ergm.userterms: A Template Package for Extending statnet

David R. Hunter
The Pennsylvania State University

Steven M. Goodreau
University of Washington

Mark S. Handcock
University of California, Los Angeles

Abstract

Exponential-family random graph models (ERGM) represent a powerful and flexible class of models for the statistical analysis of networks. **statnet** is a suite of software packages that implement these models. This paper details how the capabilities for ERGM modeling can be expanded and customized by programming additional network statistics that may be included in ERGMs. We describe a template R package called **ergm.userterms** that can be modified for this purpose. It is designed to make this process as straightforward as possible. We also explain some of the internal workings of **statnet** that will help users develop their own network analysis capabilities.

Keywords: exponential-family random graph model, Markov chain Monte Carlo, maximum likelihood estimation, p-star model.

1. Introduction

Exponential-family random graphs models ("ERG models" or "ERGMs") provide a powerful and flexible family of models for conducting statistical inference on social network structure (Frank and Strauss 1986; Handcock, Hunter, Butts, Goodreau, and Morris 2008; Hunter, Handcock, Butts, Goodreau, and Morris 2008; Goodreau, Handcock, Hunter, Butts, and Morris 2008). This framework has been implemented in the **ergm** package (Hunter *et al.* 2008), part of the **statnet** suite of packages (Handcock *et al.* 2008) for R (R Development Core Team 2010). Within this framework, there is an essentially unlimited number of potential models; for any one, the probability of a given network is a function of a set of structural network statistics posited by the model. Equivalently, the probability of any individual tie in the network existing, given all other tie states, is a function of changes in those same network statistics induced by the creation of that tie.

The use of any network statistic in the **ergm** package requires a small amount of code to calculate changes to that statistic induced by adding or removing a tie; some of this code is in R, and some in C, where the most computationally intensive tasks in **ergm** are actually performed. There are a variety of network statistics with substantive interpretations that are commonly found in the literature (e.g. counts of edges, homophilous edges, nodal degrees, triangles or k-stars), and the **ergm** package includes code to handle many of these. However, the user who wishes to build a model with a network statistic term not included in the **ergm** package must write new code to do so. We have developed a package—**ergm.userterms**—to make this process easier. The aims of this paper are to describe this package and explain all of the steps of writing up new ERGM terms in order to make this process more user-friendly.

In Section 2, we provide a statistical overview of the **ergm** framework, with an emphasis on the "change statistics" that must be coded to make the **ergm** package work. In Section 3, we provide additional detail to explain why these statistics must be coded individually, unlike in traditional generalized linear models. Section 4 outlines the methods of network storage used throughout the **statnet** package suite; a basic familiarity with these methods aids the coding of **ergm** terms considerably. In Section 5 we explain how to acquire all of the necessary tools to build an R package from source and compile C code within Windows. Sections 6 and 7 describe the R code and C code that a user must write, respectively. Finally, Section 8 provides a complete worked example.

This paper will be most easy to follow for readers who already possess a general familiarity with the **ergm** and **network** packages; those without such familiarity may wish to read relevant papers in the *Journal of Statistical Software* special volume on **statnet** first (Handcock *et al.* 2008; Butts 2008; Hunter *et al.* 2008). Also, users who wish to follow along with the examples or be able to use the instructions from this paper to write their own statistics should be sure to obtain **ergm** version 2.3 or later before beginning any coding. Prior versions used different (and less user-friendly) routines for coding terms. Although these routines are still allowed for backward compatibility, not all routines in this paper will work in prior versions; moreover, mixing the two approaches will likely cause unnecessary confusion.

2. Background

At the core of the **ergm** package (Handcock, Hunter, Butts, Goodreau, Krivitsky, and Morris 2003a) for R (R Development Core Team 2010) is a sophisticated Markov chain Monte Carlo engine for simulating random networks. As explained by Hunter *et al.* (2008), simulating a Markov chain on a set of networks whose stationary distribution is given by the exponential-family random graph model

$$P_{\boldsymbol{\theta}_0}(\mathbf{Y}=\mathbf{y}) = \frac{\exp\{\boldsymbol{\theta}_0^\top \mathbf{g}(\mathbf{y})\}}{\kappa(\boldsymbol{\theta})} \tag{1}$$

is vitally important not only for simulation but also for estimation. In equation (1), $\boldsymbol{\theta}_0 \in \mathbb{R}^p$ is a fixed parameter vector, $\mathbf{g}(\mathbf{y})$ is a user-defined p-vector of statistics on the network $\mathbf{y}$ assumed to come from some set $\mathcal{Y}$ of networks, and

$$\kappa(\boldsymbol{\theta}) = \sum_{\mathbf{z}\in\mathcal{Y}} \exp\{\boldsymbol{\theta}_0^\top \mathbf{g}(\mathbf{z})\} \tag{2}$$

is the normalizing constant.

The MCMC scheme implemented in **ergm** is called a Metropolis-Hastings algorithm. In general terms, such an algorithm proceeds from step k to step $k+1$, from one network $\mathbf{y}^k$ to the next, by (a) selecting a candidate for the next network $\mathbf{y}^{k+1}$, (b) computing a *Hastings ratio* based on $\mathbf{y}^k$ and the candidate network, and (c) deciding whether $\mathbf{y}^{k+1}$ should be set to the candidate network or, alternatively, whether to stay put for another iteration and set $\mathbf{y}^{k+1} = \mathbf{y}^k$. The selection (a) is done so that any possible network, say $\mathbf{z}$, has probability $q(\mathbf{z}, \mathbf{y}^k)$ of being selected, where q is some probability function known to the user. (The q

function can and, in general, does place probability zero on some values of $\mathbf{z}$, depending on the value of $\mathbf{y}^k$.) The Hastings ratio (b) is equal to

$$\frac{P_{\boldsymbol{\theta}_0}(\mathbf{Y}=\mathbf{y}^k)}{P_{\boldsymbol{\theta}_0}(\mathbf{Y}=\mathbf{z})}\frac{q(\mathbf{y}^k,\mathbf{z})}{q(\mathbf{z},\mathbf{y}^k)}, \tag{3}$$

where $\mathbf{z}$ is the candidate network selected in (a). Finally, (c) is done by selecting a real number, say u, uniformly from the unit interval (0,1) and comparing u with the Hastings ratio. Then

$$\mathbf{y}^{k+1} = \begin{cases} \mathbf{z} & \text{if } u \le \text{Hastings ratio;} \\ \mathbf{y}^k & \text{if } u > \text{Hastings ratio.} \end{cases}$$

In particular, note that if the Hastings ratio is greater than one, the candidate $\mathbf{z}$ is always accepted as the value for $\mathbf{y}^{k+1}$.

