
ergm 4.0: NEW FEATURES AND IMPROVEMENTS

A PREPRINT

Pavel N. Krivitsky
School of Mathematics and Statistics
University of New South Wales

p.krivitsky@unsw.edu.au

David R. Hunter
Department of Statistics
Penn State University

dhunter@stat.psu.edu

Martina Morris
Departments of Sociology and Statistics
University of Washington

morrism@uw.edu

Chad Klumb
Departments of Sociology and Statistics
University of Washington

June 10, 2021

Abstract

The **ergm** package supports the statistical analysis and simulation of network data. It anchors the **statnet** suite of packages for network analysis in R introduced in a special issue in *Journal of Statistical Software* in 2008. This article provides an overview of the functionality and performance improvements in the 2021 **ergm** 4.0 release. These include more flexible handling of nodal covariates, operator terms that extend and simplify model specification, new models for networks with valued edges, improved handling of constraints on the sample space of networks, performance enhancements to the Markov chain Monte Carlo and maximum likelihood estimation algorithms, broader and faster searching for networks with certain target statistics using simulated annealing, and estimation with missing edge data. We also identify the new packages in the **statnet** suite that extend **ergm**'s functionality to other network data types and structural features, and the robust set of online resources that support the **statnet** development process and applications.

Keywords statistical software · statnet · ERGM · exponential-family random graph models · valued networks

1 Introduction

The **statnet** suite of packages for R (R Core Team, 2021) was first introduced in 2008, in volume 24 of *Journal of Statistical Software*, a special issue devoted to **statnet**. Together, these packages, which had already gone through the maturing process of multiple releases, provided an integrated framework for the statistical analysis of network data: from data storage and manipulation, to visualization, estimation and simulation. Since that time the existing packages have undergone continual updates to improve and add capabilities, and many new packages have been added to extend the range of network data that can be modeled (e.g., dynamic, valued, sampled, multilevel). It is the **ergm** package, however, that provides the statistical foundation for all of the other modeling packages in the **statnet** suite. Version 4.0 of **ergm**, released in 2021, is a major upgrade, representing more than a decade of changes and improvements since (Hunter et al., 2008). In addition to new functionality, updates to the central MCMC and SAN algorithms have produced order of magnitude improvements in computational speed and efficiency. This paper summarizes the key changes of interest to end users.

The exponential-family random graph model (ERGM) is a general statistical framework for modeling the probability of a link (or tie) between nodes in a network. It is the basis of the **ergm** package and most of its related packages in the **statnet** suite. We consider networks over a set of nodes $N = \{1, 2, \dots, n\}$. If $\mathbb{Y} \subseteq N \times N$ denotes a set of potential pairwise relationships among them, a binary network sample space can be regarded as $\mathcal{Y} \subseteq 2^{\mathbb{Y}}$, a subset of the power set of potential relationships. More generally, we can define \mathbb{S} to be a (possibly multivariate) set of possible relationship values. Then, the sample space $\mathcal{Y} \subseteq \mathbb{S}^{\mathbb{Y}}$ is a set whose elements are of the form $\{Y_{i,j} : (i,j) \in \mathbb{Y}\}$, where each $Y_{i,j}$, which we will call a dyad, maps the node pair $(i,j) \in \mathbb{Y}$ into \mathbb{S} and denotes the value of the relationship of $(i,j) \in \mathbb{Y}$.

We begin by briefly presenting the fully general ERGM framework, referring interested readers to Schweinberger et al. (2020) for additional technical details. A random network \mathbf{Y} is distributed according to an ERGM, written $\mathbf{Y} \sim \text{ERGM}_{\mathcal{Y},h,\eta,\mathbf{g}}(\boldsymbol{\theta})$, if

$$\Pr_{\boldsymbol{\theta},\mathcal{Y},h,\eta,\mathbf{g}}(\mathbf{Y} = \mathbf{y}) = \frac{h(\mathbf{y}) \exp\{\boldsymbol{\eta}(\boldsymbol{\theta})^\top \mathbf{g}(\mathbf{y})\}}{\kappa_{h,\eta,\mathbf{g}}(\boldsymbol{\theta}, \mathcal{Y})}, \quad \mathbf{y} \in \mathcal{Y}. \quad (1)$$

In Equation (1), \mathcal{Y} is the sample space of networks; $\boldsymbol{\theta}$ is a q -dimensional parameter vector; $h(\mathbf{y})$ is a reference measure, typically a constant in the case of binary ERGMs; $\boldsymbol{\eta}$ is a mapping from $\boldsymbol{\theta}$ to the p -vector of canonical parameters, given by the identity mapping in non-curved ERGMs; \mathbf{g} is a p -vector of sufficient statistics; and $\kappa_{h,\eta,\mathbf{g}}(\boldsymbol{\theta}, \mathcal{Y})$ is the normalizer given by $\sum_{\mathbf{y}' \in \mathcal{Y}} h(\mathbf{y}') \exp\{\boldsymbol{\eta}(\boldsymbol{\theta})^\top \mathbf{g}(\mathbf{y}')\}$, which is often intractable for models that seek to reproduce the dependence across ties induced by social effects such as triadic closure. The *natural parameter space* of the model is $\boldsymbol{\Theta}_N \stackrel{\text{def}}{=} \{\boldsymbol{\theta} : \kappa_{h,\eta,\mathbf{g}}(\boldsymbol{\theta}, \mathcal{Y}) < \infty\}$.

For a particular network application, one would typically employ a special case of the fully general Equation (1). For instance, for binary ERGMs we typically define $h(\mathbf{y})$ to be a constant, as discussed in Section 6.1, in which case the sufficient statistics can be thought of as modifying a uniform baseline distribution over the potentially observable networks. Many of the features of **ergm** and the related packages that comprise the **statnet** suite address the statistical complications that arise from modeling network data using special cases of the ERGM in Equation (1).

In particular, the statistical framework implemented in **ergm** is computationally intensive for models that specify dyadic dependence. So the package relies on a central Markov chain Monte Carlo (MCMC) algorithm for estimation and simulation, along with maximum pseudo-likelihood estimation and simulated annealing in some contexts. Substantial improvements have been made in all of these algorithms, producing efficiency and speed gains of up to two orders of magnitude. Roughly speaking, the second half of this article provides an overview of the principles implemented to achieve these gains, while the first half describes the most important new capabilities that have been added to **ergm** and its related packages since volume 24 of *Journal of Statistical Software* appeared in 2008. This includes both the capabilities introduced in the 4.0 release itself and in releases 2.2.0–3.10.4, which postdate the *JoSS* volume. (Versions in which each new capability was introduced can be obtained by running `news(package="ergm")`.)

In the examples throughout the paper we assume the reader is familiar with the basic syntax and features of **ergm** included in the 2008 *JoSS* volume. Where possible we demonstrate new, more general, functionality by comparison, using the old syntax and the new to produce the same result, then moving on with the new syntax to demonstrate the additional utilities.

The source code for **ergm** 4.0, along with the LICENSE information under GPL-3, is available at <https://github.com/statnet/ergm>.

2 Extension packages in the statnet suite

The statistical models supported by the **statnet** suite have been extended by a growing number of new packages that provide additional functionality in the general ERGM framework. While the focus of this article is the base **ergm** package, in this section we provide a brief overview of the extension packages and their specific applications. Open source package development is on GitHub under the **statnet** organization. Online tutorials, found at <https://github.com/statnet/Workshops/wiki>, exist for **ergm** and many of these extension packages, and most packages also include extended vignettes. Some of the key extension packages, and the resources that support them, include:

Building custom terms for models One of the unique aspects of this modeling framework is that each network statistic in an ERGM requires a specialized algorithm for computing the value of the

statistic from the data. The **ergm** package has over 150 of the most common terms encoded—see `vignette('ergm-term-crossRef')` for the full list—but the existing terms are a small subset of the possible terms one can use in an ERGM. For those who need a custom term, the package **ergm.userterms** (Hunter et al., 2013) is designed to simplify the process of coding up new terms for use in ERG model specification. Online workshop materials provide an overview of the process, and demonstrate the use of this package (Hunter & Goodreau, 2019).

Modeling temporal (dynamic) network data The **statnet** suite contains several packages that provide a robust framework for storing, visualizing, describing and modeling temporal network data: The **networkDynamic** package extends **network** to provide data storage and management utilities, the **tsna** package extends **sna** (Butts, 2008) to provide descriptive statistics for network objects that change over time, the **ndtv** package provides a wide range of utilities for visualizing dynamic networks and saving both static and animated output in standard formats, and **tergm** extends **ergm** to fit the class of separable temporal ERGMs, from both sampled and fully observed network data (Krivitsky & Handcock, 2014). There are two online workshops that demonstrate these tools: one that demonstrates a typical workflow from data inspection to temporal modeling (Morris & Krivitsky, 2015), and another that focuses on descriptive analyses and visualization (Bender-deMoll, 2016).

Modeling valued edges The **ergm** itself contains a framework for modeling real-valued edges (see Section 6 and Section 4). Several other packages provide specialized components for specific types of valued edges: **ergm.count** for counts, **ergm.rank** for ordered categories. The relevant theory supporting these packages may be found in Krivitsky (2012) and Krivitsky & Butts (2017), respectively. **latentnet** for latent space models also supports non-binary responses, although in a somewhat different manner (Krivitsky et al., 2009; Krivitsky & Handcock, 2008). Package vignettes and online workshop materials provide an overview of the theory, and demonstrate the use of these packages (Krivitsky & Butts, 2019).

Working with egocentrically sampled network data In the social and health sciences, egocentrically sampled network data is the most common form of data available, because it can be collected using standard sample survey methods. The **ergm.ego** package provides methods for estimating ERGMs from egocentrically sampled network data, with a principled framework for statistical inference. The theory and an application of these methods may be found in Krivitsky & Morris (2017). Online workshop materials provide an overview of the framework and demonstrate the use of the package (Morris & Krivitsky, 2019).

Multimode, multilayer, and multilevel networks In the social sciences, it is increasingly common to collect and fit ERGMs on data on multiple relationship types (Krivitsky et al., 2020; Wang, 2012) and ensembles of networks (Slaughter & Koehly, 2016). These capabilities are implemented in an extension package **ergm.multi**. We refer the reader to the package manual and workshops for further information.

Modeling diffusion and epidemics on networks One of the most active application areas for ERGMs and TERGMs is in the field of epidemic modeling. The **EpiModel** package is built on the **statnet** platform, and provides a unique set of tools for statistically principled modeling of epidemics on networks (Jenness et al., 2018). A robust set of online training materials is available at the EpiModel website.

3 Enhanced handling of nodal covariates

Version 4.0 of **ergm** standardizes and provides greater flexibility for handling covariates used by terms in an ERGM. In particular, these covariates can be modified “on-the-fly” during model specification. A vignette called `nodal_attributes` is included in the package and illustrates some of the new capabilities.

Here, we describe some of these enhancements using **ergm**’s `faux.mesa.high` dataset, a simulated in-school friendship network based on data collected on 205 students. We will focus on the `Grade` attribute, an ordinal categorical variable with values 7 through 12 that can be accessed via the `%v%` operator:

```
data(faux.mesa.high)
(faux.mesa.high %v% "Grade")[1:20] # Look at first 20 nodes' (students') grade levels

## [1] 7 7 11 8 10 10 8 11 9 9 9 11 9 11 8 10 10 7 10 7
```

Grade level is typical of the kind of covariate used to model selective mixing in social networks: different hypotheses lead to different model specifications. **ergm** 4.0 provides greater flexibility than earlier versions of **ergm** to easily define and explore different specifications.

Note we will sometimes call `summary()` and other times call `ergm()` to demonstrate the functionality and output below.

3.1 Transformations of covariates

It is sometimes desirable to specify a transformation of a nodal attribute as a covariate in a model term. Most `ergm` terms now support a new user interface, inspired by `purrr` (Henry & Wickham, 2020), to specify transformations on one or more nodal attributes. Terms typically use this new interface via arguments called `attr`, `attrs`, `by`, or `on`; the interpretation of the argument depends on its type:

character string Extract the vertex attribute with this name.

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

function The function is called on the network on the left side of the main `ergm` formula and is expected to return a vector or matrix of appropriate dimension. (Shorter vectors and matrix columns will be recycled as needed.)

formula Borrowing the interface from `tidyverse`, the expression on the right hand side of the formula is evaluated in an environment of the vertex attributes of the network, expected to return a vector or matrix of appropriate dimension. (Shorter vectors and matrix columns will be recycled as needed.) Within this expression, the network itself is accessible as either `.` or `.nw`.

AsIs object created by I() Use as is, checking only for correct length and type, with optional attribute `"name"` indicating the predictor's name.

For instance, here are three ways to compute the value of

$$g(\mathbf{y}) = \sum_{(i,j) \in \mathbb{Y}} y_{i,j} (\text{Grade}_i + \text{Grade}_j),$$

which in an ERGM may be interpreted as the linear effect of grade on overall activity of an actor:

```
summary(faux.mesa.high ~ nodecov("Grade")           # String
+ nodecov(~Grade)                                   # Formula
+ nodecov(function(nw) nw%v%"Grade"))               # Function
```

```
##          nodecov.Grade          nodecov.Grade nodecov.nw%v%"Grade"
##                3491                3491                3491
```

Here is a more complicated formula-based use of `nodecov`, where the first statistic is

$$g(\mathbf{y}) = \sum_{(i,j) \in \mathbb{Y}} y_{i,j} \left(\frac{|\text{Grade}_i - \overline{\text{Grade}}|}{n} + \frac{|\text{Grade}_j - \overline{\text{Grade}}|}{n} \right),$$

and n is the number of nodes, i.e., the network size, of the network:

```
summary(faux.mesa.high ~ nodecov(~abs(Grade-mean(Grade))/network.size())
+ nodecov(~(Grade-mean(Grade))/network.size()))
```

```
## nodecov.abs(Grade-mean(Grade))/network.size(.)  nodecov.(Grade-mean(Grade))/network.size(.)
##                2.8565140                        -0.2637716
```

The non-zero output of the second statistic above, which omits the absolute value, may be counterintuitive if you are expecting it to return the sample mean grade, $\overline{\text{Grade}}$. Node factor statistics, however, are not the sample mean grade: each node is not counted exactly once, but rather the number of cases it contributes is equal to its degree.

Taking advantage of `nodecov`'s new ability to take matrix-valued arguments, we might also evaluate a polynomial effect of `Grade`, as in the following quadratic example:¹

¹For this and other summaries, we omit the call information, deviances, and significance stars in the interests of space. The full summary information can be obtained by omitting `coef()` around the `summary()` call.

```
coef(summary(ergm(faux.mesa.high ~ edges + nodecov(~cbind(Grade, Grade2=Grade^2)))))
```

```
##              Estimate Std. Error MCMC %    z value    Pr(>|z|)
## edges          8.7297963 3.52880543      0  2.473867 0.0133659343
## nodecov.Grade -1.4597723 0.39614405      0 -3.684953 0.0002287445
## nodecov.Grade2  0.0768836 0.02154632      0  3.568294 0.0003593133
```

In the code above, the column for `Grade^2` is explicitly named `Grade2` whereas the column for `Grade` is named implicitly by R itself. Omitting the name for a column not otherwise named by R would result in a warning, as it is good practice to name all variables in the model.

Alternatively, we can use `stats::poly` for orthogonal polynomials. Here, the test for significance of the quadratic term is identical to the non-orthogonal example, up to rounding error (though the estimate is different given the orthogonal specification):

```
coef(summary(ergm(faux.mesa.high ~ edges + nodecov(~poly(Grade, 2)))))
```

```
##              Estimate Std. Error MCMC %    z value    Pr(>|z|)
## edges          -4.662459 0.07309281      0 -63.788207 0.0000000000
## nodecov.poly(Grade,2).1 -1.207241 0.68018706      0 -1.774866 0.0759199607
## nodecov.poly(Grade,2).2  2.512615 0.70416949      0  3.568196 0.0003594477
```

We can even pass a nodal covariate that is not already contained in the network object. This example randomly generates a binary-valued nodal covariate and sets its `name` attribute to be used as a label:

```
set.seed(123) # Make exact output reproducible
randomcov <- structure(I(rbinom(network.size(faux.mesa.high), 1, 0.5)), name = "random")
summary(faux.mesa.high ~ nodefactor(I(randomcov)))
```

```
## nodefactor.random.1
##                      199
```

This syntax therefore allows for simulation or estimation of models with inputs taken from arbitrary R functions or data sources, facilitating the incorporation of ERGMs into more general tool chains.

3.2 Coding categorical attributes

For model terms that use categorical attributes, **ergm** 4.0 has extended the methods for selecting and/or transforming levels via the use of the argument `levels`. Some terms, such as the `sender` and `receiver` statistics of the p_1 model (Holland & Leinhardt, 1981) and the corresponding `sociality` statistics for undirected networks, treat the node labels themselves as a categorical attribute. These terms use the `nodes=` argument, rather than the `levels=` argument, to select a subset of the nodes.

Typically, `levels` or `nodes` has a default that is sensible for the term in question. Information about the defaults may be obtained from the list of terms at `help("ergm-terms")`. Interpretation of the possible values of the `levels` and `nodes` arguments is available by typing `help(nodal_attributes)`. This interpretation is summarized as follows:

AsIs object created by `I()` Use the given level, list of levels, or vector of levels as is.

numeric or logical vector Used for indexing of a list of all possible levels (typically, unique values of the attribute) in default order (typically lexicographic). In particular, `levels=TRUE` retains all levels. Negative values exclude. Another special value is `LARGEST`, which refers to the most frequent category, so, say, to set such a category as the baseline, pass `levels=-LARGEST`. In addition, `LARGEST(n)` will refer to the `n` largest categories. `SMALLEST` works analogously, and ties in frequencies are broken arbitrarily. To specify numeric or logical levels literally, wrap them in `I()`.

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

character vector Use the given level(s) as is.

function The function is called in an environment in which the network itself is accessible as `.nw`, the list of unique values of the attribute as `.` or as `.levels`, and the attribute vector itself as `.attr`. Its return value is interpreted as above.

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

Returning to the `faux.mesa.high` example, we may treat `Grade` as a categorical variable even though its values are numeric. We see that `Grade` has six levels, numbered from 7 to 12:

```
table(faux.mesa.high %v% "Grade")
```

```
##
##  7  8  9 10 11 12
## 62 40 42 25 24 12
```

We may exclude the three smallest levels or, equivalently, include levels 7, 8, and 9. Below are five of the myriad ways to do this in the context of computing basic categorical effects on node activity, implemented by `nodefactor`. In the second expression, `I()` is necessary so that `7:9` is not treated as a vector of indices.

```
summary(faux.mesa.high ~ nodefactor(~Grade, levels = -SMALLEST(3)))
```

```
## nodefactor.Grade.7 nodefactor.Grade.8 nodefactor.Grade.9
##                153                75                65
```

```
summary(faux.mesa.high ~ nodefactor(~Grade, levels = I(7:9)))
```

```
## nodefactor.Grade.7 nodefactor.Grade.8 nodefactor.Grade.9
##                153                75                65
```

```
summary(faux.mesa.high ~ nodefactor(~Grade, levels = c("7", "8", "9")))
```

```
## nodefactor.Grade.7 nodefactor.Grade.8 nodefactor.Grade.9
##                153                75                65
```

```
summary(faux.mesa.high ~ nodefactor("Grade", levels = function(a) a %in% 7:9))
```

```
## nodefactor.Grade.7 nodefactor.Grade.8 nodefactor.Grade.9
##                153                75                65
```

```
summary(faux.mesa.high ~ nodefactor("Grade", levels = ~. %in% 7:9))
```

```
## nodefactor.Grade.7 nodefactor.Grade.8 nodefactor.Grade.9
##                153                75                65
```

Any of the arguments of Section 3.1 may also be wrapped in `COLLAPSE_SMALLEST(attr, n, into)`, a convenience function that will transform the attribute by collapsing the `n` least frequent categories into one, naming it according to the `into` argument where `into` must be of the same type (numeric, character, etc.) as the vertex attribute in question. Consider the `Race` factor of the `faux.mesa.high` network, where we use `levels=TRUE` to display all levels since the default is `levels=-1`:

```
summary(faux.mesa.high ~ nodefactor("Race", levels = TRUE))
```

```
## nodefactor.Race.Black nodefactor.Race.Hisp nodefactor.Race.NatAm nodefactor.Race.Other
##                26                178                156                1
## nodefactor.Race.White
##                45
```

Because the `Hisp` and `NatAm` categories are so much larger than the other three categories in this network, we may wish to combine the `Black`, `White`, and `Other` categories. The code below accomplishes this using `COLLAPSE_SMALLEST` while also demonstrating how to use the `magrittr` package's pipe function, `%>%`, for improved readability:

```
library(magrittr)
summary(faux.mesa.high ~ nodefactor(~Race) %>%
  COLLAPSE_SMALLEST(3, "BWO"), levels = TRUE))

##   nodefactor.Race.BWO   nodefactor.Race.Hisp nodefactor.Race.NatAm
##                72                178                156
```

3.3 Mixing matrices

Mixing matrices, which refer to the cross-tabulation of all edges by the categorical attributes of the two nodes, are a common feature in models that seek to represent selective mixing.

The `mm` model term, which stands for “mixing matrix,” generalizes the familiar `nodemix` term from the original `ergm` implementation for this purpose. Like `nodemix`, `mm` creates statistics consisting of the cells of a matrix of counts in which the columns and rows correspond to the levels of two categorical nodal covariates. For `mm`, however, these covariates may or may not be the same, making it more general. We use it here to demonstrate the `levels2` argument.

Typing `help(mm)` shows that the binary-network version of the term takes the form `mm(attrs, levels=NULL, levels2=-1)`. The `attrs` argument is a two-sided formula where the left and right sides are the rows and columns, respectively, of the mixing matrix; if only a one-sided formula or attribute name is given then the rows and columns are taken to be the same. The optional `levels` argument can similarly be a one- or two-sided formula, and it specifies the levels of the row and column variables to keep. Finally, the optional `levels2` argument may be used to select only a subset of the matrix of statistics resulting from `attrs` and `levels`.

