PLEASE NOTE: This document has been modified from the original paper to form a package vignette. The original paper is:
network: A Package for Managing Relational Data in R. *Journal of Statistical Software* 24:2, 2008.
http://www.jstatsoft.org/v24/i02/paper

network: A Package for Managing Relational Data in R

Carter T. Butts
University of California, Irvine

Abstract

Effective memory structures for relational data within R must be capable of representing a wide range of data while keeping overhead to a minimum. The **network** package provides an class which may be used for encoding complex relational structures composed a vertex set together with any combination of undirected/directed, valued/unvalued, dyadic/hyper, and single/multiple edges; storage requirements are on the order of the number of edges involved. Some simple constructor, interface, and visualization functions are provided, as well as a set of operators to facilitate employment by end users. The package also supports a C-language API, which allows developers to work directly with **network** objects within backend code.

Keywords: relational data, data structures, graphs, **network**, **statnet**, R.

1. Background and introduction

In early 2002, the author and several other members of what would ultimately become the **statnet** project (Handcock, et al. 2003) came to the conclusion that the simple, matrix-based approach to representation of relational data utilized by early versions of packages such as **sna** were inadequate for the next generation of relational analysis tools in R. Rather, what was required was a customized class structure to support relational data. This class structure would be used for all **statnet** packages, thus insuring interoperability; ideally, it would also be possible to port this structure to other languages, thereby further enhancing compatibility.

The requirements which were posed for a network data class were as follows, in descending order of priority:

1. The class had to be sufficiently general to encode all major types of network data collected presently or in the foreseeable future;

2. Class storage needed to be of sufficient efficiency to permit representation of large networks (in particular, storage which was sub-quadratic in graph order for sparse networks); and

3. It had to be possible to develop interface methods to the class which were of reasonable computational efficiency.

Clearly, there are multiple approaches which could be taken to construct such a class structure. Here we describe the result of one particular effort, specifically the **network** package (Butts, et al. 2007) for the R system for statistical computing (R Development Core Team 2007).

1.1. Historical note

The **network** package as described here evolved from a specification originally written as an unpublished working paper, "Memory Structures for Relational Data in R: Classes and Interfaces" (Butts 2002). At this time, the class in question was tentatively entitled "graph." It subsequently emerged that a similar package was being developed by Robert Gentleman under the **graph** title (as part of the BioConductor project) (Gentleman, et al. 2007), and the name of the present project was hence changed to "network" in early 2005. A somewhat later version of the above relational data specification was also shared with Gabor Csardi in mid-2004, portions of which were incorporated in the development by Gabor of the **igraph** package (Csardi & Nepusz 2006). As a result, there are currently three commonly available class systems for relational data in R, two of which (**network** and **igraph**) share some common syntax and interface concepts. It should also be noted that (as mentioned above) both standard and sparse matrix (e.g., **sparseM** Koenker & Ng 2007) classes have been and continue to be used to represent relational data in R. This article does not attempt to address the relative benefits and drawbacks of these different tools, but readers should be aware that multiple alternatives are available.

1.2. A very quick note on notation

Throughout this paper we will use "graph" or "network" (G) generically to refer to any relational structure on a given vertex set (V), and "edge" to refer to a generalized edge (i.e., an ordered pair (T, H) where T is the "tail set" of the edge and H is the corresponding "head set," and where $T, H \subseteq V(G)$). The cardinality of the vertex set we denote $|V(G)| = n$, and the cardinality of the corresponding edge set we likewise denote $|E(G)| = m$. When discussing storage/computational complexity we will often use a loose order notation, where $\mathcal{O}\big(f(x)\big)$ is intended to indicate that the quantity in question grows more slowly than $f(x)$ as $x \to \infty$. A general familiarity with the R statistical computing system (and related syntax/terminology) is assumed. Those unfamiliar with R may wish to peruse a text such as those of Venables & Ripley (2000, 2002) or Chambers (1998).

2. The `network` class

The `network` class is a (reasonably) simple object structure designed to store a single relation on a vertex set of arbitrary size. The relation stored by a `network` class object is based on a generalized edge model; thus, edges may be directed, arbitrarily valued (with multiple values

per edge), multiplex (i.e., multiple edges per directed dyad), hyper (i.e., multiple head/tail vertices per edge), etc. Storage requirements for the `network` class are on the order of the number of nodes plus the total number of edges (which is substantially sub-n^2 for sparse graphs), and retrieval of edge values has a time complexity which is no worse than $\mathcal{O}(n)$.[1] For example, a network with 100,000 vertices and 100,000 edges currently consumes approximately 74MB of RAM (R 2.6.1), versus approximately 40GB for a full sociomatrix (a savings of approximately 99.8%). When dealing with extremely large, sparse graphs it therefore follows that `network` objects are substantially more efficient than simpler representations such as adjacency matrices. The class also provides for the storage of arbitrary metadata at the edge, vertex, and network level. Thus, `network` objects may be preferred to matrix representations for reasons of generality, performance, or integrative capability; while alternative means exist of obtaining these goals separately, **network** provides a single toolkit which is designed to be effective across a wide range of applications.

In this section, we provide a basic introduction to the `network` class, from a user's point of view. We describe the conditions which are necessary for **network** to be employed, and the properties of `network` objects (and their components). This serves as background for a discussion of the use of **network** methods in practical settings, which is given in the section which follows.

2.1. Identification of vertices and edges

For purposes of storage, we presume that each vertex and each edge can be uniquely identified. (For partially labeled or unlabeled graphs, observe that this internal labeling is essentially arbitrary. See Butts & Carley 2005, for a discussion.) Vertices are labeled by positive integers in the order of entry, with edges likewise; it is further assumed that this is maintained for vertices (e.g., removing a vertex requires relabeling) but not for edges. (This last has to do with how edges are handled internally, but has the desirable side effect of making edge changes less expensive.) Vertices and edges are always stored by label. In the text that follows, any reference to a vertex or edge "ID" refers to these labeling numbers, and not to any other (external) identification that a vertex or edge may have.

2.2. Basic class structure

Functionally, a `network` object can be thought as a collection of vertices and edges, together with metadata regarding those vertices and edges (as well as the network itself). As noted above, each vertex is assumed to be identifiable, and the number of vertices is fixed. Here, we discuss the way in which edges are defined within **network**, as well as the manner in which associated metadata is stored.

Edge structure

Edges within a `network` object consist of three essential components. First, each edge contains two vectors of vertex IDs, known respectively as the *head* and *tail* lists of the edge. In addition to these lists, each edge also contains a list of attribute information. This is discussed in more

[1] Edge retrieval actually scales with degree, and average retrieval time is hence approximately constant for many data sources. For an argument regarding constraints on the growth of mean degree in interpersonal networks, see e.g., Mayhew & Levinger (1976).

detail below. The content and interpretation of the head and tail lists are dependent on the type of network in which they reside. In a directed network, an edge connects the elements of its tail list with those of its head list, but not vice versa: i is adjacent to j iff there exists some edge, $e = (T, H)$, such that $i \in T, j \in H$. In an undirected network, by contrast, the head and tail sets of an edge are regarded as exchangeable. Thus, i is adjacent to j in an undirected network iff there exists an edge such that $i \in T, j \in H$ or $i \in H, j \in T$. **network** methods which deal with adjacency and incidence make this distinction transparently, based on the network object's directedness attribute (see below).

