification.

The durational terms are distinguished either by their name, `mean.age`, or their name extensions: `<name>.ages`, `<name>.mean.age`, and `<name>.age.interval`. In contrast to their named `ergm-terms` counterparts, these durational terms take into account the elapsed time since each (term-relevant) dyad in the network was last toggled.

As currently implemented, the package does not support use of durational terms during estimation. But durational terms may be used as targets, monitors, or summary statistics. The ability to use these terms in the estimation of models is under development.

All terms listed here currently work with binary-valued ties only.

Operator Terms included in the `tergm` package

`Form(formula)` *The Formation Operator Term:* This term accepts a model formula `formula` and produces the corresponding model for the post-formation network: effectively a network containing both previous time step's ties and ties just formed, the union of the previous and current network. This is the equivalent of the old-style `formation` model.

`Persist(formula)` *The Persistence Operator Term:* This term accepts a model formula `formula` and produces the corresponding model for the post-dissolution/persistence network: effectively the network containing ties that persisted since the last time step.

This is the equivalent of the old-style `dissolution` model. So a larger positive coefficient for `Persist()` operator means *less* dissolution. It produces the same results as the new `Diss()` operator, except the signs of the coefficients are negated.

`Diss(formula)` *The Dissolution Operator Term:* This term accepts a model formula `formula` and produces the corresponding model for the post-dissolution network (same as `Persist()`), but with all statistics negated.

Note: This is not the equivalent of the old style `dissolution` model, because the signs of the coefficients are reversed. So a larger positive coefficient for `Diss()` operator means *more* dissolution.

`Cross(formula)` *The Crossection Operator Term:* This term accepts a model formula `formula` and produces the corresponding model for the cross-sectional network. It is mainly useful for CMLE estimation, and has no effect (i.e., `Cross(~TERM) == ~TERM`) for EGMME and dynamic simulation.

`Change(formula)` *The Change Operator Term:* This term accepts a model formula `formula` and produces the corresponding model for a network constructed by taking the dyads that have changed between time steps.

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

`degrange.mean.age(from, to=+Inf, byarg=NULL, emptyval=0)` *Average age of ties incident on nodes having degree in a given range:* The `from` and `to` arguments are vectors of distinct integers or `+Inf`, for `to`. If one of the vectors has length 1, it is recycled to the length of the other. Otherwise, they must have the same length. This term adds one network statistic to the model for each element of `from` (or `to`); the ith such statistic equals the average, among all ties incident on nodes with degree greater than or equal to `from[i]` but strictly less than `to[i]`, of the amount of time elapsed since the tie's formation. The optional argument `byarg` specifies a vertex attribute (see Specifying Vertex Attributes and Levels for details). If specified, then separate degree statistics are calculated for nodes having each separate value of the attribute.

Because this average is undefined for a network that does not have any actors with degree in the specified range, the argument `emptyval` can be used to specify the value returned if this is the case. This is, technically, an arbitrary value, but it should not have a substantial effect unless a non-negligible fraction of networks at the parameter configuration of interest has no actors with specified degree.

`degree.mean.age(d, byarg=NULL, emptyval=0)` *Average age of ties incident on nodes having a given degree:* The `d` argument is a vector of distinct integers. This term adds one network statistic to the model for each element in `d`; the ith such statistic equals the average, among all ties incident on nodes with degree exactly `d[i]`, of the amount of time elapsed since the tie's formation. The optional argument `byarg` specifies a vertex attribute (see Specifying Vertex Attributes and Levels for details). If specified, then separate degree statistics are calculated for nodes having each separate value of the attribute.

Because this average is undefined for a network that does not have any actors with degree `d[i]`, the argument `emptyval` can be used to specify the value returned if this is the case. This is, technically, an arbitrary value, but it should not have a substantial effect unless a non-negligible fraction of networks at the parameter configuration of interest has no actors with specified degree.

`edges.ageinterval(from, to=+Inf)` *Number of edges with age falling into a specified range:* This term counts the number of edges in the network for which the time elapsed since formation is greater than or equal to `from` but strictly less than `to`. In other words, it is in the semiopen interval `[from,to)`. `from` and `to` may be scalars, vectors of the same length, or one of them must have length one, in which case it is recycled.

`edge.ages` *Sum of ages of extant ties:* This term adds one statistic equaling sum, over all ties present in the network, of the amount of time elapsed since formation.

Unlike `mean.age`, this statistic is well-defined on an empty network. However, if used as a target, it appears to produce highly biased dissolution parameter estimates if the goal is to get an intended average duration.

`edgecov.ages(x, attrname=NULL)` *Weighted sum of ages of extant ties:* This term adds one statistic equaling sum, over all ties present in the network, of the amount of time elapsed since formation, multiplied by a dyadic covariate. See the help for the `edgecov` term for details for specifying the covariate.

"Weights" can be negative.

Unlike `edgecov.mean.age`, this statistic is well-defined on an empty network. However, if used as a target, it appears to produce highly biased dissolution parameter estimates if the goal is to get an intended average duration.

`edgecov.mean.age(x, attrname=NULL, emptyval=0)` *Weighted average age of an extant tie:* This term adds one statistic equaling the average, over all ties present in the network, of the amount of time elapsed since formation, weighted by a (nonnegative) dyadic covariate. See the help for the `edgecov` term for details for specifying the covariate.

The behavior when there are negative weights is undefined.

Because this average is undefined for an empty network (or a network all of whose extant edges have been weighted 0), the argument `emptyval` can be used to specify the value returned if this is the case. This is, technically, an arbitrary value, but it should not have a substantial effect unless a non-negligible fraction of networks at the parameter configuration of interest is empty and/or if only a few dyads have nonzero weights.

`mean.age(emptyval=0, log=FALSE)` *Average age of an extant tie:* This term adds one statistic equaling the average, over all ties present in the network, of the amount of time elapsed since formation.

Because this average is undefined for an empty network, the argument `emptyval` can be used to specify the value returned if it is. This is, technically, an arbitrary value, but it should not have a substantial effect unless a non-negligible fraction of networks at the parameter configuration of interest is empty.

