Journal of Statistical Software

February 2008, Volume 24, Issue 6. *http://www.jstatsoft.org/*

Social Network Analysis with sna

Carter T. Butts
University of California, Irvine

Abstract

Modern social network analysis—the analysis of relational data arising from social systems—is a computationally intensive area of research. Here, we provide an overview of a software package which provides support for a range of network analytic functionality within the R statistical computing environment. General categories of currently supported functionality are described, and brief examples of package syntax and usage are shown.

Keywords: social network analysis, graphs, **sna**, **statnet**, R.

1. Introduction and overview

Far more so than many other domains of social science, modern social network analysis (SNA) is a computationally intensive affair. Techniques based on eigensolutions (e.g., eigenvector and Bonacich centrality, multidimensional scaling), combinatorial optimization (e.g., permutation search in equivalence analysis, structural distance/covariance calculation), shortest-path computation (e.g., betweenness centrality, network diameter), and Monte Carlo integration (e.g., QAP and CUG tests) are central to the practice of SNA, and, indeed, the overwhelming majority of current research in this area could not be performed without access to inexpensive computational tools.

This dependence on computation for research in social network analysis has helped to spawn a wide array of software packages to perform network analytic tasks. From generalist tools such as **UCINET** (Borgatti *et al.* 1999), **Pajek** (Batagelj and Mrvar 2007), **STRUCTURE** (Burt 1991), **StOCNET** (Huisman and van Duijn 2003) , **MultiNet** (Richards and Seary 2006), and **GRADAP** (Stokman and Van Veen 1981) to more specialized applications such as **netdraw** (Borgatti 2007), **SIENA** (Snijders 2001), and **KrackPlot** (Krackhardt *et al.* 1994) (to name a few), a variety of software solutions are available for the network analyst. While each of these packages has its own assets, there continues to be a need for network analysis software which is simultaneously:

1. General in coverage, incorporating a range of different network analytic techniques;
2. Easily extensible, to allow for the timely incorporation of new methods and/or refinements;
3. Well-integrated with general purpose statistical, computational, and visualization tools, so as to facilitate the use of network analysis in conjunction with both end-user extensions and broader social science methodology;
4. Based on an open codebase which is available for inspection (and hence emulation, correction, and improvement) by the network community;
5. Portable, to allow use by researchers on a variety of computing platforms; and
6. Freely available to network researchers, so as to encourage its use among the widest possible range of scientists, practitioners, and students.

This "wish list" of attributes would seem to be a great deal to ask of any single, standalone program; the emergence of open statistical computing platforms such as R (R Development Core Team 2007), however, has provided a feasible means of realizing such objectives. Using R (which is itself free software in the Stallmanian sense, see Stallman 2002), researchers can easily produce and share packages which supply specialized functionality, but which are interoperable with other statistical computing tools. In this vein, the **sna** package was created as a mechanism for fulfilling the above objectives within the R environment. Additional motivations for the introduction of **sna** were to encourage the migration of the social network community to open source and/or free software solutions; to facilitate the creation of a shared framework for dissemination of new methodological developments; to further the development of statistical network analysis methods by network analysts; and to ease the integration of network methods with those of "standard" statistical analysis.

1.1. Package history

sna began life as a loose collection of S routines (called "Various Useful Tools for Network Analysis in S," or **network.S.tools**), written by the author, which were disseminated locally to social network researchers in and around the research community at Carnegie Mellon University and the University of Pittsburgh. The first external use of the toolkit of which the author is aware was the `netlogit` analysis employed by Ingram and Roberts (2000). The first version of the collection to be generally disseminated (version 0.1) was released in August of 2000, with the first R package version (**sna**, version 0.3) appearing in May of 2001. Multiple releases followed over subsequent years, with the package reaching the "1.0" landmark in August of 2005. Development has been ongoing; as of the time of this writing, the package is on version 1.5.

1.2. sna and statnet

As noted above, a major goal in introducing **sna** was the creation of a foundation for ongoing development of tools within the network analysis community. The **statnet** project (Handcock *et al.* 2003) represents the latest incarnation of that objective (much as BioConductor Gentleman *et al.* 2004, serves as a site for tool development within the bioinformatics community);

in some sense, then, **statnet** is the natural "successor" to **sna**. Reflecting this relationship, **sna** is now considered to be part of the **statnet** project, and is fully interoperable with other **statnet** packages (including **network**). **sna** may still be employed as a stand-alone package, however, for users who do not require the full range of functionality provided by **statnet**.

1.3. Functionality

At present, the **sna** package includes over 125 functions for the manipulation and analysis of network data. Supported functionality includes:

- Functions to compute descriptive indices at the graph or node level. This includes centrality and centralization indices, measures of hierarchy and prestige, brokerage, density, reciprocity, transitivity, connectedness, and the like, as well as dyad, triad, path, and cycle census statistics. Stand-alone routines to facilitate the comparison of index values across graphs via conditional uniform graph (CUG) tests are included.

- Functions to compute geodesic distances, component structure and distribution, and structure statistics (in the sense of Fararo and Sunshine 1964), and to identify isolates.

- Functions for positional and role analysis, including structural equivalence and block-modeling.

- Functions for exploratory edge set comparison, in the paradigm of Butts and Carley (2005). This includes structural covariance/correlation and distance routines, as well as tools for scaling and visualization of graph sets. Network regression (Krackhardt 1988), canonical correlation analysis, and logistic network regression are also supported; QAP (Hubert 1987; Krackhardt 1987b) and CUG tests are currently implemented for all three approaches.

- Functions to generate graph-valued deviates from various stochastic processes. So-called Erdös-Rényi graphs, inhomogeneous Bernoulli graphs, and dyad census conditioned graphs are supported, as are graphs produced by Watts-Strogatz rewiring processes (Watts and Strogatz 1998) and the biased net models of Skvoretz *et al.* (2004); Rapoport (1957).

- Functions to fit network autocorrelation (also known as spatial autocorrelation, see Anselin 1988) and biased net models.

- Functions for network inference (i.e., inferring networks from multiple reports containing missing and/or error-prone data). This includes heuristic estimators such as Krackhardt's (Krackhardt 1987a) locally aggregated structure estimators and the central graph (Banks and Carley 1994), as well as model-based methods such as the Romney-Batchelder consensus model (Romney *et al.* 1986) and the error-rate models of (Butts 2003).

- Functions for visualization and manipulation of network data (in adjacency matrix form). Standard graph layout methods such as those of Fruchterman and Reingold (1991) and Kamada and Kawai (1989), general multidimensional scaling/eigenstructure methods, and "target" diagrams (Brandes *et al.* 2003) are included by default, and

custom layout routines are also supported. Functions are included to facilitate common tasks such as extracting neighborhoods and egocentric networks, symmetrization, application of functions to attribute information on neighborhoods (e.g., computing neighbors' mean attributes), dichotomization, permutation/relabeling, and the creation of interval graphs from spell data. Data import/export is supported for several basic file formats.

The above includes many of the methods of what is sometimes called "classical" social network analysis (exemplified by Wasserman and Faust (1994), whose presentation is now canonical), as well as some more recent contributions to the literature. Although the focus of the package has been on social scientific applications, many of the included tools may also be useful for analyzing networks arising from other sources.

1.4. Terminology and data representation

As a special-purpose toolkit dedicated to social network analysis, describing **sna**'s functionality requires us to refer to standard SNA concepts and methods; readers unfamiliar with network analysis may wish to consult the cited references (particularly Wasserman and Faust 1994) for additional details. Some specific terminology and notation is described below. Throughout this paper, we will be concerned with relational data consisting of a fixed set of entities (called *vertices*) and a multiset of relationships among those entities (called *edges*). Our particular focus is on *dyadic* relationships, in which edges consist of (possibly ordered) two-element multisets on the set of vertices. The elements of an edge are referred to as its *endpoints*, with the first element known as the *tail* (or sender) and the second known as the *head* (or receiver) in the ordered case. An edge whose endpoints are identical is called a *loop*. The combination of an edge set, E, with vertex set V is said to be a *graph* (denoted $G = (V, E)$). The size, or *order* of a graph is the number of elements in its vertex set (denoted $|V|$, where $|\cdot|$ is the cardinality operator). Specific types of graphs may be identified via the constraints satisfied by E. If the elements of E are unordered multisets, G is said to be an *undirected* graph; if edges are ordered multisets, by contrast, G is said to be a *directed graph* (or *digraph*). For an undirected graph, the set of vertices tied (or *adjacent*) to vertex v is called the *neighborhood* of v (denoted $N(v)$). In the directed case, we distinguish between the set of vertices sending edges to v (the *in-neighborhood* or $N^-(v)$) and the set of vertices receiving edge from v (the *out-neighborhood*, or $N^+(v)$). A graph (directed or otherwise) is *simple* if it has no loops and if there exists no edge having multiplicity greater than one. Finally, a graph's edge set may be associated with a set of variables, such that each edge carries some value. A graph of this kind is said to be *valued*, as opposed to the contrary, *unvalued* case.

It is worth noting that use of terminology varies somewhat across the social network field—a perhaps unfortunate legacy of the field's strongly interdisciplinary nature (Freeman 2004). Thus, vertices may also be called "points" or "nodes" (or, in social contexts, "actors" or "agents"). Likewise, edges may be called "lines," "ties," or (if directed) "arcs." The term "network" is often used generically to refer to any relational structure; in other cases, it may be reserved to refer to the actually existing relational structure, with "graph" being employed for that structure's formal representation. In the latter instance, "tie" is frequently used as the corresponding term for an actually existing relationship, with "edge" denoting the formal representation of that relationship. While such terminological subtleties are not required to use **sna**, an awareness of them may reduce confusion among users seeking to make use of the

literature cited within the package manual.

With rare exceptions, **sna** routines can be used with directed or undirected graphs with or without loops. Edge values and missing data (i.e., edges whose states are unknown) are supported in many applications, as well. Note, however, that many graph theoretic concepts (e.g., connectedness) admit somewhat different definitions in the directed and undirected cases—it is thus important to verify that one is using the settings which are appropriate to the data at hand. Except for functions whose behavior is undefined in the directed case, **sna**'s functions typically default to the assumption that one's data consists of one or more simple, unvalued digraphs.

Relational data can be represented in a number of ways, several of which are currently supported by the **sna** package. The most basic of these is the *adjacency matrix*; i.e., a square matrix, A, whose elements are defined such that A_{ij} is the value of the (i, j) edge (or $\{i, j\}$ edge, in the undirected case) in the corresponding graph. By convention, A_{ij} is a dichotomous indicator variable where the corresponding graph is unvalued. Such matrices may be passed as `matrix` objects, or as two-dimensional `array`s. While adjacency matrices are convenient to work with, they are inefficient for large, sparse graphs. When working with such data, the use of `network` (Butts *et al.* 2007) or sparse matrix (Koenker and Ng 2007, **SparseM**[]) objects may be preferred. **sna** accepts all three such data types interchangeably.

In many instances, one may need to perform operations on multiple graphs at once. Where such graphs are of the same order (i.e., number of vertices), they may be conveniently represented by a three-dimensional `array` whose first dimension indexes the component adjacency matrices. Alternately, it is also possible to specify multiple graphs by means of a `list`. This allows for the user to pass graph sets of varying orders, where required. Within a graph list, single adjacency matrices, adjacency arrays, `network`, and sparse matrix objects may be mixed as desired; individual graphs are unpacked sequentially in ascending list and array index order prior to computation.

Importing relational data into R

Another preliminary issue of obvious concern is the importation of relational data into R. Where such data is stored in `matrix` or `array` form, conventional R routines such as `read.table` and `scan` may be employed in the usual manner. Similarly, natively saved `network` objects may be loaded directly into memory without external representation. In addition to these methods, **sna** includes custom routines for importing relational data in **OrgStat** NOS and **GraphViz** DOT formats. Processed relational data can be saved via the above methods, or in the DL format widely used by packages such as **Pajek** and **UCINET**. (See also the **Pajek** import function in **network**.)

Beyond these network-specific approaches, **sna** also has facilities for converting spell data (i.e., data consisting of intervals in time or other quantities) into interval graphs (West 1996). The eponymously named `interval.graph` function serves in this capacity, converting an array of spell information into one or more interval graphs; spell-level categorical covariate information may also be included. In addition to simple interval graphs, `interval.graph` will compute the valued overlap graphs proposed by Butts and Pixley (2004) for use with life history data. In this case, the overlap quantities are stored as edge values in the output adjacency matrix (or matrices, if multiple spell sets were given).

2. Package highlights

Given the wide scope of the methods implemented within the **sna** package, we cannot review them all in detail. In this section, however, we attempt to summarize the functionality of **sna** within a number of domains, highlighting specific functions and applications which are likely to be of general interest. Brief examples are also provided within each section, to illustrate basic syntax and usage. Additional background and usage details are contained within the package manual, which is distributed with the package itself.

2.1. Random graph generation

sna has a range of tools for random graph generation. Chief among these is `rgraph`, a "workhorse" function for simulating deviates from both homogeneous and inhomogeneous Bernoulli graph distributions (Wasserman and Faust 1994). Given a set of tie probabilities (which may be specified by graph or by edge), it generates one or more graphs whose edge states are independent Bernoulli trials conditional on the specified parameters.[1]

In addition to `rgraph`, **sna** has several other tools for random graph generation. These currently include `rgnm` (which draws uniform graphs and digraphs conditional on edge count), `rguman` (which draws uniform digraphs conditional on expected or realized dyad census statistics), `rgws` (which draws from a Watts-Strogatz graph process Watts and Strogatz 1998), and `rgbn` (which simulates a Skvoretz-Fararo biased net process (Skvoretz *et al.* 2004)—see also Section 2.7). Also useful are tools such as `rmperm` and the `rewire` functions, which alter an input graph by random row/column, edgewise, or dyadic permutations. Functions which condition on degree distribution and the triad census are anticipated in future versions of **sna**.

Example

To provide a sense for the syntax involved (and options available) when generating random graphs in **sna**, we here provide a brief example of R code which draws graphs from a number of models. Note that the output type in each case is an adjacency matrix; although **sna** routines accept `network` and related objects as input (per Section 1.4), the package's current random graph generators produce output in adjacency matrix or array form. The range of output types may be expanded in future package versions. To begin, we first load the **sna** library and fix the random seed (for reproducibility).

```
R> library("sna")
R> set.seed(1913)
```

As noted above, `rgraph` can be used in various ways to obtain graphs (directed or otherwise) with different expected densities. For instance, three digraphs with respective expected densities 0.1, 0.9, and 0.5 can be drawn as follows:

```
R> g <- rgraph(10, 3, tprob=c(0.1, 0.9, 0.5))
R> gden(g)

[1] 0.1000000 0.8666667 0.5333333
```

[1] `rgraph` can also be employed to simulate valued graphs via a resampling procedure.