Using this functionality, we may specify custom mixing patterns that depend upon attribute values. For instance, if we believe that the break between junior high school (grades 7–9) and high school (grades 10–12) creates a barrier to friendships across the boundary, we can create an indicator variable `Grade ≥ 10`, then compute a mixing matrix on that variable using `mm` using a single call:

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

## mm[Grade>=10=FALSE,Grade>=10=TRUE]   mm[Grade>=10=TRUE,Grade>=10=TRUE]
##                                27                                43

# Mixing with levels2 modified:
summary(faux.mesa.high ~ mm(~Grade >= 10, levels2 = NULL))

## mm[Grade>=10=FALSE,Grade>=10=FALSE]   mm[Grade>=10=FALSE,Grade>=10=TRUE]
##                                133                                27
##   mm[Grade>=10=TRUE,Grade>=10=TRUE]
##                                43
```

The `Grade>=10` indicator variable is `False` (for junior high school) and `True` (for high school), and with the undirected friendships, this produces three possible combinations of the grade indicator—`False/False`, `False/True`, and `True/True`. For the default specification, `levels = NULL` keeps all levels of the `Grade>=10` indicator variable and `levels2 = -1` eliminates the first statistic (`False/False`) in the set of 3. For the modified specification, the `levels2 = NULL` argument keeps all of the statistics.

We can also use the `mm` formula interface to filter out certain statistics from the full set of potential comparisons. An example from the `nodal_attributes` vignette within the `ergm` package using the unmodified `Grade` attribute defines `levels2` as a one-sided formula whose right side is a function that returns `TRUE` or `FALSE`, depending on whether both elements of `.levels`—the list of values taken by a pair of nodes—are in the set `c(7, 8)`. The example therefore captures mixing statistics only involving students in grades 7 or 8:

```
summary(faux.mesa.high ~ mm("Grade", levels2 = ~supply(.levels,
  function(pair) pair[[1]] %in% c(7,8) && pair[[2]] %in% c(7,8))))

## mm[Grade=7,Grade=7] mm[Grade=7,Grade=8] mm[Grade=8,Grade=8]
##                75                0                33
```


Finally, we give an example using two covariates, allowing us to capture the tendency of sets of individuals defined by values of `Grade` to mix with sets of individuals defined by values of `Race`:

```
summary(faux.mesa.high ~ mm(Grade>=10 ~ Race,
                           levels = TRUE ~ c("Hispanic", "NatAm", "White")))

##      mm[Grade>=10=TRUE,Race=Hispanic] mm[Grade>=10=FALSE,Race=NatAm] mm[Grade>=10=TRUE,Race=NatAm]
##                                     43                                115                                41
## mm[Grade>=10=FALSE,Race=White] mm[Grade>=10=TRUE,Race=White]
##                                30                                15
```

With all values of `Grade>=10` (i.e., False and True) and three values of `Race` allowed according to the `levels` argument, the full mixing matrix here would include 2×3 statistics, though the default `levels2=-1` omits the first of these so there is no `Grade>=10=FALSE,Race=Hispanic` statistic. When interpreting mixing matrix effects of this type, bear in mind that two covariates need not partition the vertex set in the same ways. Here, for instance, there can be students both above and below grade 10 with each race/ethnicity.

4 Operator terms

ergm 4.0 introduces a new type of term that we call an *operator term*, or simply *operator*. In mathematics, an operator is a function, like differentiation, that takes functions as its inputs; analogously, an operator term takes one or more ERGM formulas as input and transforms them by modifying its inputs and/or outputs. Most operator terms therefore have a general form `X(formula, ...)` where `X` is the name of the operator, typically capitalized, `formula` is a one-sided formula specifying the network statistics to be evaluated, and the remaining arguments control the transformation applied to the network before `formula` is evaluated and/or to the transformation applied to the network statistics obtained by evaluating `formula`. Online help on the operators is available via `help("ergm-terms")`, and we describe some frequently used operators below.

4.1 Network filters

Several operators allow the user to evaluate model terms on filtered versions of the network, i.e., on particular subsets of the existing nodes and/or edges.

4.1.1 Filtering edges

The operator `F(formula, filter)` evaluates the terms in `formula` on a filtered network, with filtering specified by `filter`, a formula with one binary dyad-independent **ergm** term that has exactly one statistic with a dyadwise contribution of 0 for a 0-valued dyad. That is, the term must be expressible as

$$g(\mathbf{y}) = \sum_{(i,j) \in \mathcal{Y}} f_{i,j}(y_{i,j}), \quad (2)$$

where for all possible (i,j) , $f_{i,j}(0) = 0$. One may verify that condition (2) implies that an ERGM containing the single term $g(\mathbf{y})$ has the property that the dyads $Y_{i,j}$ are jointly independent, which is why such a term is called “dyad-independent”. Examples of such terms include `nodemix`, `nodematch`, `nodefactor`, and `nodecov` and `edgescov` with appropriate covariates. Then, `formula` will be evaluated on a network constructed by taking \mathbf{y} and removing any edges for which $f_{i,j}(y_{i,j}) = 0$.

Sampson’s Monks (Sampson, 1968) can provide illustrative examples. **ergm** includes a version of these data reporting cumulative liking nominations over the three time periods Sampson asked a group of monks to identify those they liked. This directed, 18-node network is depicted in Figure 1.

```
data(sampson)
lab <- paste0(1:18, " ", substr(sample %v% "group", 1, 1), ": ", sample %v% "vertex.names")
plot(sample, displaylabels = TRUE, label = lab)
```

As an example of the `F` filter, the code below uses two different methods to summarize the number of ties between pairs of nodes in the Turks group in the `sample` dataset:

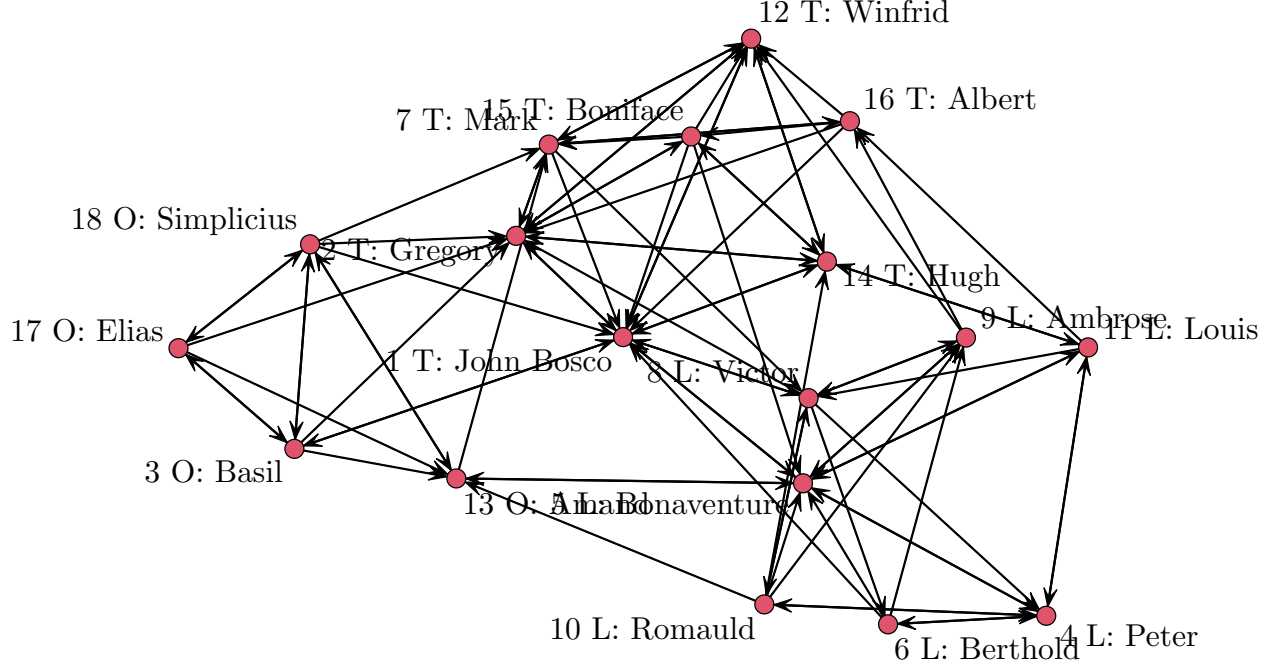


Figure 1: The monks dataset, with edges indicating directed liking relationships at any of three time points and nodes numbered from 1 to 18 and with group membership as assigned by Sampson indicated by L for Loyalists, O for Outcasts, and T for Young Turks.

```
data(sampson)
summary(samplike ~ nodematch("group", diff=TRUE, levels="Turks")
+ F(~nodematch("group"), ~nodefactor("group", levels="Turks")))
```

```
##                                nodematch.group.Turks
##                                30
## F(nodefactor("group",levels="Turks"))~nodematch.group
##                                30
```

Note that while `filter` must be dyad-independent, `formula` can have dyad-dependent terms as well.

4.1.2 Treating directed networks as undirected

The operator `Symmetrize(formula, rule)` evaluates the terms in `formula` on an undirected network constructed by symmetrizing the underlying directed network according to `rule`. The possible values of `rule`, which match the terminology of the `symmetrize` function of the `sna` package, are (a) “weak”, (b) “strong”, (c) “upper”, and (d) “lower”; for any $i < j$, these four values result in an undirected tie between i and j if and only if (a) either $y_{i,j}$ or $y_{j,i}$ equals 1, (b) both $y_{i,j}$ and $y_{j,i}$ equal 1, (c) $y_{i,j} = 1$, and (d) $y_{j,i} = 1$. For example,

```
summary(samplike ~ Symmetrize(~edges, "strong") + mutual + Symmetrize(~edges, "weak") + edges)
```

```
## Symmetrize(strong)~edges      mutual  Symmetrize(weak)~edges
##                        28          28                      60
##                        edges
##                        88
```

will compute the number of node pairs $i < j$ with reciprocated edges, equivalent to mutuality, i.e., $y_{i,j} = y_{j,i} = 1$, along with the number of node pairs in which at least one edge is present; summing these values yields the total number of directed edges.

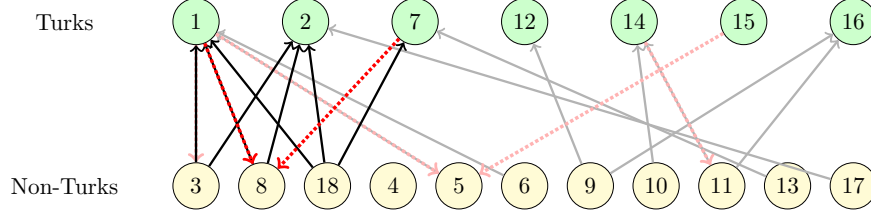


Figure 2: A bipartite induced subgraph between Turks (green) and Non-Turks (yellow). Edges involved in at least one undirected 4-cycle are emphasized. When directed edges from Non-Turks to Turks (black) are viewed as bipartite (undirected) edges, we obtain 4-cycles (3, 1, 18, 2), (3, 1, 8, 2), and (8, 1, 18, 2). When directed edges from Turks to Non-Turks (dotted red) are also included, we obtain the additional 4-cycles (8, 1, 18, 7) and (8, 2, 18, 7).

4.1.3 Extracting subgraphs

The operator `S(formula, attrs)` evaluates the terms in `formula` on an induced subgraph constructed from vertices identified by `attrs`. The `attrs` argument either takes a value as explained in Section 3.2 for the `nodes=` argument or, to obtain a bipartite network, a two-sided formula with the left-hand side specifying the tails and the right-hand side specifying the heads. For instance, suppose that we wish to model the density and mutuality dynamics within the group “Young Turks” as different from those of the rest of the network:

```
coef(summary(ergm(samplike ~ edges + mutual + S(~edges + mutual, ~(group == "Turks")))))
```

	Estimate	Std. Error	MCMC %	z value	Pr(> z)
## edges	-2.046036	0.2324354	0	-8.802600	1.336829e-18
## mutual	2.404570	0.4696226	0	5.120218	3.051821e-07
## S((group=="Turks"))~edges	2.726183	0.7997314	0	3.408873	6.523179e-04
## S((group=="Turks"))~mutual	-2.077173	1.1158313	0	-1.861547	6.266694e-02

Thus, the density within the group is statistically significantly higher, whereas the reciprocation within the group is lower, though not statistically significantly at the 5% level.

As another example, illustrated in Figure 2, consider the directed edges from non-Young Turks to Young Turks. Creating the induced subgraph from these edges results in a bipartite network—which is always taken to be undirected even though the edges were originally directed—we may count the number of four-cycles:

```
summary(samplike ~ S(~cycle(4), (group != "Turks") ~ (group == "Turks")))
```

```
## S((group!="Turks"),(group=="Turks"))~cycle4
## 3
```

On the other hand, if we treat the original network as undirected using `Symmetrize` before creating the induced bipartite subgraph, we see additional four-cycles. This example also illustrates that operator terms may be nested arbitrarily:

```
summary(samplike ~ Symmetrize(~S(~cycle(4), (group != "Turks") ~ (group == "Turks")), "weak"))
```

```
## Symmetrize(weak)~S((group!="Turks"),(group=="Turks"))~cycle4
## 5
```

Finally, we illustrate a common use case in which `Symmetrize` is used to analyze mutuality in a directed network as a function of a predictor. The `faux.dixon.high` dataset is a directed friendship network of seventh through twelfth graders. Suppose we wish to check how strongly the tendency toward mutuality in friendships is affected by students’ closeness in grade level.

```
data(faux.dixon.high)
FDHfit <- ergm(faux.dixon.high ~ edges + mutual + absdiff("grade")
+ Symmetrize(~absdiff("grade"), "strong"))
coef(summary(FDHfit))
```

	Estimate	Std. Error	MCMC %	z value	Pr(> z)
edges	-3.19981337	0.04915861	0	-65.0916209	0.000000e+00
mutual	3.05696930	0.12457834	0	24.5385295	5.733989e-133
absdiff.grade	-0.98352402	0.04410829	0	-22.2979411	3.868953e-110
Symmetrize(strong)-absdiff.grade	0.08080227	0.15693864	0	0.5148654	6.066471e-01

After correcting for the overall network density, the propensity for friendships to be reciprocated, and the predictive effect of grade difference on friendship formation, the difference in grade level has no significant effect on the tendency to form mutual friendships (p -value = 0.607).

4.2 Interaction effects

For binary ERGMs, interactions between dyad-independent `ergm` terms can be specified in a manner similar to `lm` and `glm` via the `:` and `*` operators. (See Section 4.1 for a definition of dyad-independent.)

Let us first consider the colon (`:`) operator. Generally, if term A creates p_A statistics and term B creates p_B statistics, then $A:B$ will create $p_A \times p_B$ new statistics. If A and B are dyad-independent terms, expressed for $a = 1, \dots, p_A$ and $b = 1, \dots, p_B$ as

$$g_A(\mathbf{y}) = \sum_{(i,j) \in \mathbb{Y}} x_{i,j}^A y_{i,j} \text{ and } g_B(\mathbf{y}) = \sum_{(i,j) \in \mathbb{Y}} x_{i,j}^B y_{i,j}$$

for appropriate covariate matrices X^A and X^B , then the corresponding interaction term is

$$g_{A:B}(\mathbf{y}) = \sum_{(i,j) \in \mathbb{Y}} x_{i,j}^A x_{i,j}^B y_{i,j}. \quad (3)$$

As an example, consider the `Grade` and `Sex` effects, expressed as model terms via `nodefactor`, in the `faux.mesa.high` dataset:

```
summary(faux.mesa.high ~ nodefactor("Grade", levels = TRUE):nodefactor("Sex"))

## nodefactor.Grade.7:nodefactor.Sex.M nodefactor.Grade.8:nodefactor.Sex.M
##                                70                                99
## nodefactor.Grade.9:nodefactor.Sex.M nodefactor.Grade.10:nodefactor.Sex.M
##                                63                                46
## nodefactor.Grade.11:nodefactor.Sex.M nodefactor.Grade.12:nodefactor.Sex.M
##                                38                                26
```

In the call above, we deliberately include all `Grade`-factor levels via `levels=TRUE`, whereas we employ the default behavior of `nodefactor` for the `Sex` factor, which leaves out one level. Thus, the 6-level `Grade` factor and the 2-level `Sex` factor, with one level of the latter omitted, produce 6×1 interaction terms in this example.

The `*` operator, by contrast, produces all interactions in addition to the main effects or statistics. Therefore, in the scenario described above, $A*B$ will add $p_A + p_B + p_A \times p_B$ statistics to the model. Below, we use the default behavior of `nodefactor` on both the 6-level `Grade` factor and the 2-level `Sex` factor, together with an additional `edges` term, to produce a model with $1 + 5 + 1 + 5 \times 1$ terms:

```
m <- ergm(faux.mesa.high ~ edges + nodefactor("Grade") * nodefactor("Sex"))
print(summary(m), digits = 3)

## Call:
## ergm(formula = faux.mesa.high ~ edges + nodefactor("Grade") *
##       nodefactor("Sex"))
##
## Maximum Likelihood Results:
##
##                                Estimate Std. Error MCMC % z value Pr(>|z|)
## edges                        -3.028      0.173      0  -17.53 < 1e-04 ***
## nodefactor.Grade.8            -1.424      0.263      0   -5.41 < 1e-04 ***
## nodefactor.Grade.9            -1.166      0.229      0   -5.10 < 1e-04 ***
## nodefactor.Grade.10           -1.633      0.357      0   -4.58 < 1e-04 ***
## nodefactor.Grade.11           -0.328      0.237      0   -1.38  0.16714
```

```
## nodefactor.Grade.12          -0.794      0.324      0   -2.45  0.01429 *
## nodefactor.Sex.M            -1.764      0.240      0   -7.36 < 1e-04 ***
## nodefactor.Grade.8:nodefactor.Sex.M  1.386      0.202      0    6.86 < 1e-04 ***
## nodefactor.Grade.9:nodefactor.Sex.M  1.012      0.211      0    4.79 < 1e-04 ***
## nodefactor.Grade.10:nodefactor.Sex.M  1.347      0.264      0    5.11 < 1e-04 ***
## nodefactor.Grade.11:nodefactor.Sex.M  0.419      0.240      0    1.75  0.08074 .
## nodefactor.Grade.12:nodefactor.Sex.M  1.059      0.290      0    3.65  0.00026 ***
## --
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Null Deviance: 28987 on 20910 degrees of freedom
## Residual Deviance: 2189 on 20898 degrees of freedom
##
## AIC: 2213 BIC: 2308 (Smaller is better. MC Std. Err. = 0)
```

Equation (3) implies that the change statistic corresponding to dyad (i, j) is given by $x_{i,j}^A x_{i,j}^B$; that is, the change statistic for the interaction is the product of the change statistics. One may define interaction change statistics for arbitrary pairs of terms similarly—that is, by taking the interaction change statistic as the product of the corresponding change statistics—though in the case of dyad-dependent terms it is unclear that a change statistic obtained as the product of change statistics corresponds to any ERGM sufficient statistic in the sense of Equation (1). Therefore, attempting to create interactions involving dyad-dependent terms will create an error by default in **ergm**. If one wishes to create such interactions anyway, the default behavior may be changed using the **interact.dependent** term option as described in Section 12.6.2. Interactions involving curved ERGM terms are not supported in **ergm** 4.0.

Since interaction terms are defined by multiplying change statistics dyadwise and then (for dyad-independent terms) summing over all dyads, interactions of terms are not the same as products of those terms. For instance, given a nodal covariate "a", the interaction of **nodecov**("a") with itself is different than the effect of the square of the covariate, as we observe in the case of the **wealth** covariate of the (undirected) Florentine marriage dataset:

```
data(florentine)
summary(flomarriage ~ nodecov("wealth"):nodecov("wealth") + nodecov(~wealth^2))

## nodecov.wealth:nodecov.wealth          nodecov.wealth^2
##                               284058                      187814
```

4.3 Reparametrizing the model

The operator **Sum(formulas, label)** allows arbitrary linear combinations of existing statistics to be added to the model. Suppose $\mathbf{g}_1(\mathbf{y}), \dots, \mathbf{g}_K(\mathbf{y})$ is a set of K vector-valued network statistics, each corresponding to one or more **ergm** terms and of arbitrary dimension. Also suppose that A_1, \dots, A_K is a set of known constant matrices all having the same number of rows such that each matrix multiplication $A_k \mathbf{g}_k(\mathbf{y})$ is well-defined. Then it is now possible to define the statistic

$$\mathbf{g}_{\text{Sum}}(\mathbf{y}) = \sum_{k=1}^K A_k \mathbf{g}_k(\mathbf{y}).$$

The first argument to **Sum** is a formula or a list of K formulas, each representing a vector statistic. If a formula has a left-hand side, the left-hand side will be used to define the corresponding A_k matrix: If it is a scalar or a vector, A_k will be a diagonal matrix thus multiplying each element by its corresponding element; and if it is a matrix, A_k will be used directly. When no left-hand side is given, A_k is defined as 1. To simplify this function for some common cases, if the left-hand side is "sum" or "mean", the sum (or mean) of the statistics in the formula is calculated.

As an example, consider a vector of statistics consisting of the numbers of friendship ties received by each subgroup of Sampson's monks:

```
summary(samplike ~ nodeifactor("group", levels = TRUE))

## nodeifactor.group.Loyal nodeifactor.group.Outcasts nodeifactor.group.Turks
##                               29                               13                               46
```

We may create a single statistic equal to the friendship ties received by both groups of non-Outcasts by adding the first and third components of the `nodefactor` vector, either by left-multiplying by $\begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$ or by deselecting the second component at the `nodefactor` level and summing the remaining two:

```
summary(samplike ~ Sum(cbind(1, 0, 1) ~ nodefactor("group", levels = TRUE), "nf.L_T") +
        Sum("sum" ~ nodefactor("group", levels = -2), "nf.L_T"))
```

```
## Sum~nf.L_T Sum~nf.L_T
##          75          75
```

Whereas the `Sum` operator operates on network statistics, `Curve(formula, params, map, gradient=NULL, minpar=-Inf, maxpar=+Inf, cov=NULL)` operates on the parameters. The `formula` argument specifies a vector statistic $\mathbf{g}_k(\mathbf{y})$ involving one or more terms and, if curved terms are specified, a mapping $\boldsymbol{\eta}_k(\boldsymbol{\theta})$. The remaining arguments follow the curved ERGM template: The function `map` takes arguments `x`, `n`, and `...` that maps the parameter vector (whose length and names are specified by the `params` argument) into the domain of $\boldsymbol{\eta}_k$, transforming an ERGM term $\boldsymbol{\eta}_k(\boldsymbol{\theta}_k)^\top \mathbf{g}_k(\mathbf{y})$ to $\boldsymbol{\eta}_k(\boldsymbol{\eta}_*(\boldsymbol{\theta}_k))^\top \mathbf{g}_k(\mathbf{y})$, where $\boldsymbol{\eta}_*$ is the function specified by `map`. The function `gradient` takes the same arguments as `map` and returns the gradient matrix, `minpar` and `maxpar` specify the box constraints of the domain of `map`, and `cov` provides an optional argument to `map`. If `formula` is not curved, $\boldsymbol{\eta}_k(\boldsymbol{\theta})$ is simply the identity function.

To simplify this function for some common special cases, if `map="rep"`, the parameter vector will simply be replicated to make it as long as required by $\boldsymbol{\eta}_k(\boldsymbol{\theta})$ and the gradient will be evaluated automatically. Similarly, if the user is certain that `map` is affine/linear, the gradient will be calculated automatically if `gradient="linear"` is specified.

To illustrate this, consider a simple model with the baseline edge effect and a single attractiveness effect for monks who are not Outcasts. Following are four different ways to specify this model:

```
# Calculated nodal covariate:
f1 <- samplike ~ edges + nodefactor(~group != "Outcasts")
summary(f1)
```

```
##                      edges nodefactor.group!="Outcasts".TRUE
##                      88                      75
```

```
# Transform the statistic:
f2 <- samplike ~ edges +
        Sum(cbind(1,0,1) ~ nodefactor("group",levels=TRUE), "nf.L_T")
summary(f2)
```

```
##      edges Sum~nf.L_T
##      88      75
```

```
# Transform the parameters:
f3 <- samplike ~ edges + Curve(~nodefactor("group", levels=TRUE), "nf.L_T",
                             function(x,n,...) c(x,0,x), gradient="linear")
summary(f3)
```

```
##      edges      nodefactor.group.Loyal nodefactor.group.Outcasts
##      88      29      13
##      nodefactor.group.Turks
##      46
```

```
# Select groups, replicate parameter:
f4 <- samplike ~ edges + Curve(~nodefactor("group", levels=-2), "nf.L_T", "rep")
summary(f4)
```

```
##      edges nodefactor.group.Loyal nodefactor.group.Turks
##      88      29      46
```

We may now verify that all four fitted models return the same parameter estimates:

```
cbind(coef(ergm(f1)), coef(ergm(f2)), coef(ergm(f3)), coef(ergm(f4)))
```

```
##                [,1]      [,2]      [,3]      [,4]
## edges          -1.4423838 -1.4423838 -1.4423838 -1.4423838
## nodeifactor.group!="Outcasts".TRUE 0.6661217 0.6661217 0.6661217 0.6661217
```

In addition to the `Sum` operator, there is a `Prod` operator that at the time of this writing is implemented for positive statistics by first applying the `Log` operator (which returns the natural logarithm, `log` in R, of the statistics passed to it), then the `Sum` operator, and finally the `Exp` operator (which takes the exponential function `exp` in R). As a simple illustration, we may verify that the `Sum` and `Prod` operators do in fact produce network statistics as expected if we simply use each with a list of formulas having no left hand side:

```
summary(faux.dixon.high ~ edges + mutual + Sum(list(~edges, ~mutual), "EdgesAndMutual")
        + Prod(list(~edges, ~mutual), "EdgesAndMutual"))
```

```
##                edges                mutual    Sum~EdgesAndMutual  Exp~Sum~EdgesAndMutual
##                1197                219                1416                262143
```

5 Sample space constraints

In Section 1, we saw that the sample space \mathcal{Y} is a subset of the power set $2^{\mathbb{Y}}$ of all potential relationships. Many applications in fact take $\mathcal{Y} = 2^{\mathbb{Y}}$, though it is sometimes desirable to restrict the sample space by placing constraints on which elements of $\mathcal{Y} \subseteq 2^{\mathbb{Y}}$ are allowed in \mathcal{Y} . As a simple example, a bipartite network allows only edges connecting nodes from one subset, or mode, to nodes from its complement. This particular constraint is so commonly used that it is hard-coded into **network** and **ergm**. As another example, consider the inverse of a bipartite setting, in which edges are only allowed *within* subsets of the node set, a situation often referred to as a block-diagonal constraint. As still another, some applications impose a cap on the degree of any node, which constrains the sample space to include only those networks in which every node has a legal degree.

In all of the cases above, correct statistical inference for ERGMs depends on correctly incorporating constraints into the fitting process. They are specified using the `constraints` argument, a one-sided formula whose terms specify the constraints on the sample space. For example, `constraints = ~edges` specifies $\mathcal{Y}^{\text{edges}} = \{\mathbf{y}' \in \mathcal{Y} : |\mathbf{y}'| = |\mathbf{y}|\}$, where \mathbf{y} is the observed network, specified on the left-hand side. Some constraints, such as `fixedas(y1,y0)` focus on constraining \mathbb{Y} —in this case, as $\mathbb{Y}^{\text{fixedas}(y1,y0)} = \{(i,j) \in \mathbb{Y} : (i,j) \in \mathbf{y}^1 \wedge (i,j) \notin \mathbf{y}^0\}$.

Multiple constraints can be specified on a formula, separated by `+` to impose a new constraint in addition to prior or (in some instances) `-` to relax preceding constraints. Earlier versions of the **ergm** package implemented a number of constraints, as described for example in Section 3 of Morris et al. (2008). Since that time, many additional types of constraints and methods for imposing them have been added, some of which we describe in this section. A full list of currently implemented constraints is obtained via `?'ergm-constraints'`.

5.1 Arbitrary combinations of dyad-independent constraints

In general, every combination of constraints requires a somewhat different Metropolis–Hastings proposal algorithm, and so it may be impractical support every possible combination of constraints. *Dyad-independent* constraints, which affect \mathcal{Y} only through \mathbb{Y} , and do not induce stochastic dependencies among the dyad states, are an exception. These include constraining specific dyads (such as the above-mentioned **observed** and **fixedas** constraints), dyads incident on specific actors (such as the **egocentric** constraint), and block-diagonal structure; and *any* combination of dyad-independent constraints is a dyad-independent constraint. For some such combinations, **ergm** and other packages provide optimized implementations. For the rest, **ergm** falls back to a general but efficient implementation that uses run-length encoding tools provided by package **rle** (Krivitsky, 2020) to efficiently store sets of non-constrained dyads and rejection sampling to efficiently select a dyad for the proposal.

Here, we illustrate some of **ergm**’s capabilities using a dataset due to Coleman (1964) that is small enough that computational inefficiency will not present problems. These data are self-reported friendship ties among 73 boys measured at two time points during the 1957–1958 academic year and they are included as a $2 \times 73 \times 73$ array and documented in the **sna** package. We use the Coleman data to create a **network** object with 2×73 nodes:

```

library(sna)
data(coleman)
cole <- matrix(0, 2 * 73, 2 * 73)
# Upper left and lower right blocks:
cole[1:73, 1:73] <- coleman[1, , ]
cole[73 + (1:73), 73 + (1:73)] <- coleman[2, , ]
# Upper right and lower left blocks:
diag(cole[1:73, 73 + (1:73)]) <- diag(cole[73 + (1:73), 1:73]) <- 1
ncole <- network(cole)
ncole %v% "Semester" <- rep(c("Fall", "Spring"), each = 73)
ncole

## Network attributes:
## vertices = 146
## directed = TRUE
## hyper = FALSE
## loops = FALSE
## multiple = FALSE
## bipartite = FALSE
## total edges= 652
## missing edges= 0
## non-missing edges= 652
##
## Vertex attribute names:
## Semester vertex.names
##
## No edge attributes

```

By construction, the `ncole` network includes the Fall 1957 semester data and the Spring 1958 data as the upper left 73×73 and lower right 73×73 blocks, respectively. In addition, the upper right and lower left blocks indicate which nodes are the same person; that is, $y_{i,j} = 1$ whenever i and j are the same boy measured at two different times. This latter information is redundant because the ordering of the 73 boys is the same in both fall and spring, yet we include it to illustrate some techniques using entries that are not on the main block diagonal and because in principle it might not always be the case that the same individuals are observed at both time points.

5.2 Constraints via the Dyads operator

The `Dyads(fix=NULL, vary=NULL)` operator takes one or two `ergm` formulas that may contain only dyad-independent terms. For the terms in the `fix=` formula, dyads that affect the network statistic (i.e., have nonzero change statistic) for *any* the terms will be fixed at their current values. For the terms in the `vary=` formula, only those that change *at least one* of the terms will be allowed to vary, and all others will be fixed. If both formulas are given, the dyads that vary either for one or for the other will be allowed to vary. A formula passed without an argument name will default to `fix=`, for consistency with other constraints' semantics.

The key to our treatment of the `ncole` network using the `Dyads` operator is the `Semester` vertex attribute:

```
table(ncole %v% "Semester")
```

```

##
## Fall Spring
## 73 73

```

In particular, the `nodematch("Semester")` term has a change statistic equal to one for exactly those dyads representing boys measured during the same semester, and this change statistic is zero otherwise. Therefore, in our 146-node directed network there are 146×145 , or 21,170, total dyads, of which $2 \times 73 \times 72$, or 10,512, have nonzero change statistics for `nodematch("Semester")`. We can easily see exactly how many total edges there are and how many of these are in the upper left or lower right blocks:

```
summary(ncole ~ edges + nodematch("Semester"))
```

```

##          edges nodematch.Semester
##          652          506

```


We can now calculate directly the log-odds, or logit, for both the block diagonal and the off-block diagonal subnetworks, then verify that the `Dyads` operator can accomplish these same calculations using a constrained ERGM. First, we fix dyads with nonzero change statistics, which corresponds to the block off-diagonal entries:

```
logit <- function(p) log(p/(1-p))
cbind(logit((652 - 506) / (21170 - 10512)),
      coef(ergm(ncole ~ edges, constraints = ~Dyads(fix = ~nodematch("Semester")))))

##           [,1]      [,2]
## edges -4.276666 -4.276666
```

Next, we allow dyads with nonzero change statistics to vary, which corresponds to the block diagonal entries:

```
cbind(logit(506 / 10512),
      coef(ergm(ncole ~ edges, constraints = ~Dyads(vary = ~nodematch("Semester")))))

##           [,1]      [,2]
## edges -2.984404 -2.984404
```

If we remove the constraints entirely, `ncole` has 652 edges out of a possible 21,170:

```
cbind(logit(652/21170), coef(ergm(ncole ~ edges)))

##           [,1]      [,2]
## edges -3.449013 -3.449013
```

A significant limitation of this specific constraint is that its initialization requires testing every possible dyad and therefore takes up time in proportion to the square of the number of nodes.

5.3 Constraints via blocks

The `blocks` operator constrains changes to any dyads that involve certain pairs of categories defined by a particular nodal covariate. We may reproduce the examples of Section 5.2 using `blocks`. First, consider the full complement of statistics produced by the `nodemix` model term:

```
summary(ncole ~ nodemix("Semester", levels = TRUE, levels2 = TRUE))

##      mix.Semester.Fall.Fall  mix.Semester.Spring.Fall  mix.Semester.Fall.Spring
##                        243                        73                        73
## mix.Semester.Spring.Spring
##                        263
```

The `levels = TRUE` argument ensures that `nodemix` considers every value of "group" in constructing a mixing matrix of possible dyad combinations. The `levels2 = TRUE` argument ensures that, from the full complement of such possible combinations, every one is included as a statistic. By default, `levels = TRUE` whereas `levels2 = -1`, since we frequently want to exclude at least one possible mixing combination to avoid collinearity in a model that also includes the `edges` term.

We may now use `levels2` in conjunction with `blocks` to select exactly which of the `nodemix` combinations should be constrained as fixed to reproduce the examples of Section 5.2. First, we fix all dyads where the `group` values do not match:

```
coef(ergm(ncole ~ edges, constraints = ~blocks("Semester", levels2 = c(1, 4))))

##      edges
## -4.276666
```

Second, we fix the dyads where `group` values do match:

```
coef(ergm(ncole ~ edges, constraints = ~blocks("Semester", levels2 = c(2, 3))))
```

```
##      edges
## -2.984404
```

Additional examples using `levels2`, among other nodal attribute features, are contained in the `nodal_attributes` vignette within the **ergm** package.

5.4 Additional constraints

Multiple different constraints on the sample space of possible networks, as defined by the values of certain network statistics, may be implemented beyond those discussed already in this section. The `bd` constraint, for instance, may be used to enforce a maximum allowable degree for any node, via the `maxout` argument. This capability is exploited by the MCMC proposals introduced in Section 11.1 as well as the code in the Appendix. A comprehensive list of available constraints is available via `?'ergm-constraints'`.

6 Modeling Networks with Valued Edges

As of version 4.0, the **ergm** package can handle some types of networks whose ties are not merely binary, indicating presence or absence, but may have nonzero values other than unity. For example, the value of a tie might represent a count, such as the number of times a particular relationship has occurred; or it might represent an ordinal variable, if node i ranks a subset of its neighbors. Valued ties can increase complexity relative to binary ties in, for example, specifying the model and ensuring that the chosen statistics are meaningful for the types of edge values being modeled. Whether the scale of measurement of tie values is ordinal, interval, or ratio, it becomes necessary to specify the distribution of these values and to create functions to aggregate these values into ERGM statistics.

In the `ergm()`, `simulate()`, and `summary()` functions, the valued mode is typically activated by passing a `response=` argument, giving the name of the edge attribute containing the value of the response. Non-edges are assumed to have value 0. The argument may also be a formula whose right-hand side is an expression in terms of the edge attributes that evaluates to the response value and whose left-hand side, if present, gives the name to be used. If it evaluates to a logical (TRUE/FALSE) value (e.g., `response=threeContacts~(contacts>=3)`), a binary ERGM is used.

6.1 Reference specification

Krivitsky (2012) pointed out that sufficient statistics alone do not suffice to specify an ERGM on a network whose relations are valued. Consider a simple ERGM of the form

$$\Pr(\mathbf{Y} = \mathbf{y}; \theta) \propto h(\mathbf{y}) \exp \left(\theta \sum_{(i,j) \in \mathbb{Y}} y_{i,j} \right), \quad (4)$$

where $y_{i,j} \in \{0, 1, \dots\}$ is an unbounded count. If $h(\mathbf{y})$ is any constant, then $Y_{i,j} \stackrel{\text{i.i.d.}}{\sim} \text{Geometric}[p = 1 - \exp(\theta)]$.

On the other hand, if $h(\mathbf{y}) = 1 / \prod_{(i,j) \in \mathbb{Y}} y_{i,j}!$, then $Y_{i,j} \stackrel{\text{i.i.d.}}{\sim} \text{Poisson}[\mu = \exp(\theta_k)]$. For this reason, Krivitsky (2012) called a distribution with $h(\mathbf{y}) = 1$ and a sample space of nonnegative integers a *Geometric-reference ERGM* and one with $h(\mathbf{y}) = 1 / \prod_{(i,j) \in \mathbb{Y}} y_{i,j}!$ a *Poisson-reference ERGM*.

For `ergm()`, `simulate()`, and other functions, reference distributions are specified with a `reference=` argument, which is a one-sided formula with one term. The **ergm** package allows `Unif(a,b)` and `DiscUnif(a,b)` references, specifying $h(\mathbf{y}) = 1$, the former on a dyad space $y_{i,j} \in [a, b]$, the latter on $y_{i,j} \in \{a, a+1, \dots, b\}$. A companion package, **ergm.count**, allows the additional references `Poisson` and `Geometric`, described above, as well as `Binomial(trials)` for $h(\mathbf{y}) = \prod_{(i,j) \in \mathbb{Y}} \binom{n_{\text{trials}}}{y_{i,j}}$ in the case $y_{i,j} \in \{0, \dots, n_{\text{trials}}\}$. For rank-order relational data, a `CompleteOrder` reference distribution is implemented in the **ergm.rank** package for situations where rankings are compete. Where ties are permitted, `DiscUnif()` can be used as a reference. See Krivitsky & Butts (2019) for further details on both the **ergm.count** and **ergm.rank** packages, and their vignettes.

Reference distributions are explained in more detail in Section 3 of Krivitsky & Butts (2019). This reference also illustrates how the **network** package may be used to visualize some kinds of valued networks (Section 2)

and even how the **latentnet** package can handle latent-space models with valued ties (Section 4). Online help on the reference distributions that are implemented by all packages currently loaded in an R session can be obtained by typing `help("ergm-references")`.

6.2 Dyad-Independent statistics

As in Section 4.1, a component of the vector $\mathbf{g}(\mathbf{y})$ is called a dyad-independent statistic if, when one builds an ERGM using it as the *only* model statistic, the joint distribution (1) of the network factors into the product of its marginal dyad distributions. That is, the univariate version of equation (1) may be written

$$\Pr(\mathbf{Y} = \mathbf{y}; \theta) = \prod_{(i,j) \in \mathbb{Y}} \Pr(Y_{i,j} = y_{i,j}) = \frac{h(\mathbf{y})}{\kappa_{h,\eta,g}(\theta, \mathcal{Y})} \prod_{(i,j) \in \mathbb{Y}} \exp\{\eta(\theta) g_{i,j}(\mathbf{y})\} \quad (5)$$

for $\mathbf{y} \in \mathcal{Y}$ and for some appropriately chosen $g_{i,j}(\mathbf{y})$. Equation (4) shows that the sum of the values $y_{i,j}$, which implies $g_{i,j}(\mathbf{y}) = y_{i,j}$, is one such example. Another example is the sum of the nonzero indicators that arises if we define $g_{i,j}(\mathbf{y}) = \mathbb{I}\{y_{i,j} > 0\}$. Each of these basic dyad-independent statistics is implemented in **ergm**:

sum(pow=1) *Sum of edge values* This is simply the summation of edge values. For most valued ERGMs, this is the natural intercept term. In particular, for reference distributions such as **Poisson** and **Binomial**, using this term produces intercept effects of Poisson log-linear and binomial logistic regressions, respectively. Optionally, the dyad values can be raised to a power before being summed.

nonzero *Number of nonzero edge values* This term counts nonzero edge values. It can be used to model zero-inflation that is common in networks: It is often the case that a network is sparse but has edges with relatively high weights when they are present.

Binary ERGM statistics cannot be used directly for valued networks nor vice versa, but most dyad-independent binary ERGM statistics have been generalized by imposing a covariate on one of the two above forms. They have the same arguments as their binary ERGM counterparts, with an additional argument: **form**, which has two possible values: **"sum"** (the default) and **"nonzero"**. The former creates a statistic of the form $\sum_{(i,j) \in \mathbb{Y}} x_{i,j} y_{i,j}$, where $y_{i,j}$ is the value of dyad (i,j) and $x_{i,j}$ is the term's covariate associated with it. The latter computes a sum of indicator variables, one for each dyad, indicating whether the corresponding edge has a nonzero value. When **form="sum"** is used, typically a GLM-like effect results, whereas **form="nonzero"** can be used to model sparsity effects. (Krivitsky, 2012) Krivitsky & Butts (2019) gives an example of the **form=** argument with the **nodematch** term.

Other terms that control a dyad's distribution are **atleast(threshold = 0)**, **atmost(threshold = 0)**, **equalto(value = 0, tolerance = 0)**, **greaterthan(threshold = 0)**, **ininterval(lower = -Inf, upper = +Inf, open = c(TRUE, TRUE))**, and **smallerthan(threshold = 0)**. Each of these terms counts the dyad values that satisfy the criterion identified by its name.

6.3 Mutuality

The binary **mutuality** term in **ergm** counts the number of pairs of mutual ties. Its valued counterpart is **mutuality(form)**, which permits the following values of **form**. For each of these, a higher coefficient will tend to increase the similarity of reciprocating dyad values.

"product" *Sum of products of reciprocating edge values* This is the most direct generalization. However, for a **Poisson**-reference ERGM in particular, a positive coefficient on this term produces an infinite normalizing constant and therefore lies outside the parameter space.

"geometric" *Sum of geometric mean of reciprocating edge values* This form solves the product form's problem by taking a square root of the product. It can be viewed as the uncentered covariance of variance-stabilized counts.

"min" *Minimum of reciprocating edge values* This effect is, perhaps, the easiest to interpret, at the cost of statistical power.

"nabsdiff" *Absolute difference of reciprocating edge values* This effect is more symmetrical than **min**.

We refer the reader to Krivitsky (2012) for a further discussion of the effects.

6.4 Actor heterogeneity

Different actors may have different overall propensities to interact. This has been modeled using random effects, as in the p_2 model, and using degeneracy-prone terms like k -star counts. For valued ERGMs, the following term, also introduced by Krivitsky (2012) and discussed in more detail there, models actor heterogeneity:

nodesqrtcovar(center, transform) *Covariance between $y_{i,j}$ incident on same actor* The default **transform="sqrt"** will take a square root of dyad values before calculating, and the default **center=TRUE** will center the transformed values around their global mean, gaining stability at the cost of locality.

6.5 Triadic effects

To generalize the notion of triadic closure, **ergm** implements very flexible **transitiveweights(twopath, combine, affect)** and similar **cyclicalweights** statistics. The transitive weight statistic has the general form

$$g_v(\mathbf{y}) = \sum_{(i,j) \in \mathbb{Y}} v_{\text{affect}}(y_{i,j}, v_{\text{combine}}(v_{2\text{-path}}(\mathbf{y}_{i,k}, \mathbf{y}_{k,j})_{k \in N \setminus \{i,j\}})),$$

which can be customized by varying three functions:

- $v_{2\text{-path}}$** Given $\mathbf{y}_{i,k}$ and $\mathbf{y}_{k,j}$, what is the strength of the two-path they form? The options are "min", to take the minimum of the two-path's constituent values, and "geomean", to take their geometric mean, gaining statistical power at a greater risk of model instability.
- v_{combine}** Given the strengths of the two-paths $\mathbf{y}_{i \rightarrow k \rightarrow j}$ for all $k \neq i, j$, what is the combined strength of these two-paths between i and j ? The choices are "max", for the strength of the strongest of the two-paths—analogueous to **transitivity** or **gwesp(0)** binary ERGM effects—and "sum", the sum of the path strengths. The latter choice is better able to detect effects but is more subject to degeneracy; it is analogueous to **triangles**.
- v_{affect}** Given the combined strength of the two-paths between i and j , how should they affect $Y_{i,j}$? The choices are "min", the minimum of the combined strength and the focus two-path, and "geomean", again more able to detect effects but more likely to cause degeneracy.

Usage of the **transitiveweights** and **cyclicalweights** terms is illustrated in Section 3.1 of Krivitsky & Butts (2019).

6.6 Using binary ERGM terms in valued ERGMs

ergm also allows general binary terms to be passed to valued models. The mechanism that allows this is the operator term **B(formula, form)**, which is further described in the **ergm** online help under **help("ergm-terms")**. Here, **formula=** is a one-sided formula whose right hand side contains the binary **ergm** terms to be used. Allowable values of the **form** argument are **form="sum"** and **form="nonzero"**, which have the effects described in Section 6.2, with **form="sum"** only valid for dyad-independent **formula=** terms; or a one-sided formula may be passed to **form=**, containing one *valued ergm* term, with the following properties:

- dyadic independence;
- dyadwise contribution of either 0 or 1;
- dyadwise contribution of 0 for a 0-valued dyad.

That is, it must be expressible as

$$g(y) = \sum_{(i,j) \in \mathbb{Y}} g_{i,j}(y_{i,j}),$$

where for all i, j , and \mathbf{y} , $g_{i,j}(y_{i,j}) \in \{0, 1\}$ and $g_{i,j}(0) \equiv 0$. Such terms include **nonzero**, **ininterval()**, **atleast()**, **atmost()**, **greaterthan()**, **lessthan()**, and **equalto()**. The operator will then construct a binary network \mathbf{y}^B such that $y_{i,j}^B = 1$ if and only if $g_{i,j}(y_{i,j}) = 1$, and evaluate the binary terms in **formula=** on it.

6.7 Modeling Ordinal Values Using Binary Operator Terms

To illustrate the use of binary ergm terms on a valued network as described above, we construct an example that uses the `B` (for “binary”) operator. The code snippet below gives an example of a valued ergm that uses the `DiscUnif`, or discrete uniform, reference distribution, which is included in the **ergm** package itself; that is, there is no need to load the **ergm.count** or **ergm.rank** packages to run the following example. The example fits a multinomial logistic regression model that assumes that the edge values independent of one another and take ordinal values that have the same interpretation for each dyad. (In general, rating and ranking data may not allow edge values to be compared across egos (Krivitsky & Butts, 2017); the **ergm.rank** package contains terms that remain valid in this more complex setting.) Models for independently observed ordinal random variables have a long history in the statistical literature; relevant references specific to network models include Robins et al. (1999) and, in a Bayesian framework, Caimo & Gollini (2020).

First, we build a valued network by pooling the three binary friendship nomination networks due to Sampson (1968), exactly as in Section 2.1 of Krivitsky & Butts (2019).

```
data(samplk)
# Create a sociomatrix totaling the nominations.
samplk.tot.m <- as.matrix(samplk1) + as.matrix(samplk2) + as.matrix(samplk3)
samplk.tot <- as.network(samplk.tot.m, directed=TRUE, matrix.type="a",
                        ignore.eval=FALSE, names.eval="nominations")
```

We will use the `B` operator to construct new statistics consisting of the number of edges with value k or higher, where k is 1, 2, or 3.