Note that in the familiar case of dyadic networks (the focus of packages such as **sna** (Butts 2007)), the head and tail lists of any given edge must have exactly one element. This need not be true in general, however. An edge with a head or tail list containing more than one element is said to be *hypergraphic*, reflecting a one-to-many, many-to-one, or many-to-many relationship. Hyperedges are permitted natively within **network**, although some methods may not support them – a corresponding network attribute is used by **network** methods to determine whether these edges are present, as explained below. Finally, another fundamental distinction is made between edges in which H and T are disjoint, versus those in which these endpoint lists have one or more elements in common. Edges of the latter type are said to be *loop-like*, generalizing the familiar notion of "loop" (self-tie) from the theory of dyadic graphs. Loop-like edges allow vertices to relate to themselves, and are disallowed in many applications. Applicable methods are expected to interpret such edges intelligently, where present.

`network` *attributes*

As we have already seen, each `network` object contains a range of metadata in addition to relational information. This metadata – in the form of attributes – is divided into information stored at the network, vertex, and edge levels. In all three cases, attributes are stored in `lists`, and are expected to be named. While there is no limit to the user-defined attributes which may be stored in this manner, certain attributes are required of all `network` objects. At the network level, such attributes describe general properties of the network as a whole; specifically, they may be enumerated as follows:

`bipartite` This is a `numeric` attribute, which is used to indicate the presence of an intrinsic bipartition in the `network` object. Formally, a bipartition is a partition of a network's vertices into two classes, such that no vertex in either class is adjacent to any vertex in the same class. While such partitions occur naturally, they may also be specifically enforced by the nature of the data in question. (This is the case, for instance, with two-mode networks (Wasserman & Faust 1994), in which edges represent connections between two distinct classes of entities.) Where `bipartite > 0`, **network** methods will automatically assume that vertices with IDs less than or equal to `bipartite` belong to one such class, with those with IDs greater than `bipartite` belonging to the other. This information may be used in selecting default modes for data display, calculating numbers of possible edges, etc. When `bipartite == 0`, by contrast, no such bipartition is assumed. It should be emphasized that `bipartite` is intended to reflect bipartitions which are required *ex ante,* rather than those which happen to arise empirically. There is also no performance advantage to the use of `bipartite`, since **network** only stores edges which are defined; it can make data processing more convenient, however, when working with intrinsically bipartite structures.

`directed` This is a `logical` attribute, which should be set to `TRUE` iff edges are to be interpreted as directed. As explained earlier, **network** methods will regard edge endpoint lists as exchangeable when `directed` is `FALSE`, allowing for automatic handling of both directed and undirected networks. For obvious reasons, misspecification of this attribute may lead to surprising results; it is generally set when a `network` object is created, and considered fixed thereafter.

`hyper` This attribute is a `logical` variable which is set to `TRUE` iff the network is allowed to contain hyperedges. Since the vast majority of network data is dyadic, this attribute defaults to `FALSE` for must construction methods. The setting of `hyper` to `TRUE` has potentially serious implications for edge retrieval, and so methods should not activate this option unless hypergraphic edges are explicitly to be permitted.

`loops` As noted, loop-like edges are frequently undefined in practical settings. The `loops` attribute is a `logical` which should be set to `TRUE` iff such edges are permitted within the network.

`multiple` In most settings, an edge is uniquely defined by its head and tail lists. In other cases, however, one must represent data in which multiple edges are permitted between the same endpoints. ("Same" here includes the effect of directedness; an edge from set H to set T is not the same as an edge from set T to set H, unless the network is undirected.) The `multiple` attribute is a `logical` variable which is set to `TRUE` iff such multiplex edges are permitted within the network. Where `multiple` is `FALSE`, **network** methods will assume all edges to be unique – like `directed`, the possibility of multiplex edges thus can substantially impact both behavior and performance. For this reason, `multiple` is generally set to `FALSE` by default, and should not be set to `TRUE` unless it is specifically necessary to permit multiple edges between the same endpoint sets.

`n` Finally, `n` is a `numeric` attribute containing the number of elements in the vertex set. Applicable methods are expected to adjust this attribute up or down, should vertices be added or deleted from the network.

While these attributes are clearly reserved, any number of others may be added. Attributes specifically pertaining to edges and/or vertices can be stored at the network level, but this is generally non-optimal – such attributes would have to be manually updated to reflect edge or vertex changes, and would require the creation of custom access methods. The preferred approach is to store such information directly at the edge or vertex level, as we discuss below.

Vertex attributes

As with the network as a whole, it is often useful to be able to supply attribute data for individual vertices (e.g., names, attributes, etc.). Each vertex thus has a `list` of named attributes, which can be used to store arbitrary information on a per-vertex basis; there is no restriction on the type of information which may be stored in this fashion, nor are all vertices constrained to carry information regarding the same attributes. Each vertex does carry two special attributes, however, which are assumed to be available to all class methods. These are `vertex.names`, which must be a `character` containing the name of the vertex, and the `logical` attribute `na`. Where `TRUE`, `na` indicates that the associated vertex is unobserved; this is useful in cases for which individual entities are known to belong to a given network,

but where data regarding those entities is unavailable. By default, `na` is set to `FALSE` and `vertex.names` is set equal to the corresponding vertex ID.

Edge attributes

Just as vertices can carry attributes, so too can edges. Each edge is endowed with a `list` of named attributes, which can be used to carry arbitrary information (e.g., tie strength, onset and termination times, etc.). As with vertex attributes, any information type may be employed and there is no requirement that all edges carry the same attributes. The one attribute required to be carried by each edge is `na`, a `logical` which (like the vertex case) is used to indicate the missingness of a given edge. Many **network** methods provide the option of filtering out missing edges when retrieving information, and/or returning the associated information (e.g., adjacency) as missing.

3. Using the `network` class

In addition to the class itself, **network** provides a range of tools for creating, manipulating, and visualizing `network` objects.[2] Here, we provide an overview of some of these tools, with a focus on the basic tasks most frequently encountered by end users. Additional information on these functions is also provided within the package manual. For the examples below, we begin by loading the network package into memory; we also set the random seed, to ensure that examples using random data match the output shown here. Within R, this may be accomplished via the following:

```
> library(network)
> set.seed(1702)
```

Throughout, we will represent R code in the above format. Readers may wish to try the demonstrations listed here for themselves, to get a better feel for how the package operates.

3.1. Importing data

It almost goes without saying that an important aspect of **network** functionality is the ability to import data from external sources. **network** includes functionality for the importation of **Pajek** project files (Batagelj 2007), a popular and versatile network data format, via the `read.paj` routine. Other formats supported by **sna** can be used as well, by importing to adjacency matrix form (using the relevant **sna** routines) and then coercing the result into a `network` object as described below. The **foreign** package can be used to import adjacency, edgelist, or incidence matrices from other computing environments in much the same way. Future package versions may include support for converting to and from other related classes, e.g., those of **RBGL** (Carey, et al. 2007) and **Rgraphviz** (Gentry, et al. 2007).