In **ergm**, there are different possible choices of the probability distribution $q(\mathbf{z},\mathbf{y}^k)$ used to select $\mathbf{z}$, but the default choice limits the possible choices of $\mathbf{z}$ to those that differ from $\mathbf{y}^k$ by exactly one edge indicator; in other words, $\mathbf{z}$ involves a single edge toggle of $\mathbf{y}^k$. Though more complicated possibilities for z are possible using the **ergm** package, suppose here that $\mathbf{z}$ is identical to $\mathbf{y}^k$ in all except the (i,j) entry. Notationally, we would write $z_{ij} = 1 - y^k_{ij}$ but $\mathbf{z}^c_{ij} = (y^k_{ij})^c$, where $\mathbf{z}^c_{ij}$ denotes the entire network $\mathbf{z}$ *except for* the (i,j) entry. In this case, by substituting Equation (1) into Expression (3), we see that the Hastings ratio may be simplified to

$$\frac{q(\mathbf{y}^k,\mathbf{z})}{q(\mathbf{z},\mathbf{y}^k)}\exp\{\pm\boldsymbol{\theta}_0^\top\delta(\mathbf{y}^k)_{ij}\}, \tag{4}$$

where

$$\delta(\mathbf{y})_{ij} \overset{\text{def}}{=} \mathbf{g}(\mathbf{y}^+_{ij}) - \mathbf{g}(\mathbf{y}^-_{ij}) \tag{5}$$

denotes the *vector of change statistics*, found by subtracting the two vectors of $\mathbf{g}$-statistics evaluated at the networks formed by leaving all of $\mathbf{y}$ unchanged except for the (i,j) entry, which is set to 1 in $\mathbf{y}^+_{ij}$ and 0 in $\mathbf{y}^0_{ij}$. The sign of the $\boldsymbol{\theta}_0^\top\delta(\mathbf{y}^k)_{ij}$ term in (4) depends on the value of y^k_{ij}: If $y^k_{ij}=1$, then the term gets a plus sign; otherwise it gets a minus.

The key conclusion of all of the above development is that the calculation of change statistic vectors is of vital importance to the running of both the simulation and the estimation routines in the **ergm**. While the **ergm** package itself provides a large library of possible change-statistic calculation routines (Morris, Handcock, and Hunter 2008), individual users sometimes wish to estimate or simulate from a model that includes specialized statistics not among those already coded in the **ergm** package; to do so, it is necessary for them to write code to calculate the associated change statistics for the Metropolis-Hastings algorithm. This article describes an R package called **ergm.userterms** that is designed to make this process as straightforward as possible. It also explains some of the internal workings of the **ergm** package that will help users develop their own network change statistic code.

In Section 3, we discuss the unique syntax implemented in the **ergm** package and explain why it was necessary to extend the existing formula-based syntax (as used, say, by the `lm` and `glm` functions in R) to handle models of the form found in Equation (1).

3. Syntax for a call to the `ergm` function

A traditional generalized linear model, as explained in the classic book by McCullagh and Nelder (1989), consists of a specification of the probabilistic dependence of some *response* variable, usually denoted Y, on some function of a linear combination of some other *predictor* variables, usually denoted X. The distribution of Y is generally a member of some known exponential family; In its simplest form (ignoring any dispersion parameters), we may write the density or mass function of Y, which depends on some parameter vector $\boldsymbol{\phi}$, as

$$f_Y(y) = \exp\{\mathbf{a}(y)^\top \mathbf{b}(\boldsymbol{\phi}) - c(\boldsymbol{\phi})\}.$$

Standard exponential family theory (e.g., Brown 1986) reveals that $E(Y) = (\partial/\partial\boldsymbol{\phi})c(\boldsymbol{\phi})$. In a generalized linear model, we presume that

$$E(Y) = \text{link}(\mathbf{X}^\top \boldsymbol{\beta})$$

for some "link" function. Implicit in this formulation is the fact that $\mathbf{X}$ is considered a fixed quantity, independent of Y; indeed, $\mathbf{X}$ is often termed the *independent* variable. (Even when $\mathbf{X}$ is random, one typically specifies a generalized linear model for the conditional distribution of Y given $\mathbf{X}$, so that $\mathbf{X}$ may be considered constant in this context.)

In contrast, model (1) does not in general conform to the above specifications of a generalized linear model. While it is true that the distribution of the network $\mathbf{Y}$—the "response"—is expressed in exponential family form, there is no way to specify some fixed $\mathbf{X}$ and some link function so that $E(\mathbf{Y}) = \text{link}(\mathbf{X}^\top \boldsymbol{\beta})$. Essentially, this is because an ERGM involves an inherent auto-dependence between the "response" and any traditional notion of the "predictors".

In addition, there is no closed-form expression for the likelihood function in an ERGM that can be easily evaluated. Equation (1) is generally impossible to use for the purposes of calculation due to the fact that the κ function of Equation (2) involves an enormous number of summands for even simple networks. Thus, even the estimation methods necessitated by maximum likelihood estimation for an ERGM differ from those used by a generalized linear model. In the former case, we rely on MCMC sampling to generate an approximation to the likelihood, whereas in the latter the likelihood may be evaluated, and thus maximized, directly.

Let us consider a typical call to the `glm` function in R:

```
R> glm ( response ~ predictor1 + predictor2 + predictor3, link = binomial)
```

In a general ERGM, there is no way to define `predictor1` through `predictor3`, nor is there a closed-form way to relate the expected value of the random network to the linear combination of the network statistics. Thus, there is no way to specify a link function and the standard R formula (that is, `response ~ predictors`) does not apply. Nonetheless, there are still analogies to standard generalized linear models, so the `ergm` function was designed to employ a modified version of the standard R formula. Instead of predictors, we refer to elements of the right-hand side of the formula as *terms*:

```
R> ergm ( network ~ term1 + term2 + term3)
```

Numerous examples of calls to the `ergm` function may be found in Hunter *et al.* (2008) and Goodreau *et al.* (2008), and a list of numerous possibilities for `term1` through `term3`, is given in Morris *et al.* (2008). Every term results in one or more network statistics being appended to the $\mathbf{g}(\mathbf{y})$ vector. For example, the function call

```
R> ergm ( network ~ edges + degree(c(1, 3, 4)))
```

results in a model with four network statistics: The `edges` term adds a single statistic, the number of edges in $\mathbf{y}$, whereas the `degree(c(1, 3, 4))` term adds three, the number of nodes with degree 1, with degree 3, and with degree 4.

Remark: For certain choices of the vector $\mathbf{g}(\mathbf{y})$ of graph statistics, the resulting model (1) actually implies that all of the individual y_{ij} edge indicators are jointly independent. In these cases, model (1) is actually equivalent to a logistic regression model, which *is* a generalized linear model. See Section 4.3 of Hunter *et al.* (2008) for more details on these so-called dyadic independence models.

4. Network storage in ergm

An understanding of the internal storage of a network in the **ergm** package aids the writing of code for a change statistic. We therefore describe this storage in some detail here. This storage method is quite different from that of the **network** package, which allows for much more general network-type objects than those of the **ergm** package (Butts 2008). The **ergm** storage methods, on the other hand, are built to handle only edges from one node to another so that adding, deleting, and accessing the structure of the network is as fast as possible.

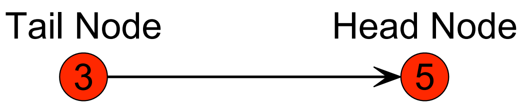

Figure 1: A directed edge, sometimes termed an arc, from node 3 to node 5.

A network with n nodes is internally represented in **ergm** as a set of $2n$ edgelists, n for "in-edges" and n for "out-edges". If the directed edge $3 \longrightarrow 5$ exists in the network, then we call 3 the "tail" node and 5 the "head" node of this edge. We would say node 3 has an *out-edge* to node 5 and that node 5 has an *in-edge* from node 3, as depicted in Figure 1. Thus, the 3rd out-edge list should contain 5 and the 5th in-edge list should contain 3. This redundant storage scheme, requiring double the memory that a single representation of the network would require, results in a corresponding gain in efficiency, since every edge may be accessed via either of its nodes.