```
summary(samplk.tot ~ B(~edges, ~atleast(1)) + B(~edges, ~atleast(2))
        + B(~edges, ~atleast(3)), response = "nominations")
```

```
## B(atleast(1))~edges B(atleast(2))~edges B(atleast(3))~edges
##                88                50                30
```

Since there are 18×17 , or 306, possible edges, the summary statistics above tell us that the valued network we have constructed has 30 edges with value 3, 20 with value 2, 38 with value 1, and the remaining 218 with value 0. The ERGM with these statistics has independent edges, where the probabilities an edge takes the values 0, 1, 2, or 3 are given by $1/D$, $\exp\{\theta_1\}/D$, $\exp\{\theta_1 + \theta_2\}/D$, and $\exp\{\theta_1 + \theta_2 + \theta_3\}/D$, respectively, where

$$D = 1 + \exp\{\theta_1\} + \exp\{\theta_1 + \theta_2\} + \exp\{\theta_1 + \theta_2 + \theta_3\}.$$

We may verify that **ergm**’s stochastic fitting algorithm obtains MLEs very close to the exact values:

```
mod <- ergm(samplk.tot ~ B(~edges, ~atleast(1)) + B(~edges, ~atleast(2))
            + B(~edges, ~atleast(3)), response = "nominations", reference = ~DiscUnif(0,3))
coef(mod) # Approximate MLEs for theta1, theta2, and theta3
```

```
## B(atleast(1))~edges B(atleast(2))~edges B(atleast(3))~edges
##          -1.7648623          -0.6045124          0.4053206
```

```
true <- c(EdgeVal0=218, EdgeVal1=38, EdgeVal2=20, EdgeVal3=30)
est <- c(1, exp(cumsum(coef(mod))), use.names=FALSE)
rbind(True_Proportions = true / sum(true), Estimated_Proportions = est / sum(est))
```

```
##                EdgeVal0 EdgeVal1 EdgeVal2 EdgeVal3
## True_Proportions    0.7124183 0.1241830 0.06535948 0.09803922
## Estimated_Proportions 0.7117245 0.1218546 0.06657414 0.09984677
```

This example could have used the `equalto` terms in place of all the `atleast` terms above. Then, the estimated proportions would have been proportional to 1, $\exp\{\theta_1\}$, $\exp\{\theta_2\}$, and $\exp\{\theta_3\}$ instead of 1, $\exp\{\theta_1\}$, $\exp\{\theta_1 + \theta_2\}$, and $\exp\{\theta_1 + \theta_2 + \theta_3\}$. Such a model does not assume ordinality of the edge values, so it could be used for a multinomial logit model in which the edges take categorical non-ordered values.

7 Markov chain Monte Carlo enhancements

The simulation of random networks distributed according to a particular ERGM with a known value of the parameter η is central to nearly all functionality of the **ergm** package. Clearly simulation is useful to examine population characteristics of an ERGM using Monte Carlo methods; potentially less obvious is the role that simulation plays in the process of maximum likelihood estimation itself. Markov chain Monte Carlo (MCMC) methods are the means by which **ergm** implements simulation of networks, and these methods are therefore the workhorses of the package.

7.1 Summary of Metropolis–Hastings algorithms

As explained by Hunter et al. (2008), the goal of MCMC is to create a Markov chain whose stationary distribution is exactly equal to the ERGM with a given value of η . After letting the chain run for a long time, its state may be taken to be an approximate draw from the ERGM in question. The **ergm** package does this via a Metropolis–Hastings algorithm, a special case of MCMC in which at each iteration, a move from the current network to a new network is proposed according to some probability distribution. The M-H algorithm operates by allowing only two possibilities following this proposal: Either the chain remains at the current network for the next iteration, or the proposed network becomes the current network for the next iteration. The latter possibility occurs with probability

$$\min \left\{ 1, \frac{\Pr_{\theta, \mathbf{y}, h, \eta, \mathbf{g}}(\mathbf{Y} = \mathbf{y}^{\text{proposed}})}{\Pr_{\theta, \mathbf{y}, h, \eta, \mathbf{g}}(\mathbf{Y} = \mathbf{y}^{\text{current}})} \times \frac{q(\mathbf{y}^{\text{current}} \mid \mathbf{y}^{\text{proposed}})}{q(\mathbf{y}^{\text{proposed}} \mid \mathbf{y}^{\text{current}})} \right\}, \quad (6)$$

where q denotes the proposal distribution; more specifically, $q(\mathbf{a} \mid \mathbf{b})$ is the probability of proposing \mathbf{a} if the current state is \mathbf{b} .

It is useful to introduce a *change statistic* or *change score*

$$\Delta_{i,j} \mathbf{g}(\mathbf{y}) \stackrel{\text{def}}{=} \mathbf{g}[\mathbf{y} \cup \{(i, j)\}] - \mathbf{g}[\mathbf{y} \setminus \{(i, j)\}], \quad (7)$$

the effect on the vector of statistics if one were to change the state of the (i, j) relationship from 0 to 1 while holding the rest of the network \mathbf{y} fixed. Let us assume for now that q only allows for changing, or toggling, at most one dyad, which is to say that $q(\mathbf{a} \mid \mathbf{b})$ must be zero whenever \mathbf{a} and \mathbf{b} differ by more than a single edge. If we call the ratio in Expression (6) the “acceptance ratio,” or AR, then for the proposed toggle of dyad $y_{i,j}$,

$$\log \text{AR} = \pm \eta^\top \Delta_{i,j} \mathbf{g}(\mathbf{y}) + \log \frac{q(\mathbf{y}^{\text{current}} \mid \mathbf{y}^{\text{proposed}})}{q(\mathbf{y}^{\text{proposed}} \mid \mathbf{y}^{\text{current}})}, \quad (8)$$

where the sign in front of $\eta^\top \Delta_{i,j} \mathbf{g}(\mathbf{y})$ is positive when $y_{i,j}^{\text{proposed}} = 1$ and negative when $y_{i,j}^{\text{proposed}} = 0$.

It is useful to consider a couple of special cases of the Metropolis–Hastings algorithm (6). When we define $q(\mathbf{y}^{\text{proposed}} \mid \mathbf{y}^{\text{current}})$ to be proportional to $\Pr_{\theta, \mathbf{y}, h, \eta, \mathbf{g}}(\mathbf{Y} = \mathbf{y}^{\text{proposed}})$, the value of AR is always 1, which implies that the proposal is always accepted and the resulting algorithm is called Gibbs sampling. Another special case is the symmetric proposal, in which $q(\mathbf{a} \mid \mathbf{b}) = q(\mathbf{b} \mid \mathbf{a})$ for all \mathbf{a} and \mathbf{b} , in which case $\log \text{AR}$ is simply $\pm \eta^\top \Delta_{i,j} \mathbf{g}(\mathbf{y})$. In particular, perhaps the most basic network-based Metropolis–Hastings algorithm for binary networks with N possible edges operates by selecting $\mathbf{y}^{\text{proposed}}$ uniformly from among all N networks that differ from $\mathbf{y}^{\text{current}}$ by exactly one edge toggle; thus, $q(\mathbf{y}^{\text{proposed}} \mid \mathbf{y}^{\text{current}})$ equals N^{-1} or 0, depending on whether or not $\mathbf{y}^{\text{proposed}}$ and $\mathbf{y}^{\text{current}}$ differ by exactly one toggle.

Simulation of networks from an ERGM using MCMC can be done using `simulate`, documented at `?simulate.ergm`. In place of its original `statsonly=` argument, `simulate` methods now take a more versatile `output` argument, which defaults to “network” for returning a list of generated network objects, “stats” for network statistics, “edgelist” for a more compact representation of the network, or a user-defined function to be evaluated on the sampler state and returned. For example, the following code produces triangle counts for fifty random undirected 10-node networks where each edge occurs independently with probability 2/3. Also, the model statistics—in this case, edge counts—are attached to the result as an attribute “stats”:

```
# Below, we use the fact that logit(2/3) = log(2)
triangles <- function(nwState, ...) summary(as.network(nwState) ~ triangles)
nw10 <- network(10, directed = FALSE)
out <- simulate(nw10 ~ edges, nsim = 50, coef = log(2), output = triangles)
unlist(out, use.names = FALSE)
```

```
## [1] 44 28 33 39 30 48 46 23 28 39 20 41 41 43 32 32 26 30 35 42 37 40 48 27 30 38 25 33 24 12
## [31] 28 16 14 37 18 32 35 36 28 36 28 26 32 55 26 28 45 49 19 28
```

```
head(attr(out, "stats"))
```

```
##      edges
## [1,]    32
## [2,]    29
## [3,]    30
## [4,]    31
## [5,]    29
## [6,]    33
```

7.2 MCMC Proposal hints

In Equation (6), basically any proposal distribution q leads in theory to a Markov chain with the correct stationary distribution. In practice, however, some choices of q may result in rejection of nearly all proposals, a situation informally called “slow mixing.” Thus, it is helpful to have access to different proposals to deploy in different situations.

For example, most real-world social networks are sparse: The vast majority of potential ties are not realized. This results in the basic uniform proposal spending a lot of computing effort proposing toggles to non-edges that are rejected by the Metropolis–Hastings algorithm.

The TNT, or tie/no tie, proposal was introduced in the **ergm** package specifically to address this slow mixing due to sparsity. As explained in Morris et al. (2008), TNT (approximately) uniformly distributes probability mass $1/2$ each over the set of non-edges and the set of existing edges. For sparse networks, the latter set is much smaller than the former, so TNT speeds mixing by making many more ‘off’-toggle proposals than would occur if all dyads had the same probability of being proposed for a toggle.

ergm 4.0 introduces a concept of a *hint* to enable the user to inform the sampling and estimation algorithm about the properties of the network model that, while they do not affect its stationary distribution, can be helpful in sampling. This information is specified via the `MCMC.prop=` or `obs.MCMC.prop=` control parameters as a one-sided formula. For example, `MCMC.prop=~sparse` (the default) informs the proposal selection algorithm that the sampling should be optimized for sparse networks, which typically means enabling the above-described TNT algorithm. As another example, the code in the Appendix that creates the output from Section 11 includes the following line within an **ergm** call:

```
MCMC.prop = ~strat(attr = ~race, empirical = TRUE) + sparse
```

The `strat` hint in this case instructs the proposal distribution to take the `race` attribute of nodes into account when proposing dyads to toggle; in particular, `empirical = TRUE` instructs the proposal to weight every possible node-pair `race` combination according to the proportions of such combinations observed in the network used at the beginning of the Markov chain. Alternatively, the user may pass the `strat` hint an explicit matrix of weights via the `pmat` argument. Additional information about hints currently implemented in **ergm** is available via `?'ergm-hints'`.

7.3 MCMC Proposal constraints

As explained in Section 5, it is possible to constrain the sample space of possible networks, which in essence means that some networks have a probability of zero. Technically, such constraints need not influence our choice of q in Equation (6), since any proposed network whose probability under the model is zero cannot be accepted by the Metropolis–Hastings algorithm. However, in practice it is a waste of computing effort to propose such networks in the first place. The **ergm** package allows certain types of constraints to be respected by q , leading to substantial gains in efficiency when these constraints exist.

The new **ergm** proposal `BDStratTNT`, in addition to supporting the `strat` and `sparse` hints described in Section 7.2, allows the user to fix edge states of dyads of specified mixing types according to a vertex attribute via the `blocks` constraint. It also allows for upper bounds on a node’s degree via the `bd` constraint’s `maxout`, `maxin`, and `attribs` arguments. Documentation for all MCMC proposals is available via `help('ergm-proposals')`.

In addition, the **tergm** package includes a generalization of **BDStratTNT** that specifically supports dynamic models by considering a dyad’s discordance, i.e., whether the dyad is in the same state that it was in at the beginning of the time step when the Markov chain for that step was initialized. Details about the **tergm** proposals are available via `help('ergm-proposals', package=tergm)`.

As an example application of the **BDStratTNT** proposal, the lengthy **ergm** calls in the Appendix, which produce the examples of Section 11, contain the following line:

```
constraints = ~bd(maxout = 1) + blocks(attr = ~sex, levels2 = diag(TRUE, 2))
```

This line ensures that the sample space of possible networks includes only networks in which no node has a degree of 2 or more—the networks in these examples are undirected—and in which no changes in dyad status between nodes of the same **sex** value are allowed. These constraints are used to model a network of monogamous heterosexual relationships.

7.4 Adaptive MCMC via effective sample size

ergm 4.0 implements adaptive MCMC sampling. MCMC-based approximate maximum likelihood estimation, sometimes abbreviated MCMLE, for an ERGM depends only on the MCMC sample of the sufficient statistic values and is agnostic to the underlying graphs once the statistics have been calculated (Hunter & Handcock, 2006; Krivitsky, 2012; Krivitsky & Butts, 2017). Furthermore, while different algorithms approach the problem in different ways, the estimation ultimately entails matching the mean of the simulated statistic under θ to the observed statistic. Thus, the sampling algorithm can focus on obtaining a particular *effective sample size* of the multivariate sufficient statistic.

The user or the estimation algorithm specifies the target effective sample size, typically via the control parameter `MCMC.effectiveSize` or, for estimation, `MCMLE.effectiveSize`, as well as the initial MCMC thinning interval (`MCMC.interval`) and sample size (`MCMC.samplesize`). The algorithm then iterates the following steps:

1. Run the Markov chain `MCMC.samplesize`×`MCMC.interval` steps forward to obtain an initial sample of size `MCMC.samplesize`.
2. If the size of the Markov chain’s cumulative sample size exceeds `2`×`MCMC.samplesize`, discard every other draw and double `MCMC.interval` for future runs.
3. Identify a candidate “burn-in” period by fitting a multivariate broken stick regression model to the sampled statistics or estimating functions. That is, considering an MCMC sample of S statistics $\mathbf{g}(\mathbf{Y}^{(1)}), \dots, \mathbf{g}(\mathbf{Y}^{(S)})$, we find a least-squares fit for

$$\boldsymbol{\eta}'(\boldsymbol{\theta})^\top \mathbf{g}(\mathbf{Y}^{(s)}) = \beta_0 + \beta_1 \max(0, s_0 - s) + \mathbf{e}^{(s)}, \quad s = 1, \dots, S,$$

where $s_0 > 0$ is the candidate burn-in, $\beta_0, \beta_1 \in \mathbb{R}^q$ are (nuisance) parameters, and $\mathbf{e}^{(s)} \in \mathbb{R}^q$ are the residuals. In practice, this estimator has a closed form given s_0 , so we perform a bisection search over the possible s_0 .

4. Evaluate a multivariate extension of the Geweke (1991) convergence diagnostic after discarding sample units up to the estimated s_0 .
5. If nonconvergence is detected, repeat from Step 1, accumulating draws.
6. Calculate the effective sample size of the retained draws using the method of Vats et al. (2019). If satisfied, return.
7. Extrapolate to estimate the additional number of Markov chain steps to obtain the target effective sample size given the current ratio of the sample size to the effective sample size. Advance the estimated number of steps, accumulating draws.
8. Continue from Step 2.

8 Maximum likelihood estimation enhancements

Frequentist inference for ERGMs calls for finding an estimator given observed data on a network or networks, along with estimates of the variability of that estimator that are generally expressed in the form of standard errors. We consider the gold standard of estimation to be the maximum likelihood estimator (MLE), with the maximum pseudo-likelihood estimator (MPLE) is an alternative that has its own advantages and disadvantages relative to the MLE. Calculating estimates like these along with their standard errors is the core functionality of the **ergm** package, and in this section we describe multiple enhancements to the package as of version 4.0.

The likelihood function is, by definition, the function of Equation (1) when that expression is viewed as a function of $\boldsymbol{\theta}$. The natural logarithm of the likelihood function is often denoted $\ell(\boldsymbol{\theta})$. The MLE $\hat{\boldsymbol{\theta}}$ is the maximizer of $\ell(\boldsymbol{\theta})$. Alternatively, the MLE is a zero of the score function (gradient of the log-likelihood) (Hunter & Handcock, 2006, Equation 3.1), i.e., $\text{sc}(\hat{\boldsymbol{\theta}}) \stackrel{\text{def}}{=} \mathbb{E}_{\hat{\boldsymbol{\theta}}; \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}} \mathbf{U} = \mathbf{0}$, where

$$\mathbf{U}(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) = \boldsymbol{\eta}'(\boldsymbol{\theta})^\top [\mathbf{g}(\mathbf{y}^{\text{obs}}) - \mathbf{g}(\mathbf{Y})]$$

In many cases, the normalizing constant $\kappa_{h, \boldsymbol{\eta}, \mathbf{g}}(\boldsymbol{\theta}, \mathcal{Y})$ of Equation (1) is computationally intractable, and a number of computational approaches (e.g., Snijders, 2002; Hummel et al., 2012) have been proposed for approximating the MLE. The **ergm** package defaults to the importance sampling approach of Hunter & Handcock (2006): The likelihood ratio is expressed as an expectation with respect to one of the parameter configurations (that of the t th guess, denoted $\boldsymbol{\theta}^t$), and a simulation from that configuration is used to maximize this ratio with respect to $\boldsymbol{\theta}$ to obtain the next guess $\boldsymbol{\theta}^{t+1}$. In particular,

$$\boldsymbol{\theta}^{t+1} = \arg \max_{\boldsymbol{\theta}} \left(\{ \boldsymbol{\eta}(\boldsymbol{\theta}) - \boldsymbol{\eta}(\boldsymbol{\theta}^t) \}^\top \mathbf{g}(\mathbf{y}^{\text{obs}}) - \log \frac{1}{S} \sum_{s=1}^S \exp[\{ \boldsymbol{\eta}(\boldsymbol{\theta}) - \boldsymbol{\eta}(\boldsymbol{\theta}^t) \}^\top \mathbf{g}(\mathbf{y}^{\boldsymbol{\theta}^t, s})] \right), \quad (9)$$

where $\mathbf{y}^{\boldsymbol{\theta}^t, 1}, \dots, \mathbf{y}^{\boldsymbol{\theta}^t, S}$ is an approximate sample from $\text{Pr}_{\boldsymbol{\theta}^t; \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}}$ obtained via MCMC. It is because an MCMC-generated sample is used to obtain an approximate MLE that this and related procedures are sometimes called MCMCMLE; the fact that this approach is a central pillar of **ergm** underscores how integral the MCMC algorithms are to nearly all facets of the package.

8.1 Standard errors for maximum pseudo-likelihood estimation

To define the maximum pseudo-likelihood estimator (MPLE) for a binary network, we must first define the pseudo-likelihood function. To this end, let us consider that Equation (8) arises due to the fact that under the ERGM of Equation (1),

$$\log \frac{\text{Pr}_{\boldsymbol{\theta}, \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}}(\mathbf{y} \cup \{(i, j)\})}{\text{Pr}_{\boldsymbol{\theta}, \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}}(\mathbf{y} \setminus \{(i, j)\})} = \log \frac{\text{Pr}_{\boldsymbol{\theta}, \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}}(\mathbf{y} \cup \{(i, j)\} \mid (i, j)^c)}{\text{Pr}_{\boldsymbol{\theta}, \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}}(\mathbf{y} \setminus \{(i, j)\} \mid (i, j)^c)} = \boldsymbol{\eta}^\top \boldsymbol{\Delta}_{i, j} \mathbf{g}(\mathbf{y}). \quad (10)$$

If we imagine that all $y_{i, j}$ are mutually independent Bernoulli random variables with distributions given by (10)—i.e., that $\text{logit } \text{Pr}_{\boldsymbol{\theta}, \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}}(Y_{i, j} = 1) = \boldsymbol{\eta}^\top \boldsymbol{\Delta}_{i, j} \mathbf{g}(\mathbf{y})$ —then multiplying their individual probability mass functions gives a function called the *pseudo-likelihood*, whose maximum is known as the maximum pseudo-likelihood estimator (MPLE).

As we discuss in Section 12.3, the **ergm** function uses logistic regression to obtain MPLEs in non-curved models. The **glm** function in R returns not only estimated logistic regression coefficients (parameters) but also standard errors for those coefficients. Earlier versions of **ergm** had simply reported these standard errors without modification whenever the user asked for MPLE model output; however, these standard errors are inaccurate when the edges are not independent. Indeed, a straightforward Taylor approximation (Schmid & Hunter, 2021) gives

$$\text{var}_{\hat{\boldsymbol{\theta}}_{\text{MPLE}}; \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}} \approx [J(\boldsymbol{\theta})]^{-1} \text{var}_{\boldsymbol{\theta}; \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}}(\mathbf{U}(\boldsymbol{\theta})) [J(\boldsymbol{\theta})]^{-1}, \quad (11)$$

where we assume that the ERGM with parameter $\boldsymbol{\theta}$ is the true model for the distribution that gave rise to the observed network. Since $\boldsymbol{\theta}$ is of course unknown, in practice we may approximate the right hand side above by substituting $\hat{\boldsymbol{\theta}}_{\text{MPLE}}$ for $\boldsymbol{\theta}$.

In the case where the MPLE is actually the MLE, which happens when the logistic regression model coincides with the ERGM because the latter is dyad-independent, then $J(\boldsymbol{\theta}) = \text{var}_{\boldsymbol{\theta}; \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}}[\mathbf{U}(\boldsymbol{\theta})]$ for all $\boldsymbol{\theta}$, so the expression on the right hand side simplifies to $[J(\boldsymbol{\theta})]^{-1}$. Indeed, the standard errors returned by the logistic regression algorithm are those given by $[J(\hat{\boldsymbol{\theta}}_{\text{MPLE}})]^{-1}$. Yet for non-dyad-independent models, these logistic-regression-based standard errors are poor approximations to the true standard deviations of the parameter estimates. The **ergm** package therefore implements the technique described by Schmid & Hunter (2021), using Equation 11 with $\hat{\boldsymbol{\theta}}_{\text{MPLE}}$ in place of $\boldsymbol{\theta}$ to provide standard errors. For this purpose, the middle term must be estimated by the sample covariance matrix from a random sample obtained using Markov chain Monte Carlo with $\hat{\boldsymbol{\theta}}_{\text{MPLE}}$ as the true parameter value.

Alternatively, **ergm** will approximate MPLE standard errors via a bootstrap method, as proposed by Schmid & Desmarais (2017).

8.2 Log-likelihood estimation

Likelihood-based estimation relies not only on the value of the maximizer $\hat{\boldsymbol{\theta}}$ of the log-likelihood function, but also on the maximum value $\ell(\hat{\boldsymbol{\theta}})$ that function obtains. Model selection criteria such as AIC and BIC are based on this maximized log-likelihood—they are equal to $-2\ell(\hat{\boldsymbol{\theta}}) + 2p$ and $-2\ell(\hat{\boldsymbol{\theta}}) + p \log d$, respectively, where p is the number of model parameters and d is the number of observed, non-fixed potential relations in the network—as are the standard chi-squared tests based on drop-in-deviance.

Hunter & Handcock (2006) point out that the null deviance, which is equal to $-2\ell(\mathbf{0})$, is straightforward to calculate; in the case of binary networks, it equals $2e \log 2$, where again e is the number of observed, non-fixed edges. Yet log-likelihood values are sometimes computationally intractable, as mentioned in Section 1. A novel method currently employed by the **ergm** package is to first identify all dyadic dependent terms in the model, then find the MLE and corresponding log-likelihood value in the constrained parameter space that fixes the values of the coefficients corresponding to those terms at zero. This calculation is straightforward using logistic regression, as explained in Section 8.1. If we denote the MLE of this sub-model as $\tilde{\boldsymbol{\theta}}$, then our task becomes estimation of $\ell(\hat{\boldsymbol{\theta}}) - \ell(\tilde{\boldsymbol{\theta}})$, since the second term in this expression is known.

Section 5 of Hunter & Handcock (2006) addresses the problem of likelihood ratio testing, which on the logarithmic scale is exactly the problem of calculating the difference of two log-likelihoods such as $\ell(\hat{\boldsymbol{\theta}}) - \ell(\tilde{\boldsymbol{\theta}})$. That paper describes the idea of path sampling (Gelman & Meng, 1998), which is based on the following observation: if we define a smooth path in parameter space from $\tilde{\boldsymbol{\theta}}$ to $\hat{\boldsymbol{\theta}}$, that is, a differentiable function \mathbf{m} that maps the closed unit interval $[0, 1]$ into the parameter space so that $\mathbf{m}(0) = \tilde{\boldsymbol{\theta}}$ and $\mathbf{m}(1) = \hat{\boldsymbol{\theta}}$, then

$$\log \kappa_{h,\boldsymbol{\eta},\mathbf{g}}(\hat{\boldsymbol{\theta}}, \mathcal{Y}) - \log \kappa_{h,\boldsymbol{\eta},\mathbf{g}}(\tilde{\boldsymbol{\theta}}, \mathcal{Y}) = \int_0^1 \mathbb{E}_{\mathbf{m}(u); \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}} \left\{ \frac{d}{du} \boldsymbol{\eta}[\mathbf{m}(u)] \right\}^\top \mathbf{g}(\mathbf{Y}) du \quad (12)$$

by the fundamental theorem of calculus, where $\kappa_{h,\boldsymbol{\eta},\mathbf{g}}(\boldsymbol{\theta}, \mathcal{Y})$ is the normalizing constant of Equation (1). Pulling the expectation and differentiation operators outside of the integral, the expression remaining under the integral sign is also an expectation with respect to a random variable U uniformly distributed on $(0, 1)$. Thus, Equation (12) implies that

$$\log \kappa_{h,\boldsymbol{\eta},\mathbf{g}}(\hat{\boldsymbol{\theta}}, \mathcal{Y}) - \log \kappa_{h,\boldsymbol{\eta},\mathbf{g}}(\tilde{\boldsymbol{\theta}}, \mathcal{Y}) = \mathbb{E}_{\mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}} \left\{ \frac{d}{dU} \boldsymbol{\eta}[\mathbf{m}(U)] \right\}^\top \mathbf{g}(\mathbf{Y}), \quad (13)$$

where the expectation is taken with respect to the joint distribution of U and \mathbf{Y} , where $U \sim \text{Unif}(0, 1)$ and $\mathbf{Y} \mid U$ is distributed according to the ERGM of Equation (1) with parameter $\mathbf{m}(U)$. Here, we introduce a shifted version of the vector $\mathbf{g}(\cdot)$ of sufficient statistics:

$$\mathbf{z}(\mathbf{y}) \stackrel{\text{def}}{=} \mathbf{g}(\mathbf{y}) - \mathbf{g}(\mathbf{y}^{\text{obs}})$$

where if missing data are present we replace $\mathbf{g}(\mathbf{y}^{\text{obs}})$ by $\mathbb{E}_{\boldsymbol{\theta}; \mathcal{Y}, h, \boldsymbol{\eta}, \mathbf{g}} \mathbf{g}(\mathbf{Y} \mid \mathbf{y}^{\text{obs}})$; yet here we assume for simplicity of notation that $\mathbf{z}(\mathbf{y})$ does not depend on $\boldsymbol{\theta}$. If $\mathbf{z}(\cdot)$ is substituted for $\mathbf{g}(\cdot)$, then $\ell(\boldsymbol{\theta}) = -\log \kappa_{h,\boldsymbol{\eta},\mathbf{z}}(\boldsymbol{\theta}, \mathcal{Y})$, so for instance Equation (13) gives a convenient expression for $\ell(\hat{\boldsymbol{\theta}}) - \ell(\tilde{\boldsymbol{\theta}})$.

In practice, the problem with Equation (13) is that simulating the first network \mathbf{Y} from a given parameter configuration $\boldsymbol{\theta} = \mathbf{m}(U)$ requires a long “burn-in” period, which makes the direct approach of drawing a U_k then a $\mathbf{Y}_k \mid U_k$ for $k = 1, \dots, K$ impractical. On the other hand, once “burned in,” subsequent draws $\mathbf{Y}_2, \dots, \mathbf{Y}_K$ from the same distribution cost relatively little additional effort. For this reason, the **ergm** package currently implements a technique known as bridge sampling (Meng & Wong, 1996) as an approximation of Equation (13).

Bridge sampling in **ergm** partitions the unit interval into J sub-intervals, each of length $1/J$, where u_j is taken to be the center of the j th sub-interval for $j = 1, \dots, J$. For each j , we simulate a random sample $\mathbf{Y}_{j1}, \dots, \mathbf{Y}_{jK}$ of networks from the ERGM with parameter $\mathbf{m}(u_j)$. Since the u_j values may be viewed as a rough approximation of a uniform sample on $[0, 1]$, the idea of Equation (13) leads to

$$\ell(\hat{\boldsymbol{\theta}}) - \ell(\tilde{\boldsymbol{\theta}}) \approx -\frac{1}{JK} \sum_{j=1}^J \sum_{k=1}^K \nabla \mathbf{m}(u_j) \nabla \boldsymbol{\eta}[\mathbf{m}(u_j)] \mathbf{z}(\mathbf{Y}_{jk}). \quad (14)$$

(The analogous Equation 5.4 of Hunter & Handcock (2006) omits the needed factor $-1/J$.)

Equation (14) entails two different approximations of Equation 13: One in approximating the expectation of \mathbf{Y} using simulated networks and one in approximating the $\text{Unif}(0, 1)$ distribution by u_0, \dots, u_J . The first of these is due to Monte Carlo error, so it may be quantified; this is the source of the standard errors reported for AIC and BIC for dyadic-dependent models, as seen for instance in the model fits in Section 10.1. The user can specify a control parameter `bridge.target.se` to `control.logLik.ergm()` (or via `snctrl()`) to continue bridge sampling until the estimated standard error of due to the first approximation is below it. The second approximation leads to a bias even in the idealized case where $K \rightarrow \infty$ and that only vanishes as $J \rightarrow \infty$. Our experiments suggest this bias is small, and it may be addressed in future updates to the package.

8.3 Curved MPLE and curved ERGMs as “first-class” models

Curved ERGMs—those for which $\eta(\theta) \neq \theta$ —were introduced by Hunter & Handcock (2006) to facilitate estimation of the decay parameter in the geometrically-weighted triadic degree and triadic terms. They stated a score function for such models and outlined an MCMLE algorithm that could be used to update θ given a sample of sufficient statistics from the previous guess.

However, the implementation prior to **ergm** 4.0 was, in this respect, incomplete: while it could estimate curved ERGMs, it required the end-user to specify the initial values for their parameters. This is because maximum pseudo-likelihood estimation (MPLE) for curved models had not been derived and implemented. Non-MCMLE methods did not support curved ERGMs at all.

The score function for the curved ERGM MPLE was derived by Krivitsky (2017) and is implemented in **ergm** 4.0. Thus, to fit a curved model with geometrically weighted degree, where previously one had to specify