`gden`, which we shall encounter again later, is an `sna` function which returns the density of one or more input graphs; as expected, the observed densities here closely match their expectations. The `tprob` parameter, used above to set the probability of each edge on a per-graph basis, can also be used in other ways. For instance, passing a matrix of Bernoulli parameters to `tprob` will cause `rgraph` to sample from the corresponding inhomogeneous Bernoulli graph model (in which the probability of an (i, j) edge is equal to `tprob[i,j]`. For example, consider a simple model for a digraph of order 10, in which the probability of an (i, j) edge is equal to $j/10$. Such a graph can be drawn easily as follows:

```
R> g.p <- sapply((1:10) / 10, rep, 10)
R> g <- rgraph(10, tprob = g.p)
R> g

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    0    0    0    1    0    0    1    1     1
 [2,]    0    0    0    1    0    1    0    0    1     1
 [3,]    0    0    0    0    0    1    0    1    0     1
 [4,]    0    0    0    0    1    1    1    1    1     1
 [5,]    0    1    0    0    0    0    1    1    1     1
 [6,]    0    0    1    0    1    0    1    0    1     1
 [7,]    0    1    1    0    1    0    0    1    1     1
 [8,]    0    0    1    1    1    0    1    0    1     1
 [9,]    0    0    0    1    1    0    1    1    0     1
[10,]    0    0    0    0    0    0    1    1    1     0

R> apply(g, 2, mean)

[1] 0.0 0.2 0.3 0.3 0.6 0.3 0.6 0.7 0.8 0.9
```

Since `rgraph` disallows loops by default, diagonal entries are ignored in the above cases; thus, the column means here have expectation $0.9(j/10)$. The observed means are quite close to this, but obviously vary due to the underlying Bernoulli process. For random graphs with exact constraints on edge count, we must use `rgnm`. For instance, to take 5 draws from the uniform distribution on the order 10 graphs having 12 edges we would proceed as follows:

```
R> g <- rgnm(5, 10, 12)
R> apply(g, 1, sum)

[1] 12 12 12 12 12
```

As the dyadic counterpart to both `rgraph` and `rgnm`, `rguman` models digraphs whose distributions are parameterized by dyad states. As each dyad corresponds to a pair of edge variables, it can be readily classified into the three isomorphism classes of mutual (both edges present), asymmetric (one edge present), or null (no edges present). The number of dyads in each class within a graph is known as its *dyad census*, and has been used as a simple basis for modeling network structure at least since the work of Holland and Leinhardt (1970). `rguman` can be employed either to generate uniform digraphs conditional on an exact dyad census constraint,

or to draw from a multinomial graph model of independent dyads with fixed expected counts. The former case can be used to generate graphs of particular types. For instance, the trivial cases of complete, complete tournament, and null graphs can be generated by placing all dyads within the appropriate isomorphism class:

```
R> k10 <- rguman(1, 10, mut = 45, asym = 0, null = 0, method = "exact")
R> t10 <- rguman(1, 10, mut = 0, asym = 45, null = 0, method = "exact")
R> n10 <- rguman(1, 10, mut = 0, asym = 0, null = 45, method = "exact")
R> k10
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    1    1    1    1    1    1    1    1     1
 [2,]    1    0    1    1    1    1    1    1    1     1
 [3,]    1    1    0    1    1    1    1    1    1     1
 [4,]    1    1    1    0    1    1    1    1    1     1
 [5,]    1    1    1    1    0    1    1    1    1     1
 [6,]    1    1    1    1    1    0    1    1    1     1
 [7,]    1    1    1    1    1    1    0    1    1     1
 [8,]    1    1    1    1    1    1    1    0    1     1
 [9,]    1    1    1    1    1    1    1    1    0     1
[10,]    1    1    1    1    1    1    1    1    1     0
```

```
R> t10
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    0    0    0    0    0    1    0    0     0
 [2,]    1    0    1    0    1    1    0    0    0     1
 [3,]    1    0    0    1    1    0    0    1    0     0
 [4,]    1    1    0    0    0    1    0    1    0     1
 [5,]    1    0    0    1    0    1    1    1    1     0
 [6,]    1    0    1    0    0    0    1    1    1     0
 [7,]    0    1    1    1    0    0    0    1    1     0
 [8,]    1    1    0    0    0    0    0    0    1     1
 [9,]    1    1    1    1    0    0    0    0    0     0
[10,]    1    0    1    0    1    1    1    0    1     0
```

```
R> n10
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    0    0    0    0    0    0    0    0     0
 [2,]    0    0    0    0    0    0    0    0    0     0
 [3,]    0    0    0    0    0    0    0    0    0     0
 [4,]    0    0    0    0    0    0    0    0    0     0
 [5,]    0    0    0    0    0    0    0    0    0     0
 [6,]    0    0    0    0    0    0    0    0    0     0
 [7,]    0    0    0    0    0    0    0    0    0     0
 [8,]    0    0    0    0    0    0    0    0    0     0
```

```
 [9,]    0    0    0    0    0    0    0    0    0    0
[10,]    0    0    0    0    0    0    0    0    0    0
```

When not in "exact" mode, `rguman` draws dyads as independent multinomial random variables with specified type probabilities. This can be used to obtain random structures with varying degrees of bias toward or away from mutuality. Thus, to obtain a random graph in which reciprocated ties are overrepresented, one might use a model like the following:

```
R> g <- rguman(1, 100, mut = 0.15, asym = 0.05, null = 0.8)
R> mean(g[upper.tri(g)] * t(g)[upper.tri(g)])

[1] 0.1482828

R> mean(g[upper.tri(g)] != t(g)[upper.tri(g)])

[1] 0.04646465

R> mean((!g)[upper.tri(g)] * t(!g)[upper.tri(g)])

[1] 0.8052525
```

By contrast with the expectation under the above model, a Bernoulli graph with the same expected density would have a mean mutuality rate of approximately 0.03 (with asymmetric dyads outnumbering mutual dyads by a factor of approximately 9.4). Thus, the behavior of the multinomial dyad model can deviate substantially from that of the Bernoulli graph family, despite their underlying similarity.

More extensive departures from independence require alternatives to the simple independent edge/dyad paradigm. One such alternative is the Skvoretz-Fararo family of biased net processes, which are discussed in more detail in Section 2.7. As we will see, these processes are specified in terms of the conditional probability of an edge given other edges within the graph; this immediately suggests the use of a Gibbs sampler (see, e.g. (Gilks *et al.* 1996)) to draw realizations of the graph process. Such a sampler is implemented via the `rgbn` function, which uses an iterative edge updating scheme to form a Markov chain whose equilibrium distribution corresponds to the distribution of (directed) graphs resulting from the Skvoretz-Fararo process. Thinning and burn-in parameters may be specified by the user, along with model parameters (which, by default, correspond to the uniform random digraph model). Parameters may be adjusted to produce "parent" or reciprocity biases (π), "sibling" or shared partner biases (σ), and "double role" biases or parent/sibling interaction effects (ρ), as well as baseline density effects (d); parameters vary from 0 to 1, with 0 indicating no bias. The command to draw a sample of 5 order 10 networks with both reciprocity and triangle formation biases will then look something like the following:

```
R> g <- rgbn(5, 10, param = list(pi = 0.05, sigma = 0.1, rho = 0.05,
+     d = 0.15))
```

with the magnitude of the specified effects depending on the exact choice of parameters.

Finally, we note that random graphs can also be produced by modifying existing networks. For instance, the Watts and Strogatz (1998) "rewiring" process takes an input network and (with specified probability) exchanges each non-null dyad with a randomly chosen null dyad sharing exactly one endpoint with the original dyad. Such a process obviously conserves edges, e.g.:

```
R> g <- matrix(0, 10, 10)
R> g[1,] <- 1
R> g2 <- rewire.ws(g, 0.5)[1,,]
R> g2
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    1    0    1    1    1    1    0    0    0     0
 [2,]    0    0    0    0    0    0    0    0    0     1
 [3,]    0    1    0    0    0    0    0    0    0     0
 [4,]    0    0    1    0    0    0    0    0    0     0
 [5,]    0    0    0    0    0    0    0    0    0     0
 [6,]    0    0    0    0    1    0    0    0    0     0
 [7,]    0    0    0    0    0    0    0    0    0     0
 [8,]    0    0    0    0    0    0    0    0    0     0
 [9,]    0    0    0    0    0    0    0    0    0     0
[10,]    0    0    0    0    0    0    0    0    1     0
```

```
R> sum(g - g2) == 0
```

```
[1] TRUE
```

Another example of an edge-preserving random transformation is the random permutation of vertex order. `rmperm` can be employed for this purpose, as for example in the following permutation of the graph `g2` above:

```
R> g3 <- rmperm(g2)
R> all(sort(apply(g2, 2, sum)) == sort(apply(g3, 2, sum)))
```

```
[1] TRUE
```

Row/column permutation preserves the "unlabeled" structure of the input graph (i.e., it draws from the graph's isomorphism class), and plays an important role in certain test procedures for matrix comparison (Hubert 1987; Krackhardt 1987b).

2.2. Visualization and data manipulation

Visualization and manipulation of relational data is a central task of relational analysis, and **sna** has a number of functions which are intended to facilitate this process. Some of these functions are quite basic: for instance, `diag.remove`, `lower.tri.remove`, and `upper.tri.remove`

extend the assignment behavior of R's `diag`, `lower.tri`, and `upper.tri` functions to arrays; `gvectorize` and `sr2css`, convert network data from one form to another; `symmetrize`, `make.stochastic`, and `event2dichot` perform basic data-normalizing operations on graphs or graph sets; `add.isolates` adds isolates to one or more input graphs; `stackcount` determines the number of graphs in an input stack, etc. Several other functions bear further explanation. For instance, `eval.edgeperturbation` is a wrapper function which computes the difference in the value of a graph statistic resulting from forcing the selected edge or edges to be present, versus forcing them to be absent (holding all other edges constant). Such differences are used extensively in computation for simulation and inference from exponential random graph processes (see, e.g., Snijders 2002), and have also been used to assess structural robustness (Dodds *et al.* 2003; Borgatti *et al.* 2006). `eval.edgeperturbation` is flexible, and can be used with any graph-level index function. Its use is straightforward, i.e.:

```
R> g <- rgraph(5)
R> eval.edgeperturbation(g, 1, 2, centralization, betweenness)

[1] 0.07291667
```

Unfortunately, the drawback to the flexibility of this routine is its inefficiency; `eval.edgeperturbation` cannot take advantage of any special properties of the change-score being calculated, and hence is inefficient for properties such as triad counts whose changes can be calculated much more quickly than the base statistic. This function is hence a useful utility for simple, exploratory applications, and does not replace the specialized (but less flexible) change-score functions used within packages such as **ergm**.

Another pair of useful, but idiosyncratic, utility functions are `rperm` and `numperm`, which produce permutation vectors with specified characteristics. (Recall that permuting a graph's adjacency matrix is equivalent to altering the "identities" of its vertices while leaving the underlying, "unlabeled" structure unchanged.) Although not graph manipulation functions per se, these routines are of importance for generating restricted permutations for use in QAP tests (Hubert 1987) and comparison of partially labeled graphs (Butts and Carley 2005). `rperm` draws a (uniform) random permutation vector such that vertices may only be exchanged if they belong to the same (user-supplied) equivalence class. `numperm` is a deterministic function, which returns the nth (unconstrained) permutation in lexical sort order; this is useful for exhaustive search through a (hopefully small) permutation set, or when sampling permutations without replacement.

In addition to the above, two families of graph manipulation functions bear discussing in more detail. These are functions to compute properties of neighborhoods, and functions for graph visualization. Here, we briefly discuss each family in turn, before proceeding to a review of **sna**'s descriptive index routines.

Neighborhood and ego net functions

The egocentric network (or "ego net") of vertex v in graph G is defined as $G[v \cup N(v)]$ (i.e., the subgraph of G induced by v and its neighborhood). `ego.extract` is a utility function which, for a given input graph (or set thereof) extracts the egocentric networks for one or more vertices. This can be a useful shortcut for computing local structural properties, or for simulating the effects of ego net sampling (see Marsden 2005). For directed graphs, it

is further possible to specify the use of incoming, outgoing, or combined neighborhoods for generating the induced subgraphs.

While `ego.extract` is useful for assessing local structural properties, it does not provide for computation on *attributes* (i.e., exogenous covariates) of vertex neighbors. This functionality is supplied by `gapply`. For each vertex in its input set, `gapply` first identifies all members of its neighborhood; neighborhoods may be in, out, or combined, and higher-order neighborhoods may be selected (as discussed below). Once each neighborhood has been identified, `gapply` applies a user-specified function to the neighbors' covariates (which may be supplied as a numeric vector). This provides a very quick and easy way to calculate properties such as the size of a given vertex's 3rd-order neighborhood, the fraction of its alters with a given characteristic, the average value of its alters on a specified covariate, etc.

In addition to the above, it is sometimes useful to be able to examine more complex neighborhood structures in their own right (e.g., as hypothetical influence matrices for network autocorrelation modeling). `neighborhood` provides for such computations, returning for a given graph the adjacency matrix whose i, j cell is an indicator for the membership of vertex j in vertex i's selected neighborhood. Specifically, the adjacency matrix associated with the 0th order neighborhood is defined as the identity matrix for order, and for orders $k > 0$ depends on the type of adjacency involved. For input graph $G = (V, E)$, let the *base relation*, R, be given by the underlying graph of G (i.e., $G \cup G^T$) if total neighborhoods are sought, the transpose of G if incoming neighborhoods are sought, or G otherwise. The *partial neighborhood* structure of order $k > 0$ on R is then defined to be the digraph on V whose edge set consists of the ordered pairs (i, j) having geodesic distance k in R. The corresponding *cumulative neighborhood* is formed by the ordered pairs having geodesic distance less than or equal to k in R. `neighborhood` computes either partial or cumulative neighborhoods of arbitrary order, and with arbitrary choice of edge direction.

To illustrate **sna**'s egocentric network tools, we begin by generating a sample network and extracting ego nets based on in, out, and combined neighborhoods. The resulting lists of ego nets are then easily subjected to other analyses, as seen below:

```
R> g <- rgraph(10, tp = 1.5 / 9)
R> g.in <- ego.extract(g, neighborhood = "in")
R> g.out <- ego.extract(g, neighborhood = "out")
R> g.comb <- ego.extract(g, neighborhood = "combined")
R> g.comb[1:3]

$`1`
     [,1] [,2] [,3] [,4]
[1,]    0    1    1    0
[2,]    1    0    0    0
[3,]    0    0    0    0
[4,]    1    0    0    0

$`2`
     [,1] [,2] [,3] [,4]
[1,]    0    1    0    0
[2,]    1    0    0    0
```

```
[3,]    1    0    0    0
[4,]    1    0    1    0

$`3`
     [,1] [,2] [,3] [,4]
[1,]    0    1    1    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    1    1    0    0

R> all(sapply(g.in, NROW) == degree(g, cmode = "indegree") + 1)

[1] TRUE

R> all(sapply(g.out, NROW) == degree(g, cmode = "outdegree") + 1)

[1] TRUE

R> all(sapply(g.comb, NROW) <= degree(g) + 1)

[1] TRUE

R> ego.size <- sapply(g.comb, NROW)
R> if(any(ego.size > 2))
+     sapply(g.comb[ego.size > 2], function(x){gden(x[-1,-1])})

         1          2          3          4          5          6          7
0.00000000 0.16666667 0.16666667 0.00000000 0.00000000 0.00000000 0.00000000
         8          9         10
0.00000000 0.08333333 0.00000000
```

Note that egocentric network density is often calculated as the density of ties among alters, i.e. neglecting ego's contribution (since ego must be tied to all alters by design). This is the form of density calculated above. In doing so, we have made use of the fact that `ego.extract` always places ego in the first row/column of each extracted adjacency matrix, thereby facilitating its removal where required. This example also makes use of `degree` and `gden` to calculate degree and graph density, respectively; these are discussed in more detail below.

Where computation on attributes of neighboring vertices is required (as opposed to the ego nets themselves), we turn to `gapply`. As the following example illustrates, `gapply` can be used to count features of vertex neighborhoods (degree being the most trivial example); other statistics (e.g., means, quantiles, etc.) can be used as well.

```
R> g <- rgraph(6)
R> all(gapply(g, 1, rep(1, 6), sum) == degree(g, cmode = "outdegree"))

[1] TRUE
```

```
R> all(gapply(g, 2, rep(1, 6), sum) == degree(g, cmode = "degree"))

[1] TRUE

R> all(gapply(g, c(1, 2), rep(1, 6), sum) == degree(symmetrize(g),
+    cmode = "freeman") / 2)

[1] TRUE

R> gapply(g, c(1, 2), 1:6, mean)

[1] 4.00 3.00 3.00 5.50 3.25 3.25

R> gapply(g, c(1, 2), 1:6, mean, distance = 2)

[1] 4.0 3.8 3.6 3.4 3.2 3.0
```

To obtain adjacency matrices for neighborhoods themselves, we employ the `neighborhood` function:

```
R> g <- rgraph(10, tp = 2/9)
R> neigh <- neighborhood(g, 9, neighborhood.type = "out", return.all = TRUE)
R> par(mfrow=c(3,3))
R> for(i in 1:9)
+    gplot(neigh[i,,],main = paste("Partial Neighborhood of Order", i))
R> neigh <- neighborhood(g, 9, neighborhood.type="out", return.all = TRUE,
+    partial = FALSE)
R> par(mfrow = c(3, 3))
R> for(i in 1:9)
+    gplot(neigh[i,,], main = paste("Cumulative Neighborhood of Order", i))
```

Typical output for the above is shown in Figures 1 (partial neighborhoods) and 2 (cumulative neighborhoods). These displays highlight the difference between partial and cumulative neighborhoods, illustrating each at all orders of depth. The rapidity with which such neighborhoods "fill out" the network is instructive of properties such as local clustering; we will revisit this issue when we discuss the `structure.statistics` function below.