Undirected networks are effectively stored as directed networks in **ergm** but with the "directed" flag set to zero rather than one. Essentially, this means that the adjacency matrix of an undirected network is represented as having only zeros below the main diagonal; the "tail" node is taken to be the lower-numbered node in each undirected node pair. For example, the undirected edge $2 \longleftrightarrow 4$ would actually be stored as the directed edge $2 \longrightarrow 4$. Every routine written to handle undirected networks must therefore check that the directed flag is set to zero and that every edge is referenced as tail $\longrightarrow$ head where tail $<$ head.

Storage of each node's in-edge list and out-edge list is implemented using a standard binary tree structure (Cormen, Leiserson, and Rivest 1990, Chapter 13). This structure allows for efficient lookup, insertion, and deletion operations that typically take $O(\log d)$ time, where d is the degree, or number of neighbors, a particular node has. A set of binary trees representing the network of Figure 2 is given in Figure 3. In this figure, every row of the adjacency matrix

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

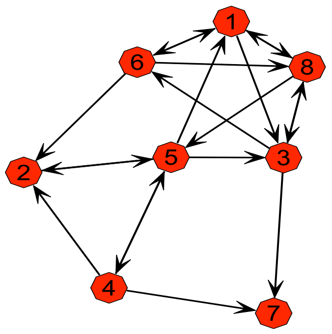

Figure 2: Adjacency matrix (left) and corresponding graphical representation of a directed 8-node network (right). The rows of the adjacency matrix define the out-edge lists of Figure 3.

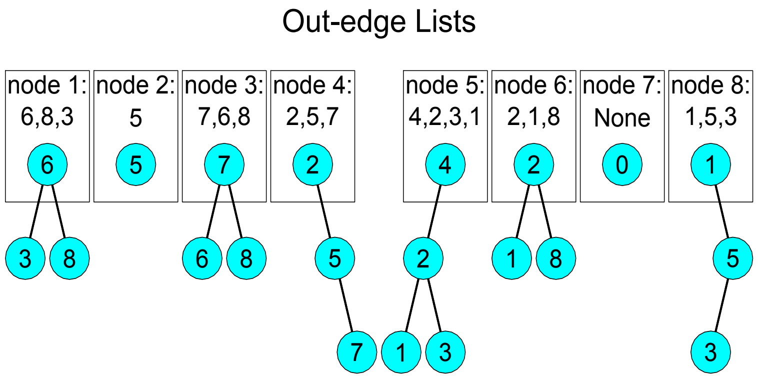

Figure 3: This is one possible **ergm** representation of the out-edges for the network of Figure 2. The network would be redundantly represented as a set of in-edge trees.

in Figure 2 is represented by its own tree, indexed by the tail node, where each numbered circular dot represents the head in a $(\text{tail}, \text{head})$ network edge. Even the root of the tree is a head node, so in case the tail has no edges, the root of the (empty) tree would have label 0. Though the depiction of the trees in Figure 3 has its own network-like appearance, this structure has nothing to do with the original network; each tree is merely an efficient storage tool for a list of node labels. In the trees, each node may have up to two "child" nodes, shown below it, one to the left and one to the right. The rule for constructing the tree, which enables the fast lookup, insertion, and deletion operations of a binary tree, is that the left child's label (if that child exists) must always be smaller than its parent's label while the right child's label must always be larger. The designation of the root node's value in each tree is completely arbitrary, as a particular list of node labels might be used to populate a tree in any order desired. To avoid the worst-case performance that can result from a list of values being passed in strictly increasing or decreasing order, each node's edgelist is randomly permuted before it is stored as a tree.

The binary tree routines, all written in the C language, are contained in the `src/edgetree.c` file in the **ergm.userterms** package. The **ergm** package includes the same code. These routines may be used to initialize or destroy a network object, to manipulate that object by adding or deleting edges, and to query that object to ascertain the presence of an edge. The C language functions `NetworkInitialize` and `NetworkDestroy` are used internally by the **ergm** package to create a network object and destroy it when it is no longer needed by the C code. These two functions are not generally called by the user, though interested users may wish to look at them. Also, the definition of the `Network` type, in the `src/edgetree.h` file, reveals that a network keeps updated lists of every node's in- and out-degree in addition to the actual edges. A user may exploit these statistics when writing code for various change statistics as described in Section 7.

5. Acquiring and setting up the necessary tools

Coding new change statistics for **ergm** requires two steps that users might not have previously encountered: building an R package from source, and writing and compiling C code. These steps requires additional tools beyond those needed by most R end users; in this section we walk through the process of acquiring and setting up such tools in Windows or UNIX-like operating systems (e.g., MacOS and Linux), with the emphasis on Windows. Those users who are already familiar with this task may wish to skip ahead to Section 6.

Windows users require a bit of preliminary setup for the building-from-source step, which we describe in the separate Section 5.1. Users of MacOS X must install the `gcc` compiler for C, which is found on the Xcode Tools CD or DVD that accompanies the MacOS system. Alternatively, the Xcode Tools may be found in the `/Applications/Installers` directory or downloaded from the web from `http://developer.apple.com/tools/xcode/`. This is explained in the "Installation of source packages" section of the R MacOS X FAQ that may be reached from `http://cran.r-project.org/faqs.html`. Users of UNIX-like systems must merely ensure that the `gcc` compiler is available, details of which may be found at `gcc.gnu.org`.

5.1. Setup for Windows

Windows users will need three tools: Perl, CLT and MinGW.[1] The easiest way to acquire all three is through the website "Building R for Windows" (currently at `http://www.murdoch-sutherland.com/Rtools`), which also provides an excellent overview of the process described in this section. Simply download the latest version of `Rtools.exe` from this website and install. While doing so, be sure to check "Edit the system PATH"; allow it to edit your path so that `c:\Program Files\Rtools\bin`, `c:\Program Files\Rtools\perl\bin`, and `c:\Program Files\Rtools\MinGW\bin` are up front. After these three entries, insert an additional entry containing the bin directory of your current R installation. This is typically `C:\Program Files\R\R-X\bin`, where `X` is your R version number.

5.2. Obtaining source code for the ergm.userterms package

When one is planning only to use an R contributed package as-is and not edit the source code, one need only install and load it using the `install.package` and `library` commands, respectively. When wishes to edit the content, one must download the source code, and then

[1] This list of necessary tools is current as of this writing. The authoritative and up-to-date source for the set of tools necessary is Appendix D ("The Windows Toolset") of the R Installation and Administration manual (`http://cran.r-project.org/doc/manuals/R-admin.html#The-Windows-toolset`)

build from there. For all operating systems, one should:

1. Go to `www.r-project.org`, click on the CRAN link in the left-hand margin, select a nearby mirror site, and finally click on the "contributed extension packages" link.
2. Select **ergm.userterms**.
3. Select the source code file (with extension `tar.gz`) next to "Package Source".
4. Save this file in `R-working-directory\src\library`. Be very careful not to place it in `R-working-directory\library`. In fact, you can technically place the file anywhere on your machine, as long as it is not in `R-working-directory\library`. In Windows, your R working directory will typically be `C:\Program Files\R\R-X`, where `X` is your R version number).
5. Untar the file by opening a DOS window in Windows (`Start>Run` and type `cmd`) or a terminal window in UNIX-like systems, navigating to the folder where you just saved the source code file using `cd`, and typing `tar xfz name.of.sourcecode.file`. A directory named `ergm.userterms` will be extracted in your current directory.