To get mean log age instead of mean age, set `log=TRUE`.

`nodefactor.mean.age(attr, levels=NULL, emptyval=0, log=FALSE)` *Average ages of extant half-ties incident on nodes of specified attribute levels:* This term adds one statistic for each level of `attr`, equaling the average, over all half-ties incident on nodes of that level, of the amount of time elapsed since formation.

To control what levels are included, use the `levels` argument. Note that the default `levels` value for `nodefactor.mean.age` retains all levels, unlike the default for `nodefactor`, which omits the first level.

See Specifying Vertex Attributes and Levels for details on specifying vertex attributes and levels.

The argument `emptyval` functions like it does for `mean.age`, except that a different value may be specified for each level of `attr`. The length of `emptyval` should either be 1 (in which case that value is used for every level of `attr`) or should be equal to the number of retained levels of `attr`, in which case the `i`th value in `emptyval` is used for the `i`th retained level of `attr`.

To get mean log age instead of mean age, set `log=TRUE`; this is applied to all levels.

`nodemix.mean.age(attr, b1levels=NULL, b2levels=NULL, levels=NULL, levels2=NULL, emptyval=0, log=FALSE)` *Average ages of extant ties of specified mixing types:* This term adds one statistic for each mixing type of `attr`, equaling the average, over all ties of that mixing type, of the amount of time elapsed since formation.

The `levels`-related arguments function just like they do for the ordinary `nodemix` term. See Specifying Vertex Attributes and Levels for details on specifying vertex attributes and levels.

The argument `emptyval` functions like it does for `mean.age`, except that a different value may be specified for each mixing type of `attr`. The length of `emptyval` should either be 1 (in which case that value is used for every mixing type of `attr`) or should be equal to the number of retained mixing types of `attr`, in which case the `i`th value in `emptyval` is used for the `i`th retained mixing type of `attr`.

To get mean log age instead of mean age, set `log=TRUE`; this is applied to all mixing types.

References

- Handcock M. S., Hunter D. R., Butts C. T., Goodreau S. G., Krivitsky P. N. and Morris M. (2012). _Fit, Simulate and Diagnose Exponential-Family Models for Networks_. Version 3.1. Project home page at <URL: https://statnet.org>, <URL: CRAN.R-project.org/package=ergm>.
- Krivitsky, P.N. (2012). Modeling of Dynamic Networks based on Egocentric Data with Durational Information. *Pennsylvania State University Department of Statistics Technical Report*, 2012(2012-01). `https://web.archive.org/web/20170830053722/https://stat.psu.edu/research/technical-report-files/2012-technical-reports/TR1201A.pdf`
- Krivitsky, P.N. (2012). Modeling Tie Duration in ERGM-Based Dynamic Network Models. *Pennsylvania State University Department of Statistics Technical Report*, 2012(2012-02).

See Also

`ergm-terms` (from the `ergm` package), `ergm`, `network`, `%v%`, `%n%`

`impute.network.list` *Impute missing dyads in a series of networks*

Description

This function takes a list of networks with missing dyads and returns a list of networks with missing dyads imputed according to a list of imputation directives.

Usage

```
impute.network.list(
  nwl,
  imputers = c(),
  nwl.prepend = list(),
  nwl.append = list()
)
```

Arguments

`nwl`	A list of `network` objects or a `network.list` object.
`imputers`	A character vector giving one or more methods to impute missing dyads. Currenly implemented methods are as follows:

- `next` Impute the state of the same dyad in the next network in the list (or later, if that one is also missing). This imputation method is likely to lead to an underestimation of the tie-change rates. The last network in the list cannot be imputed this way.
- `previous` Impute the state of the same dyad in the previous network in the list (or earlier, if that one is also missing). The first network in the list cannot be imputed this way.
- `majority` Impute the missing dyad with the value of the majority among the non-missing dyads in that time step's network. A network that has exactly the same number of ties as non-missing non-ties cannot be imputed this way.
- `0` Assume missing dyads are all non-ties.
- `1` Assume missing dyads are all ties.

If `length(imputers)>1` the specified imputation methods will be applied in succession. For example, `imputers=c("next","previous","majority","0")` would first try to impute a missing dyad with the next time step's value. If it, and all of the later values for that dyad are missing, it will try to impute it with the previous time step's value. If it, and all of the earlier values for that dyad are missing as well, it will try to impute it with the value of the majority of non-missing dyads for that time step. If there is an exact tie, it will impute 0.

`nwl.prepend`	An optional list of networks to treat as preceding those in `nwl`. They will not be imputed or returned, but they can be useful for imputing dyads in the first network in `nwl`, when using `"previous"` imputer.
`nwl.append`	An optional list of networks to treat as following those in `nwl`. They will not be imputed or returned, but they can be useful for imputing dyads in the last network in `nwl`, when using `"next"` imputer.

Value

A list of networks with missing dyads imputed.

See Also

`network`, `is.na`

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

Description

These functions test whether an ERGM is duration dependent or not.

The method for `NULL` always returns `FALSE` by convention.

Usage

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

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

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

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

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

Arguments

`object`	An ERGM formula, `ergm_model` object, or `ergm_state` object.
`...`	Unused at this time.
`response, basis`	See `ergm()`.

Value

TRUE if the ERGM terms in the model are duration dependent; FALSE otherwise.

Methods (by class)

- ergm_model: Test if the ergm_model has duration-dependent terms, which call for lasttoggle data structures.
- ergm_state: Test if the ergm_state has duration-dependent terms, which call for lasttoggle data structures.

lasttoggle *Lasttoggle*

Description

A data structure used by tergm for tracking of limited information about dyad edge histories.

Details

The tergm package handles durational information attached to network objects by way of the time and lasttoggle network attributes. The lasttoggle data structure is a 3-column matrix; the first two columns are tails and heads (respectively) of dyads, and the third column is the last time at which the dyad was toggled. The default last toggle time is -INT_MAX/2. Last toggle times for non-edges are periodically cleared in the C code. The time network attribute is simply an integer, and together with the lasttoggle data it determines the age of an extant tie as time + 1 minus the last toggle time for that dyad. The default value for time is 0.