In addition to these methods, `network` objects can be loaded into R using native tools such as `load` (for saved objects) or `data` (for packaged data sets). With respect to the latter, **network** contains two sample data sets: `flo`, John Padgett's Florentine wedding data (from Wasserman & Faust 1994); and `emon`, a set of interorganizational networks from search and

[2] These tools are currently implemented via S3 methods.

rescue operations collected by Drabek, et al. (1981). `flo` consists of a single adjacency matrix, and is useful for illustrating the process of converting data from adjacency matrix to `network` form. `emon`, on the other hand, consists of a list of seven `network` objects with vertex and edge metadata. `emon` is thus especially useful for illustrating the use of `network` objects for rich data storage (in addition to being an interesting data set in its own right). Loading these data sets is as simple as invoking the `data` command, like so:

```
> data("flo")
> data("emon")
```

Further information on each of these data sets is given in the **network** manual. We shall also use these data sets as illustrative examples at various points within this paper.

3.2. Creating and viewing `network` objects

While importation is sometimes possible, in other cases we must create our own `network` objects. **network** supports two basic approaches to this task: create the object from scratch, or build it from existing relational data via coercion. Both methods are useful, and we illustrate each here.

In the most minimal case, we begin by creating an empty network to which edges may be added. This task is performed by the `network.initialize` routine, which serves as a constructor for the `network` class. `network.initialize` takes the order of the desired graph (i.e., n) as a required argument, and the required network attributes discussed in Section 2.2.2 may be passed as well. In the event that these are unspecified, it is assumed that a simple digraph (directed, no loops, hyperedges, multiplexity, or bipartitions) is desired. For example, one may create and print an empty digraph like so:

```
> net <- network.initialize(5)
> net


 Network attributes:
  vertices = 5
  directed = TRUE
  hyper = FALSE
  loops = FALSE
  multiple = FALSE
  bipartite = FALSE
  total edges= 0
    missing edges= 0
    non-missing edges= 0

 Vertex attribute names:
    vertex.names

No edge attributes
```

network has default `print` and `summary` methods, as well as low-level operators for assignment and related operations. These do not show much in the above case, since the network in question caries little information. To create a `network` along with a specified set of edges, the preferred high-level constructor is the eponymous `network`. Like `network.initialize`, this function returns a newly allocated `network` object having specified properties. Unlike the former, however, `network` may be called with adjacency and/or attribute information. Adjacency information may be passed by using a full or bipartite adjacency matrix, incidence matrix, or edgelist as the function's first argument. These input types are defined as follows:

Adjacency matrix: This must consist of a square `matrix` or two-dimensional `array`, whose i, jth cell contains the value of the edge from i to j; as such, adjacency matrices may only be used to specify dyadic networks. By default, edges are assumed to exist for all non-zero matrix values, and are constructed accordingly. Edge values may be retained by passing `ignore.eval = FALSE`, as described in the manual page for the `network.adjacency` constructor. The `matrix.type` for an adjacency matrix is `"adjacency"`.

Bipartite adjacency matrix: This must consist of a rectangular `matrix` or two-dimensional `array` whose row and column elements reflect vertices belonging to the lower and upper sets of a bipartition (respectively). Otherwise, the matrix is interpreted as per a standard adjacency matrix. (Thus, a bipartite adjacency matrix is simply the upper off-diagonal block of the full adjacency matrix for a bipartite graph, where vertices have been ordered by partition membership. See also Doreian, et al. (2005).) The `matrix.type` for a bipartite adjacency matrix is `"bipartite"`.

Incidence matrix: This must consist of a rectangular `matrix` or two-dimensional `array` whose row elements represent vertices, and whose column elements represent edges. A non-zero value is placed in the i, jth cell if vertex i is an endpoint of edge j. In the directed case, negative values signify membership in the tail set of the corresponding edge, while positive values signify membership in the edge's head set. Unlike adjacency matrices, incidence matrices can thus be used to describe hypergraphic edges (directed or otherwise). Note, however, that an undirected hypergraph composed of two-endpoint edges is not the same as a simple graph, since the edges of the former are necessarily loop-like. When `loops`, `hyper`, and `directed` are all `FALSE`, therefore, the two positive row-elements of an incidence matrix for each column are taken to signify the head and tail elements of a dyadic edge. (This is without loss of generality, since such an incidence matrix would otherwise be inadmissible.) When specifying that an incidence matrix is to be used, `matrix.type` should be set to `"incidence"`.

Edge list: This must consist of a rectangular `matrix` or two-dimensional `array` whose row elements represent edges. The i, 1st cell of this structure is taken to be the ID of the tail vertex for the edge with ID i, with the i, 2st cell containing the ID of the edge's head vertex. (Only dyadic networks may be input in this fashion.) Additional columns, if present, are taken to contain edge attribute values. The `matrix.type` for an edge list is `"edgelist"`.

As one might suspect, the `network` function actually operates by first calling `network.initialize` to create the required object, and then calling an appropriate edge

set constructor based on the input type. This fairly modular design allows for the eventual inclusion of a wider range of input formats (although the above covers the formats currently in widest use within the social network community). Although `network` attempts to infer the matrix type from context, is wise to fix the function's behavior via the `matrix.type` argument when passing information which is not in the default, adjacency matrix form. As a simple example of the `network` constructor in action, consider the following:

```
> nmat <- matrix(rbinom(25, 1, 0.5), nr = 5, nc = 5)
> net <- network(nmat, loops = TRUE)
> net


 Network attributes:
  vertices = 5
  directed = TRUE
  hyper = FALSE
  loops = TRUE
  multiple = FALSE
  bipartite = FALSE
  total edges= 9
    missing edges= 0
    non-missing edges= 9

 Vertex attribute names:
    vertex.names

No edge attributes


> summary(net)


Network attributes:
  vertices = 5
  directed = TRUE
  hyper = FALSE
  loops = TRUE
  multiple = FALSE
  bipartite = FALSE
 total edges = 9
   missing edges = 0
   non-missing edges = 9
 density = 0.36

Vertex attributes:
  vertex.names:
   character valued attribute
```

```
   5 valid vertex names

No edge attributes

Network adjacency matrix:
  1 2 3 4 5
1 1 0 1 0 1
2 1 0 0 1 0
3 1 0 1 1 0
4 0 0 1 0 0
5 0 0 0 0 0
```

```
> all(nmat == net[,])
```

```
[1] TRUE
```

Here, we have generated a random adjacency matrix (permitting diagonal elements) and used this to construct a digraph (with loops) in `network` object form. Since we employed an adjacency matrix, there was no need to set the matrix type explicitly; had we failed to set `loops = TRUE`, however, the diagonal entries of `nmat` would have been ignored. The above example also demonstrates the use of an important form of operator overloading which can be used with dyadic network objects: specifically, dyadic network objects respond to the use of the subset and subset assignment operators `[` and `[<-` as if they were conventional adjacency matrices. Thus, in the above case, `net[,]` returns `net`'s adjacency matrix (a fact we verify by comparing it with `nmat`). This is an extremely useful "shorthand" which can be used to simplify otherwise cumbersome network operations, especially on small networks.

The use of `network` function to create objects from input matrices has a functional parallel in the use of coercion methods to transform other objects into `network` form. These operate in the same manner as the above, but follow the standard R syntax for coercion, e.g.:

```
> net <- as.network(nmat, loops = TRUE)
> all(nmat == net[,])
```

```
[1] TRUE
```

By default, `as.network` assumes that square input matrices should be treated as adjacency matrices, and that diagonal entries should be ignored; here we have overridden the latter behavior by invoking the additional argument `loops = TRUE`. Matrix-based input can also be given in edgelist or incidence matrix form, as selected by the `matrix.type` argument. This and other options are described in greater detail within the package documentation.