Visualization

Network visualization has been a fundamental aspect of social network analysis since its inception (Freeman 2004), and this functionality is an important feature of **sna**. The primary "workhorse" routine for graph visualization within **sna** is `gplot`, which displays an input network using a two-dimensional layout. Many options are available to `gplot`, including the ability to specify characteristics such as size, color, and shape for individual vertices, edges, and edge labels. Vertex layout is controlled via a modular collection of layout functions (`gplot.layout.*`) which are called transparently by `gplot` itself. Built-in functions include the well-known algorithms of Fruchterman and Reingold (1991), Kamada and Kawai (1989),

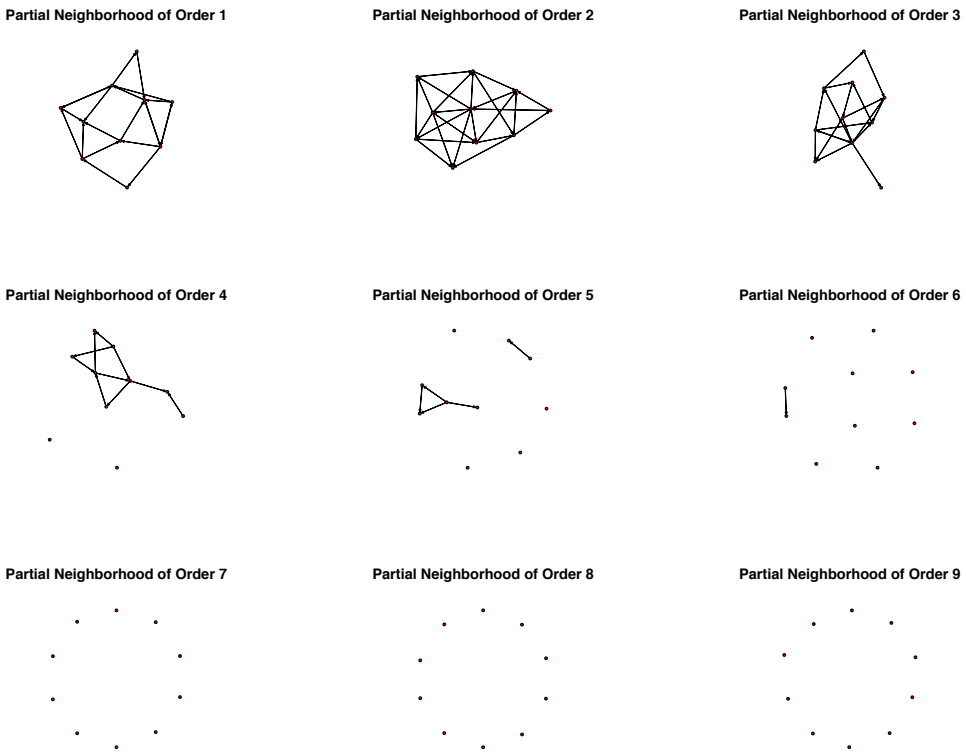

Figure 1: Sample partial neighborhoods of increasing order; vertex v is adjacent to vertex v' in the ith panel iff v' belongs to the ith order partial neighborhood of v.

and Hall (1970), as well as layouts based on general multidimensional scaling and eigenstructure procedures, circular layouts, and random placement. User-supplied functions can also be employed by creating an appropriate `gplot.layout` routine; required arguments are described in the `gplot.layout` manual page. For "target diagrams," in which graphs are plotted along concentric circles based on the magnitude of a specified covariate, `gplot.target` supplies a useful front-end to `gplot`. The layout method used in this case is that of Brandes *et al.* (2003), which may also be employed directly within `gplot`. Should no available layout suffice, coordinates may be set manually—interactive vertex placement is also supported.

While two-dimensional visualization is favored in most settings, it can also be useful to examine complex networks in three dimensions. Installing R's optional **rgl** enables `gplot3d`, which allows interactive network visualization in three dimensions. Available settings are similar to `gplot`, with layout algorithms analogously controlled by the `gplot3d.layout.*` functions. Interface and output methods are as per **rgl**, and may vary slightly by platform.

Where highly customized displays are desired, it may be useful to have access to the low-level tools used by `gplot` and `gplot3d` to display vertices and edges. `gplot.vertex`, `gplot.arrow`, `gplot.loop`, `gplot3d.arrow`, and `gplot3d.loop` can all be used directly to place `gplot`

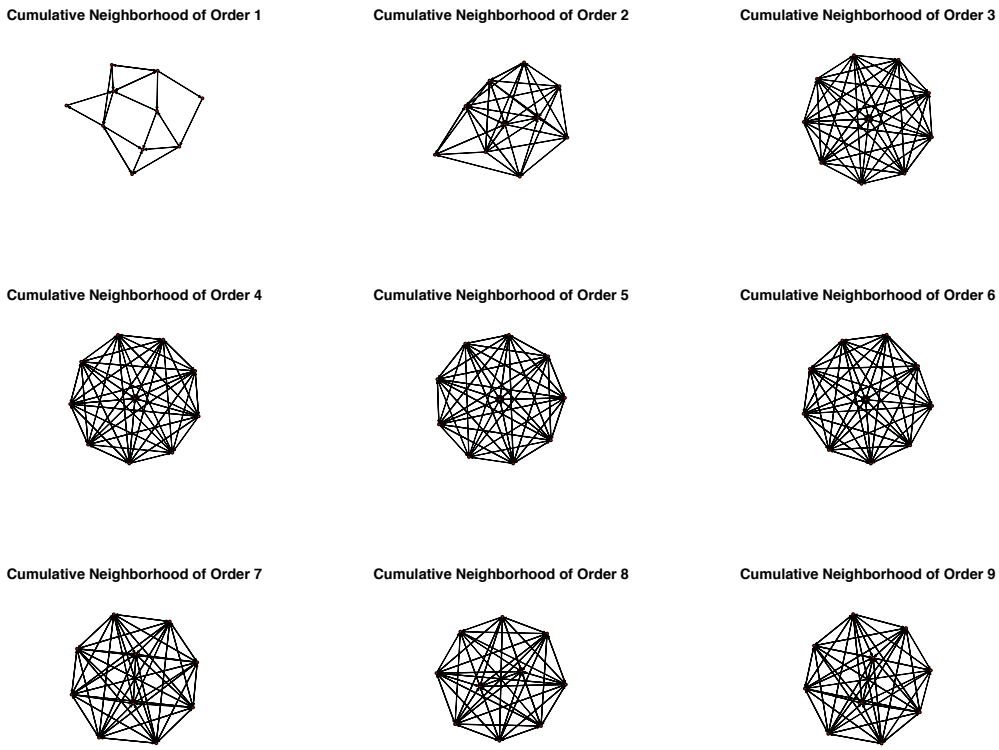

Figure 2: Sample cumulative neighborhoods of increasing order; vertex v is adjacent to vertex v' in the ith panel iff v' belongs to the ith order cumulative neighborhood of v.

elements within arbitrary displays. Options for these functions are flexible, and similar in form to those employed in the `gplot` front-end routines. It is also possible to change the behavior of the front-end visualization functions by modifying these functions, should this become necessary for more exotic applications.

All of the above functions display relational information in sociogram form, i.e., as closed shapes connected by edges. It is also possible to visualize adjacency matrices directly (i.e., as a tabular display) using the `plot.sociomatrix` function. While this is rarely useful as an exploratory tool, it can be helpful when visualizing block structure (see Section 2.5 below), or when examining matrices which are too large to display effectively using the standard `print` method.

`gplot` is a versatile routine with many options, only a few of which can be illustrated here. Curved edges, variable vertex shapes, labels, etc. are among the currently supported features. (Primitive interactive vertex placement is also supported via the `interactive` option, which can be useful in refining complex displays.) Some examples of the use of `gplot` (and `plot.sociomatrix`) are shown here:

```
R> g <- rgraph(5, diag = TRUE)
```

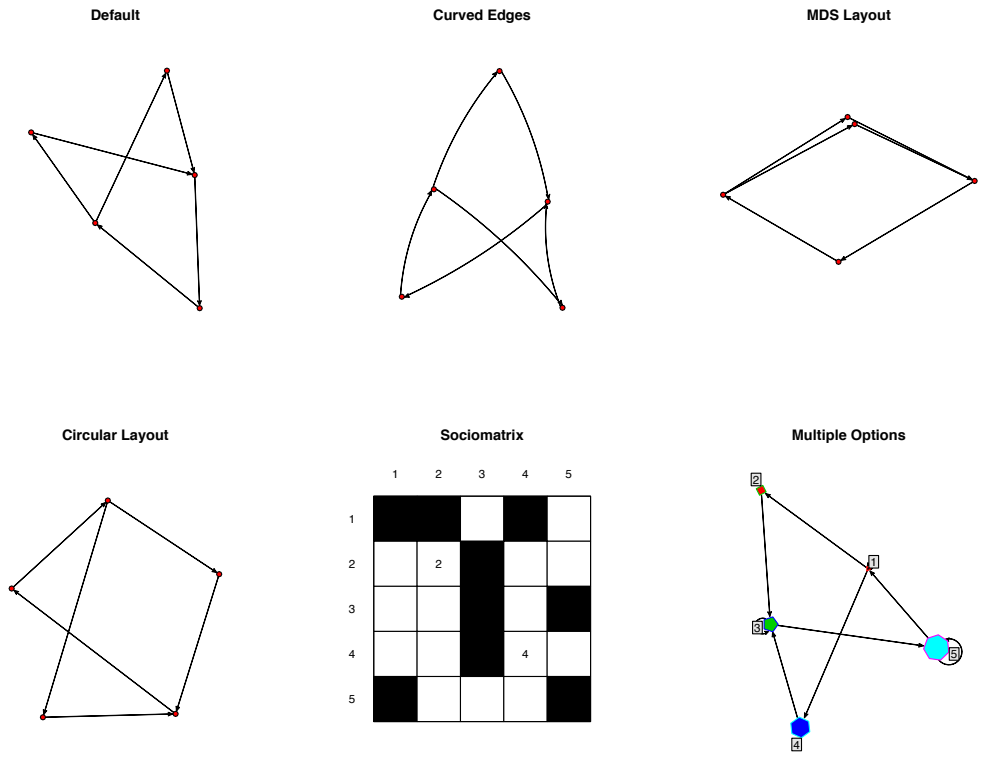

Figure 3: Sample visualizations using `gplot`, with multiple layout and display options.

```
R> par(mfrow = c(2, 3))
R> gplot(g, main = "Default")
R> gplot(g, usecurv = TRUE, main = "Curved Edges")
R> gplot(g, mode = "mds", main = "MDS Layout")
R> gplot(g, mode = "circle", main = "Circular Layout")
R> plot.sociomatrix(g, main = "Sociomatrix")
R> gplot(g, diag = TRUE, vertex.cex = 1:5, vertex.sides = 3:8,
+    vertex.col = 1:5, vertex.border = 2:6, vertex.rot = (0:4) * 72,
+    displaylabels = TRUE, label.bg = "gray90", main = "Multiple Options")
```

Output from the above is shown in Figure 3.

Three-dimensional display using `gplot3d` can be especially useful when examining networks with non-planar structure. In the following example, we see how `gplot3d` can be used to visualize the behavior of a three-dimensional Watts-Strogatz rewired lattice process. (This example requires the **rgl** package to execute.)

```
R> gplot3d(rgws(1, 5, 3, 1, 0))
R> gplot3d(rgws(1, 5, 3, 1, 0.05))
```

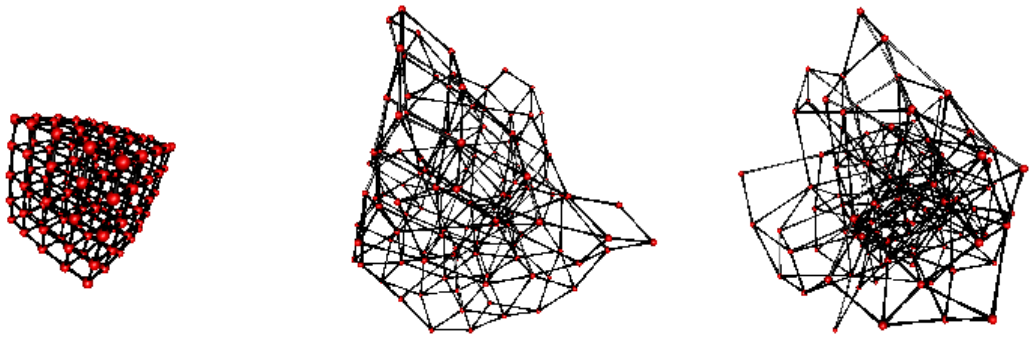

Figure 4: Three-dimensional visualizations of a Watts-Strogatz process at increasing rewiring rates.

```
R> gplot3d(rgws(1, 5, 3, 1, 0.2))
```

Snapshots of the resulting visualizations are shown in Figure 4. While not evident from the sampled output, the usual interactive features of **rgl** (e.g., rotation, zooming, etc.) are available when using `gplot3d` – this can in and of itself be useful when examining large, complex structures.

As noted, the lower-level routines used by `gplot` to produce vertices and edges can be employed directly within other displays. For instance, consider the following:

```
R> par(mfrow = c(1, 3))
R> plot(0, 0, type = "n", xlim = c(-1.5, 1.5), ylim = c(-1.5, 1.5), asp = 1,
+    xlab = "", ylab = "", main = "gplot.vertex Example")
R> gplot.vertex(cos((1:10) / 10 * 2 * pi), sin((1:10) / 10 * 2 * pi),
+    col = 1:10, sides = 3:12, radius = 0.1)
R> plot(1:2, 1:2, xlab = "", ylab = "", main = "gplot.arrow Example")
R> gplot.arrow(1, 1, 2, 2, width = 0.01, col = "red", border = "black")
R> plot(0, 0, type = "n", xlim = c(-2, 2), ylim = c(-2, 2), asp = 1,
+    xlab = "", ylab = "", main = "gplot.loop Example")
R> gplot.loop(c(0, 0), c(1, -1), col = c(3, 2), width = 0.05, length = 0.4,
+    offset = sqrt(2) / 4, angle = 20, radius = 0.5, edge.steps = 50,
+    arrowhead = TRUE)
R> polygon(c(0.25, -0.25, -0.25, 0.25, NA, 0.25, -0.25, -0.25, 0.25), c(1.25,
+    1.25, 0.75, 0.75, NA, -1.25, -1.25, -0.75, -0.75), col = c(2, 3))
```

The corresponding output, shown in Figure 5, suggests some of the flexibility of the `gplot` tools. These functions may be used to add elements to existing `gplot` output, or to create alternative display mechanisms. They may also be used within non-network contexts, as polygon-based alternatives to R's built-in `points` and `arrows` commands.

2.3. Descriptive indices

The literature of social network analysis is rich with descriptive indices of various sorts,

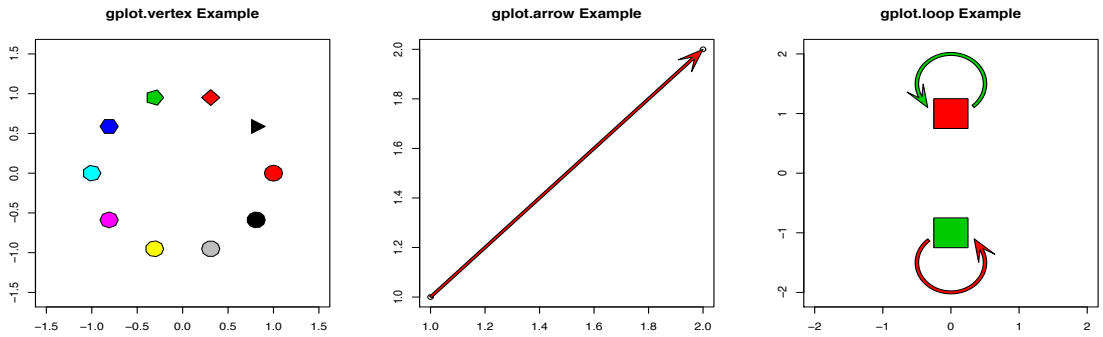

Figure 5: Examples of the use of `gplot` supplemental functions.

all of which seek to quantify particular aspects of relational structure. Broadly speaking, the most commonly used indices may be divided into two classes: *node-level indices* (NLIs), which express properties of the positions of particular vertices; and *graph-level indices* (GLIs), which express properties of entire graphs. More formally, node-level indices can be thought of as mappings of the general form $f : V \times \mathbb{G} \mapsto \mathbb{R}$, where $\mathbb{G}$ is the set of graphs on which f is defined (with associated vertex set V). Graph-level indices, by contrast, are of the form $f : \mathbb{G} \mapsto \mathbb{R}$. Although this framework is easily extended to incorporate covariates, indices of this type are uncommon; we will see an important counterexample below, however.