NetSeries *A network series specification for conditional modeling.*

Description

A function for specifying the LHS of a temporal network series ERGM.

Usage

```
NetSeries(..., order = 1, NA.impute = NULL)
```

Arguments

`...`	series specification, in one of three formats: 1. A list of identically- dimensioned and directed networks. 2. Several networks as arguments. 3. A `networkDynamic` object and a numeric vector of time indices.
`order`	how many previous networks to store as an accessible covariate of the model.
`NA.impute`	How missing dyads in transitioned-from networks are be imputed when using conditional estimation. See argument `imputers` of `impute.network.list` for details.

Value

A network object with temporal metadata.

Note

It is not recommended to modify the network returned by `NetSeries` except by adding and removing edges, and even that must be done with some care, to avoid putting it into an inconsistent state.

It is almost always better to modify the original networks and regenerate the series.

See Also

Help on model specification for specific terms.

Examples

```
data(samplk)

# Method 1: list of networks
monks <- NetSeries(list(samplk1,samplk2,samplk3))
ergm(monks ~ Form(~edges)+Diss(~edges))
ergm(monks ~ Form(~edges)+Persist(~edges))

# Method 2: networks as arguments
monks <- NetSeries(samplk1,samplk2,samplk3)
ergm(monks ~ Form(~edges)+Diss(~edges))
ergm(monks ~ Form(~edges)+Persist(~edges))

# Method 3: networkDynamic and time points:
## TODO
```

`simulate.network` *STERGM wrappers for TERGM simulation*

Description

The `simulate.network` and `simulate.networkDynamic` wrappers are provided for backwards compatibility. It is recommended that new code make use of the `simulate_formula.network` and `simulate_formula.networkDynamic` functions instead. See `simulate.tergm` for details on these new functions.

Usage

```
## S3 method for class 'network'
simulate(
  object,
  nsim = 1,
  seed = NULL,
  formation,
  dissolution,
  coef.form,
  coef.diss,
  constraints = ~.,
  monitor = NULL,
  time.slices = 1,
  time.start = NULL,
  time.burnin = 0,
  time.interval = 1,
  time.offset = 1,
  control = control.simulate.network(),
  output = c("networkDynamic", "stats", "changes", "final", "ergm_state"),
  stats.form = FALSE,
  stats.diss = FALSE,
  verbose = FALSE,
  ...
)

## S3 method for class 'networkDynamic'
simulate(
  object,
  nsim = 1,
  seed = NULL,
  formation,
  dissolution,
  coef.form = attr(object, "coef.form"),
  coef.diss = attr(object, "coef.diss"),
  constraints = ~.,
  monitor = NULL,
  time.slices = 1,
  time.start = NULL,
  time.burnin = 0,
  time.interval = 1,
  time.offset = 1,
  control = control.simulate.network(),
  output = c("networkDynamic", "stats", "changes", "final", "ergm_state"),
  stats.form = FALSE,
  stats.diss = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`object`	an object of type `network` or `networkDynamic`
`nsim`	Number of replications (separate chains of networks) of the process to run and return. The `networkDynamic` method only supports `nsim=1`.
`seed`	Random number integer seed. See `set.seed`.
`formation, dissolution`	One-sided `ergm`-style formulas for the formation and dissolution models, respectively. The dissolution model is parameterized in terms of tie persistence.
`coef.form`	Parameters for the formation model.
`coef.diss`	Parameters for the dissolution (persistence) model.
`constraints`	A one-sided formula specifying one or more constraints on the support of the distribution of the networks being modeled, using syntax similar to the `formula` argument. Multiple constraints may be given, separated by "+" operators. Together with the model terms in the formula and the reference measure, the constraints define the distribution of networks being modeled. The default is `~.`, for an unconstrained model. See the ERGM constraints documentation for the constraints implemented in the **ergm** package. Other packages may add their own constraints. Note that not all possible combinations of constraints are supported.
`monitor`	A one-sided formula specifying one or more terms whose value is to be monitored. If `monitor` is specified as a character (one of `"formation"`, `"dissolution"`, and `"all"`) then the function `.extract.fd.formulae` is used to determine the corresponding formula; the user should be aware of its behavior and limitations.
`time.slices`	Number of time slices (or statistics) to return from each replication of the dynamic process. See below for return types. Defaults to 1, which, if `time.burnin==0` and `time.interval==1` (the defaults), advances the process one time step.
`time.start`	An optional argument specifying the time point at which the simulation is to start. See Details for further information.
`time.burnin`	Number of time steps to discard before starting to collect network statistics.
`time.interval`	Number of time steps between successive recordings of network statistics.

`time.offset`	Argument specifying the offset between the point when the state of the network is sampled (`time.start`) and the the beginning of the spell that should be recorded for the newly simulated network state.
`control`	A list of control parameters for algorithm tuning, constructed using `control.simulate.network`. These are mapped to `control.simulate.formula.tergm` controls by assigning: • `MCMC.prop.form` to `MCMC.prop`, • `MCMC.prop.args.form` to `MCMC.prop.args`, and • `MCMC.prop.weights.form` to `MCMC.prop.weights`.
`output`	A character vector specifying output type: one of `"networkDynamic"` (the default), `"stats"`, `"changes"`, `"final"`, and `"ergm_state"`, with partial matching allowed.
`stats.form, stats.diss`	Logical: Whether to return formation/dissolution model statistics. This is not the recommended method: use the `monitor` argument instead. Note that if either `stats.form` or `stats.diss` is `TRUE`, all generative model statistics will be returned.
`verbose`	Logical: If `TRUE`, extra information is printed as the Markov chain progresses.
`...`	Further arguments passed to or used by methods.

Details

Note that return values may be structured differently than in past versions.

Remember that in `stergm`, the dissolution formula is parameterized in terms of tie persistence: negative coefficients imply lower rates of persistence and postive coefficients imply higher rates. The dissolution effects are simply the negation of these coefficients.

Because the old `dissolution` formula in `stergm` represents tie persistence, it maps to the new `Persist()` operator in the `tergm` function, NOT the `Diss()` operator

Value

Depends on the `output` argument. See `simulate.tergm` for details. Note that some formation/dissolution separated information is also attached to the return value for calls made through `simulate.network` and `simulate.networkDynamic` in an attempt to increase backwards compatibility.

Examples