The above methods can be used in conjunction with `data`, `load`, or `read` functions to convert imported relational data into `network` form. For example, we may apply this to the Florentine data mentioned in the previous section:

```
> nflo <- network(flo, directed = FALSE)
> nflo


 Network attributes:
  vertices = 16
  directed = FALSE
  hyper = FALSE
  loops = FALSE
  multiple = FALSE
  bipartite = FALSE
  total edges= 20
    missing edges= 0
    non-missing edges= 20

 Vertex attribute names:
    vertex.names

No edge attributes
```

Although the network's adjacency structure is summarized here in edgelist form, it may be queried in other ways. For instance, the following example demonstrates three simple methods for examining the neighborhood of a particular vertex:

```
> nflo[9,]


 [1] 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1


> nflo[9,1]


[1] 1


> nflo[9,4]


[1] 0


> is.adjacent(nflo, 9, 1)


[1] TRUE
```

```
> is.adjacent(nflo, 9, 4)
```

```
[1] FALSE
```

As the example shows, overloading can be used to extract partial as well as complete adjacency information from a `network` object. A more cumbersome (but slightly faster) method is to use a direct call to `is.adjacent`, the general indicator method for network adjacency. Calling the indicator method avoids the call parsing required by the extraction operator, which is the source of the performance difference. In practice, however, the impact of call parsing is quite minimal, and users are unlikely to detect a difference between the two approaches. (Where such overhead is an issue, it will generally be more efficacious to conduct adjacency queries directly from the backend code; this will be discussed below, in the context of the C-language API.)

In addition to adjacency, **network** supplies methods to query many basic properties of `network` objects. Although complex structural descriptives (e.g., centrality scores Wasserman & Faust 1994) are the province of other packages, **network**'s built-in functionality is sufficient to determine the types of edges allowed within a `network` object and constraints such as enforced bipartitions, as well as essential quantities such as size (number of vertices), edge count, and density (the ratio of observed to potential edges). Use of these indicator methods is straightforward, as illustrated by the following examples.

```
> network.size(nflo)                    #Number of vertices
```

```
[1] 16
```

```
> network.edgecount(nflo)               #Number of edges
```

```
[1] 20
```

```
> network.density(nflo)                 #Network density
```

```
[1] 0.1666667
```

```
> has.loops(nflo)                       #Can nflo have loops?
```

```
[1] FALSE
```

```
> is.bipartite(nflo)                    #Is nflo coded as bipartite?
```

```
[1] FALSE

> is.directed(nflo)                    #Is nflo directed?

[1] FALSE

> is.hyper(nflo)                       #Is nflo hypergraphic?

[1] FALSE

> is.multiplex(nflo)                   #Are multiplex edges allowed?

[1] FALSE
```

3.3. Coercing `network` objects to other forms

Just as one may often seek to coerce data from other forms into `network` object, so to does one sometimes need to coerce `network` objects into other data types. **network** currently supports several such coercion functions, all of which take network objects as input and produce matrices of one type or another. The class method for `as.matrix` performs this task, converting network objects to adjacency, incidence, or edgelist matrices as desired (adjacency being the default). Scalar-valued edge attributes, where present, may be used to set edge values using the appropriate functional arguments. Similar functionality is provided by `as.sociomatrix` and the extraction operator, although these are constrained to produce adjacency matrices. These equivalent approaches may be illustrated with application to the Florentine data as follows:

```
> as.sociomatrix(nflo)

             Acciaiuoli Albizzi Barbadori Bischeri Castellani Ginori Guadagni
Acciaiuoli            0       0         0        0          0      0        0
Albizzi               0       0         0        0          0      1        1
Barbadori             0       0         0        0          1      0        0
Bischeri              0       0         0        0          0      0        1
Castellani            0       0         1        0          0      0        0
Ginori                0       1         0        0          0      0        0
Guadagni              0       1         0        1          0      0        0
Lamberteschi          0       0         0        0          0      0        1
Medici                1       1         1        0          0      0        0
Pazzi                 0       0         0        0          0      0        0
```

```
Peruzzi              0       0         0        1         1      0        0
Pucci                0       0         0        0         0      0        0
Ridolfi              0       0         0        0         0      0        0
Salviati             0       0         0        0         0      0        0
Strozzi              0       0         0        1         1      0        0
Tornabuoni           0       0         0        0         0      0        1
             Lamberteschi Medici Pazzi Peruzzi Pucci Ridolfi Salviati Strozzi
Acciaiuoli              0      1     0       0     0       0        0       0
Albizzi                 0      1     0       0     0       0        0       0
Barbadori               0      1     0       0     0       0        0       0
Bischeri                0      0     0       1     0       0        0       1
Castellani              0      0     0       1     0       0        0       1
Ginori                  0      0     0       0     0       0        0       0
Guadagni                1      0     0       0     0       0        0       0
Lamberteschi            0      0     0       0     0       0        0       0
Medici                  0      0     0       0     0       1        1       0
Pazzi                   0      0     0       0     0       0        1       0
Peruzzi                 0      0     0       0     0       0        0       1
Pucci                   0      0     0       0     0       0        0       0
Ridolfi                 0      1     0       0     0       0        0       1
Salviati                0      1     1       0     0       0        0       0
Strozzi                 0      0     0       1     0       1        0       0
Tornabuoni              0      1     0       0     0       1        0       0
             Tornabuoni
Acciaiuoli           0
Albizzi              0
Barbadori            0
Bischeri             0
Castellani           0
Ginori               0
Guadagni             1
Lamberteschi         0
Medici               1
Pazzi                0
Peruzzi              0
Pucci                0
Ridolfi              1
Salviati             0
Strozzi              0
Tornabuoni           0
```

```
> all(nflo[,]==as.sociomatrix(nflo))
```

```
[1] TRUE
```

```
> all(as.matrix(nflo)==as.sociomatrix(nflo))
```

```
[1] TRUE
```

```
> as.matrix(nflo,matrix.type="edgelist")
```

```
      [,1] [,2]
 [1,]    9    1
 [2,]    6    2
 [3,]    7    2
 [4,]    9    2
 [5,]    5    3
 [6,]    9    3
 [7,]    7    4
 [8,]   11    4
 [9,]   15    4
[10,]   11    5
[11,]   15    5
[12,]    8    7
[13,]   16    7
[14,]   13    9
[15,]   14    9
[16,]   16    9
[17,]   14   10
[18,]   15   11
[19,]   15   13
[20,]   16   13
attr(,"n")
[1] 16
attr(,"vnames")
 [1] "Acciaiuoli"   "Albizzi"      "Barbadori"    "Bischeri"     "Castellani"
 [6] "Ginori"       "Guadagni"     "Lamberteschi" "Medici"       "Pazzi"
[11] "Peruzzi"      "Pucci"        "Ridolfi"      "Salviati"     "Strozzi"
[16] "Tornabuoni"
```

Note that vertex names (per the `vertex.names` attribute) are used by `as.sociomatrix` to set adjacency matrix row/column names where present.

The less-flexible `as.sociomatrix` function also plays an important role with respect to coercion in the **sna** package; the latter's `as.sociomatrix.sna` dispatches to **network**'s `as.sociomatrix` routine when **network** is loaded and a `network` object is given. The intent in both packages is to maintain an interoperable and uniform mechanism for guaranteeing adjacency matrix representations of input data (which are necessary for backward compatibility with some legacy functions).