5.3. Building ergm.userterms

For Windows, in a DOS window, go to the same directory where you saved the **ergm.userterms** source file in the previous step and type:

```
RCMD INSTALL ergm.userterms
```

For UNIX-like systems, use the same syntax but split `RCMD` into two words: `R CMD`. You can now load the basic **ergm.userterms** package within R with the `library(ergm.userterms)` command.

In Sections 6 and 7 we will describe the process of editing the R and C code to make new statistics. Any time you have made changes that you want to see incorporated into the version of **ergm.userterms** that you are using in R, simply repeat the instructions here in section 5.3.

6. Writing change statistics using ergm.userterms: The R side

A typical call to the `ergm` function might look like this:

```
R> ergm ( network ~ edges + degree(1:3) + absdiff("age"))
```

We refer to `edges`, `degree`, and `absdiff` as *terms*. In the **ergm** package, there are roughly 70 terms available as of this writing. Documentation for these terms is in Morris *et al.* (2008) and is also available by typing `help("ergm-terms")`. Terms may take arguments, such as the vector `1:3` or the nodal covariate name `"age"` above. Implementing a new term that can be used by the `ergm` function involves adding two functions: One written in R and one written in C. This section describes the former, while Section 7 describes the latter.

The R function, whose purpose is to initialize the internal representation of the model term just after a call to the `ergm` function, must be called `InitErgmTerm.termname`. It should have a specified format and should perform certain tasks. We will examine these by considering the `absdiff` term from the **ergm** package:

```
 InitErgmTerm.absdiff <- function(nw, arglist, ...) {
  ### Check the network and arguments to make sure they are appropriate.
  a <- check.ErgmTerm(nw, arglist, directed=NULL, bipartite=NULL,
                      varnames = c("attrname","pow"),
                      vartypes = c("character","numeric"),
                      defaultvalues = list(NULL,1),
                      required = c(TRUE,FALSE))
  ### Process the arguments
  nodecov <- get.node.attr(nw, a$attrname)
  ### Construct the list to return
  list(name="absdiff",
       coef.names = paste(paste("absdiff", if(a$pow != 1) a$pow else "",
         sep = ""), a$attrname, sep = "."),
       pkgname = "ergm.userterms",
       inputs = c(a$pow, nodecov),
       dependence = FALSE )
}
```

The `absdiff` term has one required argument, called `attrname`, and one optional argument, called `pow`. This term will add to the model a network statistic equal to

$$\sum_{i,j} y_{ij} |X_i - X_j|^p,$$

where X_i and X_j are the values of the nodal covariate named `attrname` (and assumed already to be part of the network object specified in the `ergm` function call) and p is the value of `pow`. By default, `pow` is one.

We now examine the `InitErgmTerm.absdiff` function line by line:

Line 1: Any `InitErgmTerm` function must take three arguments: `nw`, `arglist`, and `...`. The first of these will be the network object, from which any necessary information may be extracted. The second, `arglist`, will be the list of arguments passed by the user of the term, if any. Finally, the ellipsis ("`...`") is necessary primarily for backward compatibility, as some existing InitErgmTerm arguments may be passed additional arguments, and without the ellipsis, this could generate an error message.

Line 3: The call to the `check.ErgmTerm` function should be performed by every `InitErgmTerm` function, and its result is typically given the name `a`. The first two arguments to `check.ErgmTerm` are the network and argument list; these should not be modified. However, the `directed` and `bipartite` arguments may be set to `TRUE` or `FALSE` if the term should only be applicable to the specified types of networks. (An error results if a term is not appropriate, for example, if a directed network is used in a call with a term for which `directed=FALSE`.) Leaving `directed=NULL` and `bipartite=NULL` indicates that the term may be used on either directed or undirected, either bipartite or unipartite networks. In short, the values of `FALSE`, `TRUE`, and `NULL` for the argument `directed` indicate that the term can be used on only undirected, only directed, or both types of networks, respectively. Likewise for `bipartite`.

Line 4: Each argument to a term (whether required or optional) has a name, and these names are specified by `varnames`. The `check.ErgmTerm` function will return a list in which each item is named corresponding to its varname. In the example, the list will have items named `attrname` and `pow`. In the case of a term with no arguments (such as `edges`), use `varnames=NULL`.

Line 5: In this example, the argument `attrname` is of type `character` and the argument `pow` is of type `numeric`.

Lines 6 and 7: The `attrname` argument is required; i.e., an error results if the user does not specify this argument. Therefore, the default `NULL` is irrelevant. However, since `pow` is not required, its value will be set to the default of 1 whenever the user does not supply it.

Line 9: This line extracts a vector of nodal covariate values from the network object. The name of this covariate was passed in by the user and is the character string called `attrname` in the list returned by the `check.ErgmTerm` function.

Line 11: Each `InitErgmTerm` function should return, upon exit, a list whose items are all named. Some names are required, and some are optional. A full list of these is given in the header of the `InitErgmTerm.users.R` file in the `R` subdirectory of the **ergm.userterms** package. The first named item shown here, `name`, is required. It gives the name of the C function (when "`d_`" is prepended) that will calculate the change statistic(s) for this term; see Section 7 for details. In this case, we know that the function `d_absdiff` will be responsible for this calculation.

Line 12: The `coef.names` is another required element in the output list (the only one other than `name`). It should give a vector of names for the statistic(s) that are to be added to the model by this term. The present example adds only a single statistic whose name contains both `absdiff` and the name of the nodal attribute, along with the exponent `pow` if it is not one. The length of the vector of statistic names determines how many statistics **ergm** will expect the term to add to the model.

Line 14: The `pkgname` is the name of the R package in which the C function that calculates change statistics can be found. By default, this is **ergm**, but for new terms defined in the **ergm.userterms** package the default must be overridden.

Line 15: The `inputs` vector includes any information from the network object (other than the values of the ties) that must be made available to the C function that calculates the change statistic. Because `absdiff` relies on the values of a nodal covariate, these values must be included in this case. In addition, the value of the `pow` exponent must be available. All these values are concatenated into a single numeric vector and given the name `inputs` in the list. Although the order in which they are included is arbitrary, the single values and shorter vectors are typically placed before the nodal attribute vector in order to make the indexing easier when they are retrieved within the **C**.

Line 16: The `dependence=FALSE` means that this term does not, by itself, result in an ERGM in which the dyads are dependent. If all terms in a model have `dependence=FALSE` set, then the entire model is a dyadic independence model. By default, if `dependence` is omitted then it is assumed `TRUE`.