```
logit<-function(p)log(p/(1-p))
coef.form.f<-function(coef.diss,density) -log(((1+exp(coef.diss))/(density/(1-density)))-1)

# Construct a network with 20 nodes and 20 edges
n<-20
target.stats<-edges<-20
g0<-network.initialize(n,dir=TRUE)
g1<-san(g0~edges,target.stats=target.stats,verbose=TRUE)
```

```
S<-10

# To get an average duration of 10...
duration<-10
coef.diss<-logit(1-1/duration)

# To get an average of 20 edges...
dyads<-network.dyadcount(g1)
density<-edges/dyads
coef.form<-coef.form.f(coef.diss,density)

# ... coefficients.
print(coef.form)
print(coef.diss)

# Simulate a networkDynamic
dynsim<-simulate(g1,formation=~edges,dissolution=~edges,
                 coef.form=coef.form,coef.diss=coef.diss,
                 time.slices=S,verbose=TRUE)

# "Resume" the simulation.
dynsim2<-simulate(dynsim,formation=~edges,dissolution=~edges,time.slices=S,verbose=TRUE)
```

`simulate.tergm` *Draw from the distribution of a Temporal Exponential Family Random Graph Model*

Description

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

Usage

```
## S3 method for class 'tergm'
simulate(
  object,
  nsim = 1,
  seed = NULL,
  coef = coefficients(object),
  constraints = object$constraints,
  monitor = object$targets,
  time.slices = 1,
  time.start = NULL,
  time.burnin = 0,
  time.interval = 1,
  control = control.simulate.tergm(),
  output = c("networkDynamic", "stats", "changes", "final", "ergm_state"),
  nw.start = NULL,
  stats = FALSE,
  verbose = FALSE,
  ...
)

## S3 method for class 'network'
simulate_formula(
  object,
  nsim = 1,
  seed = NULL,
  coef = NULL,
  constraints = ~.,
  monitor = NULL,
  time.slices = 1,
  time.start = NULL,
  time.burnin = 0,
  time.interval = 1,
  time.offset = 1,
  control = control.simulate.formula.tergm(),
  output = c("networkDynamic", "stats", "changes", "final", "ergm_state"),
  stats = FALSE,
  verbose = FALSE,
  ...,
  basis = ergm.getnetwork(object),
  dynamic = FALSE
)

## S3 method for class 'networkDynamic'
simulate_formula(
  object,
  nsim = 1,
  seed = NULL,
  coef = attr(basis, "coef"),
  constraints = ~.,
  monitor = NULL,
  time.slices = 1,
  time.start = NULL,
  time.burnin = 0,
  time.interval = 1,
  time.offset = 1,
  control = control.simulate.formula.tergm(),
  output = c("networkDynamic", "stats", "changes", "final", "ergm_state"),
  stats = FALSE,
  verbose = FALSE,
  ...,
  basis = eval_lhs.formula(object),
  dynamic = FALSE
```