Node-level indices

Of the node-level indices, the most well-developed are the *centrality* indices. Formal characterization of centrality indices as a distinct class of NLIs has proved elusive (though see efforts by Sabidussi (1966) and Brandes and Erlebach (2005) chapters 3–5), but all intuitively reflect some sense in which a vertex occupies a prominent or "central" position within a graph. Among the most widely used centrality indices are those of Freeman (1979) which reflect a standardized "paring down" of a range of similar measures used in earlier work. These indices—*degree*, *betweenness*, and *closeness*—are implemented in **sna** via the eponymous `degree`, `betweenness`, and `closeness` functions. Degree, a standard graph theoretic concept, is given by $c_d(v, G) \equiv |N(v)|$ for undirected G. In the directed case, three notions of degree are generally encountered: outdegree ($c_{d^+}(v, G) \equiv |N^+(v)|$); indegree ($c_{d^-}(v, G) \equiv |N^-(v)|$); and total or "Freeman" degree ($c_{d^t}(v, G) \equiv c_{d^+}(v, G) + c_{d^-}(v, G)$). All of these are supported via `degree`. Betweenness measures the extent to which a given vertex lies on non-redundant geodesics between third parties. The index is formally defined as $c_b(v, G) \equiv \sum_{(v',v'')\subset V\setminus v} \frac{g'(v',v,v'',G)}{g(v',v'',G)}$, where $g(v, v', G)$ is the number of (v, v') geodesics in G, $g(v, v', v'', G)$ is the number of (v, v'') geodesics in G containing v', and $\frac{g'(v',v,v'',G)}{g(v',v'',G)}$ is taken equal to 0 where $g(v', v'', G) = 0$. A close variant, *stress centrality*, is identical save for the denominator of the geodesic count ratio, which is set to 1 (Shimbel 1953); this is implemented by `stresscent` in **sna**. Finally, closeness is given by $c_c(v, G) \equiv \frac{n-1}{\sum_{v'\in V} d(v,v')}$, where $d(v, v')$ is the geodesic distance from vertex v to vertex v'. Closeness is ill-defined on graphs which are not strongly connected, unless distances between disconnected vertices are taken to be infinite. In this case, $c_c(v, G) = 0$ for any v lacking a path to any vertex, and hence all

closeness scores will be 0 for graphs having multiple weak components. Due to this fragility, closeness is less often deployed than the other two of Freeman's measures.

Another important family of measures includes the *eigenvector* and *Bonacich power* centralities, both of which are based on spectral properties of the graph adjacency matrix. Eigenvector centrality (implemented in **sna** via `evcent`) is simply the absolute value of the principal eigenvector of A (where A is the graph adjacency matrix). This can be interpreted variously as a measure of "coreness" (or membership in the largest dense cluster), "recursive" or "reflected" degree (i.e., v is central to the extent to which it has many ties to other central nodes), or of the ability of v to reach other vertices through a multiplicity of short walks. Bonacich (1987) extended this notion via a measure equal to $c_{bp}(G) = \alpha(\mathbf{I} - \beta\mathbf{A})^{-1}\mathbf{A1}$, where a solution exists. This index approaches the eigenvector centrality as β approaches the reciprocal of the principal eigenvalue of $\mathbf{A}$, and degree as β approaches 0. Setting $\beta < 0$ reverses the sense of the dependence of centrality scores across vertices: where β is negative, vertices become *more* central by being attached to *less* central alters. This effect was intended to capture the behavior of equilibrium payoffs in bilateral exchange networks with credible exclusion threats; as with the positive case, parameter magnitude in this instance reflects the degree of weight afforded distant edges. The `bonpow` command in **sna** implements the Bonacich power measure, for user-specified values of β. The scaling parameter, α is by convention set so as to result in a centrality vector of length equal to $|V|$—in general, it should be remembered that this measure is uniquely defined only up to a rescaling operation. Closely related to `evcent` and `bonpow` are `prestige` (which calculates various prestige measures) and `infocent` (which calculates the *information centrality* of Stephenson and Zelen 1989). Although a range of indices is included within `prestige`, all measure the extent to which individuals secure the direct or indirect nomination of others; several variants of eigenvector centrality are included for this purpose. Information centrality provides an indication of the extent to which each individual has a large number of short walks to other actors in the network. It is similar to eigenvector centrality in being walk-based, but weights short walks more heavily (and long walks less heavily) than the former.

An example of a more specialized family of node-level indices is given by the Gould and Fernandez (1989) *brokerage scores*. The total brokerage of a given vertex, v, is defined as the number of ordered pairs (v', v'') such that $(v', v), (v, v'') \in E$, and $(v', v'') \notin E$—that is, the number of pairs for which v serves as a *local bridge*. Now, let us posit a vector of states, $\mathbf{s}$, with V such that s_i is the state of $v_i \in V$. ("State" in this case can be any exogenous covariate, although Gould and Fernandez initially intended it to be a categorical indicator of group membership.) Gould and Fernandez define five specific types of brokerage (or *brokerage roles*), based on the states of the three vertices within a locally bridged pair. For an ordered triad (v_i, v_j, v_k) with brokering vertex v_j, the possible brokerage roles are coordinating ($s_i = s_j = s_k$), itinerant ($s_i = s_k$, $s_i \neq s_j$), gatekeeping ($s_j = s_k$, $s_i \neq s_j$), representative ($s_i = s_j$, $s_j \neq s_k$), and liaison ($s_i \neq s_j$, $s_j \neq s_k$, $s_i \neq s_k$). The brokerage score for vertex v with respect to a particular role is defined as the number of ordered triads of the appropriate type for which v is a broker. The `brokerage` function computes these (and total) brokerage scores for all vertices, as well as the total amount of brokerage within each role performed throughout the network. First and second moments for brokerage scores under a null hypothesis of random association (holding fixed s and the expected density) are also provided as well as the z-tests suggested by Gould and Fernandez. It should be cautioned that the authors did not prove that the statistics in question are asymptotically normal under

the null model, and hence the statistical foundation for their associated tests is somewhat dubious; when in doubt, it may be wise to perform a simulation-based conditional uniform graph or permutation test.

To illustrate the use of node-level index routines within **sna**, we compute various centrality indices on a random digraph generated by `rgraph`. In the case of the Bonacich power measure, we also illustrate the impact of various decay parameter settings. For comparison, we begin by showing indegree, outdegree, total degree, closeness, betweenness, stress, Harary's graph centrality, eigenvector centrality, and information centrality on the same network:

```
R> dat <- rgraph(10)
R> degree(dat, cmode = "indegree")

 [1] 4 4 8 2 4 5 4 4 3 6

R> degree(dat, cmode = "outdegree")

 [1] 6 3 5 2 5 4 4 4 5 6

R> degree(dat)

 [1] 10  7 13  4  9  9  8  8  8 12

R> closeness(dat)

 [1] 0.7500000 0.5625000 0.6923077 0.5000000 0.6923077 0.6428571 0.6000000
 [8] 0.6428571 0.6923077 0.7500000

R> betweenness(dat)

 [1]  8.7666667  2.2000000 11.3500000  0.3333333  5.7833333  6.4833333
 [7]  2.4500000  2.0333333  2.4166667  8.1833333

R> stresscent(dat)

 [1] 21  6 27  1 14 15  6  7  7 21

R> graphcent(dat)

 [1] 0.5000000 0.3333333 0.5000000 0.3333333 0.5000000 0.5000000 0.3333333
 [8] 0.5000000 0.5000000 0.5000000

R> evcent(dat)

 [1] 0.3967806 0.2068905 0.3482775 0.1443617 0.3098004 0.3179091 0.2885521
 [8] 0.2734192 0.3642163 0.4121985
```

```
R> infocent(dat)

 [1] 3.712599 3.102093 3.955891 2.695898 3.712425 3.413946 3.094442 3.425508
 [9] 3.077481 3.704181
```

As the above illustrate, the various standard centrality measures differ greatly in scale; they are, however, generally positively correlated. Other measures, such as the Bonacich power score (`bonpow`) have properties which can differ substantially depending on user-specified parameters. In the case of `bonpow`, we have already noted that the score's behavior is controlled by a decay parameter (set by the `exponent` argument) which determines the nature and strength of ego's dependency upon his or her alters. Simple calculations (shown below) verify that the `bonpow` measure is proportional to outdegree when `exponent = 0` and is equivalent to eigenvector centrality when `exponent` is set to the reciprocal of the first eigenvalue of the adjacency matrix. `bonpow`'s most interesting behavior occurs when `exponent < 0`, expressing the notion that ego becomes stronger when attached to weak alters (and vice versa). As the example below illustrates, the behavior of the measure in this case is essentially unrelated to both eigenvector and degree, reflecting a very different set of assumptions regarding the underlying social process.

```
R> bonpow(dat, exponent = 0) / degree(dat, cmode = "outdegree")

 [1] 0.2192645 0.2192645 0.2192645 0.2192645 0.2192645 0.2192645 0.2192645
 [8] 0.2192645 0.2192645 0.2192645

R> all(abs(bonpow(dat, exponent = 1 / eigen(dat)$values[1], rescale = TRUE) -
+    evcent(dat, rescale = TRUE)) < 1e-10)

[1] TRUE

R> bonpow(dat, exponent = -0.5)

 [1]  1.0764391  1.2917269 -0.1230216  0.9534175  0.4613310  0.4920864
 [7]  0.4613310  0.9226621  0.3075540  2.1528782
```

As noted above `brokerage` requires a vector of group memberships (i.e., vertex states) in addition to the network itself. Here, we randomly assign vertices to one of three groups, using the resulting vector to calculate brokerage scores:

```
R> memb <- sample(1:3, 10, replace = TRUE)
R> summary(brokerage(dat, memb))

Gould-Fernandez Brokerage Analysis

Global Brokerage Properties
          t    E(t)   Sd(t)       z Pr(>|z|)
w_I   5.0000  5.8638  2.7314 -0.3162   0.7518
```

```
w_O  25.0000 19.5459  7.0713  0.7713   0.4405
b_IO 18.0000 19.5459  6.2244 -0.2484   0.8039
b_OI 17.0000 19.5459  6.2244 -0.4090   0.6825
b_O  28.0000 23.4551  5.3349  0.8519   0.3943
t    93.0000 87.9565 13.6124  0.3705   0.7110

Individual Properties (by Group)

        Group ID: 1
     w_I w_O b_IO b_OI b_O  t        w_I        w_O       b_IO       b_OI
[1,]   3   2    3    5   0 13  2.4874100  0.1931462  0.4058476  1.4190904
[2,]   0   0    1    0   0  1 -0.8042244 -1.1401201 -0.6073953 -1.1140168
[3,]   0   2    4    1   0  7 -0.8042244  0.1931462  0.9124690 -0.6073953
[4,]   0   1    1    3   0  5 -0.8042244 -0.4734869 -0.6073953  0.4058476
           b_O          t
[1,] -1.186381  0.8682544
[2,] -1.186381 -1.6099084
[3,] -1.186381 -0.3708270
[4,] -1.186381 -0.7838541

        Group ID: 2
     w_I w_O b_IO b_OI b_O  t w_I        w_O       b_IO       b_OI        b_O
[1,]   0   3    0    0   2  5 NaN 0.03375725 -0.7426778 -0.7426778 -0.7530719
[2,]   0   6    0    0  10 16 NaN 1.52052825 -0.7426778 -0.7426778  2.4025111
             t
[1,] -0.7838541
[2,]  1.4877951

        Group ID: 3
     w_I w_O b_IO b_OI b_O  t        w_I       w_O       b_IO       b_OI
[1,]   1   4    6    2   7 20  0.2929871 1.5264125  1.9257119 -0.1007739
[2,]   0   3    2    3   3 11 -0.8042244 0.8597794 -0.1007739  0.4058476
[3,]   1   2    1    2   3  9  0.2929871 0.1931462 -0.6073953 -0.1007739
[4,]   0   2    0    1   3  6 -0.8042244 0.1931462 -1.1140168 -0.6073953
           b_O           t
[1,] 3.0624213  2.31384939
[2,] 0.6345344  0.45522729
[3,] 0.6345344  0.04220016
[4,] 0.6345344 -0.57734055
```

Unlike the centrality routines described above, `brokerage` produces a range of output in addition to the raw brokerage scores. The first table consists of the observed aggregate brokerage scores by group for each of the brokerage roles (coordinator (`w_I`), itinerant broker (`w_O`), gatekeeper (`b_IO`), representative (`b_OI`), liaison (`b_O`), and combined (`t`)), along with the corresponding expectations, standard deviations, associated z-scores, and p-values under the Gould-Fernandez random association model (to which the caveats noted earlier apply). The second set of tables similarly provides the observed brokerage scores and G-F z-scores

for each individual, organized by group. It should be noted that very small groups cannot support certain brokerage roles, and (likewise) certain brokerage roles can only be realized when a sufficient number of groups are present. z-scores are considered to be undefined when their associated role preconditions are unmet, and are returned as `NaN`s.

Graph-level indices

Like node-level indices, graph-level indices are intended to provide succinct numerical summaries of structural properties; in the latter case, however, the properties in question are those pertaining to global structure. Perhaps the simplest of the GLIs is *density*, conventionally defined as the fraction of potentially observable edges which are present within the graph. Density is computed within **sna** using the `gden` function, which returns the density scores for one or more input graphs (taking into account directedness, loops, and missing data where applicable). Two more fundamental GLI classes are the *reciprocity* and *transitivity* measures, computed within **sna** by `grecip` and `gtrans`, respectively. By default, `grecip` returns the fraction of dyads which are symmetric (i.e., mutual or null) within the input graph(s). It can, however, be employed to return the fraction of non-null dyads which are symmetric, or the fraction of reciprocated edges (the "edgewise" reciprocity). All of these correspond to slightly different notions of reciprocity, and are thus appropriate in somewhat different circumstances. Likewise, `gtrans` provides several options for assessing structural transitivity. Of particular importance is the distinction between transitivity in its strong ($(i,j),(j,k) \in E \Leftrightarrow (i,k) \in E$, for $(i,j,k) \in V$) and weak ($(i,j),(j,k) \in E \Rightarrow (i,k) \in E$) forms. Intuitively, weak transitivity constitutes the notion embodied in the familiar saying that "a friend of a friend is a friend"—where a two-path exists from i to k, i should also be tied to k directly. Strong transitivity is akin to a notion of "third party support": direct ties occur if and only if supported by an associated two-path. Weak transitivity is preferred for most purposes, although strong transitivity may be of interest as more strict indicator of local clustering. By default, `gtrans` returns the fraction of possible ordered triads which satisfy the appropriate condition (out of those at risk), although absolute counts of transitive triads can also be obtained.

Another classic family of indices which can be calculated using **sna** consists of the *centralization* scores. Following Freeman (1979), the centralization of graph G with respect to centrality measure c is given by

$$C(G) = \sum_{i=1}^{|V|} \left[\left(\max_{v \in V} c\left(v, G\right) \right) - c\left(v_i, G\right) \right], \tag{1}$$

i.e. the total deviation from the maximum observed centrality score. This can be usefully rewritten as

$$C(G) = |V| \left[c^*(G) - \bar{c}(G) \right], \tag{2}$$

where $c^*(G) = \max_{v \in V} c\left(v, G\right)$ and $\bar{c}(G) = \frac{1}{|V|} \sum_{i=1}^{|V|} c\left(v_i, G\right)$ are the maximum and mean centrality scores, respectively. The Freeman centralization index is thus equal to the difference between the maximum and mean centrality scores, scaled by the number of vertices; its dimensions are those of the underlying centrality measure. In practice, it is common to work with the normalized centrality score obtained by dividing $C(G)$ by its maximum across all graphs of the same order as G. This index is dimensionless, and varies between 0 (for a graph in which all vertices have the same centrality scores[2]) and 1 (for a graph of maximum con-

[2]For instance, when all vertices are automorphically equivalent.

centration). Generally, maximum centralization scores occur on the *star* graphs (i.e., $K_{1,n}$),[3] although this is not always the case—eigenvector centralization, for instance, is maximized for the family $K_2 \cup N_n$. Within **sna**, both normalized and raw centralization scores may be obtained via the `centralization` function. Arbitrary centrality functions may be passed to `centralization`, which are used to generate the underlying score vector; in the normalized case, the centrality function is asked to return the theoretical maximum deviation, as well. This is handled transparently for all included centrality functions within **sna**; the mechanism may also be employed with user-supplied functions, provided that they supply the required arguments. Examples are supplied in the **sna** manual.