3.4. Creating and modifying edges and vertices

In addition to coercion of data to `network` form, the **network** package contains many mechanisms for creating, modifying, and removing edges and vertices from `network` objects. The simplest means of manipulating edges for most users is the use of the overloaded extraction and assignment operators, which (as noted previously) simulate the effects of working with an adjacency matrix. Thus, a statement such as `g[i,j] <- 1` adds an edge between `i` and `j` (if one is not already present), `g[i,j] <- 0` removes an existing edge, and `g[i,j]` itself is a dichotomous indicator of adjacency. Subset selection and assignment otherwise works in the same fashion as for R matrices, including the role of `logicals` and element lists. (One minor exception involves the effects of assignment on undirected and/or loopless graphs: **network** will enforce symmetry and/or empty diagonal entries, and will ignore any assignments which are contrary to this.) The uses of assignment by overloading are hence legion, as partially illustrated by the following:

```
> #Add edges to an empty network
> net <- network.initialize(5,loops=TRUE)
> net[nmat>0] <- 1                        #One way to add edges
> all(nmat==net[,])                       #Should be TRUE

[1] TRUE

> net[,] <- 0                             #Remove the edges
> net[,] <- nmat                          #Not quite kosher, but _will_ work....
> all(nmat==net[,])                       #Should still be TRUE

[1] TRUE

> net[,] <- 0                             #Remove the edges
> for(i in 1:5)                           #Add the hard way!
+   for(j in 1:5)
+     if(nmat[i,j])
+       net[i,j] <- 1
> all(nmat==net[,])                       #Should STILL be TRUE

[1] TRUE

> net[,] <- 0                             #Remove the edges
> add.edges(net,row(nmat)[nmat>0],col(nmat)[nmat>0])
> all(nmat==net[,])                       #When will it all end??

[1] TRUE
```

```
> net[,] <- as.numeric(nmat[,])
> all(nmat==net[,])                    #When will it all end??
```

```
[1] TRUE
```

The above example also introduces `add.edges`, to which the overloaded assignment operator is a front end. `add.edges` is more cumbersome to employ than the assignment operators, but is substantially more powerful. In particular, it can be used to add edges of arbitrary type, with arbitrary attribute data. A comparison of usage is instructive; we begin by creating an empty digraph, and adding a single edge:

```
> #Add edges (redux)
> net<-network.initialize(5)                #Create empty graph
> add.edge(net,2,3)                         #Create 2->3 edge
> net[,]                                    #Trust, but verify
```

```
  1 2 3 4 5
1 0 0 0 0 0
2 0 0 1 0 0
3 0 0 0 0 0
4 0 0 0 0 0
5 0 0 0 0 0
```

```
> add.edges(net,c(3,5),c(4,4))              #3 and 5 send ties to 4
> net[,]                                    #Again, verify edges
```

```
  1 2 3 4 5
1 0 0 0 0 0
2 0 0 1 0 0
3 0 0 0 1 0
4 0 0 0 0 0
5 0 0 0 1 0
```

```
> net[,2]<-1                                #Everyone sends ties to 2
> net[,]                                    #Note that loops are not created!
```

```
  1 2 3 4 5
1 0 1 0 0 0
2 0 0 1 0 0
3 0 1 0 1 0
4 0 1 0 0 0
5 0 1 0 1 0
```

Observe that the (2,2) loop is not created, since `loops` is `FALSE` for this network. This automatic behavior is *not* true of `add.edges`, unless optional edge checking is turned on (by means of the `edge.check` argument). For this reason, explicit use of `add.edges` is discouraged for novice users.

In addition to edge addition/removal, vertices can be added or removed via `add.vertices` and `delete.vertices`. The former adds the specified number of vertices to a `network` object (along with any supplied attribute information), while the latter deletes a specified list of vertices from its argument. Usage is straightforward:

```
> #Deleting vertices
> delete.vertices(net,4)                                    #Remove vertex 4
> net[,]                                                    #It's gone!
```

```
  1 2 3 5
1 0 1 0 0
2 0 0 1 0
3 0 1 0 0
5 0 1 0 0
```

```
> add.vertices(net,2)                                       #Add two new vertices
> net[,]                                                    #Both are isolates
```

```
     1 2 3 5 <NA> <NA>
1    0 1 0 0    0    0
2    0 0 1 0    0    0
3    0 1 0 0    0    0
5    0 1 0 0    0    0
<NA> 0 0 0 0    0    0
<NA> 0 0 0 0    0    0
```

As the above illustrates, vertex names are not automatically created for newly added vertices (but can be subsequently assigned). New vertices are always added as isolates (i.e., without existing ties), and any edges having a deleted vertex as an endpoint are removed along with the deleted vertex.

The use of `is.adjacent` (and friends) to perform adjacency testing has been shown above. While this is adequate for many purposes, it is sometimes necessary to examine an edge's contents in detail. As we have seen, each edge can be thought of as a list made up of a vector of tail vertex IDs, a vector of head vertex IDs, and a vector of attributes. The utility function `get.edges` retrieves edges in this form, returning them as lists with elements `inl` (tail), `outl` (head), and `atl` (attributes). `get.edges` allows for edges to be retrieved by endpoint(s), and is usable even on multiplex networks. Incoming or outgoing edges (or both) can be selected, as per the following example:

```
> #Retrieving edges
> get.edges(net,1)                                  #Out-edges sent by vertex 1
```

```
[[1]]
[[1]]$inl
[1] 2

[[1]]$outl
[1] 1

[[1]]$atl
[[1]]$atl$na
[1] FALSE
```

```
> get.edges(net,2,neighborhood="in")                #In-edges to vertex 2
```

```
[[1]]
[[1]]$inl
[1] 2

[[1]]$outl
[1] 4

[[1]]$atl
[[1]]$atl$na
[1] FALSE



[[2]]
[[2]]$inl
[1] 2

[[2]]$outl
[1] 3

[[2]]$atl
[[2]]$atl$na
[1] FALSE



[[3]]
[[3]]$inl
[1] 2

[[3]]$outl
[1] 1
```

```
[[3]]$atl
[[3]]$atl$na
[1] FALSE
```

```
> get.edges(net,1,alter=2)                          #Out-edges from 1 to 2
```

```
[[1]]
[[1]]$inl
[1] 2

[[1]]$outl
[1] 1

[[1]]$atl
[[1]]$atl$na
[1] FALSE
```

The `alter` argument in the last case tells `get.edges` to supply only edges from vertex 1 to vertex 2. As with other applications of `get.edges`, this will return all applicable edges in the multiplex case.