```
)
```

Arguments

`object`	for `simulate.tergm`, an object of type `tergm` giving a model fit; for `simulate_formula.network` and `simulate_formula.networkDynamic`, a formula specifying the model `simulate_formula.network` understands the `lasttoggle` "API".
`nsim`	Number of replications (separate chains of networks) of the process to run and return. The `networkDynamic` method only supports `nsim=1`.
`seed`	Random number integer seed. See `set.seed`.
`coef`	Parameters for the model.
`constraints`	A one-sided formula specifying one or more constraints on the support of the distribution of the networks being modeled. Multiple constraints may be given, separated by "+" operators. Together with the model terms in the formula and the reference measure, the constraints define the distribution of networks being modeled. The default is `~.`, for an unconstrained model. See the ERGM constraints documentation for the constraints implemented in the **ergm** package. Other packages may add their own constraints. Note that not all possible combinations of constraints are supported.
`monitor`	A one-sided formula specifying one or more terms whose value is to be monitored. If `monitor` is specified as a character (one of `"formation"`, `"dissolution"`, and `"all"`) then the function `.extract.fd.formulae` is used to determine the corresponding formula; the user should be aware of its behavior and limitations.
`time.slices`	Number of time slices (or statistics) to return from each replication of the dynamic process. See below for return types. Defaults to 1, which, if `time.burnin==0` and `time.interval==1` (the defaults), advances the process one time step.
`time.start`	An optional argument specifying the time point at which the simulation is to start. See Details for further information.
`time.burnin`	Number of time steps to discard before starting to collect network statistics.
`time.interval`	Number of time steps between successive recordings of network statistics.
`control`	A list of control parameters for algorithm tuning. Constructed using `control.simulate.tergm` or `control.simulate.formula.tergm`. For backwards compatibility, control lists from `control.simulate.stergm` and `control.simulate.network` are allowed in calls to `simulate.tergm`; they are mapped to `control.simulate.tergm` by assigning: • `MCMC.prop.form` to `MCMC.prop`, • `MCMC.prop.args.form` to `MCMC.prop.args`, • `MCMC.prop.weights.form` to `MCMC.prop.weights`.
`output`	A character vector specifying output type: one of `"networkDynamic"` (the default), `"stats"`, `"changes"`, `"final"`, and `"ergm_state"`, with partial matching allowed. See Value section for details.
`nw.start`	A specification for the starting network to be used by `simulate.tergm`, optional for EGMME fits, but required for CMLE and CMPLE fits:

a numeric index `i` use `i`th time-point's network, where the first network in the series used to fit the model is defined to be at the first time point;

`"first"` **or** `"last"` the first or last time point used in fitting the model; or

`network` specify the network directly.

`networkDynamic`s cannot be used as starting networks for `simulate.tergm` at this time. (They can be used as starting networks for `simulate_formula.networkDynamic`, of course.)

`stats`	Logical: Whether to return model statistics. This is not the recommended method: use `monitor` argument instead.
`verbose`	Logical: If TRUE, extra information is printed as the Markov chain progresses.
`...`	Further arguments passed to or used by methods.
`time.offset`	Argument specifying the offset between the point when the state of the network is sampled (`time.start`) and the the beginning of the spell that should be recorded for the newly simulated network state.
`basis`	For the `network` and `networkDynamic` methods, the network to start the simulation from. (If `basis` is missing, the default is the left hand side of the `object` argument.)
`dynamic`	Logical; if `TRUE`, dynamic simulation is performed in `tergm`; if `FALSE` (the default), ordinary `ergm` simulation is performed instead. Note that when `dynamic=FALSE`, default argument values for `ergm`'s `simulate` methods are used.

Details

The dynamic process is run forward and the results are returned. For the method for `networkDynamic`, the simulation is resumed from the last generated time point of `basis` (or the left hand side of `object` if `basis` is missing), by default with the same model and parameters.

The starting network for the `tergm` object method (`simulate.tergm`) is determined by the `nw.start` argument.

- If `time.start` is specified, it is used as the initial time index of the simulation.
- If `time.start` is not specified (is `NULL`), then if the `object` carries a time stamp from which to start or resume the simulation, either in the form of a `"time"` network attribute (for the `network` method — see the `lasttoggle` "API") or in the form of an `net.obs.period` network attribute (for the `networkDynamic` method), this attribute will be used. (If specified, `time.start` will override it with a warning.)
- Othewise, the simulation starts at 0.

Value

Depends on the `output` argument:

`"stats"`	If `stats == FALSE`, an `mcmc` matrix with monitored statistics, and if `stats == TRUE`, a list containing elements `stats` for statistics specified in the `monitor` argument, and `stats.gen` for the model statistics. If `stats == FALSE` and no monitored statistics are specified, an empty list is returned, with a warning. When `nsim>1`, an `mcmc.list` (or list of them) of the statistics is returned instead.

`"networkDynamic"`
: A `networkDynamic` object representing the simulated process, with ties present in the initial network having onset `-Inf` and ties present at the end of the simulation having terminus `+Inf`. The method for `networkDynamic` returns the initial `networkDynamic` with simulated changes applied to it. The `net.obs.period` network attribute is updated (or added if not existing) to reflect the time period that was simulated. If the network does not have any `persistent.ids` defined for vertices, a vertex.pid will be attached in a vertex attribute named `'tergm_pid'` to facilitate 'bookkeeping' between the networkDynamic argument and the simulated network time step. Additionally, attributes (`attr`, not network attributes) are attached as follows:

  `formula`, `monitor`: Model and monitoring formulas used in the simulation, respectively.

  `stats`, `stats.gen`: Network statistics as above.

  `coef`: Coefficients used in the simulation.

  `changes`: A four-column matrix summarizing the changes in the `"changes"` output. (This may be removed in the future.)

  When `nsim>1`, a `network.list` of these `networkDynamic`s is returned.

`"changes"`
: An integer matrix with four columns (`time`, `tail`, `head`, and `to`), giving the time-stamped changes relative to the current network. `to` is `1` if a tie was formed and `0` if a tie was dissolved. The convention for `time` is that it gives the time point during which the change is effective. For example, a row `c(5,2,3,1)` indicates that between time 4 and 5, a tie from node 2 to node 3 was formed, so that it was absent at time point 4 and present at time point 5; while a row `c(5,2,3,0)` indicates that in that time, that tie was dissolved, so that it is was present at time point 4 and absent at time point 5. Additionally, the same attributes (`attr`, not network attributes) as with `output=="networkDynamic"` are attached. When `nsim>1`, a list of these change matrices is returned.

`"final"`
: A `network` object representing the last network in the series generated. `lasttoggle` and `time` attributes are also included. Additionally, the same attributes (`attr`, not network attributes) as with `output=="networkDynamic"` are attached. When `nsim>1`, a `network.list` of these `network`s is returned.

`"ergm_state"`
: The `ergm_state` object resulting from the simulation. Attributes are attached as for other output types.

Note that when using `simulate_formula.networkDynamic` with either `"final"` or `"ergm_state"` for `output`, the nodes included in these objects are those produced by `network.collapse` at the start time.

Examples

```
data(samplk)

# Fit a transition from Time 1 to Time 2
samplk12 <- tergm(list(samplk1, samplk2)~
                  Form(~edges+mutual+transitiveties+cyclicalties)+
                  Diss(~edges+mutual+transitiveties+cyclicalties),
                  estimate="CMLE")
```

```
# direct simulation from tergm object
sim1 <- simulate(samplk12, nw.start="last")

# equivalent simulation from formula with network LHS;
# must pass dynamic=TRUE for tergm simulation
sim2 <- simulate(samplk2 ~ Form(~edges+mutual+transitiveties+cyclicalties) +
                           Diss(~edges+mutual+transitiveties+cyclicalties),
                           coef = coef(samplk12),
                           dynamic=TRUE)

# the default simulate output is a networkDynamic, and we can simulate
# with a networkDynamic LHS as well
sim3 <- simulate(sim2 ~ Form(~edges+mutual+transitiveties+cyclicalties) +
                        Diss(~edges+mutual+transitiveties+cyclicalties),
                        coef = coef(samplk12),
                        dynamic=TRUE)
```

snctrl *Statnet Control*

Description

A utility to facilitate argument completion of control lists, reexported from `statnet.common`.

Currently recognised control parameters

This list is updated as packages are loaded and unloaded.

See Also

`statnet.common::snctrl()`

split.network *A `split()` method for `network::network` objects.*

Description

Split a network into subnetworks on a factor.

Usage