```
ergm(floamarriage ~ edges + gwdegree(0.25)) # Initial guess for the decay parameter = 0.25
```

one may now specify

```
ergm(floamarriage ~ edges + gwdegree)

##
## Call:
## ergm(formula = floamarriage ~ edges + gwdegree)
##
## Last MCMC sample of size 457 based on:
##      edges      gwdegree gwdegree.decay
##      -1.5593      -0.1246       0.4305
##
## Monte Carlo Maximum Likelihood Coefficients:
##      edges      gwdegree gwdegree.decay
##      -1.57180      -0.07424       0.42424
```

and that the initial value is determined automatically.

In situations where the decay parameter is fixed and known, its value may be specified directly or via the partial `offset` capability, also new in **ergm** 4.0. Thus, the two **ergm** calls below are equivalent, though their coefficient estimates differ slightly because the fitting algorithm is stochastic:

```
coef(ergm(samplike~edges+gwesp(0.25, fix=TRUE),
  control=control.ergm(seed=123, MCMLE.maxit=2)))

##      edges gwesp.fixed.0.25
##      -1.6555796       0.4202939

coef(ergm(samplike~edges+offset(gwesp(),c(FALSE,TRUE)), offset.coef=0.25,
  control=control.ergm(seed=123, MCMLE.maxit=2)))

##      edges      gwesp offset(gwesp.decay)
##      -1.6335907      0.4032849       0.2500000
```

8.4 Contrastive Divergence

Contrastive divergence (CD) is a technique taken from the computer science literature and proposed in the context of ERGMs by Asuncion et al. (2010). Much more detail about its use for ERGMs is provided by Krivitsky (2017). Essentially, CD provides a spectrum of estimation algorithms with MPLE at one extreme and MLE at the other. Little is presently known about the efficacy of algorithms lying between these two extremes.

The **ergm** package implements CD estimates, which may be obtained by passing `estimate="CD"` to the **ergm** function. Since not much is known about the quality of these estimates, their most promising use at present is as starting values for MCMC-based maximum likelihood estimation as described at the beginning of Section 8; that is, they are used for the initial guess θ^t for $t = 0$. By default, they are used where available implementations of MPLE are inapplicable, in particular valued ERGMs or binary ERGMs with dyad-dependent sample space constraints: unlike MPLE, which must be rederived for each reference distribution and sample space constraint, contrastive divergence can reuse the proposals from the MCMC implementation (Krivitsky, 2017).

8.5 Confidence stopping criterion

The **ergm** package implements several methods to determine when to declare convergence in an MCMLE algorithm and report the results. They are selected via the `MCMLE.termination` parameter as follows:

MCMLE.termination="Hotelling" Convergence is declared if an autocorrelation-adjusted Hotelling T^2 test is unable to reject the null hypothesis that the estimating function equals zero, which for non-curved models on fully observed networks means simply that the expected value of the simulated statistic equals the observed statistic at a high level ($\alpha = 0.5$ by default). Krivitsky (2017) provides additional details.

MCMLE.termination="Hummel" The algorithm of Hummel et al. (2012) is used: Convergence is declared if for two consecutive parameter updates, the observed statistic is sufficiently deep in the interior of the convex hull of the sample of simulated statistics. (For curved models, sample values of estimating functions are used instead.) However, this criterion can be problematic for partially observed networks: as discussed in Section 10.2, this criterion requires that *every* point in the constrained (conditional) sample be in the interior of the convex hull of the unconstrained, which can be problematic when the fraction of the missing dyads and the dimension of the parameter vector are moderately high.

MCMLE.termination="confidence" Loosely based on the algorithms of Vats et al. (2019), this method, which is the default in **ergm** 4.0, implements a form of equivalence testing. The general idea of equivalence testing is to define the null hypothesis to be that the difference between the observed and the expected statistics large enough to be interesting. Thus, rejecting the null hypothesis entails deciding that the difference is small enough that convergence is declared.

9 Simulated Annealing

ergm has enhanced its flexibility in the use of simulated annealing (SAN) to randomly generate networks with a particular set of network statistics. This capability is used by the package in the process of finding MLE, particularly when estimating from sufficient statistics (Section 12.2). At least some of the methods enabled by SAN are the subject of ongoing research. For instance, Schmid & Hunter (2020) find that SAN can be used to find effective starting values for the iterative algorithm described at the beginning of Section 8. The rest of this section describes simulated annealing and details some of its capabilities and uses within the **ergm** package.

9.1 Formulation of SAN algorithm

Let \mathbf{g} be a vector of target statistics for the network we wish to construct. That is, we are given an arbitrary network $\mathbf{y}^0 \in \mathcal{Y}$, and we seek a network $\mathbf{y} \in \mathcal{Y}$ such that $\mathbf{g}(\mathbf{y}) \approx \mathbf{g}$ —ideally equality is achieved, but in practice we may have to settle for a close approximation. The variant of simulated annealing used in **ergm** is as follows.

The energy function is defined $E_W(\mathbf{y}) = \{\mathbf{g}(\mathbf{y}) - \mathbf{g}\}^\top W \{\mathbf{g}(\mathbf{y}) - \mathbf{g}\}$, with W a symmetric positive (barring multicollinearity in statistics) definite matrix of weights. This function achieves 0 only if the target is reached. A good choice of this matrix yields a more efficient search.

A standard simulated annealing loop is used, as described below, with some modifications. In particular, we allow the user to specify a vector of offsets $\boldsymbol{\eta}$ to bias the annealing, with $\eta_k = 0$ denoting no offset. As illustrated in the example below, offsets can be used with SAN to forbid certain statistics from ever increasing or decreasing. As with `ergm`, offset terms are specified using the `offset()` decorator and their coefficients specified with the `offset.coef` argument. By default, finite offsets are ignored by, but this can be overridden by setting the control argument `SAN.ignore.finite.offsets = FALSE`.

The number of simulated annealing runs is specified by the `SAN.maxit` control parameter and the initial value of the temperature T is set to `SAN.tau`. The value of T decreases linearly until $T = 0$ at the last run, which implies that all proposals that increase $E_W(\mathbf{y})$ are rejected. The weight matrix W is initially set to I_p/p , where I_p is the identity matrix of an appropriate dimension.

For weight W and temperature T , the simulated annealing iteration proceeds as follows:

1. Test if $E_W(\mathbf{y}) = 0$. If so, then exit.
2. Generate a perturbed network \mathbf{y}^* from a proposal that respects the model constraints. (This is typically the same proposal as that used for MCMC.)
3. Store the quantity $\mathbf{g}(\mathbf{y}^*) - \mathbf{g}(\mathbf{y})$ for later use.
4. Calculate acceptance probability

$$\alpha = \exp[-\{E_W(\mathbf{y}^*) - E_W(\mathbf{y})\}/T + \boldsymbol{\eta}^\top \{\mathbf{g}(\mathbf{y}^*) - \mathbf{g}(\mathbf{y})\}].$$

(If $|\eta_k| = \infty$ and $g_k(\mathbf{y}^*) - g_k(\mathbf{y}) = 0$, their product is defined to be 0.)

5. Replace \mathbf{y} with \mathbf{y}^* with probability $\min(1, \alpha)$.

After the specified number of iterations, T is updated as described above, and W is recalculated by first computing a matrix S , the sample covariance matrix of the proposed differences stored in Step 3 (i.e., whether or not they were rejected), then $W = S^+ / \text{tr}(S^+)$, where S^+ is the Moore–Penrose pseudoinverse of S . The differences in Step 3 closely reflect the relative variances and correlations among the network statistics.

In Step 2, the many options for MCMC proposals, including those newly added to the `ergm` package as covered in Section 11.1, can provide for effective means of speeding the SAN algorithm’s search for a viable network. This phenomenon is illustrated in Section 11.3.

The example below illustrates the use of offsets in a 100-node network in which each node has a `sex` attribute with possible values "M" and "F". Suppose that we wish to construct a network with 30 edges in which no edges are allowed between nodes of the same `sex` value, nor that result in any node having more than one edge—constraints that would arise, for example, if we wished to model a network of heterosexual, monogamous relationships. SAN can find such a network by placing offset parameters valued at `-Inf` on ERGM terms corresponding to `nodematch("sex")` and `concurrent`:

```
nw <- network.initialize(100, directed = FALSE)
nw %v% "sex" <- rep(c("M", "F"), 50)
example <- san(nw ~ edges + offset(nodematch("sex")) + offset(concurrent),
               offset.coef = c(-Inf, -Inf), target.stats = 30)
summary(example ~ edges + nodematch("sex") + concurrent)
```

```
##          edges nodematch.sex      concurrent
##          30           0           0
```

The output of the `summary` function above verifies that the constraints are satisfied by the generated network `example`.

10 Estimation in the presence of missing edge data

It is quite common that network data are incomplete in various ways. The `ergm` package includes the capability to handle missing edge data, whereas other types of missingness such as missing nodal information are not addressed. Handcock & Gile (2010) formulated a framework for modeling networks with missing ties and expressed the log-likelihood as

$$\ell(\boldsymbol{\theta}) = \log \Pr(\mathbf{Y} \in \mathcal{Y}(\mathbf{y}^{\text{obs}}); \boldsymbol{\theta}) = \log \sum_{\mathbf{y}' \in \mathcal{Y}(\mathbf{y}^{\text{obs}})} \Pr(\mathbf{Y} = \mathbf{y}'; \boldsymbol{\theta}),$$

where $\mathcal{Y}(\mathbf{y}^{\text{obs}})$ is defined as the set of networks whose partial observation could have produced \mathbf{y}^{obs} : essentially, all of the ways to impute the missing ties in \mathbf{y}^{obs} . They then proposed to maximize this likelihood by taking advantage of the fact that, if

$$\kappa_{\mathcal{Y}'}(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \sum_{\mathbf{y}' \in \mathcal{Y}'} h(\mathbf{y}') \exp\{\boldsymbol{\eta}(\boldsymbol{\theta})^\top \mathbf{g}(\mathbf{y}')\},$$

the log-likelihood can be expressed as $\ell(\boldsymbol{\theta}) = \log \kappa_{\mathcal{Y}(\mathbf{y}^{\text{obs}})}(\boldsymbol{\theta}) - \log \kappa_{\mathcal{Y}}(\boldsymbol{\theta})$, resulting in

$$\text{sc}(\hat{\boldsymbol{\theta}}) = \nabla_{\boldsymbol{\theta}} \ell(\hat{\boldsymbol{\theta}}) = \boldsymbol{\eta}'(\hat{\boldsymbol{\theta}})^\top [\mathbb{E}_{\mathcal{Y}(\mathbf{y}^{\text{obs}})}\{\mathbf{g}(\mathbf{Y}); \hat{\boldsymbol{\theta}}\} - \mathbb{E}_{\mathcal{Y}}\{\mathbf{g}(\mathbf{Y}); \hat{\boldsymbol{\theta}}\}] = \mathbf{0},$$

with MCMLE approximation also possible for the first term by fixing a particular $\boldsymbol{\theta}^t$ and drawing a sample from $\text{ERGM}_{\mathcal{Y}(\mathbf{y}^{\text{obs}})}(\boldsymbol{\theta}^t)$.

10.1 Specifying missing edge data

The **ergm** package invokes the above approach automatically when a network has missing edge variables. The simplest way to encode a missing edge is to set its value to **NA**. The **network** package natively supports missing edge variables coded in this way, and **network** objects with missingness are thus handled without additional intervention.

Here we fit a simple model with edges, mutuality (reciprocated dyads), transitive ties, and cyclical ties to the Sampson Monks dataset depicted in Figure 1. For the sake of comparison, we first fit the model assuming no missing edge data, which may be quickly verified using the output of the `print(samlpke)` command:

```
print(samlpke)

## Network attributes:
##   vertices = 18
##   directed = TRUE
##   hyper = FALSE
##   loops = FALSE
##   multiple = FALSE
##   total edges= 88
##   missing edges= 0
##   non-missing edges= 88
##
## Vertex attribute names:
##   cloisterville group vertex.names
##
## Edge attribute names:
##   nominations

summary(full.fit <- ergm(samlpke ~ edges + mutual + transitivity + cyclicalities,
  eval.loglik=TRUE))

## Call:
## ergm(formula = samlpke ~ edges + mutual + transitivity + cyclicalities,
##   eval.loglik = TRUE)
##
## Monte Carlo Maximum Likelihood Results:
##
##           Estimate Std. Error MCMC % z value Pr(>|z|)
## edges          -1.8998    0.3509     0  -5.414   <1e-04 ***
## mutual           2.4806    0.4475     0   5.544   <1e-04 ***
## transitivity     0.5141    0.2984     0   1.723   0.0849 .
## cyclicalities    -0.4490    0.2492     0  -1.802   0.0716 .
## --
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Null Deviance: 424.2 on 306 degrees of freedom
## Residual Deviance: 327.8 on 302 degrees of freedom
##
## AIC: 335.8 BIC: 350.7 (Smaller is better. MC Std. Err. = 0.6976)
```


Now, suppose that Monk #1 (John Bosco) refused to respond during all three waves, rendering his replies missing:

```
samplike1 <- samplike
samplike1[1, ] <- NA
print(samplike1)
```

```
## Network attributes:
##   vertices = 18
##   directed = TRUE
##   hyper = FALSE
##   loops = FALSE
##   multiple = FALSE
##   total edges= 99
##     missing edges= 17
##     non-missing edges= 82
##
## Vertex attribute names:
##   cloisterville group vertex.names
##
## Edge attribute names:
##   nominations
```

If we pass this modified object to `ergm`, it will automatically calculate the MLE under the assumption that the monk's refusal is unrelated to his choice of relations, i.e., that the data are ignorably missing with respect to the specified model:

```
summary(m1.fit <- ergm(samplike1 ~ edges + mutual + transitiveties + cyclicalities, eval.loglik = TRUE))
```

```
## Call:
## ergm(formula = samplike1 ~ edges + mutual + transitiveties +
##       cyclicalities, eval.loglik = TRUE)
##
## Monte Carlo Maximum Likelihood Results:
##
##              Estimate Std. Error MCMC % z value Pr(>|z|)
## edges          -2.0093    0.3837      0 -5.237   <1e-04 ***
## mutual           2.4163    0.4846      0  4.987   <1e-04 ***
## transitiveties   0.4357    0.3920      0  1.111    0.266
## cyclicalities   -0.2684    0.3598      0 -0.746    0.456
## --
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##      Null Deviance: 400.6 on 289 degrees of freedom
## Residual Deviance: 312.8 on 285 degrees of freedom
##
## AIC: 320.8 BIC: 335.4 (Smaller is better. MC Std. Err. = 0.4452)
```

The degrees of freedom associated with the missing data fit have decreased because unobserved dyads do not carry information. For details regarding the ignorability assumption for edge variables, see Handcock & Gile (2010).

The estimation approach above can be extended to other types of incomplete network observation. Karwa et al. (2017) applied it to fit arbitrary ERGMs to networks whose dyad values had been stochastically perturbed—ties added and removed at random, with known probabilities—in order to preserve privacy. Another use case is multiple imputation for networks with missing data, in which multiple random versions of the full network are constructed by randomly inserting values for unobserved dyads according to probabilities that are determined based on, say, some type of logistic regression model. These mechanisms may be invoked by passing an `obs.constraints` formula, specifying how the network of interest was observed. Of particular interest are the following constraints:

observed restricts the proposal to changing only those dyads that are recorded as missing.

egocentric(`attr = NULL`, `direction = c("both", "out", "in")`) restricts the proposal to changing only those dyads that would not be observed in an egocentric sample. That is, dyads cannot be modified

that are incident on vertices for which attribute specification `attr` has value `TRUE` or, if `attr` is `NULL`, the vertex attribute `"na"` has value `FALSE`. For directed networks, `direction=="out"` only preserves the out-dyads of those actors, and `direction=="in"` preserves their in-dyads.

dyadnoise(p01,p10) Unlike the others, this is a soft constraint to adjust the sampled distribution for dyad-level noise with known perturbation probabilities, which can arise in a variety of contexts (Karwa et al., 2017). It is assumed that the observed LHS network is a noisy observation of some unobserved true network, with `p01` giving the dyadwise probability of erroneously observing a tie where the true network had a non-tie and `p10` giving the dyadwise probability of erroneously observing a nontie where the true network had a tie. `p01` and `p10` can be either both be scalars or both be adjacency matrices of the same dimension as that of the LHS network giving these probabilities.

We may use the `obs.constraints` argument to re-fit the model above:

```
samplike2 <- samplike
samplike2[1,] <- 0
samplike2 %v% "refused" <- rep(c(TRUE,FALSE),c(1,17))
samplike2 # same as print(samplike2)

## Network attributes:
##   vertices = 18
##   directed = TRUE
##   hyper = FALSE
##   loops = FALSE
##   multiple = FALSE
##   total edges= 82
##   missing edges= 0
##   non-missing edges= 82
##
## Vertex attribute names:
##   cloisterville group refused vertex.names
##
## Edge attribute names:
##   nominations

summary(m2.fit <- ergm(samplike2 ~ edges + mutual + transitivity + cyclicalities,
  obs.constraints = ~ egocentric(~!refused, "out"))

## Call:
## ergm(formula = samplike2 ~ edges + mutual + transitivity +
##   cyclicalities, obs.constraints = ~egocentric(~!refused, "out"))
##
## Monte Carlo Maximum Likelihood Results:
##
##              Estimate Std. Error MCMC % z value Pr(>|z|)
## edges          -2.0386    0.4142    0  -4.922   <1e-04 ***
## mutual           2.4494    0.4894    0   5.005   <1e-04 ***
## transitivity     0.4594    0.4158    0   1.105    0.269
## cyclicalities    -0.2782    0.3703    0  -0.751    0.452
## --
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Null Deviance: 400.6 on 289 degrees of freedom
## Residual Deviance: 313.8 on 285 degrees of freedom
##
## AIC: 321.8 BIC: 336.4 (Smaller is better. MC Std. Err. = 0.4377)
```

Finally, since the observational process can be viewed as a part of the network dataset, we may specify it using the `%ergmlhs%` operation, giving a third way to fit the model above:

```
samplike2 %ergmlhs% "obs.constraints" <- ~egocentric(~!refused, "out")
summary(m3.fit <- ergm(samplike2 ~ edges + mutual + transitivity + cyclicalities))

## Call:
```

```

## ergm(formula = samplike2 ~ edges + mutual + transitivityes +
##       cyclicalities)
##
## Monte Carlo Maximum Likelihood Results:
##
##               Estimate Std. Error MCMC % z value Pr(>|z|)
## edges          -2.0436    0.3885     0 -5.260 <1e-04 ***
## mutual           2.4295    0.4811     0  5.049 <1e-04 ***
## transitivityes   0.4749    0.3654     0  1.300  0.194
## cyclicalities   -0.2946    0.3128     0 -0.942  0.346
## --
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Null Deviance: 400.6 on 289 degrees of freedom
## Residual Deviance: 313.9 on 285 degrees of freedom
##
## AIC: 321.9 BIC: 336.6 (Smaller is better. MC Std. Err. = 0.4657)

```

10.2 Partial Stepping for Observation Process

Handcock (2003) observed that for a non-curved family, i.e., where $\eta(\theta) \equiv \theta$, Equation (9) does not have a unique maximizer if $\mathbf{g}(\mathbf{y}^{\text{obs}})$ is not in the convex hull of the sample $\mathbf{g}(\mathbf{y}^{\theta^t, s})$. This can be seen from its form: If it is outside of the convex hull, then one can increase the maximand arbitrarily by selecting $\theta' = \alpha \mathbf{g}(\mathbf{y}^{\text{obs}}) + \theta^t$ with $\alpha \rightarrow \infty$: the first term in (9) would grow faster than any summand or combination of summands of the second. Conversely, if it is in the interior of the convex hull, then for any direction for θ' , some combinations of summands would grow faster than the first term.

Hummel et al. (2012) therefore proposed to translate $\mathbf{g}(\mathbf{y}^{\text{obs}})$ in the direction of the centroid of $\mathbf{g}(\mathbf{y}^{\theta^t, s})$ until it was sufficiently deeply in its convex hull, making an update that would be guaranteed to be a unique maximizer in the correct general direction. Krivitsky (2017) extended this approach to curved ERGMs.

When there are missing data or an observation process, the form of the maximand becomes

$$\log \frac{1}{S} \sum_{s=1}^S \exp[\{\eta(\theta') - \eta(\theta^t)\}^\top \mathbf{g}(\mathbf{y}^{\theta^t, s} | \mathbf{y}^{\text{obs}})] - \log \frac{1}{S} \sum_{s=1}^S \exp[\{\eta(\theta') - \eta(\theta^t)\}^\top \mathbf{g}(\mathbf{y}^{\theta^t, s})],$$

where $\mathbf{y}^{\theta^t, s} | \mathbf{y}^{\text{obs}}$ are draws from the distribution $\text{ERGM}_{\mathbf{y}(\mathbf{y}^{\text{obs}})}(\theta^t)$. From its form, it can be seen that it can be maximized to infinity if *any* of $\mathbf{g}(\mathbf{y}^{\theta^t, s} | \mathbf{y}^{\text{obs}})$ is outside of the convex hull of $\mathbf{g}(\mathbf{y}^{\theta^t, s})$, as that summand could then dominate all others in both summations.

ergm therefore scales all $\mathbf{g}(\mathbf{y}^{\theta^t, s} | \mathbf{y}^{\text{obs}})$ toward the centroid of $\mathbf{g}(\mathbf{y}^{\theta^t, s})$ until they are all sufficiently deep in the convex hull. (That they are being scaled towards a point guarantees that such a scaling factor exists unless the rank of $\mathbf{y}^{\theta^t, s} | \mathbf{y}^{\text{obs}}$ is higher than that of $\mathbf{g}(\mathbf{y}^{\theta^t, s})$.)

11 Computing efficiency tests on large networks

Version 4.0 of the **ergm** package enables substantial gains in computing efficiency relative to earlier versions of the packages. There are many reasons for these gains, including better algorithms (e.g., improvements to simulated annealing and maximum pseudo-likelihood estimation), better use of parallelism, and new MCMC proposals. As pointed out elsewhere, improved MCMC proposals lead to performance improvements across a wide range of **ergm** package functionality due to the pervasive use of simulation in the **ergm** workflow. Where these improvements have greatest impact, we see speedups of 2 orders of magnitude for comparable computing tasks. This section highlights some of the key changes and demonstrates their impacts, using in some cases networks with one million nodes.

The code used to produce the results in each subsection of Section 11 is given in the corresponding subsection of the Appendix. Each test is based on a hypothetical network, where the nodes are persons and the edges represent cohabiting. The distribution of nodal attributes and edges is based on aggregated statistics from the National Survey of Family Growth (NSFG) (National Center for Health Statistics, 2020). The nodes have demographic attributes—sex, age, and race—sampled using the NSFG post-stratification weights that have been adjusted to match the demographics of King County in Washington State. The edges represent

heterosexual cohabitation relationships observed in the data, which imposes two constraints on the network that we can exploit for computing efficiency gains: the networks are bipartite, i.e., only edges between male and female nodes are allowed, and nodal degree is capped at one. In addition to demographic attributes, the test dataset includes two more nodal variables as distributed in the adjusted NSFG data: sexual identity and whether the node has at least one persistent non-cohabiting partner. The data and model are a simplified version of an actual applied research project that models the spread of HIV.

11.1 Impact of new proposals on Markov chain mixing

The trace plots of Figure 3 show how a Markov chain based on a particular ERGM for a million-node network approaches equilibrium, starting from an empty network, using each of three different proposals. The horizontal axis is the base 10 logarithm of the cumulative number of proposals made, and the vertical axis is the statistic value. Statistics were sampled every 1000 proposals, so the horizontal axis starts at 3.

The model has 15 statistics. Here is the formula used for simulation:

```
MillionNodeFormula <-
  nw ~ edges + nodefactor("sex.ident", levels = 3) + nodecov("age") + nodecov("agesq") +
    nodefactor("race", levels = -5) + nodefactor("othr.net.deg", levels = -1) +
    nodematch("race", diff = TRUE) + absdiff("sqrt.age.adj")
```

Additionally, constraints are imposed to prevent edges between nodes with the same `sex` attribute and also to prevent concurrent partnerships, so a node's degree in this network can be only 0 or 1.

We see that the `BDStratTNT` proposal stratifying on only race holds an order of magnitude advantage over TNT for three of the four statistics shown, and this advantage increases to 3 or more orders of magnitude for the statistic representing homophily for race group B (`nodematch.race.B`). The `BDStratTNT` proposal stratifying on both race and age roughly matches the performance of the race-only stratification for the statistics representing density (`edges`) and homophily for race group B, but it does significantly better for the terms representing the effects of age differences (`absdiff.sqrt.age.adj`) and having at least one persistent relationship in another network (`nodefactor.othr.net.deg.1`).

The improvement in `absdiff.sqrt.age.adj` is expected, since the model favors ties on dyads where the nodes have similar ages, and such dyads will have toggles proposed more frequently when proposals are stratified according to age mixing.

There is a modest sex asymmetry in the age mixing, with females tending to be about 1.5 years younger than their male partners; taking this asymmetry into account in the proposal stratification via the `pmat` argument of the `strat` hint did not produce significant additional gains. The improvement in `nodefactor.othr.net.deg.1` may arise because nodes having positive `othr.net.deg` tend to be younger than the population average age, so proposal age stratification can hasten equilibration of the `nodefactor.othr.net.deg.1` statistic.

11.2 Gains in Effective Sample Size

Table 1 shows univariate effective sample sizes (ESS), as calculated by the `coda` package (Plummer et al., 2006), for some of the statistics generated from a Markov chain run on a 50,000-node network with an interval of 100 steps between each sampled vector to produce 10 million vectors. ESS gives a way to compare various Markov chains that all have the same equilibrium distribution, since chains that do not mix well do not produce samples that vary much, which in turn reduces their ESS. Table 3 in Section A.2 of the Appendix presents the same results in units of ESS per minute of computing time, revealing differences not only in MCMC mixing efficiency but also computing efficiency.

The ERGM used here includes the same 15 statistics as the `MillionNodeFormula` of Section 11.1. The first column in Table 1 indicates, in abbreviated form, the hints and/or constraints that were specified, in addition to the default `sparse` hint used for all rows. Thus, the first row uses the TNT proposal, while all other rows use `BDStratTNT` with varying levels of complexity in the hints and constraints passed to the proposal.

We see that, for the statistics included in Table 1, the `BDStratTNT` proposal that stratifies on the modified `race` effect and respects the heterosexual nature of the model and the bound placed on degree—no node is ever allowed to have more than one tie—gives roughly a 50-fold increase in minimum ESS relative to the TNT proposal, the only proposal tested that was available in `ergm` prior to version 3.10. For the same pair of proposals, the improvement in minimum ESS across all 15 statistics was roughly 90-fold. Here, the modified

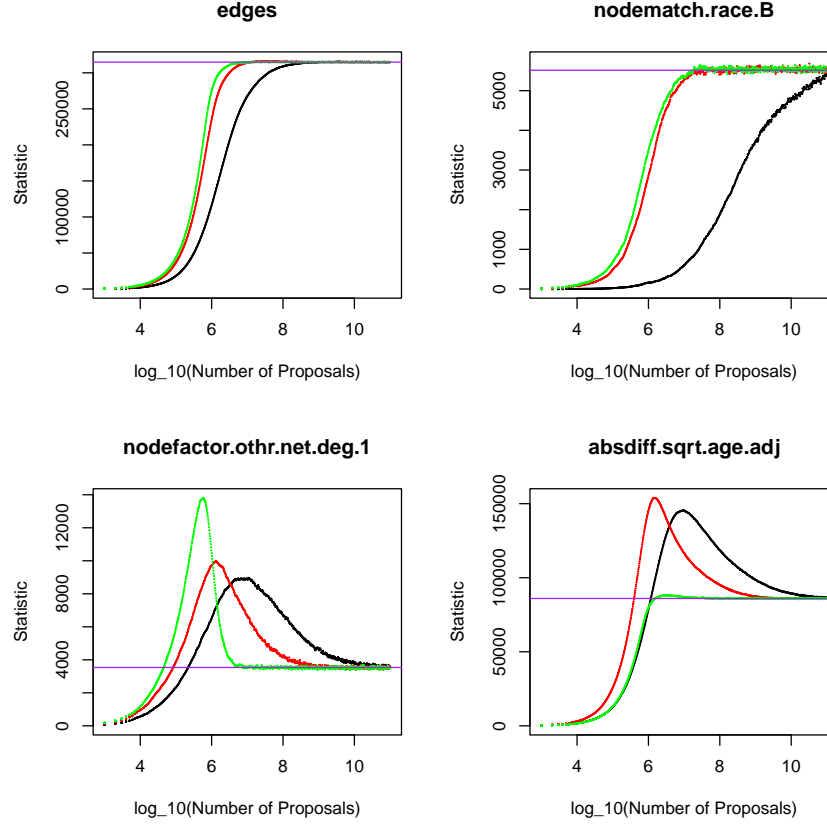


Figure 3: Approach to equilibrium of Markov chains using TNT (black), BDStratTNT stratified only on race (red), and BDStratTNT stratified jointly on race and age (green). Target values are shown as horizontal lines.

Table 1: Effective sample sizes for 4 of the 15 statistics from an MCMC sample of size 10 million with interval 100 using the hints and constraints shown in the leftmost column in addition to the `sparse` hint, which is used in all cases.

	edges	race B homoph.	other net deg. 1+	$\sqrt{\text{age diff.}}$
"TNT" <code>bd(sex,1)</code>	2994	224	5219	2323
<code>bd(1)+blocks(sex)</code>	25752	1184	11028	12684
<code>bd(1)+blocks(sex)+strat(race)</code>	27286	4377	11717	14031
<code>bd(1)+blocks(sex)+strat(race.mod)</code>	20299	13050	10714	10840
<code>bd(1)+blocks(sex)+strat(race,age)</code>	28340	4866	16054	6328

`race` effect is implemented by passing a `pmat` argument to the `strat` hint that is constructed so as to give more weight, in selecting potential MCMC edge toggles, to smaller race groups than their edge fraction would dictate. See Section A.2 of the Appendix for additional details.

11.3 Impact of MCMC improvements on simulated annealing speed

The trace plots in Figure 4 show how various statistics approach their target values during a run of `ergm`'s simulated annealing algorithm, starting from an empty million node network, with each of three different proposals. The horizontal axis is the base 10 logarithm of the number of proposals made, and the vertical axis is the statistic value, with the target value indicated as the horizontal purple line. Statistics are sampled every 1000 proposals, so the horizontal axis starts at 3. The SAN run takes place at a fixed temperature of 0, with the matrix of weights being the diagonal matrix of reciprocal squared target statistics divided by their

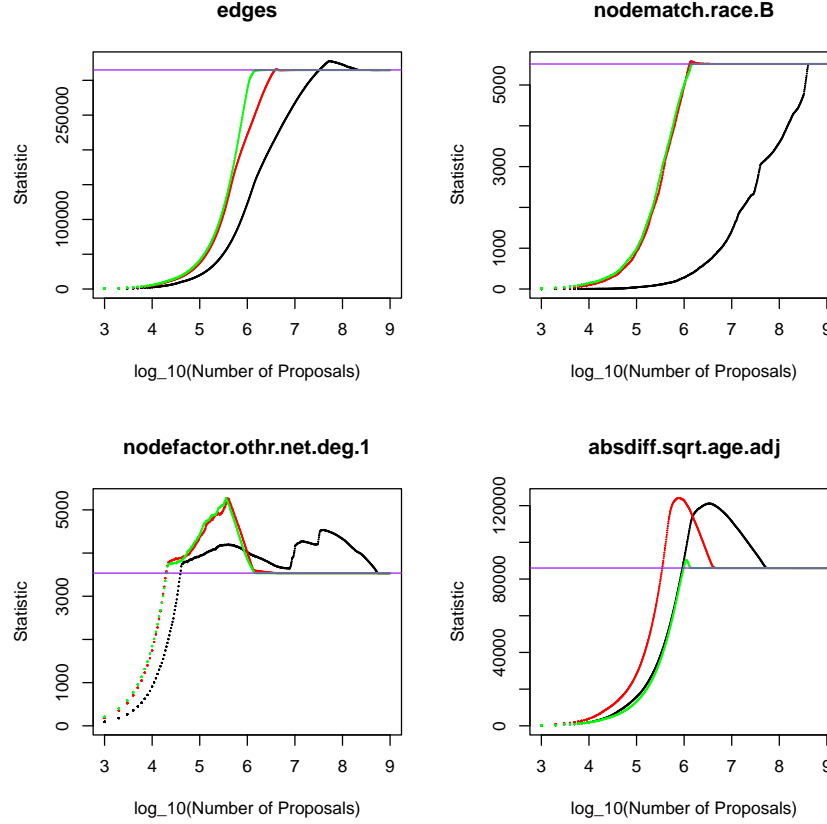


Figure 4: Approach to target values of SAN runs using proposals TNT (black), BDStratTNT stratified only on race (red), and BDStratTNT stratified jointly on race and age (green). Target values are shown as horizontal lines.

sum. This choice of temperature and weight matrix settings was made to try to minimize the effect of these choices on the algorithm’s behavior and thereby isolate the different effects of the proposals, yet we find that using the default TNT settings explained in Section 9.1 leads to similar behavior to that reported in Figure 4.

Figure 4 shows that the BDStratTNT proposal stratifying on only race yields an advantage of roughly 1 to 2 orders of magnitude over TNT for these statistics, with the BDStratTNT proposal stratifying on both race and age yielding an additional half an order of magnitude or less.

11.4 Impact of MCMC improvements on estimation time

Table 2 shows estimation times for the model described in Section 11.1 with various **ergm** versions and settings. The **ergm** 3.10 fits use TNT while the **ergm** 4.0 fits use BDStratTNT. All fits include offsets with coefficients set to $-\infty$ in the model formula to enforce constraints in maximizing the pseudolikelihood. The “3.10 without bd” fits also utilize these offsets to enforce constraints during MCMC, while the offsets are redundant in MCMC for the “3.10 with bd” and “4.0” fits. An important difference between “3.10 with bd” and “4.0” (which also uses bd) is the implementation of bd: in **ergm** 3.10, the only bd implementation available was a rejection algorithm, while in **ergm** 4.0, the BDStratTNT proposal maintains the necessary state to avoid the need for a rejection algorithm when imposing upper bounds on degree. In this particular model, we have a target mean degree of about 0.63, meaning that approximately 86% of randomly chosen dyads cannot be toggled without violating the degree bound when the network is near equilibrium. Of the 14% that can, half cannot be toggled due to the heterosexuality constraint, which is also taken into account by BDStratTNT. The BDStratTNT proposal is thus naively about 15 times as efficient for this model as a proposal that does not constructively take the constraints into account. The stratification of proposals by **race**, also handled by BDStratTNT, yields still further improvement. This expected baseline increase in

Table 2: Model fit times for various **ergm** versions and settings using fixed short, fixed long, and adaptive MCMC intervals.

	Short	Long	Adaptive
3.10 without bd	21.58 hours (1e5 interval)	32.40 hours (1e6 interval)	N/A
3.10 with bd	3.71 hours (1e5 interval)	5.80 hours (1e6 interval)	N/A
4.0	1.60 minutes (5e3 interval)	3.55 minutes (5e4 interval)	3.23 minutes

efficiency, together with the effective sample size results described in 11.2, explains the choice of fixed interval values $1/20$ as large in the third row of Table 2 as in the first two rows.

The rightmost column of Table 2 uses adaptive MCMC to fit the model, which is new in **ergm** 4.0 and is the default. One may disable adaptive MCMC by setting `MCMLE.effectiveSize = NULL`, as in Section A.4 of the Appendix.

All fits reported in the table parallelize MCMC over 16 cores, so improvements in parallelization between **ergm** versions 3.10 and 4.0 are also represented in the final row.

A few words are in order regarding the **bd** constraint and its relationship with simulated annealing in earlier versions of **ergm**. As mentioned above, it is possible to enforce a constraint such as a bound on degree without using **bd** by adding an offset term to the model, say, `degree(k)` where **k** is one larger than the maximum allowable degree, and fixing its coefficient value at `-Inf`. However, offsets were previously ignored by **san**, the simulated annealing function used to produce an initial network, potentially resulting in an initial network not satisfying the constraints. This in turn could produce a poor initial parameter value, as obtained from the MPLE of the network generated by **san**. Indeed, we see in the first row of Table 2 that models fit in 3.10 without using **bd** produce much longer fit times for this particular model. The **san** function now respects offsets, but the facts that multiple algorithms in **ergm** might rely on constraints and not all such algorithms currently optimize their treatment of both offsets and explicit **constraints** lead us to recommend the redundancy in specifying such constraints.

12 Other enhancements

We close this paper by highlighting a number of miscellaneous enhancements to the **ergm** package since the Hunter et al. (2008) article.

12.1 Exact calculations for small networks

For small networks, it is possible to obtain full enumeration of all possible network statistic vectors over the entire sample space of possible networks. This enumeration enables exact calculations of such quantities as the log-likelihood function, the MLE, or the normalizing constant. If we consider only binary networks on an unconstrained sample space, the total number of networks is $2^{n(n-1)/2}$ for undirected networks and $2^{n(n-1)}$ for directed networks, which imposes a practical limit of $n = 8$ nodes in the undirected case or $n = 6$ in the directed case unless the user wants to compute for a long time, and the functions described in this section return an error for larger networks than these unless the `force=TRUE` option is invoked.

The `ergm.allstats` function, added to the **ergm** more than a decade ago, performs an efficient, “brute-force” tabulation of all possible network statistic vectors for an arbitrary ERGM by visiting every possible network. The `ergm.exact` function uses `ergm.allstats` to calculate exact likelihood values. Due to the computationally intractable normalizing constant $\kappa_{h,\eta,g}(\theta, \mathcal{Y})$ of Equation (1), except in the case of dyadic independence models, `ergm.exact` and `ergm.allstats` may only be used for small networks. In a test, the code below took about 254 times as long on a 9-node network as it did on an 8-node network, which is not surprising because the 9-node sample space has 2^{36-28} , or 256, times as many networks.

```
system.time({
  EmptyNW <- network.initialize(8, directed = FALSE) # Replacing 8 by 9 takes much longer!
  a <- ergm.allstats(EmptyNW ~ edges + triangle + isolates + degree(4), force = TRUE)
})
```

```
##      user  system elapsed
## 65.442   0.036   65.651
```


Naturally, many networks of interest are too large to utilize `ergm.allstats` and `ergm.exact`. Yet calculations on small networks can still provide useful test cases; for instance, see Schmid & Hunter (2020) or Vega Yon et al. (2021).

12.2 Estimation based only on sufficient statistics

In exponential family parlance, $\mathbf{g}(\mathbf{y}^{\text{obs}})$ is often referred to as the vector of sufficient statistics. Since the likelihood function of Equation (9) depends on \mathbf{y}^{obs} only via these sufficient statistics, it is not actually necessary to observe \mathbf{y}^{obs} in order to calculate an MLE. This fact is essentially a statement that MLE adheres to the so-called likelihood principle, which affirms that the likelihood function encodes all information relevant to estimation—a principle not satisfied by MPLE, as discussed by Schmid & Hunter (2020).

In some applications, such as when data are egocentrically sampled, it is possible to observe or estimate the vector $\mathbf{g}(\mathbf{y}^{\text{obs}})$ of statistics that would in principle have been observed in the network, even if other information about the network itself is absent. Estimation may still proceed by passing a `target.stats` argument containing a vector of network statistics. For example, we may reproduce (up to the stochasticity of the fitting algorithm) the analysis of the `full.fit` example in Section 10.1 by passing the vector of statistics on the `samplelike` network via `target.stats` even though the network used in the `ergm` function call has no edges at all:

```
# Complete network statistics:
ts <- summary(samplelike ~ edges + mutual + transitivity + cyclicalities)
emptynw <- network.initialize(network.size(samplelike), directed = TRUE)
ts.fit <- ergm(emptynw ~ edges + mutual + transitivity + cyclicalities, target.stats = ts)
rbind(coef(full.fit), coef(ts.fit))

##          edges  mutual transitivity cyclicalities
## [1,] -1.899816 2.480634      0.5140796    -0.4489623
## [2,] -1.909809 2.493298      0.5339571    -0.4597818
```

12.3 Logistic regression to obtain MPLE

In the case where the ERGM of Equation (1) is dyad-independent, which means that all of its terms are dyad-independent as defined in Section 6.2, the MPLE and the MLE are the same. Regardless, the MPLE may generally be obtained via standard binary logistic regression (Duijn et al., 2009).

Though this logistic regression is performed automatically when `estimate="MPLE"` is used with the `ergm` function, the `ergm` package also provides a function called `ergmMPLE` that produces the response vector and predictor matrix that may be used, for instance, to produce the logistic regression output directly via the `glm` function in R. The `ergmMPLE` function gives its output in the form of weighted response/predictor combinations, weighted according to their multiplicity, in order to conserve memory in cases where particular combinations occur frequently. In addition, `ergmMPLE` respects constraints in the sense that any dyads that are constrained not to change from their observed values are omitted.