In addition to the above, **sna** includes functions for GLIs such as Krackhardt's (1994) measures of informal organization. These indices—supplied respectively by `connectedness`, `efficiency`, `hierarchy`, and `lubness`—describe the extent to which the structure of an input graph approaches that of an outtree. `hierarchy` can also be used to calculate hierarchy based on simple reciprocity, as with `grecip`.

The use of **sna**'s GLI routines is straightforward; calling with a graph or set thereof generally results in a vector of GLI scores (as in the following example). Note below the difference between the default (dyadic) and edgewise reciprocity, the standard and "census" variants of `gtrans`, and the various Krackhardt indices. `hierarchy` defaults to one minus the dyadic reciprocity (as shown), but other options are available. Similar selective behavior is employed elsewhere within **sna** (e.g., `prestige`).

```
R> g <- rgraph(10, 5, tprob = c(0.1, 0.25, 0.5, 0.75, 0.9))
R> gden(g)

[1] 0.06666667 0.31111111 0.54444444 0.72222222 0.93333333

R> grecip(g)

[1] 0.8666667 0.3777778 0.4888889 0.6666667 0.8666667

R> grecip(g, measure = "edgewise")

[1] 0.0000000 0.0000000 0.5306122 0.7692308 0.9285714

R> grecip(g) == 1 - hierarchy(g)

[1]  TRUE  TRUE TRUE  TRUE  TRUE

R> gtrans(g)

[1] 1.0000000 0.2957746 0.5047619 0.6809651 0.9326923

R> gtrans(g, measure = "weakcensus")
```

[3] K_n is the complete graph on n vertices, with $K_{n,m}$ denoting the complete bipartite graph on n and m vertices and N_n the null or empty graph on n vertices.

```
[1]   0  21 106 254 582

R> connectedness(g)

[1] 0.4666667 1.0000000 1.0000000 1.0000000 1.0000000

R> efficiency(g)

[1] 1.00000000 0.76543210 0.50617284 0.30864198 0.07407407

R> hierarchy(g, measure = "krackhardt")

[1] 1.0 0.2 0.0 0.0 0.0

R> lubness(g)

[1] 0.2 1.0 1.0 1.0 1.0
```

`centralization`'s usage differs somewhat from the above, as it acts as a wrapper for centrality routines (which must be specified, along with any additional arguments). By default, centralization scores are computed only for a single graph; R's `apply` (for arrays) or `sapply` (for lists) may be used to calculate scores for multiple graphs at once. Both forms are illustrated in the following example:

```
R> centralization(g, degree, cmode = "outdegree")

[1] 0.1728395

R> centralization(g, betweenness)

[1] 0

R> apply(g, 1, centralization, degree, cmode = "outdegree")

[1] 0.17283951 0.27160494 0.38271605 0.06172840 0.07407407

R> apply(g, 1, centralization, betweenness)

[1] 0.000000000 0.135802469 0.043467078 0.021237507 0.004151969
```

As noted above, `centralization` is compatible with any node-level index function which returns its theoretical maximum deviation when called with `tmaxdev = TRUE`. Consider, for instance, the following:

```
R> o2scent <- function(dat, tmaxdev = FALSE, ...){
+      n <- NROW(dat)
+      if(tmaxdev)
+        return((n-1) * choose(n-1, 2))
+      odeg <- degree(dat, cmode = "outdegree")
+      choose(odeg, 2)
+    }
R> apply(g, 1, centralization, o2scent)

[1] 0.02160494 0.20370370 0.54012346 0.08950617 0.14506173
```

Thus, users can employ `centralization` "for free" when working with their own centrality routines, so long as they support the required calling argument.

2.4. Connectivity and subgraph statistics

Connectivity, in its most general sense, refers to a range of properties relating to the ability of one vertex to reach another via traversal of edges. **sna** has a number of functions to compute connectivity-related statistics, and to identify associated graph features. Of these, `component.dist` is likely the most fundamental. Given one or more input graphs, `component.dist` identifies all (maximal) components, and provides associated information on membership and size distributions. Components may be selected based on standard notions of strong, weak, unilateral, or recursive connectedness (although it should be noted that unilaterally connected components may not be uniquely defined). The convenience functions `is.connected`, `components`, and `component.largest` can be used as front-ends to `component.dist`, returning (respectively) the connectedness of the graph as a whole, the number of observed components, and the largest component in the graph. The graph of pairwise connected vertices (or *reachability graph*) is returned by `reachability`, and provides another means of assessing connectivity. More precise information is contained in the geodesic distances between vertices, which can be computed (along with numbers of geodesics between pairs) by `geodist`. An example of how these concepts may be combined is provided by Fararo and Sunshine's (1964) *structure statistics*. Let $G = (V, E)$ be a (possibly directed) graph of order N, and let $d(i, j)$ be the geodesic distance from vertex i to vertex j in G. The "structure statistics" of G are then given by the series $s_0, \ldots, s_{N-1}$, where $s_i = N^{-2} \sum_{j=1}^{N} \sum_{k=1}^{N} I(d(j, k) \leq i)$ and I is the standard indicator function. Intuitively, s_i is the expected fraction of G which lies within distance i of a randomly chosen vertex. As such, the structure statistics provide a parsimonious description of global connectivity. (They are also of importance within biased net theory, since analytical results for the expectation of these statistics exist for certain models. See Fararo (1981, 1983); Skvoretz *et al.* (2004) for related results.)

At least since Davis and Leinhardt (1972), social network analysts have recognized the importance of subgraph frequencies as an indicator of underlying structural tendencies. This theory has been considerably enriched in recent decades (see, e.g., Frank and Strauss 1986; Pattison and Robins 2002), particularly with respect to the connection between edgewise dependence conditions and structural biases (see Wasserman and Robins (2005) for an approachable introduction). It has also been recognized that constraints on properties of small

subgraphs have substantial implications for global structure (see, e.g., Faust (2007) and references), a connection which also motivates the use of such measures. Most fundamental of the subgraph statistics are those of the `dyad census`, i.e., the respective counts of mutual, asymmetric, and null dyads. The eponymous `dyad.census` function returns these quantities (with `mutuality` returning only the number of mutual dyads). The *triad census*, or frequencies of each triadic isomorphism class observed as induced subgraphs of G, is similarly computed by `triad.census`. In the undirected case, there are four such classes, versus 16 for the directed case; it is thus important to specify the directedness of one's data when employing this routine (or `triad.classify`, which can be used to classify specific triads). Similar counts of paths and cycles may be obtained using `kpath.census` and `kcycle.census`. In addition to raw counts, co-membership and incidence statistics are given by vertex (where requested). Users should be aware that path and cycle census enumeration are NP-complete problems in the general case, and hence counts of longer paths or cycles are often impractical. Short (or even mid-length) cases can usually be calculated for sufficiently sparse graphs, however.

Interpretation of subgraph census statistics is often aided by comparison with baseline models (Mayhew 1984), as in the case of *conditional uniform graph (CUG) tests*. The p-value for a one-tailed CUG test of statistic t for graph G is given by $\Pr(t(H) \geq t(G))$ or $\Pr(t(H) \leq t(G))$ (for the upper and lower tests, respectively), where H is a random graph drawn uniformly given conditioning statistics $s(H) = s(G), s'(H) = s'(G), \ldots$. Conditioning on the order of G is routine; the number of edges, dyad census, and degree distribution are also widely used. A somewhat weaker family of null distributions are those which satisfy the conditions $\mathbf{E}s(H) = s(G), \mathbf{E}s'(H) = s'(G), \ldots$ for some $s, s', \ldots$. These are equivalent to the graph distributions arising from the MLE for an exponential random graph model with sufficient statistics $s, s', \ldots$—the homogeneous Bernoulli graph with parameter p equal to the density of G is a trivial example, but more complex families are possible. Within **sna**, the `cugtest` wrapper function can be used to facilitate such comparisons. Using the `gliop` routine, `cugtest` can be used to compare functions of statistics on graph pairs (e.g., difference in triangle counts) to those expected based on one or more simple null models. (Compare to `qaptest`, discussed in Section 2.6.)

Example

To illustrate the use of the above measures, we apply them to draws from a series of biased net processes. (See Section 2.7 for a discussion of the biased net model.) We begin with a low-density Bernoulli graph model, adding first reciprocity and then triad formation biases. As can be seen, varying the types of biases specified within the model alters the nature of the resulting structures, and hence their subgraph and connectivity properties.

```
R> g1 <- rgbn(50, 10, param = list(pi = 0, sigma = 0, rho = 0, d = 0.17))
R> apply(dyad.census(g1), 2, mean)

  Mut  Asym  Null
 1.00 12.84 31.16

R> apply(triad.census(g1), 2, mean)

  003   012   102  021D  021U  021C  111D  111U  030T  030C   201  120D  120U
40.16 48.48  3.50  5.52  5.80  9.60  1.94  1.86  1.84  0.72  0.12  0.08  0.08
```

```
 120C   210   300
 0.30  0.00  0.00

R> g2 <- rgbn(50, 10, param = list(pi = 0.5, sigma = 0, rho = 0, d = 0.17))
R> apply(dyad.census(g2), 2, mean)

  Mut  Asym  Null
 8.84  9.26 26.90

R> apply(triad.census(g2), 2, mean)

  003   012   102  021D  021U  021C  111D  111U  030T  030C   201  120D  120U
25.46 27.28 23.36  1.86  2.40  4.22  8.26 11.46  0.66  0.22  9.34  0.52  0.74
 120C   210   300
 1.34  2.28  0.60

R> g3 <- rgbn(50, 10, param = list(pi = 0.0, sigma = 0.25, rho = 0, d = 0.17))
R> apply(dyad.census(g3), 2, mean)

  Mut  Asym  Null
 8.94 20.44 15.62

R> apply(triad.census(g3), 2, mean)

  003   012   102  021D  021U  021C  111D  111U  030T  030C   201  120D  120U
 4.66 22.62 10.06  4.82  5.00 12.74 10.78  9.02  9.72  2.56  3.26  3.88  3.60
 120C   210   300
 8.40  7.38  1.50

R> kpath.census(g3[1,,], maxlen = 5, path.comembership = "bylength",
+    dyadic.tabulation = "bylength")$path.count

   Agg  v1  v2   v3  v4   v5   v6  v7   v8  v9 v10
1   35   8   3    9   2   10    9   3   10   8   8
2  119  40  10   47   8   59   47  13   56  39  38
3  346 155  41  180  35  223  185  52  211 149 153
4  791 457 130  504 114  601  527 163  572 425 462
5 1351 964 303 1000 282 1143 1061 375 1104 884 990

R> kcycle.census(g3[1,,], maxlen = 5,
+    cycle.comembership = "bylength")$cycle.count

  Agg v1 v2 v3 v4 v5 v6 v7 v8 v9 v10
2   9  2  1  2  0  3  2  0  4  3   1
3  24  7  1 11  0 15  9  2 12  8   7
4  42 16  1 23  2 32 26  3 30 19  16
5  72 39  5 48  8 60 54 10 57 36  43
```

```
R> component.dist(g3[1,,])

$membership
 [1] 1 1 1 1 1 1 1 1 1 1

$csize
[1] 10

$cdist
 [1] 0 0 0 0 0 0 0 0 0 1

R> structure.statistics(g3[1,,])

   0    1    2    3    4    5    6    7    8    9
0.10 0.45 0.83 0.99 1.00 1.00 1.00 1.00 1.00 1.00
```

In addition to inspecting graph statistics directly, we can also compare them using conditional uniform graph tests. Here, for example, we employ the absolute difference in reciprocities as a test statistic, first testing against a CUG hypothesis conditioning only on order and second testing against a CUG hypothesis conditioning on both order and density.

```
R> g4 <- g1[1:2,,]
R> g4[2,,] <- g2[1,,]
R> cug <- cugtest(g4, gliop, cmode = "order", GFUN = grecip, OP = "-",
+    g1 = 1, g2 = 2)
R> summary(cug)

CUG Test Results

Estimated p-values:
        p(f(rnd) >= f(d)): 0.299
        p(f(rnd) <= f(d)): 0.708

Test Diagnostics:
        Test Value (f(d)): 0.04444444
        Replications: 1000
        Distribution Summary:
                Min:     -0.3333333
                1stQ:    -0.06666667
                Med:     0
                Mean:    -0.001288889
                3rdQ:    0.06666667
                Max:     0.3555556

R> cug <- cugtest(g4, gliop, GFUN = grecip, OP = "-", g1 = 1, g2 = 2)
R> summary(cug)
```

```
CUG Test Results

Estimated p-values:
        p(f(rnd) >= f(d)): 0.967
        p(f(rnd) <= f(d)): 0.039

Test Diagnostics:
        Test Value (f(d)): 0.04444444
        Replications: 1000
        Distribution Summary:
                Min:     -0.06666667
                1stQ:    0.1555556
                Med:     0.2222222
                Mean:    0.2215333
                3rdQ:    0.2888889
                Max:     0.5333333
```

A broader range of similar Monte Carlo tests can be employed by comparing observed statistics against those arising from `rgbn`, `rguman`, or other included models.

2.5. Position and role analysis

The study of roles and positions is a strong tradition within social network analysis (see, e.g., Breiger *et al.* 1975; Burt 1976; Wasserman and Faust 1994; Doreian *et al.* 2005), and remains a popular means of reducing the complexity of large structures. Although many notions of "role" and "position" have been proposed (see Doreian *et al.* (2005) for an extensive treatment), the most widely used is without question structural equivalence. For a simple graph, G, vertex v is said to be structurally equivalent to vertex v' iff $N(v) \setminus v' = N(v') \setminus v$ (i.e., when v and v' have the same alters). In the directed case, this same general property (*mutatis mutandis*) is required to hold for both in and outneighborhoods. Structurally equivalent vertices are *copies* in a graph theoretic sense, and are necessarily identical with respect to all structural properties; graph permutations which exchange only structural equivalent vertices are necessarily automorphisms. As a true equivalence relation, structural equivalence divides a given graph into equivalence classes, which are termed *positions*. Since all vertices occupying a given position connect to other positions in precisely the same way, analyses of relations among positions (via their reduced form blockmodel—see below) can often be used in place of analyses of relations among vertices. Where non-trivial structural equivalence is present, this may result in an appreciable reduction in the size of the vertex set.

In practice, exact structural equivalence is fairly rare (isolates and pendants being two important counterexamples). Nevertheless, one may identify vertices which are approximately structurally equivalent, in that their neighborhoods are "similar" in some well-defined sense. Common means of assessing similarity between two vertices are product-moment correlations, Euclidean distances, Hamming distances, or gamma coefficients applied to their respective rows and columns within the graph adjacency matrix. Within **sna**, `sedist` computes such indices for all pairs of vertices on one or more input graphs. Once these similarities/differences are calculated, conventional multivariate data analysis procedures (e.g., hierarchical clustering or multidimensional scaling) can be used to evaluate the extent of reduction which is possible.

This process is facilitated by the function `equiv.clust`, which is essentially a joint front-end to R's built-in hierarchical clustering function (`hclust`) and various positional distance functions, though it defaults to structural equivalence in particular. Taking a set of user-specified graphs as input, `equiv.clust` computes the distances between all pairs of positions using the selected distance function, and then performs a cluster analysis of the result. The return value is an object of class `equiv.clust`, for which various secondary analysis methods exist.

After clustering, the next phase of a positional analysis is frequently blockmodeling. Given a set of equivalence classes (in the form of an `equiv.clust` or `hclust` object, or membership vector) and one or more graphs, `blockmodel` will form a blockmodel of the input graph(s) based on the classes in question, using the specified block content type. A blockmodel can be thought of as a generalized relational structure on a set of vertex classes. The relationship between the ith and jth class is said to be the i,jth *block*, whose content is referred to as its corresponding `block type`. (This terminology originates from the observation that permuting the rows and columns of an adjacency matrix by vertex class can lead to "blocks" of discernible structure in the permuted matrix. For instance, blocks among structural equivalence classes are comprised entirely of 1s or 0s, neglecting the diagonal.) Unless a vector of classes is specified, `blockmodel` forms its eponymous models by using R's `cutree` function to cut an equivalence by height or number of clusters (as specified). After forming clusters (classes), the input graphs are reordered by class and blockmodel reduction is applied. Block types currently supported include quantitative forms such as density (mean value of the cells in the associated adjacency matrix), row or column sums, cell value descriptives, and categorical types (e.g., null, 1-covered, etc.). Once a given reduction is performed, the block structure itself can be analyzed and/or expansion can be used to generate new graphs based on the image structure.