```
## S3 method for class 'network'
split(x, f, drop = FALSE, sep = ".", lex.order = FALSE, ...)
```

Arguments

`x`	a `network::network` object.
`f, drop, sep, lex.order`	see `split()`; note that `f` must have length equal to `network.size(x)`.
`...`	additional arguments, currently unused.

Value

A `network.list` containing the networks. These networks will inherit all vertex and edge attributes, as well as relevant network attributes.

See Also

`network::get.inducedSubgraph()`

`stergm` *Separable Temporal Exponential Family Random Graph Models*

Description

`stergm` is used for finding Separable Temporal ERGMs' (STERGMs) Conditional MLE (CMLE) (Krivitsky and Handcock, 2010) and Equilibrium Generalized Method of Moments Estimator (EGMME) (Krivitsky, 2009).

Usage

```
stergm(
  nw,
  formation,
  dissolution,
  constraints = ~.,
  estimate,
  times = NULL,
  offset.coef.form = NULL,
  offset.coef.diss = NULL,
  targets = NULL,
  target.stats = NULL,
 eval.loglik = NVL(getOption("tergm.eval.loglik"), getOption("ergm.eval.loglik")),
  control = control.stergm(),
  verbose = FALSE,
  ...,
  SAN.offsets = NULL
)
```

Arguments

`nw`	A `network` object (for EGMME); or `networkDynamic` object, a `network.list` object, or a `list` containing networks (for CMLE and CMPLE). `stergm` understands the `lasttoggle` "API".
`formation, dissolution`	One-sided `ergm`-style formulas for the formation and dissolution models, respectively. In `stergm`, the dissolution formula is parameterized in terms of tie persistence: negative coefficients imply lower rates of persistence and postive coefficients imply higher rates. The dissolution effects are simply the negation of these coefficients.
`constraints`	A one-sided formula specifying one or more constraints on the support of the distribution of the networks being modeled, using syntax similar to the `formula` argument. Multiple constraints may be given, separated by "+" operators. Together with the model terms in the formula and the reference measure, the constraints define the distribution of networks being modeled. The default is `~.`, for an unconstrained model. See the ERGM constraints documentation for the constraints implemented in the **ergm** package. Other packages may add their own constraints. Note that not all possible combinations of constraints are supported.
`estimate`	One of "EGMME" for Equilibrium Generalized Method of Moments Estimation, based on a single network with some temporal information and making an assumption that it is a product of a STERGM process running to its stationary (equilibrium) distribution; "CMLE" for Conditional Maximum Likelihood Estimation, modeling a transition between two networks, or "CMPLE" for Conditional Maximum PseudoLikelihood Estimation, using MPLE instead of MLE. CMPLE is extremely inaccurate at this time.
`times`	For CMLE and CMPLE estimation, times or indexes at which the networks whose transition is to be modeled are observed. Default to `c(0,1)` if `nw` is a `networkDynamic` and to `1:length(nw)` (all transitions) if `nw` is a `network.list` or a `list`. Unused for EGMME. Note that at this time, the selected time points will be treated as temporally adjacent. Irregularly spaced time series are not supported at this time.
`offset.coef.form`	Numeric vector to specify offset formation parameters.
`offset.coef.diss`	Numeric vector to specify offset dissolution parameters.
`targets`	One-sided `ergm`-style formula specifying statistics whose moments are used for the EGMME. Unused for CMLE and CMPLE. Targets is required for EGMME estimation. It may contain any valid ergm terms. Any offset terms are used only during the preliminary SAN run; they are removed automatically for the EGMME proper. If `targets` is specified as a character (one of `"formation"` and `"dissolution"`) then the function `.extract.fd.formulae` is used to determine the corresponding formula; the user should be aware of its behavior and limitations.
`target.stats`	A vector specifying the values of the `targets` statistics that EGMME will try to match. Defaults to the statistics of `nw`. Unused for CMLE and CMPLE.

`eval.loglik`	Whether or not to calculate the log-likelihood of a CMLE STERGM fit. See `ergm` for details. Can be set globally via `option(tergm.eval.loglik=...)`, falling back to `getOption("ergm.eval.loglik")` if not set.
`control`	A list of control parameters for algorithm tuning. Constructed using `control.stergm`. Remapped to `control.tergm`.
`verbose`	logical or integer; if TRUE or positive, the program will print out progress information. Higher values result in more output.
`...`	Additional arguments, to be passed to lower-level functions.
`SAN.offsets`	Offset coefficients (if any) to use during the SAN run.

Details

This function is included for backwards compatibility, and users are encouraged to use the new `tergm` family of functions instead.

The `stergm` function uses a pair of formulas, `formation` and `dissolution` to model tie-dynamics. The dissolution formula, however, is parameterized in terms of tie persistence: negative coefficients imply lower rates of persistence and postive coefficients imply higher rates. The dissolution effects are simply the negation of these coefficients, but the discrepancy between the terminology and interpretation has always been unfortunate, and we have fixed this in the new `tergm` function.

If you are making the transition from old `stergm` to new `tergm`, note that the `dissolution` formula in `stergm` maps to the new `Persist()` operator in the `tergm` function, NOT the `Diss()` operator.

Model Terms See `ergm` and `ergm-terms` for details. At this time, only linear ERGM terms are allowed.

- For a brief demonstration, please see the tergm package vignette: `browseVignettes(package='tergm')`
- A more detailed tutorial is available on the statnet wiki: `https://statnet.org/Workshops/tergm/tergm_tutorial.html`

Value

`stergm` returns an object of class `tergm`; see `tergm()` for details and methods.

References

Krivitsky P.N. and Handcock M.S. (2014) A Separable Model for Dynamic Networks. *Journal of the Royal Statistical Society, Series B*, 76(1): 29-46. doi: 10.1111/rssb.12014

Krivitsky, P.N. (2012). Modeling of Dynamic Networks based on Egocentric Data with Durational Information. *Pennsylvania State University Department of Statistics Technical Report*, 2012(2012-01). `https://web.archive.org/web/20170830053722/https://stat.psu.edu/research/technical-report-files` `2012-technical-reports/TR1201A.pdf`

See Also

ergm, network, \

Examples

```
## Not run:
# EGMME Example
par(ask=FALSE)
n<-30
g0<-network.initialize(n,dir=FALSE)