```
data(g4)
print(lr <- ergmMPLE(g4 ~ edges + triangle, constraints = ~Dyads(fix = ~isolates)))

## $response
## [1] 0 1 1 0 0
##
## $predictor
##      edges triangle
## [1,]     1         1
## [2,]     1         1
## [3,]     1         2
## [4,]     1         2
## [5,]     1         0
##
## $weights
## [1] 2 2 2 4 1
```

In the example above, the directed `g4` network on 4 nodes has only one dyad, out of twelve, with the property that the number of isolates would change if that dyad's tie status were toggled. Therefore, the constraint

means that this dyad is held fixed whereas the other eleven can change. Notice that the weights vector sums to 11, revealing that these weights are the multiplicities of the corresponding response vector entries and predictor matrix rows; we may verify that the MPLE obtained via `ergm` matches direct logistic regression estimates:

```

rbind(
  coef(ergm(g4~edges + triangle, constraints = ~Dyads(fix=~isolates), estimate="MPLE")),
  coef(glm(response ~ predictor - 1, weights = weights, data = lr, family = "binomial")))

##           edges triangle
## [1,] -0.8066497 0.1686443
## [2,] -0.8066497 0.1686443

```

12.4 Predicting individual edge probabilities

The `predict` method, which may be called on either `formula` or `ergm` objects, calculates model-predicted conditional or unconditional tie probabilities for dyads in a binary network. In the conditional case, `predict` simply uses the output from the `ergmMPLE` function of Section 12.3. In the unconditional case, even for dyadic independence models where the conditional and unconditional probabilities coincide, `predict` simulates multiple random networks via the `simulate` method in order to estimate the tie probabilities.

In the example below, we use the small `g4` network with 4 nodes and 5 directed ties.

If we fit a simple ERGM with only the edges statistic, the maximum likelihood estimate is the logit of $5/12$ since there are $4 \times 3 = 12$ possible ties. If we use the `predict` method on this fitted `ergm` object, theoretically the conditional and unconditional probabilities are the same because this is a dyadic independence model. Nonetheless, `conditional = FALSE` forces `predict` to estimate the matrix of tie probabilities via simulation of `nsim` networks.

```

set.seed(123)
SimpleERGM <- ergm(g4 ~ edges)
predict(SimpleERGM, conditional = TRUE, output = "matrix")

##           V1           V2           V3           V4
## V1 0.0000000 0.4166667 0.4166667 0.4166667
## V2 0.4166667 0.0000000 0.4166667 0.4166667
## V3 0.4166667 0.4166667 0.0000000 0.4166667
## V4 0.4166667 0.4166667 0.4166667 0.0000000

predict(SimpleERGM, conditional = FALSE, output = "matrix", nsim = 1000)

##           V1           V2           V3           V4
## V1 0.000 0.407 0.398 0.434
## V2 0.387 0.000 0.404 0.412
## V3 0.427 0.441 0.000 0.406
## V4 0.416 0.402 0.433 0.000

```

12.5 Flattened control arguments via a single list

Many of the core functions of `ergm` and related packages have `control =` arguments that control various aspects of their working. Within just `ergm`, for instance, the `ergm`, `simulate`, and `san` functions all require various control parameters; and packages such as `ergm.ego` include additional core functions such as `ergm.ego`. Moreover, it is not unusual that, say, a call to `ergm` will invoke `simulate` and possibly even `SAN` implicitly. This means that a single `ergm` (or `ergm.ego`) call could have multiple lists of control parameters, sometimes passed as nested lists. `ergm` 4.0 implements a method that flattens these nested lists, allowing users to enter all control parameters in a single list; furthermore, this method allows for the usual tab-completion of available arguments when using most R environments.

The key to entering control arguments for all of the various functions requiring them is the single function `snctrl()`, which is shorthand for “StatNet ConTRoL”. The `snctrl()` function is used as the single value of the `control` argument in a function such as `ergm`. For instance, if we wish to force Monte-Carlo-based estimation in a simple ERGM that could be estimated exactly—because it is a dyadic independence model in which the pseudo-likelihood is the same as the likelihood—we might type

```
coef(ergm(g4 ~ edges, control = snctrl(force.main = TRUE)))
```

```
##      edges
## -0.3331718
```

If the code above is entered in RStudio, then pressing the tab key after typing “...control = snctrl(” will reveal the various possible control parameters, including `force.main`. Additional illustrations of this method of entering control parameters are in the Appendix.

ergm 4.0 is backwards-compatible with the previous method of passing control parameters via `control.ergm`, `control.simulate`, `control.san`, and others.

12.6 Setting package options

ergm 4.0 has several options that affect ERGM estimation as well as the behavior of some terms, as detailed at the time of this writing in Section 12.6.1 and 12.6.2, respectively. A current list of available options may be obtained via `help("ergm-options")` or the shorthand `options?ergm`.

12.6.1 Global Options

A number of **ergm** behaviors can be set globally using the familiar `options()` command. For example, whether `ergm()` and similar functions evaluate the likelihood of the fitted model—a very computationally intensive process, particularly for valued networks—by default is controlled by option `ergm.eval.loglik`, which itself defaults to `TRUE`. Running

```
options(ergm.eval.loglik = FALSE)
```

instructs `ergm()` to skip likelihood calculation unless overridden in the call via `ergm(..., eval.loglik=TRUE)`.

Other global options currently implemented are

ergm.loglik.warn_dyads Whether log-likelihood evaluation should issue a warning when the effective number of dyads that can vary in the sample space is poorly defined, such as if degree the sequence is constrained.

ergm.cluster.retries **ergm**’s parallel routines implement rudimentary fault-tolerance. This option controls the number of retries for a cluster call before giving up.

ergm.term This allows the default term options list, described in Section 12.6.2, to be set globally.

12.6.2 Term options

ergm 4.0 implements an interface for setting certain options for ERGM term behavior. The global setting is controlled via `options(ergm.term=list(...))` where ... are key-value pairs specifying the options. Individual options can be overwritten on an ad hoc basis within a function call. For functions that have a `control=` argument, such as `ergm()` and `simulate()`, this is done via a `term.options=` control parameter, and for those that do not, such as `summary()`, it is done by passing the options directly or by passing a `term.options=` argument with the list.

Options used as of this writing include:

version A string that can be interpreted as an R package version. If set, the term will attempt to emulate its behavior as it was that version of **ergm**. Not all past version behaviors are available.

gw.cutoff In geometrically weighted terms (`gwesp`, `gwdegree`, etc.) the highest number of shared partners, degrees, etc. for which to compute the statistic. This usually defaults to 30.

cache.sp Whether the `gwesp`, `dgwesp`, and similar terms need should use a cache for the dyadwise number of shared partners. This usually improves performance significantly and therefore defaults to `TRUE`, but it can be disabled.

interact.dependent How to handle interaction terms, using `:` or `*`, involving dyad-dependent terms. Possible values are “error” (the default), “message”, “warning”, and “silent”. Each of the last three will allow such terms, defined as described in Section 4.2 via their change statistics.

13 Discussion

Since version 2.1 of the **ergm** package was released concurrently with Hunter et al. (2008) over a decade ago, the package has undergone substantial changes. This paper describes the changes that are most likely to be of general interest, including—but not limited to—those that are new with the release of major version 4.0 (Handcock et al., 2021). Development of **ergm** and the growing list of related packages, many of which are described in Section 2 of this article, is ongoing. Thus, while this article describes many new features, it represents a snapshot of the evolving code comprising the **statnet** suite of packages for R (R Core Team, 2021).

Acknowledgments

Many individuals have contributed code for version 4.0 of **ergm**, particularly Mark Handcock, who wrote most of the code upon which Section 10 is based, and Michał Bojanowski, who produced the **predict** method of Section 12.4, among many other contributions by both of them. Skye Bender-deMoll wrote a vignette that automatically cross-references **ergm** model terms, Carter Butts originated the **trustregion** concept used in several **ergm** algorithms, and Christian Schmid contributed code implementing the MPLE standard errors described in Section 8.1. Other important contributors are Steven Goodreau, Ayn Leslie-Cook, Li Wang, and Kirk Li.

References

- Asuncion, A., Liu, Q., Ihler, A., & Smyth, P. (2010). Learning with blocks: Composite likelihood and contrastive divergence. In Y. W. Teh & M. Titterton (Eds.), *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (Vol. 9, pp. 33–40). PMLR. <http://proceedings.mlr.press/v9/asuncion10a.html>
- Bender-deMoll, S. (2016). *Temporal network tools in statnet: networkDynamic, ndtv And tsna*. Statnet Development Team. http://statnet.org/Workshops/ndtv_workshop.html
- Butts, C. T. (2008). Social network analysis with sna. *Journal of Statistical Software*, 24(6), 1–51. <https://doi.org/10.18637/jss.v024.i06>
- Caimo, A., & Gollini, I. (2020). A multilayer exponential random graph modelling approach for weighted networks. *Computational Statistics and Data Analysis*, 142, 106825. <https://doi.org/10.1016/j.csda.2019.106825>
- Coleman, J. S. (1964). *Introduction to mathematical sociology*. The Free Press of Glencoe.
- R Core Team. (2021). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. <http://www.R-project.org/>
- Duijn, M. A. J. van, Gile, K. J., & Handcock, M. S. (2009). A framework for the comparison of maximum pseudo-likelihood and maximum likelihood estimation of exponential family random graph models. *Social Networks*, 31(1), 52–62. <https://doi.org/10.1016/j.socnet.2008.10.003>
- Gelman, A., & Meng, X.-L. (1998). Simulating normalizing constants: From importance sampling to bridge sampling to path sampling. *Statistical Science*, 13, 163–185.
- Geweke, J. (1991). *Bayesian statistics 4* (J. M. Bernardo, J. O. Berger, A. P. Dawid, & A. F. M. Smith, Eds.). Federal Reserve Bank of Minneapolis, Research Department Minneapolis, MN, USA.
- Handcock, M. S. (2003). *Assessing degeneracy in statistical models of social networks*. University of Washington. <https://csss.uw.edu/files/working-papers/2003/wp39.pdf>
- Handcock, M. S., & Gile, K. J. (2010). Modeling social networks from sampled data. *Annals of Applied Statistics*, 4(1), 5–25. <https://doi.org/10.1214/08-A0AS221>
- Handcock, M. S., Hunter, D. R., Butts, C. T., Goodreau, S. M., Krivitsky, P. N., & Morris, M. (2021). *Ergm: Fit, simulate and diagnose exponential-family models for networks*. The Statnet Project (<https://statnet.org>). <https://CRAN.R-project.org/package=ergm>
- Henry, L., & Wickham, H. (2020). *purrr: Functional programming tools*. <https://CRAN.R-project.org/package=purrr>

- Holland, P. W., & Leinhardt, S. (1981). An exponential family of probability distributions for directed graphs. *Journal of the American Statistical Association*, 76(373), 33–50.
- Hummel, R. M., Hunter, D. R., & Handcock, M. S. (2012). Improving simulation-based algorithms for fitting ERGMs. *Journal of Computational and Graphical Statistics*, 21(4), 920–939. <https://doi.org/10.1080/10618600.2012.679224>
- Hunter, D. R., & Goodreau, S. M. (2019). *Extending ergm functionality within statnet: Building custom user terms*. Statnet Development Team. http://statnet.org/Workshops/ergm.userterms_tutorial.pdf
- Hunter, D. R., Goodreau, S. M., & Handcock, M. S. (2013). ergm.userterms: A template package for extending statnet. *Journal of Statistical Software*, 52(2), 1–25. <https://doi.org/10.18637/jss.v052.i02>
- Hunter, D. R., & Handcock, M. S. (2006). Inference in curved exponential family models for networks. *Journal of Computational and Graphical Statistics*, 15(3), 565–583. <https://doi.org/10.1198/106186006x133069>
- Hunter, D. R., Handcock, M. S., Butts, C. T., Goodreau, S. M., & Morris, M. (2008). ergm: A package to fit, simulate and diagnose exponential-family models for networks. *Journal of Statistical Software*, 24(3), 1–29. <https://doi.org/10.18637/jss.v024.i03>
- Jenness, S. M., Goodreau, S. M., & Morris, M. (2018). EpiModel: An R package for mathematical modeling of infectious disease over networks. *Journal of Statistical Software*, 84(8), 1–47. <https://doi.org/10.18637/jss.v084.i08>
- Karwa, V., Krivitsky, P. N., & Slavković, A. B. (2017). Sharing social network data: Differentially private estimation of exponential-family random graph models. *Journal of the Royal Statistical Society, Series C*, 66(3), 481–500. <https://doi.org/10.1111/rssc.12185>
- Krivitsky, P. N. (2012). Exponential-family random graph models for valued networks. *Electronic Journal of Statistics*, 6, 1100–1128. <https://doi.org/10.1214/12-EJS696>
- Krivitsky, P. N. (2020). *rle: Common functions for run-length encoded vectors*. <https://CRAN.R-project.org/package=rle>
- Krivitsky, P. N. (2017). Using contrastive divergence to seed Monte Carlo MLE for exponential-family random graph models. *Computational Statistics & Data Analysis*, 107, 149–161. <https://doi.org/10.1016/j.csda.2016.10.015>
- Krivitsky, P. N., & Butts, C. T. (2017). Exponential-family random graph models for rank-order relational data. *Sociological Methodology*, 47(1), 68–112. <https://doi.org/10.1177/0081175017692623>
- Krivitsky, P. N., & Butts, C. T. (2019). *Modeling valued networks with statnet*. Statnet Development Team. <http://statnet.org/Workshops/valued.html>
- Krivitsky, P. N., & Handcock, M. S. (2008). Fitting position latent cluster models for social networks with latentnet. *Journal of Statistical Software*, 24(5), 1–23. <https://doi.org/10.18637/jss.v024.i05>
- Krivitsky, P. N., & Handcock, M. S. (2014). A separable model for dynamic networks. *Journal of the Royal Statistical Society, Series B*, 76(1), 29–46. <https://doi.org/10.1111/rssb.12014>
- Krivitsky, P. N., Handcock, M. S., Raftery, A. E., & Hoff, P. D. (2009). Representing degree distributions, clustering, and homophily in social networks with latent cluster random effects models. *Social Networks*, 31(3), 204–213. <https://doi.org/10.1016/j.socnet.2009.04.001>
- Krivitsky, P. N., Koehly, L. M., & Marcum, C. S. (2020). Exponential-family random graph models for multi-layer networks. *Psychometrika*, 85, 630–659. <https://doi.org/10.1007/s11336-020-09720-7>
- Krivitsky, P. N., & Morris, M. (2017). Inference for social network models from egocentrically-sampled data, with application to understanding persistent racial disparities in HIV prevalence in the US. *Annals of Applied Statistics*, 11(1), 427–455. <https://doi.org/10.1214/16-AOAS1010>
- Meng, X.-L., & Wong, W. H. (1996). Simulating ratios of normalizing constants via a simple identity: A theoretical exploration. *Statistica Sinica*, 6, 831–860.
- Morris, M., Handcock, M. S., & Hunter, D. R. (2008). Specification of exponential-family random graph models: Terms and computational aspects. *Journal of Statistical Software*, 24(4), 1–24. <https://doi.org/10.18637/jss.v024.i04>

- Morris, M., & Krivitsky, P. N. (2015). *Temporal exponential random graph models (TERGMs) for dynamic network modeling in statnet*. Statnet Development Team. http://statnet.org/Workshops/tergm_tutorial.html
- Morris, M., & Krivitsky, P. N. (2019). *Introduction to egocentric network data analysis with ERGMs using statnet*. Statnet Development Team. http://statnet.org/Workshops/ergm.ego_tutorial.html
- National Center for Health Statistics. (2020). *2006–2015 national survey of family growth*. <https://www.cdc.gov/nchs/nsfg/index.htm>
- Plummer, M., Best, N., Cowles, K., & Vines, K. (2006). CODA: Convergence diagnosis and output analysis for MCMC. *R News*, 6(1), 7–11. <https://journal.r-project.org/archive/>
- Robins, G., Pattison, P., & Wasserman, S. (1999). Logit models and logistic regressions for social networks: III. Valued relations. *Psychometrika*, 64(3), 371–394.
- Sampson, S. F. (1968). *A novitiate in a period of change: An experimental and case study of social relationships* [Ph.D. thesis (University Microfilm, No 69-5775)]. Department of Sociology, Cornell University.
- Schmid, C. S., & Desmarais, B. A. (2017). Exponential random graph models with big networks: Maximum pseudolikelihood estimation and the parametric bootstrap. *2017 IEEE International Conference on Big Data (Big Data)*, 116–121. <https://doi.org/10.1109/bigdata.2017.8257919>
- Schmid, C. S., & Hunter, D. R. (2020). *Improving ERGM starting values using simulated annealing*. <https://arxiv.org/abs/2009.01202>
- Schmid, C. S., & Hunter, D. R. (2021). *Accounting for model misspecification when using pseudolikelihood for ERGMs*.
- Schweinberger, M., Krivitsky, P. N., Butts, C. T., & Stewart, J. R. (2020). Exponential-family models of random graphs: Inference in finite, super and infinite population scenarios. *Statistical Science*, 35(4), 627–662. <https://doi.org/10.1214/19-STS743>
- Slaughter, A. J., & Koehly, L. M. (2016). Multilevel models for social networks: Hierarchical Bayesian approaches to exponential random graph modeling. *Social Networks*, 44, 334–345. <https://doi.org/10.1016/j.socnet.2015.11.002>
- Snijders, T. A. B. (2002). Markov chain Monte Carlo estimation of exponential random graph models. *Journal of Social Structure*, 3(2). <https://www.cmu.edu/joss/content/articles/volume3/Snijders.pdf>
- Vats, D., Flegal, J. M., & Jones, G. L. (2019). Multivariate output analysis for Markov chain Monte Carlo. *Biometrika*, 106(2), 321–337. <https://doi.org/10.1093/biomet/asz002>
- Vega Yon, G. G., Slaughter, A., & de la Haye, K. (2021). Exponential random graph models for little networks. *Social Networks*, 64, 225–238. <https://doi.org/10.1016/j.socnet.2020.07.005>
- Wang, P. (2012). Exponential random graph model extensions: Models for multiple networks and bipartite networks. In D. Lusher, J. Koskinen, & G. Robins (Eds.), *Exponential random graph models for social networks: Theory, methods, and applications* (pp. 115–129). Cambridge University Press. <https://doi.org/10.1017/CB09780511894701.012>

A Supplemental Code

The results of several simulation tests involving large networks are reported in Section 11. This document provides the code used in those simulations.

A.1 MCMC speedup

This code of this section produced the results of Section 11.1 as summarized in Figure 3.

```
library(ergm)
library(parallel)
load("cohab.RData")

set.seed(0)
net_size <- 50000
nw <- network.initialize(net_size, directed = FALSE)
```

```

inds <- sample(seq_len(NROW(cohab_PopWts)), net_size, TRUE, cohab_PopWts$weight)
set.vertex.attribute(nw, names(cohab_PopWts)[-1], cohab_PopWts[inds,-1])
fit <- ergm(nw ~ edges +
  nodefactor("sex.ident", levels = 3) +
  nodecov("age") +
  nodecov("agesq") +
  nodefactor("race", levels = -5) +
  nodefactor("othr.net.deg", levels = -1) +
  nodematch("race", diff = TRUE) +
  absdiff("sqrt.age.adj") +
  offset(nodematch("sex", diff = FALSE)) +
  offset(concurrent),
  target.stats = cohab_TargetStats,
  offset.coef = c(-Inf, -Inf),
  eval.loglik = FALSE,
  constraints = ~bd(maxout = 1) + blocks(attr = ~sex, levels2 = diag(TRUE, 2)),
  control = snctrl(MCMC.prop = ~strat(attr = ~race, empirical = TRUE) + sparse,
    init.method = "MPLE",
    init.MPLE.samplesize = 5e7,
    MPLE.constraints.ignore = TRUE,
    MCMLE.effectiveSize = NULL,
    MCMC.burnin = 5e4,
    MCMC.interval = 5e4,
    MCMC.samplesize = 7500,
    parallel = 16,
    SAN.nsteps = 5e7,
    SAN.prop=~strat(attr = ~race, pmat = cohab_MixMat) + sparse))

el <- do.call(rbind, lapply(fit$newnetworks, as.edgelist))
attrs <- cbind(nw %v% "race", nw %v% "age")
colnames(attrs) <- c("race", "age")
tailattrs <- attrs[el[,1],]
headattrs <- attrs[el[,2],]
attrnames <- "race"
levs <- sort(unique(apply(attrs[,attrnames,drop=FALSE], 1, paste, collapse = ".")))
tails <- factor(apply(tailattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
  levels = levs)
heads <- factor(apply(headattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
  levels = levs)
mmr <- table(from = tails, to = heads)
mmr <- mmr + t(mmr) - diag(diag(mmr))

attrnames <- c("race", "age")
levs <- sort(unique(apply(attrs[,attrnames,drop=FALSE], 1, paste, collapse = ".")))
tails <- factor(apply(tailattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
  levels = levs)
heads <- factor(apply(headattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
  levels = levs)
mmra <- table(from = tails, to = heads)
mmra <- mmra + t(mmra) - diag(diag(mmra))

## ensure positive proposal weight for all allowed pairings
mmra[mmra == 0] <- 1/2

net_size <- 1000000
nw <- network.initialize(net_size, directed = FALSE)
inds <- rep(inds, length.out = net_size)
set.vertex.attribute(nw, names(cohab_PopWts)[-1], cohab_PopWts[inds,-1])
coef <- fit$coef[seq_len(length(fit$coef) - 2)]
coef[1] <- coef[1] + log(50000) - log(1000000)
cohab_TargetStats <- cohab_TargetStats*20

MillionNodeFormula <-
  nw ~ edges + nodefactor("sex.ident", levels = 3) + nodecov("age") + nodecov("agesq") +

```



```

    nodefactor("race", levels = -5) + nodefactor("othr.net.deg", levels = -1) +
    nodematch("race", diff = TRUE) + absdiff("sqrt.age.adj")
attribs <- matrix(FALSE, nrow = network.size(nw), ncol = 2)
attribs[nw %v% "sex" == "M", 1] <- TRUE
attribs[nw %v% "sex" == "F", 2] <- TRUE
maxout <- matrix(0, nrow = network.size(nw), ncol = 2)
maxout[nw %v% "sex" == "M", 2] <- 1
maxout[nw %v% "sex" == "F", 1] <- 1
con_list <- list("TNT"~bd(attribs = attribs, maxout = maxout),
  ~bd(maxout = 1) + blocks(attr = "sex", levels2 = diag(TRUE, 2))
  + strat(attr = ~paste(race, sep = "."), pmat = mmr),
  ~bd(maxout = 1) + blocks(attr = "sex", levels2 = diag(TRUE, 2))
  + strat(attr = ~paste(race, age, sep = "."), pmat = mmra))
names_vec <- c("\TNT\"~bd(sex,1)",
  "~bd(1)+blocks(sex)+strat(race)",
  "~bd(1)+blocks(sex)+strat(race,age)")
nsim <- 100000000
interval <- 1000
run_simulate <- function(constraint) {
  library(ergm)
  st <- Sys.time()
  x <- simulate(MillionNodeFormula,
    coef = coef,
    constraints = constraint,
    nsim = nsim,
    output = "stats",
    control = snctrl(MCMC.interval = interval, MCMC.burnin = interval))
  et <- Sys.time()
  x <- matrix(c(x), nrow = nsim, dimnames = dimnames(x))
  list(statsmatrix = x, timediff = et - st)
}

cl <- makeCluster(length(con_list))
clusterExport(cl, "nw")
clusterExport(cl, "MillionNodeFormula")
clusterExport(cl, "coef")
clusterExport(cl, "mmr")
clusterExport(cl, "mmra")
clusterExport(cl, "nsim")
clusterExport(cl, "interval")
clusterExport(cl, "attribs")
clusterExport(cl, "maxout")
rv <- clusterApply(cl, con_list, run_simulate)

stopCluster(cl)
z <- lapply(rv, `[`, "statsmatrix")
times <- lapply(rv, `[`, "timediff")
indices <- as.integer(exp(seq(from=0,to=log(nsim),length.out=1000)))

pdf("MCMC_trace_plots_constrained.pdf")
for(j in seq_len(15)) {
  plot(log(interval*indices)/log(10), z[[1]][indices,j], cex=0.1,
    ylim = c(0, max(z[[1]][,j], z[[2]][,j], z[[3]][,j])), ylab = "Statistic",
    xlab = "log10(Number of Proposals)", main = colnames(z[[1]])[j])
  points(log(interval*indices)/log(10), z[[2]][indices,j], cex=0.1, col = "red")
  points(log(interval*indices)/log(10), z[[3]][indices,j], cex=0.1, col = "green")
  abline(h = cohab_TargetStats[j], col="purple")
}
dev.off()

pdf("MCMC_trace_plots_4_panel_constrained.pdf")
par(mfrow=c(2,2))
for(j in c(1, 10, 9, 15)) {
  plot(log(interval*indices)/log(10), z[[1]][indices,j], cex=0.1,

```

Table 3: Effective sample sizes per minute for 4 of the 15 statistics from an MCMC sample of size 10 million with interval 100 using the hints and constraints shown in the leftmost column in addition to the **sparse** hint, which is used in all cases.

	edges	race B homoph.	other net deg. 1+	$\sqrt{\text{age diff.}}$
"TNT" bd(sex,1)	471	35	822	366
bd(1)+blocks(sex)	4830	222	2068	2379
bd(1)+blocks(sex)+strat(race)	4635	744	1990	2384
bd(1)+blocks(sex)+strat(race.mod)	3285	2112	1734	1754
bd(1)+blocks(sex)+strat(race,age)	2757	473	1562	616

```

ylim = c(0, max(z[[1]][,j], z[[2]][,j], z[[3]][,j])), ylab = "Statistic",
xlab = "log10(Number of Proposals)", main = colnames(z[[1]])[j])
points(log(interval*indices)/log(10), z[[2]][indices,j], cex=0.1, col = "red")
points(log(interval*indices)/log(10), z[[3]][indices,j], cex=0.1, col = "green")
abline(h = cohab_TargetStats[j], col="purple")
}
dev.off()
# Results are now contained in the z object
# save(z, times, nsim, interval, names_vec, file = "MCMC_test_results_constrained.rdata")

```

A.2 Likelihood calculation efficiency gains

The code of this section produced the results of Section 11.2, as summarized in Table 1. In addition, Table 3 shows the effective sample size (ESS) per minute of run time, which allows a comparison that takes computational complexity into account, for the same sets of hints and constraints as in Table 1. Comparing rows 1 and 4, the improvement in minimum ESS per minute across all 15 statistics, not merely the four statistics shown here, was roughly 90-fold.

As mentioned in Section 11.2, the weight matrix passed via **pmat** to the **strat** hint is modified so that smaller race groups are proposed more frequently by the Metropolis-Hastings algorithm than their edge fraction alone would indicate. The construction of this modified race weight matrix, which is called **mmr_mod**, is seen in the code below.