One additional item that can go in the output list is `emptynwstats`, which is a vector of the same length as the number of statistics generated by the term being added. This vector gives the value of the network statistic measured on a network with no edges. The reason for this is that the empty network gives a point of reference for calculating global values of the statistics using C code that only calculates change statistics. By default, `emptynwstats` is a vector of zeros, so it is only necessary to include this item for cases where the empty network does not have the value zero for some of the statistics. For instance, the `isolates` term counts the number of zero-degree nodes in the network, and this statistic equals n when the network is empty; thus, the `InitErgmTerm.isolates` function ends with the following lines:

```
list(name="isolates",
     coef.names = "isolates",
     emptynwstats = network.size(nw) )
```

Other items that can go into the output list are described in the comments at the top of the `InitErgmTerm.users.R` file in the `R` subdirectory of the **ergm.userterms** package. Those not described above deal with curved exponential family models, which we do not discuss here; nonetheless, the reader is encouraged to read the comments at the top of the `InitErgmTerm.users.R` file.

Any `InitErgmTerm` function will be automatically included in the **ergm.userterms** package if it is saved in a file ending with the extension `.R` in the `R` subdirectory. For instance, a user may simply use a text editor to add new `InitErgmTerm` functions to the existing `InitErgmTerm.users.R` file; these functions will then automatically be incorporated into the **ergm.userterms** package. By analogy, the `InitErgmTerm.absdiff` function examined here is found in the `InitErgmTerm.R` file in the `R` subdirectory in the **ergm** package; indeed, that file contains a wealth of other examples that may be examined, copied, and modified by users interested in producing their own model terms.

7. Writing change statistics using ergm.userterms: The C side

As explained at the beginning of Section 6, adding a term to be used with the **ergm** package requires two different functions: An `InitErgmTerm` function written in R and a change statistic function written in C. This section discusses the second of these, which should be placed in a file with the extension `.c` in the `src` directory, just as the examples in the **ergm.userterms** package are found in the `changestats.c` file. Any `.c` file placed in the `src` subdirectory in the **ergm.userterms** package will automatically be compiled when the package is installed from source. Just be sure to include the line:

```
#include "changestat.h"
```

at the beginning of the file.

Readers familiar with writing C code may at first not recognize the example `d_absdiff` function shown below. This is because it uses numerous macros, named using all capitals in the `changestat.h` file in the `src` directory, that are designed to make writing change statistic functions easier. For instance, an author of the absdiff term need not worry about the arguments that will be required of the d_absdiff function; these are automatically included by typing CHANGESTAT_FN(d_absdiff). Here, CHANGESTAT_FN is a macro defined to create a new function whose name is supplied by the user and whose arguments agree exactly with the arguments that will automatically be passed to it by the **ergm** package.

Here, then, is the code for the d_absdiff statistic:

```
CHANGESTAT_FN(d_absdiff) {
  double change, p; Vertex t, h; int i;
  ZERO_ALL_CHANGESTATS(i);
  FOR_EACH_TOGGLE(i) {
    t = TAIL(i); h = HEAD(i);
    p = INPUT_PARAM[0];
    if(p==1.0){
      change = fabs(INPUT_PARAM[t] - INPUT_PARAM[h]);
    }else{
      change = pow(fabs(INPUT_PARAM[t] - INPUT_PARAM[h]), p);
    }
    CHANGE_STAT[0] += IS_OUTEDGE(t,h) ? -change : change;
    TOGGLE_IF_MORE_TO_COME(i); /* Needed in case of multiple toggles */
  }
  UNDO_PREVIOUS_TOGGLES(i); /* Needed on exit in case of multiple toggles */
}
```

Following the first line, various variables are declared. The Vertex type is a safe way to declare any integer variables that may be as large as the total number of vertices in the network; there is also a similar Edge type. After the variable declarations, essentially every change statistic function has the following form:

```
  ZERO_ALL_CHANGESTATS(i);
  FOR_EACH_TOGGLE(i) {
    /* body of function */
    TOGGLE_IF_MORE_TO_COME(i); /* Needed in case of multiple toggles */
  }
  UNDO_PREVIOUS_TOGGLES(i); /* Needed on exit in case of multiple toggles */
```

The body of the change statistic function has the goal of updating the CHANGE_STAT vector, which has entries CHANGE_STAT[0], ..., CHANGE_STAT[N_CHANGE_STATS-1], as appropriate for the ith toggle specified by TAIL(i), HEAD(i). It is this portion of the function that should be changed according to the requirements of the model statistic(s) specified by the term in question.

Here, we briefly describe the logic of the code, consisting completely of pre-defined macros, surrounding the body of the change statistic function. However, it is not actually necessary to understand this logic thoroughly in order to write change statistics functions; simply copy the syntax above. The Metropolis-Hastings proposal could possibly involve multiple edge toggles, and the job of the change statistic function is to update the change statistics as appropriate for the entire set of toggles. Since some of the macros, such as ZERO_ALL_CHANGESTATS and

`FOR_EACH_TOGGLE`, require the use of a variable (such as `i` in the code above), it is important that this variable be declared as type `int` prior to its first use.

Inside the `FOR_EACH_TOGGLE(i)` loop, the value of `i` counts from 0 to `ntoggles-1`, where `ntoggles` is the total number of toggles being considered. In each case, the proposed toggle—from edge to non-edge or vice-versa, depending on the current state—is to the ordered node pair (`TAIL(i), HEAD(i)`). Even when the network is considered undirected, every edge is stored as a directed edge from the "tail" node to the "head" node. After initially zeroing all of the change statistics using the `ZERO_ALL_CHANGESTATS` macro, the function should, for each edge toggle proposed, calculate how the toggle would change the network statistics of the network. This is a subtle difference here as compared to the $\delta(y)_{th}$ function of Equation (5), where the change is always calculated as the edge (t, h) changes from 0 to 1. In a change statistic function in **ergm**, however, the sign of the change will depend on whether the proposed toggle is to an existing edge or not.

We may see how this is done by examining the body of the `d_absdiff` function below. Remember that `t` and `h` are variables of type `Vertex` (which is essentially the same as type `int`), whereas `p` and `change` are of type `double`:

```
t = TAIL(i); h = HEAD(i);
p = INPUT_PARAM[0];
if(p==1.0){
  change = fabs(INPUT_PARAM[t] - INPUT_PARAM[h]);
}else{
  change = pow(fabs(INPUT_PARAM[t] - INPUT_PARAM[h]), p);
}
CHANGE_STAT[0] += IS_OUTEDGE(t,h) ? -change : change;
```

The first step in the body of our change statistic function is to read the value of the exponent, determine whether or not it equals one, and act accordingly. When the `inputs` vector was added to the output list of the `InitErgmTerm.absdiff` function, it consisted of the exponent followed by the values of some nodal attribute, one for each of the n nodes. All of these inputs may be accessed now as the elements of the `INPUT_PARAMS` vector, which has entries `INPUT_PARAMS[0]` through `INPUT_PARAMS[N_INPUT_PARAMS-1]`. Thus, the exponent may be read as `INPUT_PARAM[0]`, which is why the code above sets `p` to this value. It then calculates the absolute value of the difference of the nodal attributes for the nodes numbered `t` and `h`, for these are the nodes involved in the proposed toggle. If necessary (i.e., if `p` is not one), the absolute difference is raised to the appropriate power. The change statistic—there is only one change statistic, `CHANGE_STAT[0]`, in this case—will then be equal to plus or minus the exponentiated absolute difference, depending on the value returned by the `IS_OUTEDGE` macro. In Section 4, we explained that each node's full list of in-edges and out-edges is constantly updated. Therefore, each edge is listed in two places, so that if the edge `(t,h)` is currently in the network, both `IS_OUTEDGE(t,h)` and `IS_INEDGE(h,t)` would return one rather than zero.[2] In this case, it suffices to check whether `(t,h)` is an outedge: If it is, then the toggle would remove the edge and so we should add `-change` to `CHANGE_STAT[0]`; otherwise, we should add `change`. Since each example in this article involves a term for which

[2]This is not the same as saying that both `IS_OUTEDGE(t,h)` and `IS_INEDGE(t,h)` would return one rather than zero. This is particularly true for undirected networks, where the edge $t \longleftrightarrow h$ is always stored as $t \longrightarrow h$ where $t < h$, so `IS_INEDGE(t,h)` would *never* return one when $t < h$.