#                     edges, degree(1), mean.age
target.stats<-c(      n*1/2,    n*0.6,        20)

dynfit<-stergm(g0,formation = ~edges+degree(1), dissolution = ~edges,
               targets = ~edges+degree(1)+mean.age,
               target.stats=target.stats, estimate="EGMME",
               control=control.stergm(SA.plot.progress=TRUE))

par(ask=TRUE)
mcmc.diagnostics(dynfit)
summary(dynfit)

## End(Not run)


# CMLE Example
data(samplk)

# Fit a transition from Time 1 to Time 2
samplk12 <- stergm(list(samplk1, samplk2),
                   formation=~edges+mutual+transitiveties+cyclicalties,
                   dissolution=~edges+mutual+transitiveties+cyclicalties,
                   estimate="CMLE")

mcmc.diagnostics(samplk12)
summary(samplk12)

# Fit a transition from Time 1 to Time 2 and from Time 2 to Time 3 jointly
samplk123 <- stergm(list(samplk1, samplk2, samplk3),
                    formation=~edges+mutual+transitiveties+cyclicalties,
                    dissolution=~edges+mutual+transitiveties+cyclicalties,
                    estimate="CMLE")

mcmc.diagnostics(samplk123)
summary(samplk123)
```

`summary_formula.networkDynamic` *Calculation of networkDynamic statistics.*

Description

A method for `summary_formula` to calculate the specified statistics for an observed `networkDynamic` at the specified time point(s). See `ergm-terms` for more information on the statistics that may be specified.

Usage

```
## S3 method for class 'networkDynamic'
summary_formula(object, at, ..., basis = NULL)
```

Arguments

`object`	An `formula` object with a `networkDynamic` as its LHS. (See `summary_formula` for more details.)
`at`	A vector of time points at which to calculate the statistics.
`...`	Further arguments passed to or used by methods.
`basis`	An optional `networkDynamic` object relative to which the statistics should be calculated.

Value

A matrix with `length(at)` rows, one for each time point in `at`, and columns for each term of the formula, containing the corresponding statistics measured on the network.

See Also

`ergm()`, `networkDynamic`, `ergm-terms`, summary.formula

Examples

```
# create a toy dynamic network
my.nD <- network.initialize(100,directed=FALSE)
activate.vertices(my.nD, onset=0, terminus = 10)
add.edges.active(my.nD,tail=1:2,head=2:3,onset=5,terminus=8)

# use a summary formula to display number of isolates and edges
# at discrete time points
summary(my.nD~isolates+edges, at=1:10)
```

`tergm` *Temporal Exponential-Family Random Graph Models*

Description

`tergm` is used for finding Temporal ERGMs' (TERGMs) and Separable Temporal ERGMs' (STERGMs) Conditional MLE (CMLE) (Krivitsky and Handcock, 2010) and Equilibrium Generalized Method of Moments Estimator (EGMME) (Krivitsky, 2009).

Usage

```
tergm(
  formula,
  constraints = ~.,
  estimate,
  times = NULL,
  offset.coef = NULL,
  targets = NULL,
  target.stats = NULL,
  SAN.offsets = NULL,
 eval.loglik = NVL(getOption("tergm.eval.loglik"), getOption("ergm.eval.loglik")),
  control = control.tergm(),
  verbose = FALSE,
  ...
)
```

Arguments

`formula`	an ERGM formula.
`constraints`	A one-sided formula specifying one or more constraints on the support of the distribution of the networks being modeled, using syntax similar to the `formula` argument. Multiple constraints may be given, separated by "+" operators. Together with the model terms in the formula and the reference measure, the constraints define the distribution of networks being modeled. The default is `~.`, for an unconstrained model. See the ERGM constraints documentation for the constraints implemented in the **ergm** package. Other packages may add their own constraints. Note that not all possible combinations of constraints are supported.
`estimate`	One of "EGMME" for Equilibrium Generalized Method of Moments Estimation, based on a single network with some temporal information and making an assumption that it is a product of a TERGM process running to its stationary (equilibrium) distribution; "CMLE" for Conditional Maximum Likelihood Estimation, modeling a transition between two networks, or "CMPLE" for Conditional Maximum PseudoLikelihood Estimation, using MPLE instead of MLE. CMPLE is extremely inaccurate at this time.
`times`	For CMLE and CMPLE estimation, times or indexes at which the networks whose transition is to be modeled are observed. Default to `c(0,1)` if `nw` is a `networkDynamic` and to `1:length(nw)` (all transitions) if `nw` is a `network.list` or a `list`. Unused for EGMME. Note that at this time, the selected time points will be treated as temporally adjacent. Irregluarly spaced time series are not supported at this time.
`offset.coef`	Numeric vector to specify offset parameters.
`targets`	One-sided `ergm`-style formula specifying statistics whose moments are used for the EGMME. Unused for CMLE and CMPLE. Targets is required for EGMME estimation. It may contain any valid ergm terms. Any offset terms are used only during the preliminary SAN run; they are removed automatically for the

	EGMME proper. If `targets` is specified as a character (one of `"formation"` and `"dissolution"`) then the function `.extract.fd.formulae` is used to determine the corresponding formula; the user should be aware of its behavior and limitations.
`target.stats`	A vector specifying the values of the `targets` statistics that EGMME will try to match. Defaults to the statistics of `nw`. Unused for CMLE and CMPLE.
`SAN.offsets`	Offset coefficients (if any) to use during the SAN run.
`eval.loglik`	Whether or not to calculate the log-likelihood of a CMLE TERGM fit. See `ergm` for details. Can be set globally via `option(tergm.eval.loglik=...)`, falling back to `getOption("ergm.eval.loglik")` if not set.
`control`	A list of control parameters for algorithm tuning. Constructed using `control.tergm`.
`verbose`	logical or integer; if TRUE or positive, the program will print out progress information. Higher values result in more output.
`...`	Additional arguments, to be passed to lower-level functions.

Details

Model Terms See `ergm` and `ergm-terms` for details. At this time, only linear ERGM terms are allowed.

- For a brief demonstration, please see the tergm package vignette: `browseVignettes(package='tergm')`
- A more detailed tutorial is available on the statnet wiki: `https://statnet.org/Workshops/tergm/tergm_tutorial.html`

Value

`tergm` returns an object of class `tergm` that inherits from `ergm` and has the usual methods (`coef.ergm()`, `summary.ergm()`, `mcmc.diagnostics()`, etc.) implemented for it. Note that `gof()` only works for the CMLE method.

References

Krackhardt, D and Handcock, MS (2006) Heider vs Simmel: Emergent features in dynamic structures. ICML Workshop on Statistical Network Analysis. Springer, Berlin, Heidelberg, 2006.

Hanneke S, Fu W, and Xing EP (2010). Discrete Temporal Models of Social Networks. *Electronic Journal of Statistics*, 2010, 4, 585-605. doi: 10.1214/09EJS548

Krivitsky P.N. and Handcock M.S. (2014) A Separable Model for Dynamic Networks. *Journal of the Royal Statistical Society, Series B*, 76(1): 29-46. doi: 10.1111/rssb.12014

Krivitsky, P.N. (2012). Modeling of Dynamic Networks based on Egocentric Data with Durational Information. *Pennsylvania State University Department of Statistics Technical Report*, 2012(2012-01). `http://stat.psu.edu/research/technical-report-files/2012-technical-reports/modeling-of-dynamic-networks-based-on-egocentric-data-with-durational-information`

See Also

`ergm()`, `network()`, `%v%`, `%n%`, `ergm-terms`

Examples