```

library(ergm)
library(parallel)
load("cohab.RData")

set.seed(0)
net_size <- 50000
nw <- network.initialize(net_size, directed = FALSE)
inds <- sample(seq_len(NROW(cohab_PopWts)), net_size, TRUE, cohab_PopWts$weight)
set.vertex.attribute(nw, names(cohab_PopWts)[-1], cohab_PopWts[inds,-1])
fit <- ergm(nw ~ edges +
  nodefactor("sex.ident", levels = 3) +
  nodecov("age") +
  nodecov("agesq") +
  nodefactor("race", levels = -5) +
  nodefactor("othr.net.deg", levels = -1) +
  nodematch("race", diff = TRUE) +
  absdiff("sqrt.age.adj") +
  offset(nodematch("sex", diff = FALSE)) +
  offset(concurrent),
  target.stats = cohab_TargetStats,
  offset.coef = c(-Inf, -Inf),
  eval.loglik = FALSE,
  constraints = ~bd(maxout = 1) + blocks(attr = ~sex, levels2 = diag(TRUE, 2)),
  control = snctrl(MCMC.prop = ~strat(attr = ~race, empirical = TRUE) + sparse,
    init.method = "MPLE",

```

```

init.MPLE.samplesize = 5e7,
MPLE.constraints.ignore = TRUE,
MCMLE.effectiveSize = NULL,
MCMC.burnin = 5e4,
MCMC.interval = 5e4,
MCMC.samplesize = 7500,
parallel = 16,
SAN.nsteps = 5e7,
SAN.prop=~strat(attr = ~race, pmat = cohab_MixMat) + sparse))

nw <- fit$newnetworks[[1]]
el <- do.call(rbind, lapply(fit$newnetworks, as.edgelist))
attrs <- cbind(nw %v% "race", nw %v% "age")
colnames(attrs) <- c("race", "age")
tailattrs <- attrs[el[,1],]
headattrs <- attrs[el[,2],]
attrnames <- "race"
levs <- sort(unique(apply(attrs[,attrnames,drop=FALSE], 1, paste, collapse = ".")))
tails <- factor(apply(tailattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
               levels = levs)
heads <- factor(apply(headattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
               levels = levs)
mmr <- table(from = tails, to = heads)
mmr <- mmr + t(mmr) - diag(diag(mmr))

attrnames <- c("race", "age")
levs <- sort(unique(apply(attrs[,attrnames,drop=FALSE], 1, paste, collapse = ".")))
tails <- factor(apply(tailattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
               levels = levs)
heads <- factor(apply(headattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
               levels = levs)
mmra <- table(from = tails, to = heads)
mmra <- mmra + t(mmra) - diag(diag(mmra))

## ensure positive proposal weight for all allowed pairings
mmra[mmra == 0] <- 1/2
mmr_mod <- mmr
mmr_mod[-(4:5),] <- mmr_mod[-(4:5)]*sqrt(6)
mmr_mod[,-(4:5)] <- mmr_mod[-(4:5),]*sqrt(6)
mmr_mod[3,] <- mmr_mod[3,]*sqrt(2)
mmr_mod[,3] <- mmr_mod[,3]*sqrt(2)
mmr_mod[4,] <- 1.5*mmr_mod[4,]
mmr_mod[,4] <- 1.5*mmr_mod[,4]

coef <- fit$coef[seq_len(length(fit$coef) - 2)]
nsim <- 10000000
interval <- 100
ff <- nw ~ edges +
  nodefactor("sex.ident", levels = 3) +
  nodecov("age") +
  nodecov("agesq") +
  nodefactor("race", levels = -5) +
  nodefactor("othr.net.deg", levels = -1) +
  nodematch("race", diff = TRUE) +
  absdiff("sqrt.age.adj")
attribs <- matrix(FALSE, nrow = network.size(nw), ncol = 2)
attribs[nw %v% "sex" == "M", 1] <- TRUE
attribs[nw %v% "sex" == "F", 2] <- TRUE
maxout <- matrix(0, nrow = network.size(nw), ncol = 2)
maxout[nw %v% "sex" == "M", 2] <- 1
maxout[nw %v% "sex" == "F", 1] <- 1

con_list <- list("TNT"~bd(attribs = attribs, maxout = maxout),
               ~bd(maxout = 1) + blocks(attr = "sex", levels2 = diag(TRUE, 2)),

```

```

~bd(maxout = 1) + blocks(attr = "sex", levels2 = diag(TRUE, 2))
+ strat(attr = ~paste(race, sep = "."), pmat = mmr),
~bd(maxout = 1) + blocks(attr = "sex", levels2 = diag(TRUE, 2))
+ strat(attr = ~paste(race, sep = "."), pmat = mmr_mod),
~bd(maxout = 1) + blocks(attr = "sex", levels2 = diag(TRUE, 2))
+ strat(attr = ~paste(race, age, sep = "."), pmat = mmra))
names_vec <- c("\TNT\"~bd(sex,1)",
               "~bd(1)+blocks(sex)",
               "~bd(1)+blocks(sex)+strat(race)",
               "~bd(1)+blocks(sex)+strat(race.mod)",
               "~bd(1)+blocks(sex)+strat(race,age)")

run_ess <- function(constraint) {
  library(statnet.common)
  library(ergm)
  x <- simulate(ff,
               coef = coef,
               constraints = constraint,
               nsim = nsim,
               output = "stats",
               control = snctrl(MCMC.interval = interval, MCMC.burnin = interval))
  y <- coda::effectiveSize(x)
  list(x = x, y = y)
}

cl <- makeCluster(length(con_list))
clusterExport(cl, "nw")
clusterExport(cl, "mmr")
clusterExport(cl, "mmr_mod")
clusterExport(cl, "mmra")
clusterExport(cl, "ff")
clusterExport(cl, "coef")
clusterExport(cl, "nsim")
clusterExport(cl, "interval")
clusterExport(cl, "attribs")
clusterExport(cl, "maxout")
rv <- clusterApply(cl, con_list, run_ess)
stopCluster(cl)

# x <- lapply(rv, `[`, "x")
y <- lapply(rv, `[`, "y")
z <- do.call(rbind, y)
rownames(z) <- names_vec

burnin <- nsim*interval
times <- list()
for(i in seq_along(con_list)) {
  print(i)
  st <- Sys.time()
  x <- simulate(ff,
               coef = coef,
               constraints = con_list[[i]],
               nsim = 1,
               control = snctrl(MCMC.burnin = burnin))
  et <- Sys.time()
  times[[i]] <- et - st
  print(et - st)
}

w <- z
for(i in seq_len(NROW(w))) {
  w[i,] <- w[i,] / as.numeric(times[[i]])
}
# Results are now contained in the z and w objects

```

```
#save(times, w, z, nsim, interval, burnin, con_list, names_vec,
#      file = "ess_benchmarks_constrained.rdata")
```

A.3 SAN speedup

The code of this section produced the results of Section 11.3 as summarized in Figure 4.

```
library(ergm)
library(parallel)
load("cohab.RData")

set.seed(0)
net_size <- 50000
nw <- network.initialize(net_size, directed = FALSE)
inds <- sample(seq_len(NROW(cohab_PopWts)), net_size, TRUE, cohab_PopWts$weight)
set.vertex.attribute(nw, names(cohab_PopWts)[-1], cohab_PopWts[inds,-1])
fit <- ergm(nw ~ edges +
  nodefactor("sex.ident", levels = 3) +
  nodecov("age") +
  nodecov("agesq") +
  nodefactor("race", levels = -5) +
  nodefactor("othr.net.deg", levels = -1) +
  nodematch("race", diff = TRUE) +
  absdiff("sqrt.age.adj") +
  offset(nodematch("sex", diff = FALSE)) +
  offset(concurrent),
  target.stats = cohab_TargetStats,
  offset.coef = c(-Inf, -Inf),
  eval.loglik = FALSE,
  constraints = ~bd(maxout = 1) + blocks(attr = ~sex, levels2 = diag(TRUE, 2)),
  control = snctrl(MCMC.prop = ~strat(attr = ~race, empirical = TRUE) + sparse,
    init.method = "MPLE",
    init.MPLE.samplesize = 5e7,
    MPLE.constraints.ignore = TRUE,
    MCMLE.effectiveSize = NULL,
    MCMC.burnin = 5e4,
    MCMC.interval = 5e4,
    MCMC.samplesize = 7500,
    parallel = 16,
    SAN.steps = 5e7,
    SAN.prop=~strat(attr = ~race, pmat = cohab_MixMat) + sparse))
el <- do.call(rbind, lapply(fit$newnetworks, as.edgelist))
attrs <- cbind(nw %v% "race", nw %v% "age")
colnames(attrs) <- c("race", "age")
tailattrs <- attrs[el[,1],]
headattrs <- attrs[el[,2],]
attrnames <- "race"
levs <- sort(unique(apply(attrs[,attrnames,drop=FALSE], 1, paste, collapse = ".")))
tails <- factor(apply(tailattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
  levels = levs)
heads <- factor(apply(headattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
  levels = levs)
mmr <- table(from = tails, to = heads)
mmr <- mmr + t(mmr) - diag(diag(mmr))

attrnames <- c("race", "age")
levs <- sort(unique(apply(attrs[,attrnames,drop=FALSE], 1, paste, collapse = ".")))
tails <- factor(apply(tailattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
  levels = levs)
heads <- factor(apply(headattrs[,attrnames,drop=FALSE], 1, paste, collapse = "."),
  levels = levs)
mmra <- table(from = tails, to = heads)
mmra <- mmra + t(mmra) - diag(diag(mmra))
```

```

## ensure positive proposal weight for all allowed pairings
mmra[mmra == 0] <- 1/2
net_size <- 1000000
nw <- network.initialize(net_size, directed = FALSE)
inds <- rep(inds, length.out = net_size)
set.vertex.attribute(nw, names(cohab_PopWts)[-1], cohab_PopWts[inds,-1])

cohab_TargetStats <- cohab_TargetStats*20
ff <- nw ~ edges +
  nodefactor("sex.ident", levels = 3) +
  nodecov("age") +
  nodecov("agesq") +
  nodefactor("race", levels = -5) +
  nodefactor("othr.net.deg", levels = -1) +
  nodematch("race", diff = TRUE) +
  absdiff("sqrt.age.adj")
attribs <- matrix(FALSE, nrow = network.size(nw), ncol = 2)
attribs[nw %v% "sex" == "M", 1] <- TRUE
attribs[nw %v% "sex" == "F", 2] <- TRUE
maxout <- matrix(0, nrow = network.size(nw), ncol = 2)
maxout[nw %v% "sex" == "M", 2] <- 1
maxout[nw %v% "sex" == "F", 1] <- 1

con_list <- list("TNT"~bd(attribs = attribs, maxout = maxout),
  ~bd(maxout = 1) + blocks(attr = "sex", levels2 = diag(TRUE, 2))
  + strat(attr = ~paste(race, sep = "."), pmat = mmr),
  ~bd(maxout = 1) + blocks(attr = "sex", levels2 = diag(TRUE, 2))
  + strat(attr = ~paste(race, age, sep = "."), pmat = mmra))
names_vec <- c("\TNT\"~bd(sex,1)",
  "~bd(1)+blocks(sex)+strat(race)",
  "~bd(1)+blocks(sex)+strat(race,age)")

samplesize <- 1000000
nsteps <- 100000000

invcov <- diag(1/(cohab_TargetStats**2))
invcov <- invcov/sum(invcov)

run_san <- function(constraint) {
  library(ergm)

  st <- Sys.time()
  rv <- san(ff,
    constraints = constraint,
    target.stats = cohab_TargetStats,
    control = snctrl(SAN.maxit = 1,
      SAN.invcov = invcov,
      SAN.nsteps = nsteps,
      SAN.samplesize = samplesize))

  et <- Sys.time()

  sm <- attr(rv, "stats")
  sm <- t(t(sm) + cohab_TargetStats)

  list(statsmatrix = sm, timediff = et - st)
}

cl <- makeCluster(length(con_list))
clusterExport(cl, "ff")
clusterExport(cl, "nw")
clusterExport(cl, "mmr")
clusterExport(cl, "mmra")
clusterExport(cl, "samplesize")
clusterExport(cl, "nsteps")

```

```

clusterExport(cl, "invcov")
clusterExport(cl, "cohab_TargetStats")
clusterExport(cl, "attribs")
clusterExport(cl, "maxout")
rv <- clusterApply(cl, con_list, run_san)
stopCluster(cl)

z <- lapply(rv, `[`, "statsmatrix")
times <- lapply(rv, `[`, "timediff")
indices <- as.integer(exp(seq(from=0,to=log(samplesize),length.out=1000)))

pdf("SAN_trace_plots_constrained.pdf")
for(j in seq_len(15)) {
  plot(log(nsteps*indices/samplesize)/log(10), z[[1]][indices,j], cex=0.1,
        ylim = c(0, max(z[[1]][,j], z[[2]][,j], z[[3]][,j])), ylab = "Statistic",
        xlab = "log10(Number of Proposals)", main = colnames(z[[1]])[j])
  points(log(nsteps*indices/samplesize)/log(10), z[[2]][indices,j], cex=0.1, col = "red")
  points(log(nsteps*indices/samplesize)/log(10), z[[3]][indices,j], cex=0.1, col = "green")
  abline(h = cohab_TargetStats[j], col="purple")
}
dev.off()

pdf("SAN_trace_plots_4_panel_constrained.pdf")
par(mfrow=c(2,2))
for(j in c(1, 10, 9, 15)) {
  plot(log(nsteps*indices/samplesize)/log(10), z[[1]][indices,j], cex=0.1,
        ylim = c(0, max(z[[1]][,j], z[[2]][,j], z[[3]][,j])), ylab = "Statistic",
        xlab = "log10(Number of Proposals)", main = colnames(z[[1]])[j])
  points(log(nsteps*indices/samplesize)/log(10), z[[2]][indices,j], cex=0.1, col = "red")
  points(log(nsteps*indices/samplesize)/log(10), z[[3]][indices,j], cex=0.1, col = "green")
  abline(h = cohab_TargetStats[j], col="purple")
}
dev.off()

# Results are now contained in the z object
# save(cohab_TargetStats, z, times, samplesize, nsteps, invcov,
#      file = "SAN_test_results_constrained.rdata")

```

A.4 Estimation speedup

The code of this section produced the results of Section 11.4 as summarized in Table 2. First, we present the code that implements the non-adaptive MCMC intervals in **ergm** version 4.0:

```

## non-adaptive version; make sure ergm 4.0 is installed
library(ergm)
load("cohab.RData")

times <- list()
for(i in seq_len(10)) {
  set.seed(i)
  net_size <- 50000
  nw <- network.initialize(net_size, directed = FALSE)
  inds <- sample(seq_len(NROW(cohab_PopWts)), net_size, TRUE, cohab_PopWts$weight)
  set.vertex.attribute(nw, names(cohab_PopWts)[-1], cohab_PopWts[inds,-1])
  st <- Sys.time()
  fit <- ergm(nw ~ edges +
    nodefactor("sex.ident", levels = 3) +
    nodecov("age") +
    nodecov("agesq") +
    nodefactor("race", levels = -5) +
    nodefactor("othr.net.deg", levels = -1) +
    nodematch("race", diff = TRUE) +
    absdiff("sqrt.age.adj") +

```



```

        offset(nodematch("sex", diff = FALSE)) +
        offset(concurrent),
        target.stats = cohab_TargetStats,
        offset.coef = c(-Inf, -Inf),
        eval.loglik = FALSE,
        constraints = ~bd(maxout = 1) + blocks(attr = ~sex, levels2 = diag(TRUE, 2)),
        control = snctrl(MCMC.prop = ~strat(attr = ~race, empirical = TRUE) + sparse,
                          init.method = "MPLE",
                          init.MPLE.samplesize = 5e7,
                          MPLE.constraints.ignore = TRUE,
                          MCMLE.effectiveSize = NULL,
                          MCMC.burnin = 5e4,
                          MCMC.interval = 5e4,
                          MCMC.samplesize = 7500,
                          parallel = 16,
                          SAN.nsteps = 5e7,
                          SAN.prop=~strat(attr = ~race, pmat = cohab_MixMat) + sparse))

    et <- Sys.time()
    times[[i]] <- et - st
  }
  print(mean(as.numeric(times)))

```

The non-adaptive code for **ergm** 3.10, which only runs one simulation instead of the ten used for version 4.0 above because of the vast difference in computing time required, is given below:

```

## install ergm 3.10.4 and contemporaneous dependencies from CRAN archive
archv <- "https://cran.r-project.org/src/contrib/Archive/"
install.packages(paste(archv, "statnet.common/statnet.common_4.3.0.tar.gz", sep=""),
                  repos = NULL, type = "source")
install.packages(paste(archv, "network/network_1.15.tar.gz", sep=""),
                  repos = NULL, type = "source")
install.packages(paste(archv, "ergm/ergm_3.10.4.tar.gz", sep=""),
                  repos = NULL, type = "source")

library(ergm)
load("cohab.RData")

set.seed(0)
net_size <- 50000
nw <- network.initialize(net_size, directed = FALSE)
inds <- sample(seq_len(NROW(cohab_PopWts)), net_size, TRUE, cohab_PopWts$weight)
set.vertex.attribute(nw, names(cohab_PopWts)[-1], cohab_PopWts[inds, -1])

attrib_mat <- matrix(FALSE, nrow = net_size, ncol = 2)
attrib_mat[nw %v% "sex" == "F", 1] <- TRUE
attrib_mat[nw %v% "sex" == "M", 2] <- TRUE

maxout_mat <- matrix(0, nrow = net_size, ncol = 2)
maxout_mat[nw %v% "sex" == "F", 2] <- 1
maxout_mat[nw %v% "sex" == "M", 1] <- 1

st <- Sys.time()
fit <- ergm(nw ~ edges +
            nodefactor("sex.ident", levels = 3) +
            nodecov("age") +
            nodecov("agesq") +
            nodefactor("race", levels = -5) +
            nodefactor("othr.net.deg", levels = -1) +
            nodematch("race", diff = TRUE) +
            absdiff("sqrt.age.adj") +
            offset(nodematch("sex", diff = FALSE)) +
            offset(concurrent),
            target.stats = cohab_TargetStats,
            offset.coef = c(-Inf, -Inf),

```

```

eval.loglik = FALSE,
constraints = ~bd(attribs = attrib_mat, maxout = maxout_mat),
control = control.ergm(init.method = "MPLE",
                        MCMC.burnin = 1e6,
                        MCMC.interval = 1e6,
                        MCMC.samplesize = 7500,
                        parallel = 16,
                        MCMLE.maxit = 1000,
                        SAN.control = control.san(SAN.nsteps = 1e9)))

et <- Sys.time()
print(et - st)

```

Finally, we present the adaptive version used for **ergm** version 4.0, which is the only version that implements adaptive MCMC intervals:

```

## adaptive version; make sure ergm 4.0 is installed
library(ergm)
load("cohab.RData")
times <- list()
for(i in seq_len(10)) {
  set.seed(i)
  net_size <- 50000
  nw <- network.initialize(net_size, directed = FALSE)
  inds <- sample(seq_len(NROW(cohab_PopWts)), net_size, TRUE, cohab_PopWts$weight)
  set.vertex.attribute(nw, names(cohab_PopWts)[-1], cohab_PopWts[inds,-1])
  st <- Sys.time()
  fit <- ergm(nw ~ edges +
              nodefactor("sex.ident", levels = 3) +
              nodecov("age") +
              nodecov("agesq") +
              nodefactor("race", levels = -5) +
              nodefactor("othr.net.deg", levels = -1) +
              nodematch("race", diff = TRUE) +
              absdiff("sqrt.age.adj") +
              offset(nodematch("sex", diff = FALSE)) +
              offset(concurrent),
              target.stats = cohab_TargetStats,
              offset.coef = c(-Inf, -Inf),
              eval.loglik = FALSE,
              constraints = ~bd(maxout = 1) + blocks(attr = ~sex, levels2 = diag(TRUE, 2)),
              control = snctrl(MCMC.prop = ~strat(attr = ~race, empirical = TRUE) + sparse,
                               init.method = "MPLE",
                               init.MPLE.samplesize = 5e7,
                               MPLE.constraints.ignore = TRUE,
                               parallel = 16,
                               SAN.nsteps = 5e7,
                               SAN.prop=~strat(attr = ~race, pmat = cohab_MixMat) + sparse))

  et <- Sys.time()
  times[[i]] <- et - st
}
print(mean(as.numeric(times)))

```