Retrieving edges themselves is useful, but does not provide the edges' ID information – particularly in multiplex networks, such information is needed to delete or modify edges. For that purpose, we employ a parallel routine called `get.edgeIDs`:

```
> #Retrieving edge IDs
> get.edgeIDs(net,1)                          #Same as above, but gets ID numbers
```

```
[1] 4
```

```
> get.edgeIDs(net,2,neighborhood="in")
```

```
[1] 7 5 4
```

```
> get.edgeIDs(net,1,alter=2)
```

```
[1] 4
```

By the same token, it is sometimes the vertex neighborhood (rather than edge neighborhood) which is of interest. The `get.neighborhood` function can be used in these cases to obtain vertex neighborhoods directly, without having to first query edges. (Since this operation is implemented in the underlying compiled code, it is considerably faster than an R-level front end would be.)

```
> #Vertex neighborhoods
> get.neighborhood(net,1)                                    #1's out-neighbors

[1] 2

> get.neighborhood(net,2,type="in")                          #2's in-neighbors

[1] 4 3 1
```

Finally, we note that edge deletion can be performed either by assignment operators (as noted above) or by the `delete.edges` function. `delete.edges` removes edges by ID, and hence is not primarily employed by end users. In conjunction with tools such as `get.edgeIDs`, however, it can be seen to be quite versatile. A typical example is as follows:

```
> #Deleting edges
> net[2,3]<-0                                                #This deletes the 2->3
>                                                            #edge
> net[2,3]==0                                                #Should be TRUE

[1] TRUE

> delete.edges(net,get.edgeIDs(net,2,neighborhood="in")) #Remove all->2
> net[,]

     1 2 3 5 <NA> <NA>
1    0 0 0 0    0    0
2    0 0 0 0    0    0
3    0 0 0 0    0    0
5    0 0 0 0    0    0
<NA> 0 0 0 0    0    0
<NA> 0 0 0 0    0    0
```

Since it works by IDs, it should be noted that `delete.edges` can be used to selectively remove edges from multiplex networks. The operator-based approach automatically removes any edges connecting the selected pair, and is not recommended for use with multiplex networks.

3.5. Working with attributes

A major advantage of `network` objects over simple matrix or list based data representations is the ability to store meta-information regarding vertices, edges, or the network as a whole. For each such attribute type, **network** contains access functions to manage the creation,

modification, and extraction of such information. Here, we briefly introduce the primary functions used for these tasks, by attribute type.

Network attributes

As indicated previously, network-level attributes are those attached to the `network` object as a whole. Such attributes are created via the `set.network.attribute` function, which takes as arguments the object to which the attribute should be attached, the name of the attribute, and the value of the attribute in question. Network attributes may contain arbitrary data, as they are stored internally via generalized vectors (`lists`). To streamline the creation of such attributes, the network attribute operator, `%n%`, has also been provided. Assignment using the operator is performed via the syntax `network %n% "attrname" <- value`, as in the second portion of the example below (which assigns the first seven lowercase letters to an attribute called "hoo" in `net`).

```
> net <- network.initialize(5)
> set.network.attribute(net, "boo", 1:10)
> net %n% "hoo" <- letters[1:7]
```

After network attributes have been created, they may be listed using the `list.network.attributes` command. Attribute extraction may then be performed by a call to `get.network.attribute`, or via the network attribute operator. In the latter case, a call of the form `network %n% "attrname"` returns the value of attribute "attrname" in the object "network." In our current example, for instance, we have created the attributes "boo" and "hoo," each of which may be accessed using either method:

```
> #List attributes
> list.network.attributes(net)
```

```
[1] "bipartite" "boo"       "directed"  "hoo"       "hyper"     "loops"
[7] "mnext"     "multiple"  "n"
```

```
> #Retrieve attributes
> get.network.attribute(net,"boo")
```

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

```
> net %n% "hoo"
```

```
[1] "a" "b" "c" "d" "e" "f" "g"
```

Finally, it is sometimes desirable to remove network attributes which have been created. This is accomplished using `delete.network.attributes`, which removes the indicated attribute from the network object (freeing the associated memory). One can verify that the attribute has been removed by checking the list of network attributes, e.g:

```
> #Delete attributes
> delete.network.attribute(net,"boo")
> list.network.attributes(net)
```

```
[1] "bipartite" "directed"  "hoo"       "hyper"     "loops"     "mnext"
[7] "multiple"  "n"
```

Vertex attributes

Vertex attributes are manipulated in the same general manner as network attributes, with the caveat that each vertex can have its own attributes. There is no requirement that all vertices have the same attributes, or that all attributes of a given name contain the same data type; however, not all extraction methods work well in the latter case. Complete functionality for arbitrary vertex creation, listing, retrieval, and deletion is provided by the `set.vertex.attribute`, `list.vertex.attributes`, `get.vertex.attribute`, and `delete.vertex.attribute` methods (respectively). These allow attribute data to be passed in list form (permitting arbitrary contents) and to be assigned to specific vertices. While the generality of these functions is helpful, they are cumbersome to use for simple tasks such as assigning scalar or character values to each vertex (or retrieving the same). To facilitate such routine tasks, **network** provides a vertex attribute operator, `%v%`. The operator may be used either for extraction or assignment, treating the right-hand value as a vector of attribute values (with the ith element corresponding to the ith vertex). By passing a `list` with a `list` for each element, one may assign arbitrary vertex values in this manner; however, the vertex operator will vectorize these values upon retrieval (and hence one must use `get.vertex.attribute` with `unlist = FALSE` to recover the full list structure). If a requested attribute is unavailable for a particular vertex, an `NA` is returned.

Typical use of the vertex attribute methods is illustrated via the following example. Note that more complex usage is also possible, as detailed in the package manual.

```
> #Add vertex attributes
> set.vertex.attribute(net,"boo",1:5)              #Create a numeric attribute
> net %v% "hoo" <- letters[1:5]                    #Now, a character attribute
> #Listing attributes
> list.vertex.attributes(net)                      #List all vertex attributes
```

```
[1] "boo"          "hoo"          "na"           "vertex.names"
```

```
> #Retrieving attributes
> get.vertex.attribute(net,"boo")                  #Retrieve 'em
```

```
[1] 1 2 3 4 5
```

```
> net %v% "hoo"
```

```
[1] "a" "b" "c" "d" "e"
```

```
> #Deleting attributes
> delete.vertex.attribute(net,"boo")              #Remove one
> list.vertex.attributes(net)                     #Check to see that it's gone
```

```
[1] "hoo"          "na"           "vertex.names"
```

Edge attributes

Finally, we come to edge attributes. The operations involved here are much like those for the network and vertex cases. List, set, get, and delete methods exist for edge attributes (`list.edge.attributes`, `set.edge.attribute`, `get.edge.attribute`, and `delete.edge.attribute`), as does an edge attribute operator (`%e%`). Operations with edges are rendered somewhat more complex, however, because of the need to employ edge IDs in referencing the edges themselves. These can be obtained via the `get.edgeIDs` function (as described above), but this adds complexity which is unnecessary in the case of simple attribute assignment on non-multiplex, dyadic graphs (where edges are uniquely identifiable by a pair of endpoints). For such cases, the convenience function `set.edge.value` allows edge values to be specified in adjacency matrix form. Also useful is the bracket operator, which can be used to assign values as well as to create edges. For network `net`, `net[sel, names.eval = "attrname"] <- value` will set the attribute named by "attrname" on the edges selected by `sel` (which follows standard R syntax for selection of cells from square matrices) to the values in `value`. By default, values for non-existent edges are ignored (although new edges can be created by adding `add.edges = TRUE` to the included arguments). Reasonable behavior for non-scalar values using this method is not guaranteed.