```
## Not run:
# EGMME Example
par(ask=FALSE)
n<-30
g0<-network.initialize(n,dir=FALSE)

#                     edges, degree(1), mean.age
target.stats<-c(      n*1/2,    n*0.6,        20)

dynfit<-tergm(g0 ~ Form(~edges + degree(1)) + Diss(~edges),
               targets = ~edges+degree(1)+mean.age,
               target.stats=target.stats, estimate="EGMME",
               control=control.tergm(SA.plot.progress=TRUE))

par(ask=TRUE)
mcmc.diagnostics(dynfit)
summary(dynfit)

## End(Not run)

# CMLE Example
data(samplk)

# Fit a transition from Time 1 to Time 2
samplk12 <- tergm(list(samplk1, samplk2)~
                  Form(~edges+mutual+transitiveties+cyclicalties)+
                  Diss(~edges+mutual+transitiveties+cyclicalties),
                  estimate="CMLE")

mcmc.diagnostics(samplk12)
summary(samplk12)

samplk12.gof <- gof(samplk12)

samplk12.gof

plot(samplk12.gof)

plot(samplk12.gof, plotlogodds=TRUE)

# Fit a transition from Time 1 to Time 2 and from Time 2 to Time 3 jointly
samplk123 <- tergm(list(samplk1, samplk2, samplk3)~
                   Form(~edges+mutual+transitiveties+cyclicalties)+
                   Diss(~edges+mutual+transitiveties+cyclicalties),
                   estimate="CMLE")

mcmc.diagnostics(samplk123)
summary(samplk123)
```

tergm.godfather *A function to apply a given series of changes to a network.*

Description

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

Usage

```
tergm.godfather(
  formula,
  changes = NULL,
  toggles = changes[, -4, drop = FALSE],
  start = NULL,
  end = NULL,
  end.network = FALSE,
  stats.start = FALSE,
  verbose = FALSE,
  control = control.tergm.godfather()
)
```

Arguments

`formula`	An `summary.formula`-style formula, with either a `network` or a `networkDynamic` as the LHS and statistics to be computed on the RHS. If LHS is a `networkDynamic`, it will be used to derive the changes to the network whose statistics are wanted. Otherwise, either `changes` or `toggles` must be specified, and the LHS `network` will be used as the starting network.
`changes`	A matrix with four columns: time, tail, head, and new value, describing the changes to be made. Can only be used if LHS of `formula` is not a `networkDynamic`.
`toggles`	A matrix with three columns: time, tail, and head, giving the dyads which had changed. Can only be used if LHS of `formula` is not a `networkDynamic`.
`start`	Time from which to start applying changes. Note that the first set of changes will take effect at `start + 1`. Defaults to the time point 1 before the earliest change passed.
`end`	Time at which to finish applying changes. Defaults to the last time point at which a change occurs.
`end.network`	Whether to return the 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 TERGMs when `output="stats"`, where initial network's statistics are not returned.
`verbose`	Whether to print progress messages.
`control`	A control list generated by `control.tergm.godfather`.

Value

If `end.network` is `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` is `TRUE`, return a `network` object with `lasttoggle` "extension", 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

`simulate.tergm()`, `simulate_formula.network()`, `simulate_formula.networkDynamic()`

`uncombine_network` *Split up a network into a list of subgraphs*

Description

Given a network created by `combine_networks()`, `uncombine_network()` returns a list of networks, preserving attributes that can be preserved.

Usage

```
uncombine_network(
  nw,
 ignore.nattr = c("bipartite", "directed", "hyper", "loops", "mnext", "multiple", "n",
    ".subnetcache"),
  ignore.vattr = c(),
  ignore.eattr = c(),
  split.vattr = ".NetworkID",
  names.vattr = NULL
)
```

Arguments

`nw`	a `network::network` created by `combine_networks()`.
`ignore.nattr, ignore.vattr, ignore.eattr`	network, vertex, and edge attributes not to be processed as described below.
`split.vattr`	name of the vertex attribute on which to split.
`names.vattr`	optional name of the vertex attribute to use as network names in the output list.

Value

a list of `network::network`s containing subgraphs on `split.vattr`. In particular,

- their basic properties (directedness and bipartednes) are the same as those of the input network;
- vertex attributes are split;
- edge attributes are assigned to their respective edges in the returned networks.

If `split.vattr` is a vector, only the first element is used and it's "popped".

See Also

`split.network()`

Examples

```
data(samplk)

o1 <- combine_networks(list(samplk1, samplk2, samplk3))
image(as.matrix(o1))

ol <- uncombine_network(o1)
```

Index