N_CHANGE_STATS is equal to one, the reader might benefit from examining the C code for a term in the **ergm** package that can involve more than one change statistic, such as degree or nodematch.

In some change statistic functions, it is necessary to look at all of the current neighbors of a particular node. The macros STEP_THROUGH_OUTEDGES and STEP_THROUGH_INEDGES can help with this. Each of these macros takes three arguments; suppose we call them (a, e, v). The first argument (which should be declared to of type Vertex) is the node whose in- or out-edges we wish to examine. The second argument (of type Edge) is the looping variable that steps through each of the neighbors of a in turn. Finally, the third variable (of type Vertex) is set by the macro to equal the number of the node corresponding to the edge. It is important to remember that for undirected networks, each edge is only stored as (t, h), where $t < h$. Therefore, to step through all of the edges of node t in an undirected network, it is necessary to step through all of its in- and out-edges by using STEP_THROUGH_OUTEDGES(t, e, v) followed by STEP_THROUGH_INEDGES(t, e, v). It is possible to observe examples of these macros in the changestats.c file in the **ergm** package.

Other macros are described briefly in the file changestat.h in the src directory. Again, many examples of the use of these macros are available in the changestats.c file.

8. Worked example: a term for "minimum degree"

Degree distributions are of interest in many social network applications, and the **ergm** package includes a variety of terms to model the propensities for nodes to display different degrees or degree distributions. The most straightforward is the degree term, which takes an argument comprising a vector of non-negative integers. The term returns a statistic for each element in the vector, representing the number of nodes whose degree equals exactly that element. The gwd term provides one parsimonious method for capturing certain features of the degree distribution; however, one might reasonably be interested in specifying other aspects, such as the number of nodes of *at least* degree n, or of *no more than* degree n. The former may be useful, for instance, in testing the hypothesis that there is some social norm against having more than one alter, but that once one has two alters, there is no added propensity for or against acquiring additional ones. In this case, we would want to know whether we observe fewer nodes with degree of 2+ in our network than we would otherwise expect, and whether this model does a good job of fitting our data. Let us define a new **ergm** term called mindegree to allow us to do this.

First, to the R side, where we must add a function called InitErgmTerm.mindegree. We can do so either in the existing InitErgmTerm.users.R file, or in another file with extension .R that we create in the R directory of the **ergm.userterms** source code. To begin, we simply define the function and its arguments, remembering the standard set of arguments that we learned in Section 6 all such functions must have:

```
InitErgmTerm.mindegree <- function(nw, arglist, ...) {
  # More lines coming....
}
```

Next, we must add the call to check.ErgmTerm, where we may specify whether we want this statistic to be usable for undirected networks, directed networks, or both, and also whether it should be usable for bipartite networks. The existing **ergm** term `degree` is limited to undirected networks, with `idegree` and `odegree` used to capture the two types of degree (in-degree and out-degree, respectively) for directed networks. Thus, it seems most reasonable for us to limit mindegree to undirected networks as well; users interested in analogous statistics for directed networks might choose to code up statistics called, for example, `min.idegree` and `min.odegree`. Likewise, bipartite networks will typically require separate degree terms for each of the two modes of the network. Filling in our progress so far yields:

```
InitErgmTerm.mindegree <- function(nw, arglist, ...) {
  a <- check.ErgmTerm(nw, arglist, directed=FALSE, bipartite=FALSE,
    # More args to check.ErgmTerm coming....
  # More lines coming....
}
```

Next we must decide on the arguments for `mindegree`. Certainly we must include at least one argument, the minimum degree. The `degree` term also allows the user to specify, via the `by` argument, a nodal attribute on which ego and alter must match in order to be counted. For the sake of demonstration, we shall follow suit. The `degree` term also has a `homophily` argument, whose value is either `TRUE` or `FALSE`, and when it is `FALSE` the meaning of the `by` argument changes. We will not duplicate the `homophily` option in `mindegree`, which is to say we will fix it as `TRUE`. Note also that the degree term allows users to pass a vector of degrees, not just a single one. In this case, it seems less likely that a user would be interested in multiple mindegrees than in multiple degrees; we will leave this feature off, and require the first argument to be a single non-negative integer, rather than a vector of arbitrary length. Note that users who are interested in having multiple minimum degree statistics in a single model still have the option of including multiple mindegree terms in their formula.

This leaves us with two arguments (let's call them "mindeg" and "by"). These are of types numeric and character, respectively; required and optional, respectively; and should probably have no default values:

```
InitErgmTerm.mindegree <- function(nw, arglist, ...) {
  a <- check.ErgmTerm(nw, arglist, directed=FALSE, bipartite=FALSE,
          varnames = c("mindeg", "by"),
          vartypes = c("numeric", "character"),
          required = c(TRUE, FALSE),
          defaultvalues = list(NULL, NULL))
  # More lines coming....
}
```

Our remaining task is to generate the list to be returned by the function, including any processing of arguments required for that list. In our case, the values of some of our entries will vary depending on whether we are using the homogeneous version of the statistic (i.e. without a nodal attribute) or the attribute-specific version. One of these is the vector of attribute values, which is commonly assigned to a vector named "nodecov" (for "nodal covariate", a synonym for nodal attribute), a convention we shall follow here. If there is no attribute, then nodecov will need to be assigned some value to indicate that. Our `C` coding task will also be made easier if we pass a flag (which we will call `attrflag`) that tells the `C` whether or not to expect nodal attribute values. Looking over the various items that need to be included in the list returned by the function, we see one other that should vary between the homogeneous and attribute-specific versions: the names of the statistic and coefficient, assigned to variable `coef.names`. If we follow the example of the `degree` term, for the homogeneous version we will want these to be named `mindegree`*x*, where *x* is the degree cutoff for the statistic; with an attribute, they should be named `mindegree.attrname`*x*, where `attrname` is the name of the attribute in the `by` argument. For example, if the degree cutoff is 2, we want the terms to be named `mindegree2` and `mindegree.attrname2`, respectively. Determining the values of `attrflag`, `nodecov`, and `coef.names` thus all begin with the same condition, so we can combine them in a single `if` statement:

```
if (is.null(a$by)) {
  attrflag <- 0
  nodecov <- NULL
  coef.names <- paste("mindegree", a$mindeg, sep=""),
} else {
  attrflag <- 1
  nodecov <- get.node.attr(nw, a$by)
  coef.names <- paste("mindegree.", a$by, a$mindeg, sep=""),
}
```

For the attribute-matching case, the object nodecov now contains a vector of the values of that attribute for all the nodes. Within the **statnet** suite, nodal attributes can be either numeric or character-based. It will much easier to write R code converting a character-based attribute into a numeric than it will be to write C code that can handle either a numeric or character vector. This is because it doesn't matter what the actual values of the attribute are for individual nodes; all that affects our statistic is whether any two nodes have the same value or not. We can simply determine all of the unique values for the attribute in our network, assign each a numeric value (based on its alphabetic position, typically), and then replace the nodes' character-based attribute values with the new numeric ranks. In R, one example of code to do this is:

```
u <- sort(unique(nodecov))
nodecov <- match(nodecov,u)
```

which has the added benefit of handling an attribute that is passed in as numeric as well.