In addition to the above, methods such as `as.sociomatrix` allow for edge attributes to be employed in some settings. These provide a more convenient (if less flexible) interface for the common case of scalar attributes on the edges of non-multiplex, dyadic networks. The following is a typical example of these routines in action, although much more exotic scenarios are certainly possible.

```
> #Create a network with some edges
> net <- network(nmat)
> #Add attributes
> set.edge.attribute(net,"boo",sum(nmat):1)
> set.edge.value(net,"hoo",matrix(1:25,5,5)) #Note: only sets for extant edges!
> net %e% "woo" <- matrix(rnorm(25),5,5)     #Ditto
> net[,,names.eval="zoo"] <- nmat*6           #Ditto if add.edges!=TRUE
> #List attributes
> list.edge.attributes(net)
```

```
[1] "boo" "hoo" "na"  "woo" "zoo"
```

```
> #Retrieving attributes
> get.edge.attribute(get.edges(net,1),"boo") #Get the attribute for 1's out-edges

[1] 3 7

> get.edge.value(net,"hoo")

[1]  2  3 11 14 17 18 21

> net %e% "woo"

[1]  0.8984226  0.4793125  3.7056453  0.7033602 -0.6100306 -0.4825138 -1.2331196

> as.sociomatrix(net,"zoo")

  1 2 3 4 5
1 0 0 6 0 6
2 6 0 0 6 0
3 6 0 0 6 0
4 0 0 6 0 0
5 0 0 0 0 0

> #Deleting attributes
> delete.edge.attribute(net,"boo")
> list.edge.attributes(net)

[1] "hoo" "na"  "woo" "zoo"
```

As this example illustrates, edge attributes are only set for actually existing edges (although the optional `add.edges` argument to the network assignment operator can be used to force addition of edges with non-zero attribute values). Also illustrated is the difference between attribute setting using `set.edge.attribute` (which is edge ID based) and function such as the assignment operator. The relative ease of the latter recommends itself for everyday use, although more complex settings may call for the former approach.

From attributes to networks

In addition to simply storing covariate information, it should be noted that one can actively use attributes to construct new networks. For instance, consider the `emon` data set used above.

Among other variables, each vertex carries an attribute called `"Location"` which contains information on whether the corresponding organization had headquarters or command post installations which were local, non-local, or both with respect to the operation from which the network was drawn. We may thus use this information to construct a very simple hypergraph, in which locations constitute edges and edge membership is defined as having an installation at the respective location. For the Mt. St. Helens network, such a network may be constructed as follows. First, we extract the location information from the relevant network object, and use this to build an incidence matrix based on location. Then we convert this incidence matrix to a hypergraphic network object (setting vertex names from the original network object for convenience).

```
> #Extract location information
> MtSHloc<-emon$MtStHelens%v%"Location"
> #Build an incidence matrix based on Local/Non-local/Both placement
> MtSHimat<-cbind(MtSHloc%in%c("L","B"),MtSHloc%in%c("NL","B"))
> #Convert incidence matrix to a hypergraph
> MtSHbyloc<-network(MtSHimat,matrix="incidence",hyper=TRUE,directed=FALSE,
+   loops=TRUE)
> #Set vertex names, for convenience
> MtSHbyloc%v%"vertex.names"<-emon$MtStHelens%v%"vertex.names"
> #Examine the result
> MtSHbyloc


 Network attributes:
  vertices = 27
  directed = FALSE
  hyper = TRUE
  loops = TRUE
  multiple = FALSE
  bipartite = FALSE
  total edges= 2
    missing edges= 0
    non-missing edges= 2

 Vertex attribute names:
    vertex.names

No edge attributes
```

Obviously, the simple location coding used here cannot lead to a very complex structure. Nevertheless, this case serves to illustrate the flexibility of the **network** tools in allowing attribute information to be used in creative ways. In addition to constructing networks from attributes, one can use attributes to store networks (useful for joint representation of cognitive and behavioral structures such as those of Krackhardt 1988; Killworth & Bernard 1976), edge timing information (for dynamic structures), etc. Appropriate use of network, edge, and

vertex attributes allows a wide range of complex relational data structures to be supported without the need for a cumbersome array of of custom data classes.

3.6. Visualizing `network` objects

In addition to manipulating `network` objects, the **network** package provides built-in support for network visualization. This capability is supplied by the package `plot` method (ported from **sna**'s `gplot`), which is dispatched transparently when `plot` is called with a `network` object. The plot method supports a range of layout and display options, which are specified through additional arguments. For instance, to visualize the Florentine marriage data we might use commands such as the following:

```
> plot(nflo, displaylabels = TRUE, boxed.labels = FALSE)
> plot(nflo, displaylabels = TRUE, mode = "circle")
```

Typical results of these commands are shown in Figure 1. Note that the plot method automatically determines whether the network being visualized is directed, and adds or suppresses arrowheads accordingly. For instance, compare the above with the Mt. Si communication network (Figure 2):

```
> plot(emon$MtSi)
```

The default layout algorithm for the plot method is that of Fruchterman & Reingold (1991), a force-directed display with good overall performance. Other layout methods are available (including the well-known energy-minimization algorithm of Kamada& Kawai 1989), and support is included for user-added functions. To create a custom layout method, one need only create a function with the prefix `network.layout` which supplies the appropriate formal arguments (see the **network** manual for details). The `plot` method can then be directed to utilize the custom layout function, as in this simple example (shown in Figure 3):

```
> library(sna)
> network.layout.degree <- function(d, layout.par){
+       id <- degree(d, cmode = "indegree")
+       od <- degree(d, cmode = "outdegree")
+       cbind(id, od)
+     }
> plot(emon$MtStHelens, mode = "degree", displaylabels = TRUE,
+     boxed.labels = FALSE, suppress.axes = FALSE, label.cex = 0.5,
+     xlab = "Indegree", ylab = "Outdegree", label.col = 3)
```

As this example illustrates, most properties of the visualization can be adjusted where necessary. This is especially helpful when visualizing structures such as hypergraphs:

```
> plot(MtSHbyloc, displaylabels = TRUE, label =
+     c(network.vertex.names(MtSHbyloc), "Local", "Non-Local"),
+     boxed.labels = FALSE, label.cex = rep(c(0.5, 1), each = c(27, 2)),
+     label.col = rep(c(3, 4), each = c(27, 2)), vertex.col = rep(c(2, 5),
+     each = c(27, 2)))
```

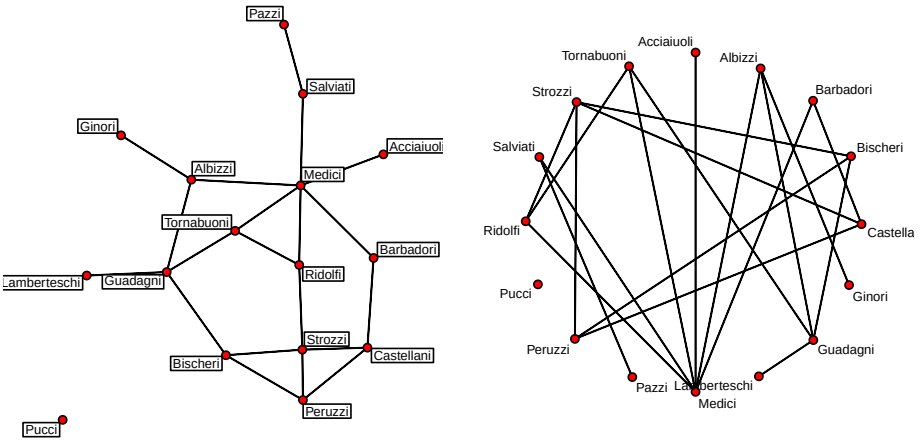

Figure 1: Sample displays of the Florentine marriage data; the left panel depicts the default Fruchterman-Reingold layout, while the right panel depicts a circular layout.