The primary use of blockmodel expansion (performed using `blockmodel.expand`) is in generating simulated draws from a hypothesized blockmodel. Expansion involves generating a new network from a block image, and thus depends on the block types from which the blockmodel is composed; at present, only density is supported. For the density block type, expansion is performed by interpreting the interclass density as an edge probability, and by drawing random graphs from the Bernoulli parameter matrix formed by expanding the density model. Thus, repeated calls to `blockmodel.expand` can be used to generate a sample for Monte Carlo null hypothesis tests under an inhomogeneous Bernoulli graph model.

Finally, we note that positional analyses have traditionally been closely associated with role algebras (White 1963; Boyd 1969; Boorman and White 1976), which seek to model empirical graph structure via the composition of multiple, simpler graphs. Although **sna**'s support for such analyses is currently limited, a composition operator, `%c%`, is available. The composition G'' of graphs G and G' on vertex set V is the graph on V such that $(v,v') \in E(G'')$ iff there exists a vertex v'' such that $(v,v'') \in G$ and $(v'',v') \in G'$. (This is equivalent to the graph formed by the boolean inner product of the graphs' respective adjacency matrices.) It should be noted that the composition of two graphs may have loops, even where the original graphs do not; thus, diagonals should not be neglected when analyzing the results of graph compositions.

Example

To demonstrate the above routines, we begin by creating an inhomogeneous Bernoulli digraph

with edge probabilities which are constant by sending vertex. (This is equivalent to drawing from a p_1 model containing only expansiveness and density effects.) We then produce an equivalence clustering and associated blockmodel, ultimately using the blockmodel to produce a new graph. As demonstrated, new graphs produced in this way need not be of the same order as the original; this is useful when simulating a hypothetical case in which individual actors may have entered or left a network without changing the underlying group structure.

```
R> g.p <- sapply(runif(20, 0, 1), rep, 20)
R> g <- rgraph(20, tprob = g.p)
R> eq <- equiv.clust(g)
R> b <- blockmodel(g, eq, h = 15)
R> g.e <- blockmodel.expand(b, rep(2, length(b$rlabels)))
R> g.e
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
 [1,]    0    0    1    1    0    0    1    0    0     1     1     1
 [2,]    0    0    1    1    0    0    1    1    0     1     1     1
 [3,]    0    0    0    0    1    1    1    1    0     0     0     0
 [4,]    0    0    1    0    1    1    1    1    0     0     0     0
 [5,]    0    0    0    0    0    0    0    0    1     1     0     0
 [6,]    0    1    1    0    0    0    1    0    1     1     0     0
 [7,]    0    0    1    1    0    1    0    1    1     1     0     1
 [8,]    0    0    1    1    0    0    1    0    0     1     0     1
 [9,]    0    0    0    1    1    1    0    1    0     0     0     0
[10,]    0    0    1    1    0    1    1    1    1     0     1     1
[11,]    0    0    0    0    0    0    1    1    0     0     0     1
[12,]    0    1    1    1    0    0    0    1    0     0     1     0
```

2.6. Exploratory edge set comparison

One important alternative to graph comparison using structural indices or subgraph statistics is direct comparison of edge sets. Within this general paradigm (see Hubert (1987); Krackhardt (1987a, 1988); Banks and Carley (1994); Butts and Carley (2005); Butts (2007) for examples), comparison is based on establishing a matching between the edges of one graph and the edges of another, leading to a measure of correspondence between the two. In the simplest case of multiple graphs on the same vertex set, the matching in question may be between those edges having the same (ordered) endpoints. One natural correspondence measure is then the Hamming distance, i.e., the number of edge changes needed to take one graph into the other. Another useful measure is Hubert's Γ, or the uncentered product-moment between the two sets of edge variables. For appropriate transformations of the original data, Γ can be interpreted as the correlation or covariance between the edge variable sets; when entire adjacency matrices are compared in this way, the result is known as the *graph correlation* or *graph covariance* (respectively). For a directed graph pair G, H, for instance, the latter is given by

$$\mathrm{cov}(G,H) = \frac{\sum_{(i,j)} \left(A^G_{ij} - \mu_G\right)\left(A^H_{ij} - \mu_H\right)}{|V|\left(|V|-1\right)} \tag{3}$$

where $\mathbf{A}^G, \mathbf{A}^H$ are the respective adjacency matrices of G and H, and $\mu_X = (|V|(|V|-1))^{-1} \sum_{(i,j)} A^X_{ij}$ is the *graph mean*. The *graph variance* is then $\text{cov}(G,G)$, and the graph correlation $\rho(G,H) = \text{cov}(G,H)/\sqrt{\text{cov}(G,G)\text{cov}(H,H)}$. Within **sna**, graph correlations and covariances can be obtained by using `gcor` and `gcov`, respectively. Hamming distances for graph sets can be similarly obtained using `hdist`.

The above situation becomes more complex when there is not a unique matching between edge sets. (Butts and Carley 2005) provide a family of generalizations for these cases, which they term *structural distances/covariances*. These measures are based on maximizing the correspondence between edge sets, under a set of permissible matchings; this results in a decomposition of the total distance/covariance into that which is attributable to fixed aspects of the structure (the structural component), versus that which depends on the (potentially variable) matching (the "labeling" component). **sna** provides tools to obtain approximate structural comparison measures, using heuristic optimization methods to seek an optimal matching. The analogs to `hdist` in this regard are `structdist` and `sdmat`, and those to `gcor` and `gcov` are `gscor` and `gscov`. For optimal matching for arbitrary bivariate statistics on graphs of identical order, the `lab.optimize` routines can also be employed. Several methods are supported, of which the default (simulated annealing) seems to be the most effective in practice.

Given a set of distances among graphs, analysis can then proceed using standard R tools for exploratory multivariate analysis such as `cmdscale` and `hclust`. Functionality specific to **sna** includes `centralgraph` (which returns the graph minimizing the Hamming distance to all graphs in the input set), `gclust.boxstats` (which shows distributions of graph statistics based on a hierarchical clustering of networks), `gclust.centralgraph` (which returns the central graphs for each element of a network clustering solution), `gdist.plotdiff` (which plots distances between networks against differences in their properties), and `gdist.plotstats` (which displays a metric MDS of networks, with star-like figures showing graph-level covariates for each structure). Similarly, network principal component analysis (Butts and Carley 2001) can be trivially implemented by the application of `eigen` to a graph covariance or correlation matrix. The ability to make use of standard tools for exploratory multivariate analysis is thus a salutary aspect of this approach.

In addition to these general tools, specific functions are available for OLS network regression (`netlm`), logistic network regression (`netlogit`), and network canonical correlation analysis (`netcancor`). These models assume multiple edge sets taken from the same set of vertices, so that there is a 1:1 mapping between edge variables across networks. In this case, the models in question are exactly analogous to their conventional (non-network) equivalents, applied to the set of vectorized adjacency matrices (as with `gvectorize`). The primary difference between the `net*` versions of these analyses and standard routines is the availability of more specialized diagnostic and testing mechanisms. Of particular note is support for various QAP (Hubert 1987) null hypotheses, which test the observed correspondence between graphs against the distribution of statistics arising from random reallocation of individuals to structural positions (i.e., permutation or relabeling). Simple QAP tests for bivariate network statistics (e.g., graph correlation) can also be performed using the stand-alone `qaptest` function. Some CUG null hypotheses are also available, where conditioning on the entire observed structure is inappropriate.

Example

We begin our demonstration of the **sna** edge set comparison routines with the simple case of graph correlation. The following illustrates the use of both simple graph correlations and structural correlations. Note that the unlabeled correlation between `g.2` and `g.3` here is 1 (since the graphs are isomorphic), but the value returned by `gscor` may sometimes be less than 1. This is because `gscor` defaults to its heuristic annealing method when seeking the structural correlation, and this method does not always identify the global maximum. Exact results can be guaranteed using exhaustive search (`method="exhaustive"`), but the computational expense of this method is prohibitive for graphs of moderate to large size; see the **sna** manual for additional options and details.

```
R> g.1 <- rgraph(5)
R> g.2 < -rgraph(5)
R> g.3 <- rmperm(g.2)
R> gcor(g.1, g.2)

[1] -0.1336306

R> gcor(g.1, g.3)

[1] 0.08908708

R> gcor(g.2, g.3)

[1] -0.4583333

R> gscor(g.1, g.2, reps = 1e5)

[1] 0.5345225

R> gscor(g.1, g.3, reps = 1e5)

[1] 0.5345225

R> gscor(g.2, g.3, reps = 1e5)

[1] 1
```

Going beyond graph correlations, `netlm` allows us to relate multiple networks in an intuitive manner:

```
R> x <- rgraph(20, 4)
R> y <- x[1,,] + 4 * x[2,,] + 2 * x[3,,]
R> nl <- netlm(y, x)
R> summary(nl)
```

```
OLS Network Model

Residuals:
           0%           25%           50%           75%          100%
-2.136676e-13 -6.547650e-16  5.123264e-16  1.345843e-15  7.075165e-14

Coefficients:
            Estimate      Pr(<=b) Pr(>=b) Pr(>=|b|)
(intercept) -1.467115e-14 0.000   1.000   0.000
x1           1.000000e+00 1.000   0.000   0.000
x2           4.000000e+00 1.000   0.000   0.000
x3           2.000000e+00 1.000   0.000   0.000
x4          -7.553990e-16 0.369   0.631   0.756

Residual standard error: 1.169e-14 on 375 degrees of freedom
Multiple R-squared: 1   Adjusted R-squared: 1
F-statistic: 3.65e+30 on 4 and 375 degrees of freedom, p-value:     0


Test Diagnostics:

        Null Hypothesis: qap
        Replications: 1000
        Coefficient Distribution Summary:

       (intercept)         x1         x2         x3         x4
Min     -2.6048970 -2.9689678 -3.5940257 -2.9888472 -1.5687343
1stQ    -0.6779707 -0.6739579 -0.6980733 -0.7469624 -0.9732831
Median  -0.0841683 -0.0090468  0.0003289 -0.0116757 -0.4346029
Mean    -0.0256936 -0.0249585 -0.0161372 -0.0055288 -0.0080178
3rdQ     0.6930508  0.6393521  0.6352920  0.7064120  0.8601390
Max      2.5434373  2.7231537  3.0464596  3.6938260  1.6294713
```

As noted earlier, OLS network regression is problematic when the dependent graph is unvalued. In this case, `netlogit` may be preferred. Its usage is directly analogous, as in the following example.

```
R> x <- rgraph(20, 4)
R> y.l <- x[1,,] + 4 * x[2,,] + 2 * x[3,,]
R> y.p <- apply(y.l, c(1, 2), function(a){1 / (1 + exp(-a))})
R> y <- rgraph(20, tprob = y.p)
R> nl <- netlogit(y, x)
R> summary(nl)

Network Logit Model

Coefficients:
```

```
            Estimate   Exp(b)     Pr(<=b) Pr(>=b) Pr(>=|b|)
(intercept)  0.3077180  1.3603173 0.680   0.320   0.503
x1           0.9411361  2.5628914 0.985   0.015   0.019
x2           4.1473292 63.2648084 1.000   0.000   0.000
x3           1.8630911  6.4436238 1.000   0.000   0.000
x4          -0.1757242  0.8388493 0.318   0.682   0.642

Goodness of Fit Statistics:

Null deviance: 526.7919 on 380 degrees of freedom
Residual deviance: 174.1572 on 375 degrees of freedom
Chi-Squared test of fit improvement:
         352.6347 on 5 degrees of freedom, p-value 0
AIC: 184.1572   BIC: 203.8580
Pseudo-R^2 Measures:
        (Dn-Dr)/(Dn-Dr+dfn): 0.481324
        (Dn-Dr)/Dn: 0.6694004
Contingency Table (predicted (rows) x actual (cols)):

      0     1
0     0     0
1    39   341

        Total Fraction Correct: 0.8973684
        Fraction Predicted 1s Correct: 0.8973684
        Fraction Predicted 0s Correct: NaN
        False Negative Rate: 0
        False Positive Rate: 1

Test Diagnostics:

        Null Hypothesis: qap
        Replications: 1000
        Distribution Summary:

       (intercept)        x1        x2        x3        x4
Min      -1.253710 -1.160806 -1.270806 -1.295749 -1.252300
1stQ     -0.215404 -0.236393 -0.229377 -0.278976 -0.250322
Median    0.078514  0.022337 -0.001591 -0.020205  0.001053
Mean      0.093105  0.025854  0.004520 -0.017570 -0.002262
3rdQ      0.408121  0.269836  0.239821  0.236166  0.252251
Max       1.704128  1.408468  1.214650  1.100783  1.533500
```

It may be noted that, in this case, the model diagnostics indicate that the model is not terribly effective at predicting the absence of ties – this is largely a consequence of the high density in the dependent graph (approximately 0.90), and is analogous to the usual challenge of predicting rare events with a logistic regression model. Nevertheless, we see that the model's

parameter estimates are quite close to the true values, and that the QAP test correctly identifies the irrelevant predictors.

2.7. Network inference and process models

A final category of functions supplied by **sna** are those implementing various network inference and process models. Although the package still contains a legacy function for fitting simple exponential random graph models via maximum pseudo-likelihood methods (`pstar`), it is strongly recommended that users employ the more modern tools of the **ergm** package for this purpose; there are several other models, however, for which **sna** provides functionality not found elsewhere in **statnet**. Perhaps foremost among these are tools for conducting *network inference,* i.e., estimation of the structure of an unknown network from noisy and/or incomplete data (Butts 2003). Several classical methods of this type are implemented by the `consensus` function, which returns the estimate of an unknown graph from a series of observed graphs. Methods supported include data analytic tools such as locally-aggregated structure (Krackhardt 1987a) and central graph (Banks and Carley 1994) estimators, as well as model-based approaches such as the consensus model of Batchelder and Romney (1988). The latter is based on the assumption that each data source has a base chance to "know" and correctly generate the true value of an edge on which they report, otherwise producing a "guess" based on a (possibly biased) Bernoulli trial. These competency and bias parameters are treated as source-level fixed effects, and the latter may be omitted if desired; estimation is by maximum likelihood. A related class of models is supported by the `bbnam` family of routines, which implements the methods of Butts (2003). The edge reporting process is in this case parameterized in terms of false positive and false negative error rates, which may be fixed at the source level, pooled, or given as known. Estimation is fully Bayesian, with error rate priors (where applicable) specified as beta distributions, and graph priors specified in inhomogeneous Bernoulli form. It should be noted that the likelihood of the reporting process assumed by the (Butts 2003) model can be reparameterized to match that of the (Batchelder and Romney 1988) model for cases in which the sum of false positive and false negative rates is less than 1; the two approaches differ primarily in their prior structure, and in the former's allowance for negatively informative reports (e.g., due to systematic deception). `bbnam` returns draws from the joint posterior distribution of the true graph and error parameters (where applicable) using a multiple-chain Gibbs sampler. The potential scale reduction measure of Gelman and Rubin (1992) (in the simplified form of Gelman *et al.* 1995) can be applied via `potscalered.mcmc` to assess convergence, and `bbnam.bf` supports basic model comparison using approximate Bayes factors. Draws from the model can be used directly, or used to construct point estimates; the helper function `npostpred` can be employed to easily obtain posterior predictive graph properties from a set of posterior draws.

Also supported by **sna** are the methods for estimating biased net parameters shown by Skvoretz *et al.* (2004). The biased net model stems from early work by Rapoport, who sought to model network structure via a hypothetical "tracing" process. This process may be described loosely as follows. One begins with a small "seed" set of vertices, each member of which is assumed to nominate (generate ties to) other members of the population with some fixed probability. These members, in turn, may nominate new members of the population, as well as members who have already been reached. Such nominations may be "biased" in one fashion or another, leading to a non-uniform growth process. Specifically, let e_{ij} be the random event that vertex i nominates vertex j when reached. Then the conditional probability

of e_{ij} is given by $\Pr(e_{ij}|T) = 1 - \big(1 - \Pr(B_e)\big) \prod_k \big(1 - \Pr(B_k)\big)^{s_k(i,j,T)}$ where T is the current state of the trace, B_e is the Bernoulli event corresponding to the baseline probability of e_{ij}, and the B_k are "bias events" (of which s_k have potentially occurred for the (i, j) directed dyad). Bias events are taken to be independent Bernoulli trials, given T, such that e_{ij} is observed with certainty if any bias event occurs. The specification of a biased net model, then, involves defining the various bias events (which, in turn, influence the structure of the network). The joint graph distribution under such a model is not in general known; as such, estimation for model parameters (bias event probabilities) is currently heuristic. `bn` currently implements the maximum pseudo-likelihood estimators of Skvoretz *et al.* (2004), as well as a method of moments estimator based on the expected triad census (also proposed by Skvoretz et al.). Heuristic goodness-of-fit statistics are provided, as well as asymptotic goodness-of-fit tests for dyad and triad statistics.