Some elements of the list to be returned by the function are straightforward (`names`, `pkgname`, `dependence`). We have already addressed `coef.names`. In the `inputs` vector, we combine everything that the C code will need to know the value of: `mindeg` as well as the `attrflag` and `nodecov` objects we made in the previous step. For an empty network, the value of the mindegree statistic depends on what our minimum degree cut-off is; if the cutoff is 0, then the statistic is equal to the number of nodes in the network; but if the cut-off is any positive integer, then the statistic is 0. Thus, our final returned list is going to be:

```
list(name = "mindegree",
  coef.names = coef.names,
  pkgname = "ergm.userterms",
  inputs = c(attrflag, a$mindeg, nodecov),
  dependence = TRUE,
  emptynwstats = (a$mindeg == 0) * network.size(nw)
)
```

Note that the dependence argument is not strictly needed, since the default is TRUE, but is included for the sake of completeness.

One additional task we may wish to add is some checks on the arguments that the user has passed. For instance, since the first argument for `mindegree` differs from that of `degree` (constrained to be a single numeric in the first case; allowed to be a vector in the latter), we may want to check that the user has followed whatever documentation we will write for the statistic, rather than assuming it is exactly parallel to `degree`:

```
  if(length(a$mindeg) > 1)
    stop("The argument mindeg to mindegree expected a vector of length ",
         "1, but received a vector of length ",length(a$mindeg))
```

Putting this all together gives us our complete R code:

```
InitErgmTerm.mindegree <- function(nw, arglist, ...) {
  a <- check.ErgmTerm(nw, arglist, directed=FALSE, bipartite=FALSE,
      varnames = c("mindeg", "by"),
      vartypes = c("numeric", "character"),
      required = c(TRUE, FALSE),
      defaultvalues = list(NULL, NULL))
  if(length(a$mindeg) > 1)
    stop("The argument mindeg to mindegree expected a vector of length ",
         "1, but received a vector of length ",length(a$mindeg))
  if (is.null(a$by)) {
    attrflag <- 0
    nodecov <- NULL
    coef.names <- paste("mindegree", a$mindeg, sep="")
  } else {
    attrflag <- 1
    nodecov <- get.node.attr(nw, a$by)
    coef.names <- paste("mindegree.", a$by, a$mindeg, sep="")
  }
  u <- sort(unique(nodecov))
  nodecov <- match(nodecov,u)
  list(name = "mindegree",
      coef.names = coef.names,
      pkgname = "ergm.userterms",
      inputs = c(attrflag, a$mindeg, nodecov),
      dependence = TRUE,
      emptynwstats = (a$mindeg == 0) * network.size(nw)
  )
}
```

Let us now turn to the C. Here again, we can choose to add our code to the existing file `changestats.c` in the `src` directory of the `ergm.userterms` source code, or create a new file (with extension `.c`) to place in the same directory. If we do the latter, we must be sure to include the line `#include "changestat.h"` at the top of the file.

Following Section 7, we know that the basic structure of our C code must be:

```
CHANGESTAT_FN(d_mindegree) {
  /* declarations to go here */
  ZERO_ALL_CHANGESTATS(i);
  FOR_EACH_TOGGLE(i) {
    /* body of function */
    TOGGLE_IF_MORE_TO_COME(i); /* Needed in case of multiple toggles */
  }
  UNDO_PREVIOUS_TOGGLES(i); /* Needed on exit in case of multiple toggles */
}
```

We will need to conduct a different analysis depending on whether `attrflag` is 0 or 1, so we should first check this value. Remember that in the R code we passed the arguments to the C (via the `inputs` element) in the order: `attrflag`, `mindeg`, `nodecov`. Remember also that, whereas R indexes its vectors starting at position 1, C does so starting at 0;

```
  attrflag = INPUT_PARAM[0];
  if(attrflag==0){
    /* homogeneous version */
  }else{
    /* nodal attribute-specific version */
  }
```

The homogeneous case (`attrflag=0`) is easier, so let us take this first. If we were to toggle y_{ij}, how might it change the number of nodes in the network with at least `mindeg` ties? If $y_{ij} = 0$ (that is, a tie currently does not exist), then we will be adding a tie. In that case, if node i currently has exactly `mindeg`-1 ties, adding tie y_{ij} will cause i to have `mindeg` ties, and the statistic increases by 1; likewise for node j. If $y_{ij} = 1$ (i.e., a tie currently exists), then toggling it removes the tie; in this case, if node i currently has exactly `mindeg` ties, removing y_{ij} drops them below our threshold, and our statistic decreases by one; likewise, again, for j. Looking in the description of the macros in `changestat.h`, we see that there are macros called `IN_DEG` and `OUT_DEG` to check for the in-degree and out-degree of a node; since we are dealing with undirected networks only, which (as mentioned earlier) are stored as directed networks, we must check both. Thus:

```
CHANGESTAT_FN(d_mindegree) {
  /* declarations to go here */
  ZERO_ALL_CHANGESTATS(i);
  FOR_EACH_TOGGLE(i) {
    t = TAIL(i); h = HEAD(i);
    attrflag = INPUT_PARAM[0];
    mindeg = INPUT_PARAM[1];
```

```
      if(attrflag==0){
        tdeg = IN_DEG[t]+OUT_DEG[t];
        hdeg = IN_DEG[h]+OUT_DEG[h];
        CHANGE_STAT[0] += IS_OUTEDGE(t,h) ?
          - (tdeg==mindeg) - (hdeg==mindeg) :
          (tdeg==mindeg-1) + (hdeg==mindeg-1);
      }else{
        /* nodal attribute-specific version */
      }
      TOGGLE_IF_MORE_TO_COME(i); /* Needed in case of multiple toggles */
    }
    UNDO_PREVIOUS_TOGGLES(i); /* Needed on exit in case of multiple toggles */
  }
```

For the version with the nodal attribute, we need to read the values of that attribute for both i and j and then for all of their alters. Since we are only considering those ties in which both the actor and their alter have the same attribute value, we must first consider whether i and j have the same value. If they do not, then toggling their tie either on or off will not change our statistic. If they do have the same attribute value, then we need to consider the number of ties each has to alters with the same attribute value. We then have the same conditions for changing the statistic, depending on whether the tie is being toggled on or toggled off. In order to determine the nodal attribute for each of the alters, we can use two macros found in `changestat.h` and described in Section 7: `STEP_THROUGH_OUTEDGES` and `STEP_THROUGH_INEDGES`. The new code, to go inside the `else` statement, is thus:

```
        t_nodecov = INPUT_PARAM[t+1];
        h_nodecov = INPUT_PARAM[h+1];
        if (t_nodecov == h_nodecov) {
          hdeg = 0;
          STEP_THROUGH_OUTEDGES(h, e, node3) { /* step through outedges of head */
            if(INPUT_PARAM[node3+1]==h_nodecov){++hdeg;}
          }
          STEP_THROUGH_INEDGES(h, e, node3) { /* step through inedges of head */
            if(INPUT_PARAM[node3+1]==h_nodecov){++hdeg;}
          }
          tdeg = 0;
          STEP_THROUGH_OUTEDGES(t, e, node3) { /* step through outedges of tail */
            if(INPUT_PARAM[node3+1]==t_nodecov){++tdeg;}
          }
          STEP_THROUGH_INEDGES(t, e, node3) { /* step through inedges of tail */
            if(INPUT_PARAM[node3+1]==t_nodecov){++tdeg;}
          }
          CHANGE_STAT[0] += IS_OUTEDGE(t,h) ?
            - (tdeg==mindeg) - (hdeg==mindeg) :
            (tdeg==mindeg-1) + (hdeg==mindeg-1);
        }else{
          CHANGE_STAT[0] = 0;
```

```
      }
```

The +1 in each of the `INPUT_PARAM` indices is because the input vector is indexed as 0=`attrflag`; 1=`mindeg`; 2 to `popsize+1` = nodal attribute value for nodes 1 to `popsize`.

Declaring all of the variables that we ended up using, and putting it all together, yields:

```
CHANGESTAT_FN(d_mindegree) {
  Vertex t, h, node3;
  int i, mindeg, hdeg, tdeg;
  Edge e;
  int attrflag;
  double t_nodecov, h_nodecov;

  ZERO_ALL_CHANGESTATS(i);
  FOR_EACH_TOGGLE(i) {
    t = TAIL(i); h = HEAD(i);
    attrflag = INPUT_PARAM[0];
    mindeg = INPUT_PARAM[1];
    if(attrflag==0){
      hdeg = IN_DEG[h]+OUT_DEG[h];
      tdeg = IN_DEG[t]+OUT_DEG[t];
      CHANGE_STAT[0] += IS_OUTEDGE(t,h) ?
        - (tdeg==mindeg) - (hdeg==mindeg) :
        (tdeg==mindeg-1) + (hdeg==mindeg-1);
    }else{
      t_nodecov = INPUT_PARAM[t+1];
      h_nodecov = INPUT_PARAM[h+1];
      if (h_nodecov == t_nodecov) {
        hdeg = 0;
        STEP_THROUGH_OUTEDGES(h, e, node3) { /* step through outedges of head */
          if(INPUT_PARAM[node3+1]==h_nodecov){++hdeg;}
        }
        STEP_THROUGH_INEDGES(h, e, node3) { /* step through inedges of head */
          if(INPUT_PARAM[node3+1]==h_nodecov){++hdeg;}
        }
        tdeg = 0;
        STEP_THROUGH_OUTEDGES(t, e, node3) { /* step through outedges of tail */
          if(INPUT_PARAM[node3+1]==t_nodecov){++tdeg;}
        }
        STEP_THROUGH_INEDGES(t, e, node3) { /* step through inedges of tail */
          if(INPUT_PARAM[node3+1]==t_nodecov){++tdeg;}
        }
        CHANGE_STAT[0] += IS_OUTEDGE(t,h) ?
          - (tdeg==mindeg) - (hdeg==mindeg) :
          (tdeg==mindeg-1) + (hdeg==mindeg-1);
      }else{
        CHANGE_STAT[0] = 0;
      }
    }
    TOGGLE_IF_MORE_TO_COME(i); /* Needed in case of multiple toggles */
  }
  UNDO_PREVIOUS_TOGGLES(i); /* Needed on exit in case of multiple toggles */
}
```

Once both the C and R files are completed, one need simply rebuild the package from source, following the instructions in Section 5.3 and the new terms should appear.

We find that one useful way to test code for new statistics is by generating a wide variety of networks for which one knows the correct values of the statistic, and then using the command:

```
summary(network ~ new.change.stat)
```

to ensure that the code is calculating the correct values.

9. Discussion

This paper describes how to extend the exponential-family random graph modeling capabilities of the **statnet** suite of packages by explaining how users may use and modify the template package **ergm.userterms** to develop custom terms that can be "plugged in" to **statnet**. These terms can then use the full capabilities of the **statnet** suite of packages. These terms will be computed at the C level and operate at native speeds. In addition, they will be able to use the parallel capabilities of **statnet** without further changes. Finally, users may make the new terms available to the **statnet** community, hence extending the analysis and modeling capabilities of **statnet**.

Additional examples abound within the `changestats.c` file in the source code; collectively these demonstrate the syntax for the other macros listed in `changestat.h` and should answer most other questions for users. When relying on existing network statistics code as examples, it is important to remember the caveat that versions of **ergm** prior to 2.3 used different methods; these are still allowed for backward compatibility, but modifying individual snippets of code from them and interacting them with newer code may cause problems.

The fact that the **ergm.userterms** package comes with a number of macros to make the coding of change statistics easier does not preclude users from writing their own macros to further streamline coding. For instance, the lines of code that currently must appear at the beginning and end of each statistic's C code could be incorporated into a macro. Macros of this type will likely appear in a future version of the `ergm.userterms` package, although their use will always remain optional.

Additional questions may be posted to the listserv for the **statnet** users' group, which readers are encouraged to join (Handcock, Hunter, Butts, Goodreau, Krivitsky, and Morris 2003b).

Acknowledgments

We would like to thank the **statnet** team, and in particular, Martina Morris, Carter Butts, Pavel Krivitsky, and Ayn Leslie-Cook.

References

Brown LD (1986). *Fundamentals of Statistical Exponential Families.* Institute of Mathematical Statistics, Hayward, Calif.

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

Cormen TH, Leiserson CE, Rivest RL (1990). *Introduction to Algorithms.* Massachusetts Institute of Technology.

Frank O, Strauss D (1986). "Markov Graphs." *Journal of the American Statistical Association*, **81**(395), 832–842.

Goodreau SM, Handcock MS, Hunter DR, Butts CT, Morris M (2008). "A **statnet** Tutorial." *Journal of Statistical Software*, **24**(9). URL http://www.jstatsoft.org/v24/i09/.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Krivitsky PN, Morris M (2003a). ***ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks.* Statnet Project http://statnetproject.org/, Seattle, WA. R package version 2.0, URL http://CRAN.R-project.org/package=ergm.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Krivitsky PN, Morris M (2003b). *Users Group for the **statnet** Package Statistical Modeling of Network Data.* Statnet Project http://statnetproject.org/, Seattle, WA. URL http://statnet.org/statnet_users_group.shtml.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2008). "**statnet**: Software Tools for the Representation, Visualization, Analysis and Simulation of Network Data." *Journal of Statistical Software*, **24**(1). URL http://www.jstatsoft.org/v24/i01/.

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

McCullagh P, Nelder J (1989). *Generalized Linear Models.* Chapman & Hall/CRC, 2nd edition.

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

R Development Core Team (2010). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, Version 2.6.1, URL http://www.R-project.org/.

Affiliation:

David R. Hunter
Department of Statistics

Pennsylvania State University
University Park, PA 16802, United States of America
E-mail: `dhunter@stat.psu.edu`
URL: `http://www.stat.psu.edu/~dhunter/`