While much attention in social network analysis is directed to structural properties per se, we may also consider models for the effect of structure on individual attributes. The linear network autocorrelation models (see Doreian (1990), and Cliff and Ord (1973); Anselin (1988) for the equivalent class of spatial autocorrelation models) constitute one important family of processes which are often used for this purpose. These models are of the form

$$\mathbf{y} = \left(\sum_{i=1}^{w} \theta_i \mathbf{W}_i \right) \mathbf{y} + \mathbf{X}\beta + \epsilon, \tag{4}$$

$$\epsilon = \left(\sum_{i=1}^{z} \psi_i \mathbf{Z}_i \right) \epsilon + \nu, \tag{5}$$

where $\mathbf{y} \in \mathbb{R}^n$ is a vector of responses, $\mathbf{X} \in \mathbb{R}^{n \times x}$ is a covariate matrix, $\mathbf{W} \in \mathbb{R}^{w \times n \times n}$ and $\mathbf{Z} \in \mathbb{R}^{z \times n \times n}$ are interaction arrays, $\beta \in \mathbb{R}^x$, $\theta \in \mathbb{R}^w$, and $\psi \in \mathbb{R}^z$ are free parameters, and $\nu \sim \text{Norm}(0, \sigma^2)$ is a vector of iid disturbances. $\mathbf{Z}$ and ψ combine to form a network moving average (MA) term, which expresses the extent to which disturbances diffuse through the network. Analogously, $\mathbf{W}$ and θ describe autocorrelation structure in the responses (network AR effects). Pragmatically, the distinction between the two effect types is the latter's inclusion of impact from neighbors' covariate scores—an AR term implies that each individual's response depends on that of their neighbors (including all covariate, disturbance, and higher-order neighborhood effects), while an MA term implies that conditional dependence between responses is limited to deviations from the expectation. It is thus possible to specify AR and MA effects in isolation, as well as jointly. Within **sna**, the `lnam` function performs maximum likelihood estimation for network autocorrelation models. To aid in identifying appropriate weight matrices for use with `lnam`, **sna** also supplies a function (`nacf`) for computation of sample network autocorrelation and autocovariance functions. `nacf` can compute correlations/covariances for partial and complete in-, out-, and combined neighborhoods of various orders, as well as autocorrelation indices such as Moran's I (Moran 1950) and Geary's C (Geary 1954). Prior inspection of network autocorrelation functions can aid in proposing weight matrices for subsequent evaluation (in analogy to similar heuristics within the time series literature; see, e.g. Brockwell and Davis 1991). Functions such as `sedist` can also be used to construct matrices based on other structural properties (e.g., structural equivalence); see Leenders (2002) for a useful discussion.

Example

To demonstrate the use of **sna**'s network inference procedures, we begin by creating a fictitious data set in which we are given reports regarding the state of the network (`g`) from 20 error-prone informants. As a fairly realistic test case, we take the informants' false positive rates (`ep`) to be beta distributed with a mean of 0.038, and their false negative rates (`em`) to be likewise beta distributed with a mean of 0.375 (about ten times higher). We then subject this data to `bbnam`, employing some fairly generic priors. Specifically, we employ an uninformative network prior (specified by `pnet`), and identical beta$(2, 11)$ priors for all error rates. The summary function for the returned network describes the resulting posterior properties, along with various diagnostics.

```
R> g <- rgraph(20)
R> ep <- rbeta(20, 1, 25)
R> em <- rbeta(20, 15, 25)
R> dat <- array(dim = c(20, 20, 20))
R> for(i in 1:20)
+    dat[i,,] <- rgraph(20, 1, tprob = (g * (1 - em[i]) + (1 - g) * ep[i]))
R> pnet <- matrix(0.5, ncol = 20, nrow = 20)
R> pem <- matrix(nrow = 20, ncol = 2)
R> pem[,1] <- 2
R> pem[,2] <- 11
R> pep <- matrix(nrow = 20, ncol = 2)
R> pep[,1] <- 2
R> pep[,2] <- 11
R> b <- bbnam(dat, model = "actor", nprior = pnet, emprior = pem,
+    epprior = pep, burntime = 300, draws = 100)
R> summary(b)

Butts' Hierarchical Bayes Model for Network Estimation/Informant Accuracy

Multiple Error Probability Model

Marginal Posterior Network Distribution:

      a1   a2   a3   a4   a5   a6   a7   a8   a9  a10  a11  a12  a13  a14  a15
a1  0.00 0.00 0.00 1.00 1.00 0.00 1.00 1.00 0.00 0.00 1.00 1.00 0.00 0.00 0.00
a2  0.00 0.00 1.00 1.00 1.00 0.00 0.00 1.00 1.00 1.00 0.00 0.00 0.00 0.00 1.00
a3  0.00 1.00 0.00 1.00 1.00 1.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 1.00 1.00
a4  0.01 1.00 1.00 0.00 0.00 0.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00 0.00 1.00
a5  1.00 1.00 1.00 1.00 0.00 1.00 0.00 0.00 1.00 0.00 1.00 1.00 1.00 1.00 0.00
a6  0.00 0.00 1.00 0.00 0.00 0.00 1.00 0.00 1.00 1.00 0.18 1.00 0.00 0.00 1.00
a7  1.00 1.00 0.00 1.00 0.00 0.00 0.00 1.00 0.00 0.00 0.00 1.00 0.00 0.00 1.00
a8  0.00 1.00 1.00 1.00 1.00 1.00 0.00 0.00 1.00 0.00 0.00 1.00 0.00 1.00 0.00
a9  0.00 0.00 1.00 0.00 1.00 0.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00 1.00 1.00
a10 0.00 0.00 0.00 0.00 0.00 0.00 1.00 1.00 1.00 0.00 1.00 0.00 0.00 1.00 0.00
a11 0.00 0.00 1.00 1.00 1.00 0.00 0.00 0.00 0.00 0.00 0.00 1.00 1.00 0.00 1.00
a12 1.00 1.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00
```

```
a13 0.00 0.00 0.00 1.00 1.00 1.00 1.00 1.00 0.00 0.00 1.00 1.00 0.00 0.00 0.00
a14 1.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 0.00
a15 1.00 1.00 0.00 1.00 0.00 0.00 1.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 0.00
a16 0.00 1.00 1.00 0.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 0.00 1.00
a17 1.00 0.00 1.00 0.00 0.00 1.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00 1.00 0.00
a18 1.00 0.00 1.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 1.00 1.00 0.00 1.00 1.00
a19 0.00 0.00 1.00 0.00 1.00 1.00 0.00 1.00 0.00 0.00 1.00 1.00 1.00 1.00 1.00
a20 0.00 1.00 0.00 1.00 1.00 0.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00
     a16  a17  a18  a19  a20
a1  1.00 1.00 1.00 0.00 0.00
a2  1.00 0.00 0.00 1.00 1.00
a3  0.00 0.00 1.00 0.00 1.00
a4  0.00 1.00 0.00 1.00 1.00
a5  1.00 1.00 0.00 0.00 1.00
a6  0.00 0.00 0.00 1.00 0.00
a7  1.00 0.00 0.00 0.00 0.00
a8  0.00 0.00 1.00 0.00 1.00
a9  1.00 1.00 1.00 1.00 0.00
a10 0.00 1.00 1.00 1.00 0.00
a11 1.00 1.00 0.00 1.00 1.00
a12 1.00 0.00 1.00 1.00 0.00
a13 0.00 0.00 1.00 0.00 1.00
a14 0.00 0.00 0.00 0.00 0.00
a15 1.00 0.00 1.00 0.00 1.00
a16 0.00 0.00 1.00 0.00 0.00
a17 0.00 0.00 1.00 0.00 1.00
a18 0.00 0.00 0.00 1.00 0.00
a19 0.00 0.00 0.00 0.00 1.00
a20 1.00 1.00 1.00 1.00 0.00

Marginal Posterior Global Error Distribution:

             e^-       e^+
Min    0.1443951 0.0004238
1stQ   0.3126975 0.0167584
Median 0.3678306 0.0294646
Mean   0.3783663 0.0493688
3rdQ   0.4423027 0.0574099
Max    0.6909116 0.2262239

Marginal Posterior Error Distribution (by observer):

Probability of False Negatives (e^-):

       Min   1stQ Median   Mean   3rdQ    Max
o1  0.3132 0.3599 0.3798 0.3864 0.4073 0.5071
o2  0.2613 0.2944 0.3115 0.3187 0.3419 0.3995
```

```
o3  0.4148 0.4724 0.4937 0.4948 0.5213 0.5649
o4  0.2511 0.3075 0.3246 0.3257 0.3448 0.4085
o5  0.1814 0.2417 0.2681 0.2678 0.2887 0.3434
o6  0.2881 0.3531 0.3761 0.3766 0.4046 0.4488
o7  0.2395 0.3028 0.3211 0.3244 0.3449 0.3951
o8  0.1444 0.2011 0.2209 0.2212 0.2398 0.2922
o9  0.3708 0.4358 0.4529 0.4578 0.4787 0.5503
o10 0.3210 0.3724 0.3967 0.3982 0.4259 0.4751
o11 0.3064 0.3847 0.4093 0.4109 0.4371 0.5007
o12 0.2367 0.3132 0.3354 0.3349 0.3607 0.4455
o13 0.3534 0.4144 0.4386 0.4382 0.4600 0.5337
o14 0.2438 0.2985 0.3235 0.3229 0.3452 0.4184
o15 0.2585 0.3299 0.3510 0.3519 0.3706 0.4704
o16 0.2502 0.3298 0.3481 0.3509 0.3699 0.4268
o17 0.1759 0.2273 0.2488 0.2503 0.2668 0.3372
o18 0.3959 0.4468 0.4646 0.4710 0.4922 0.5812
o19 0.4944 0.5736 0.6007 0.5975 0.6189 0.6909
o20 0.3737 0.4433 0.4631 0.4671 0.4916 0.5607

Probability of False Positives (e^+):

          Min      1stQ    Median      Mean      3rdQ       Max
o1  0.0195433 0.0397919 0.0490722 0.0510872 0.0585109 0.1069030
o2  0.1067928 0.1395067 0.1555455 0.1569023 0.1714084 0.2262239
o3  0.0084268 0.0165518 0.0224858 0.0236948 0.0293221 0.0551761
o4  0.0712109 0.1047058 0.1137249 0.1180402 0.1320136 0.1723854
o5  0.0034994 0.0103378 0.0150617 0.0169536 0.0212638 0.0468961
o6  0.0004238 0.0040509 0.0068522 0.0082363 0.0098606 0.0279960
o7  0.0061597 0.0136434 0.0192100 0.0207973 0.0266508 0.0484633
o8  0.0072124 0.0204896 0.0260316 0.0282562 0.0350608 0.0593586
o9  0.0804463 0.1092987 0.1213202 0.1246571 0.1372326 0.1935724
o10 0.0065188 0.0135991 0.0194675 0.0223006 0.0278075 0.0594150
o11 0.0173415 0.0358252 0.0445098 0.0464278 0.0551955 0.0828446
o12 0.0185894 0.0416346 0.0499440 0.0516976 0.0573815 0.1202316
o13 0.0029818 0.0108936 0.0155202 0.0170049 0.0209790 0.0401566
o14 0.0044849 0.0108034 0.0166631 0.0178764 0.0226294 0.0486647
o15 0.0084143 0.0199868 0.0271149 0.0290795 0.0355966 0.0606914
o16 0.0009067 0.0078736 0.0124531 0.0139218 0.0187929 0.0455700
o17 0.0066611 0.0216195 0.0273388 0.0290307 0.0346110 0.0691573
o18 0.0846863 0.1344580 0.1508170 0.1485688 0.1628176 0.2036186
o19 0.0037608 0.0117982 0.0171030 0.0179751 0.0225298 0.0466090
o20 0.0214701 0.0348032 0.0433397 0.0448676 0.0516594 0.0936080

MCMC Diagnostics:

        Replicate Chains: 5
        Burn Time: 300
```

```
        Draws per Chain: 20 Total Draws: 100
        Potential Scale Reduction (G&R's sqrt(Rhat)):
                Max: 1.003116
                Med: 0.9992194
                IQR: 0.0004545115

R> cor(em, apply(b$em, 2, median))

[1] 0.9187894

R> cor(ep, apply(b$ep, 2, median))

[1] 0.971649

R> mean(apply(b$net, c(2, 3), median) == g)

[1] 1
```

Although the priors do not reflect the true error distribution, `bbnam` still does a good job of pinning down the error rates (and the network itself, which is actually somewhat easier to estimate in many cases). In practice, the `bbnam` model is fairly robust to choice of priors, so long as the error rate priors do not put a large degree of mass on the "perverse" region for which `em + ep > 1`. Multiple actors whose error rates satisfy this condition with high probability in the posterior, or posterior graph distributions which are strongly multimodal, can be indicators either of excessively "perverse" priors or of extreme disagreement among informants (e.g., as would result from systematic deception). Either possibility warrants a re-examination of both the user's modeling assumptions and of the data itself.

Having obtained a Bayesian point estimate, we can also evaluate the performance of various classical network estimators. The `consensus` function allows us to calculate several, including the union and intersection LAS, central graph, and Romney-Batchelder model:

```
R> mean(consensus(dat, method = "LAS.intersection") == g)

[1] 0.7725

R> mean(consensus(dat, method = "LAS.union") == g)

[1] 0.905

R> mean(consensus(dat, method = "central.graph") == g)

[1] 0.9575

R> mean(consensus(dat, method = "romney.batchelder") == g)
```

```
Estimated competency scores:
 [1] 0.5384305 0.5152780 0.4482434 0.5333154 0.7128820 0.5920044 0.6278100
 [8] 0.7532642 0.3863239 0.5535066 0.5120474 0.6065419 0.5147395 0.6447705
[15] 0.6046575 0.6121955 0.7115359 0.3448647 0.3351731 0.4501279
Estimated bias parameters:
 [1] 0.13137940 0.35170786 0.06013660 0.28684742 0.09962490 0.04767398
 [7] 0.08915006 0.15302781 0.22559772 0.07431412 0.11489655 0.15412247
[13] 0.05894590 0.08052288 0.09550557 0.06195760 0.14675686 0.24625026
[19] 0.04302486 0.10195838
[1] 1
```

For this scenario, the intersection LAS is an especially poor choice (since it exacerbates the effects of false negatives); the central graph and Romney-Batchelder models are far better. The performance of the central graph will degrade quickly, however, when either false positive or false negative rates approach or exceed 0.5. The two likelihood-based methods (`bbnam` and Romney-Batchelder) can still be quite robust in such such cases, provided that total error rates (false positive plus false negative) are less than 1.

As a final example of **sna**'s model-based methods, we here illustrate the use of `lnam` to fit a linear network autocorrelation model. We show in this case an example which includes both AR and MA components, estimating both effects simultaneously. (This example requires the **numDeriv** package.)

```
R> w1 <- rgraph(50)
R> w2 <- rgraph(50)
R> x <- matrix(rnorm(50 * 5), 50, 5)
R> r1 <- 0.2
R> r2 <- 0.3
R> sigma <- 0.1
R> beta <- rnorm(5)
R> nu <- rnorm(50, 0, sigma)
R> e <- qr.solve(diag(50) - r2 * w2, nu)
R> y <- qr.solve(diag(50) - r1 * w1, x %*% beta + e)
R> fit <- lnam(y, x, w1, w2)
R> summary(fit)

Call:
lnam(y = y, x = x, W1 = w1, W2 = w2)

Residuals:
     Min       1Q   Median       3Q      Max
-0.52052 -0.18305  0.01156  0.15557  0.62082

Coefficients:
        Estimate Std. Error Z value Pr(>|z|)
X1     -0.331259   0.010831  -30.58   <2e-16 ***
X2      0.535608   0.009448   56.69   <2e-16 ***
X3     -0.685068   0.007138  -95.98   <2e-16 ***
```

```
X4      0.691812   0.008417   82.19   <2e-16 ***
X5      0.016491   0.007890    2.09   0.0366 *
rho1.1  0.194935   0.002575   75.71   <2e-16 ***
rho2.1  0.307491   0.021167   14.53   <2e-16 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

      Estimate Std. Error
Sigma  0.09597   9.22e-05

Goodness-of-Fit:
        Residual standard error: 0.2913 on 43 degrees of freedom (w/o Sigma)
        Multiple R-Squared: 0.96, Adjusted R-Squared: 0.9534
        Model log likelihood: 58.47 on 42 degrees of freedom (w/Sigma)
        AIC: -100.9 BIC: -85.65

        Null model: meanstd
        Null log likelihood: -82.48 on 48 degrees of freedom
        AIC: 169.0 BIC: 172.8
        AIC difference (model versus null): 269.9
        Heuristic Log Bayes Factor (model versus null):  258.4
```

In addition to the above diagnostics, `plot(fit)` produces residual plots and a "net influence plot" which depicts the total influence of each vertex on each other vertex in network form; (i, j) pairs for which i's net influence on j is estimated to be at least two standard deviations greater than the mean net influence are designated by green edges, while corresponding pairs for which i's net influence on j is estimated to be at least two standard deviations lower (i.e., more negative) than the mean net influence are designated by red edges. Sample output for the above example is provided in Figure 6.

3. Closing comments

The methodological literature on social network analysis is large and growing, and no one package can hope to implement all known measures and techniques. **sna** provides a collection of routines which is diverse, and which covers many of the methods currently seeing wide use within the field. Together with the other packages of the **statnet** ensemble, it is hoped that the inclusion of such tools within a freely available, widely used statistical computing platform will help further the integration of network analytic methods with more conventional approaches to modern data analysis.

Acknowledgments

The author would like to thank the many persons who have contributed to **sna** in some fashion, including (but not limited to) David Barron, Matthijs den Besten, Alex Montgomery, David Krackhardt, David Dekker, Kurt Hornik, Ulrik Brandes, Mark S. Handcock, and the **statnet**

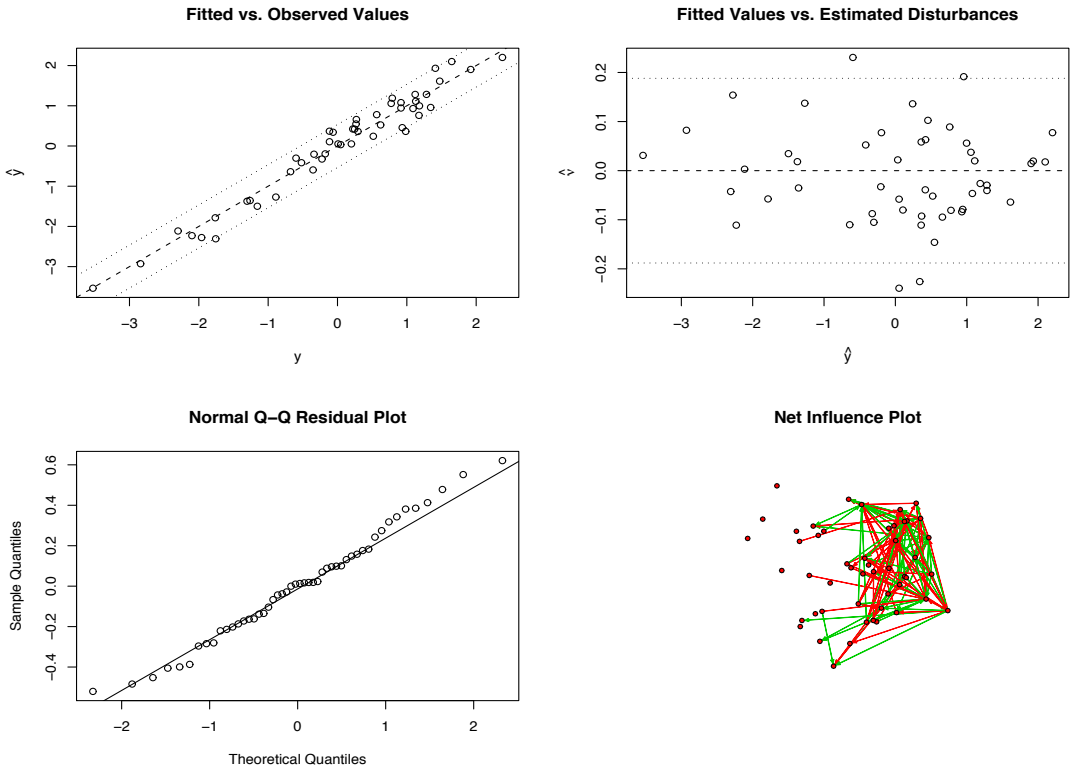

Figure 6: Plot method output for `lnam`.

team. This paper is based upon work supported by National Institutes of Health award 5 R01 DA012831-05, subaward 918197, and by NSF award IIS-0331707.

References

Anselin L (1988). *Spatial Econometrics: Methods and Models*. Kluwer, Norwell, MA.

Banks D, Carley KM (1994). "Metric Inference for Social Networks." *Journal of Classification*, **11**(1), 121–149.

Batagelj V, Mrvar A (2007). ***Pajek**: Package for Large Network Analysis*. University of Ljubljana, Slovenia. URL `http://vlado.fmf.uni-lj.si/pub/networks/pajek/`.

Batchelder WH, Romney AK (1988). "Test Theory Without an Answer Key." *Psychometrika*, **53**(1), 71–92.

Bonacich P (1987). "Power and Centrality: A Family of Measures." *American Journal of Sociology*, **92**, 1170–1182.

Boorman SA, White HC (1976). "Social Structure from Multiple Networks II. Role Structures." *American Journal of Sociology*, **81**, 1384–1446.

Borgatti SP (2007). ***NetDraw**: Network Visualization Software*. Version 2.067, URL http://www.analytictech.com/.

Borgatti SP, Carley K, Krackhardt D (2006). "Robustness of Centrality Measures Under Conditions of Imperfect Data." *Social Networks*, **28**, 124–136.

Borgatti SP, Everett MG, Freeman LC (1999). ***UCINET** 6.0 for Windows: Software for Social Network Analysis*. Analytic Technologies, Natick. URL http://www.analytictech.com/.

Boyd JP (1969). "The Algebra of Group Kinship." *Journal of Mathematical Psychology*, **6**, 139–167.

Brandes U, Erlebach T (eds.) (2005). *Network Analysis: Methodological Foundations*. Springer-Verlag, Berlin.

Brandes U, Kenis P, Wagner D (2003). "Communicating Centrality in Policy Network Drawings." *IEEE Transactions on Visualization and Computer Graphics*, **9**(2), 241–253.

Breiger RL, Boorman SA, Arabie P (1975). "An Algorithm for Clustering Relational Data with Applications to Social Network Analysis and Comparison with Multidimensional Scaling." *Journal of Mathematical Psychology*, **12**, 323–383.

Brockwell PJ, Davis RA (1991). *Time Series: Theory and Methods*. Springer-Verlag, New York, second edition.

Burt RS (1976). "Positions In Networks." *Social Forces*, **55**, 93–122.

Burt RS (1991). ***STRUCTURE***. Columbia University. Software package version 4.2, URL http://faculty.chicagogsb.edu/ronald.burt/teaching/.

Butts CT (2003). "Network Inference, Error, and Informant (In)Accuracy: A Bayesian Approach." *Social Networks*, **25**(2), 103–140.

Butts CT (2007). "Permutation Models for Relational Data." *Sociological Methodology*, **37**, 257–281.

Butts CT, Carley KM (2001). "Multivariate Methods for Interstructural Analysis." *CASOS working paper*, Center for the Computational Analysis of Social and Organization Systems, Carnegie Mellon University.

Butts CT, Carley KM (2005). "Some Simple Algorithms for Structural Comparison." *Computational and Mathematical Organization Theory*, **11**(4), 291–305.

Butts CT, Handcock MS, Hunter DR (2007). ***network**: Classes for Relational Data*. Statnet Project http://statnetproject.org/, Seattle, WA. R package version 1.3, URL http://CRAN.R-project.org/package=network.

Butts CT, Pixley JE (2004). "A Structural Approach to the Representation of Life History Data." *Journal of Mathematical Sociology*, **28**(2), 81–124.

Cliff AD, Ord JK (1973). *Spatial Autocorrelation*. Pion, London.

Davis JA, Leinhardt S (1972). "The Structure of Positive Interpersonal Relations in Small Groups." In J Berger (ed.), "Sociological Theories in Progress, Volume 2," pp. 218–251. Houghton Mifflin, Boston.

Dodds PS, Watts DJ, Sabel CF (2003). "Information Exchange and the Robustness of Organizational Networks." *Proceedings of the National Academy of Sciences*, **100**(2), 12516–12521.

Doreian P (1990). "Network Autocorrelation Models: Problems and Prospects." In IDA Griffith (ed.), "Spatial Statistics: Past, Present, and Future," pp. 369–389. Institute of Mathematical Geography, Ann Arbor.

Doreian P, Batagelj V, Ferlioj A (2005). *Generalized Blockmodeling*. Cambridge University Press, Cambridge.

Fararo TJ (1981). "Biased Networks and Social Structure Theorems. Part I." *Social Networks*, **3**, 137–159.

Fararo TJ (1983). "Biased Networks and the Strength of Weak Ties." *Social Networks*, **5**, 1–11.

Fararo TJ, Sunshine MH (1964). *A Study of a Biased Friendship Net*. Youth Development Center, Syracuse, NY.

Faust K (2007). "Very Local Structure in Social Networks." *Sociological Methodology*, **37**, 209–256.

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

Freeman LC (1979). "Centrality in Social Networks: Conceptual Clarification." *Social Networks*, **1**(3), 223–258.

Freeman LC (2004). *The Development of Social Network Analysis: A Study in the Sociology of Science*. Empirical Press, Vancouver.

Fruchterman TMJ, Reingold EM (1991). "Graph Drawing by Force-directed Placement." *Software – Practice and Experience*, **21**(11), 1129–1164.

Geary R (1954). "The Contiguity Ratio and Spatial Mapping." *The Incorporated Statistician*, **5**, 115–145.

Gelman A, Carlin JB, Stern HS, Rubin DB (1995). *Bayesian Data Analysis*. Chapman & Hall/CRC, London.

Gelman A, Rubin DB (1992). "Inference from Iterative Simulation Using Multiple Sequences." *Statistical Science*, **7**, 457–511.

Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, Ellis B, Gautier L, Ge Y, Gentry J, Hornik K, Hothorn T, Huber W, Iacus S, Irizarry R, Leisch F, Li C, Maechler M, Rossini AJ, Sawitzki G, Smith C, Smyth G, Tierney L, Yang JYH, Zhang

J (2004). "**Bioconductor**: Open Software Development for Computational Biology and Bioinformatics." *Genome Biology*, **5**, R80. URL http://genomebiology.com/2004/5/10/R80/.

Gilks WR, Richardson S, Spiegelhalter DJ (eds.) (1996). *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC, New York.

Gould R, Fernandez R (1989). "Structures of Mediation: A Formal Approach to Brokerage in Transaction Networks." *Sociological Methodology*, **19**, 89–126.

Hall KM (1970). "An r-dimensional Quadratic Placement Algorithm." *Management Science*, **17**, 219–229.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003). ***statnet**: Software Tools for the Statistical Modeling of Network Data*. Statnet Project http://statnetproject.org/, Seattle, WA. R package version 2.0, URL http://CRAN.R-project.org/package=statnet.

Holland PW, Leinhardt S (1970). "A Method for Detecting Structure in Sociometric Data." *American Journal of Sociology*, **70**, 492–513.

Hubert LJ (1987). *Assignment Methods in Combinatorial Data Analysis*. Marcel Dekker, New York.

Huisman M, van Duijn MAJ (2003). "**StOCNET**: Software for the Statistical Analysis of Social Networks." *Connections*, **25**(1), 7–26.

Ingram P, Roberts PW (2000). "Friendships Among Competitors in the Sydney Hotel Industry." *American Journal of Sociology*, **106**, 387–423.

Kamada T, Kawai S (1989). "An Algorithm for Drawing General Undirected Graphs." *Information Processing Letters*, **31**(1), 7–15.

Koenker R, Ng P (2007). ***SparseM**: Sparse Linear Algebra*. R package version 0.73, URL http://CRAN.R-project.org/package=SparseM.

Krackhardt D (1987a). "Cognitive Social Structures." *Social Networks*, **9**(2), 109–134.

Krackhardt D (1987b). "QAP Partialling as a Test of Spuriousness." *Social Networks*, **9**(2), 171–186.

Krackhardt D (1988). "Predicting with Networks: Nonparametric Multiple Regression Analyses of Dyadic Data." *Social Networks*, **10**, 359–382.

Krackhardt D (1994). "Graph Theoretical Dimensions of Informal Organizations." In KM Carley, MJ Prietula (eds.), "Computational Organizational Theory," pp. 88–111. Lawrence Erlbaum Associates, Hillsdale, NJ.

Krackhardt D, Blythe J, McGrath C (1994). "**KrackPlot** 3.0: An Improved Network Drawing Program." *Connections*, **17**(2), 53–55.

Leenders TTAJ (2002). "Modeling Social Influence Through Network Autocorrelation: Constructing the Weight Matrix." *Social Networks*, **24**(1), 21–47.

Marsden PV (2005). "Recent Developments in Network Measurement." In PJ Carrington, J Scott, S Wasserman (eds.), "Models and Methods in Social Network Analysis," chapter 2, pp. 8–30. Cambridge University Press, Cambridge.

Mayhew BH (1984). "Baseline Models of Sociological Phenomena." *Journal of Mathematical Sociology*, **9**, 259–281.

Moran PAP (1950). "Notes on Continuous Stochastic Phenomena." *Biometrika*, **37**, 17–23.

Pattison P, Robins GL (2002). "Neighbourhood-Based Models for Social Networks." *Sociological Methodology*, **32**, 301–337.

Rapoport A (1957). "A Contribution to the Theory of Random and Biased Nets." *Bulletin of Mathematical Biophysics*, **15**, 523–533.

R Development Core Team (2007). *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/.

Richards WD, Seary AJ (2006). ***MultiNet*** *for Windows*. Version 4.75, URL http://www.sfu.ca/~richards/Multinet/Pages/multinet.htm.

Romney AK, Weller SC, Batchelder WH (1986). "Culture as Consensus: A Theory of Culture and Informant Accuracy." *American Anthropologist*, **88**(2), 313–338.

Sabidussi G (1966). "The Centrality Index of a Graph." *Psychometrika*, **31**, 581–603.

Shimbel A (1953). "Structural Parameters of Communication Networks." *Bulletin of Mathematical Biophysics*, **15**, 501–507.

Skvoretz J, Fararo TJ, Agneessens F (2004). "Advances in Biased Net Theory: Definitions, Derivations, and Estimations." *Social Networks*, **26**, 113–139.

Snijders TAB (2001). ***SIENA****: Simulation Investigation for Empirical Network Analysis*. Version 3.1, URL http://stat.gamma.rug.nl/snijders/siena.html.

Snijders TAB (2002). "Markov Chain Monte Carlo Estimation of Exponential Random Graph Models." *Journal of Social Structure*, **3**(2).

Stallman RM (2002). *Free Software, Free Society: Selected Essays of Richard M. Stallman*. GNU Press/Free Software Foundation, Boston, MA.

Stephenson K, Zelen M (1989). "Rethinking Centrality: Methods and Applications." *Social Networks*, **11**, 1–37.

Stokman FN, Van Veen FJAM (1981). ***GRADAP****, Graph Definition and Analysis Package User's Manual*. Interuniversity Project Group **GRADAP**, University of Amsterdam-Gröningen-Nijmegen. URL http://www.assess.com/.

Wasserman S, Robins G (2005). "An Introduction to Random Graphs, Dependence Graphs, and p^*." In PJ Carrington, J Scott, S Wasserman (eds.), "Models and Methods in Social Network Analysis," chapter 10, pp. 192–214. Cambridge University Press, Cambridge.

Wasserman SS, Faust K (1994). *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences. Cambridge University Press, Cambridge.

Watts DJ, Strogatz SH (1998). "Collective Dynamics of 'Small-World' Networks." *Nature*, **393**, 440–442.

West DB (1996). *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, NJ.

White HC (1963). *An Anatomy of Kinship*. Englewood Cliffs, NJ, Prentice Hall.

Affiliation:

Carter T. Butts
Department of Sociology and Institute for Mathematical Behavioral Sciences
University of California, Irvine
Irvine, CA 92697-5100, United States of America
E-mail: buttsc@uci.edu
URL: http://www.faculty.uci.edu/profile.cfm?faculty_id=5057

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 24, Issue 6 *Submitted:* 2007-06-01
February 2008 *Accepted:* 2007-12-25