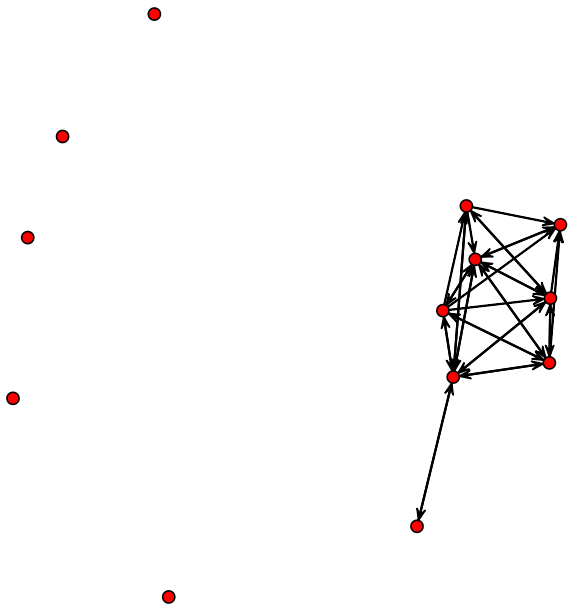

Figure 2: Sample display of the Mt. Si EMON data, using the default Fruchterman-Reingold layout.

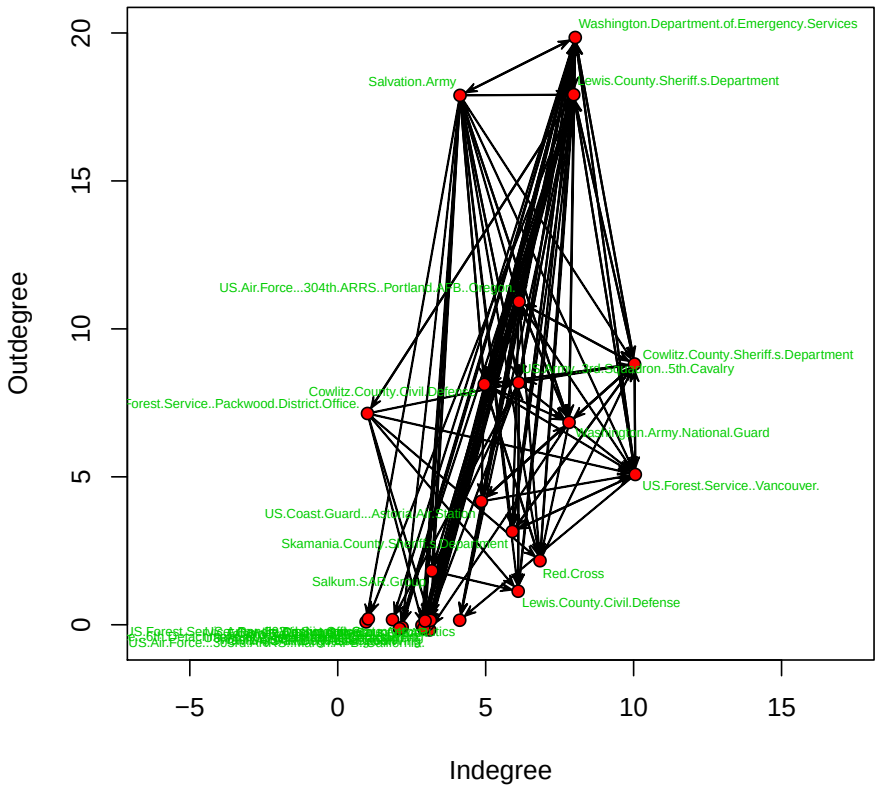

Figure 3: Sample display of the Mt. St. Helens EMON data, using a custom indegree/outdegree layout.

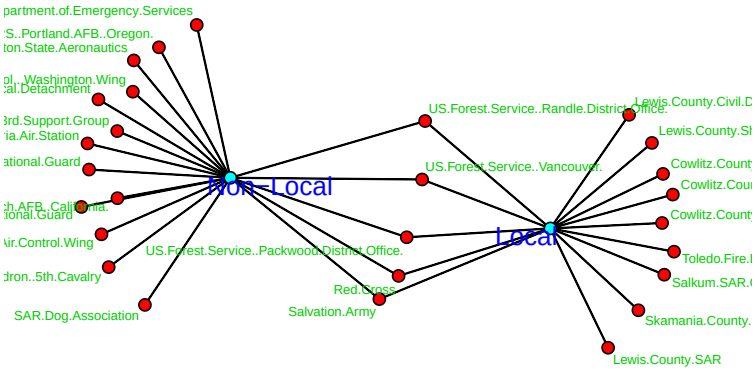

Figure 4: Sample display of the Mt. St. Helens location hypergraph, showing division between locally, non-locally, and dual headquartered organizations.

Note that the `plot` method automatically recognizes that the network being passed is hypergraphic, an employs a two-mode representation for visualization purposes (see Figure 4). Supplying custom labeling and vertex coloring helps clarify the interpretation. For instance, here we can immediately see the division between organizations who maintained headquarters exclusively at local or remote locations during the Mount St. Helens search and rescue operation, as well as those organizations (e.g. the Salvation Army and Red Cross) which bridged the two. Though simple, examples such as this demonstrate how the default *plot* settings can be adjusted to produce effective visualizations of even complex relational data.

4. C-language API

While the functionality described thus far has been aimed at users working within an interpreted R environment, many **network** package features can also be accessed through a C-language application programming interface (API). Although this API still makes use of R data structures, it provides mechanisms for direct manipulation of those structures via

compiled code. While invisible to most end users, the API has a number of attractions for developers. Chief among these is performance: in the author's experience, a reasonably well-designed C function can run as much as one to two orders of magnitude faster than an equivalent R implementation. For many day-to-day applications, such gains are unlikely to be worth the considerable increase in implementation and maintenance costs associated with choosing C over R; however, they may prove vital when performing computationally demanding tasks such as Markov chain Monte Carlo simulation, large-graph computations, and small-N solutions for non-polynomial time problems (e.g., cycle counting). Another useful feature of the C API is its ability to make the complex data storage capabilities of `network` objects accessible to developers whose projects involve existing backend code. Instead of performing data extraction on a `network` object and passing the result to the compiled routine, the **network** API allows for such routines to work with such objects directly. Finally, a third useful asset of the **network** API is the capacity it provides for generating user-transparent functionality which transcends what is feasible with R's pass-by-value semantics. The use of compiled code to directly modify objects without copying has been fundamental to the functionality of the package since version 1.0, as can be gleaned from an examination of the package source code.

The mechanism by which the API is currently implemented is fairly simple. A shared header file (which must be included in the user's application) defines a series of macros which point to the package's internal routines. During program execution, a global registration function is used to map these macros to their internal symbols; following this, the macros may be called normally. Other then ensuring that the **network** library is loaded prior to invoking the registration function, no other measures are necessary. In particular, the calling routine does not have to be linked against the **network** library, although the aforementioned header/registration routines must be included at compile time.[3]

4.1. Using the network API

To use the **network** API within one's own code, the following steps are necessary:

1. The required **network** header and function registration files must be added to the developer's source tree.
2. The **network** header file must be included during compilation.
3. The `netRegisterFunctions` function must be invoked at the entry point to any C program using the API.
4. The **network** API functions must be used as required.

The command `netRegisterFunctions` takes and returns no arguments, being invoked solely for its side effect. Although it must be called at each entry to the C backend (i.e., each invocation of `.Call` or `.External` from R), its effects persist until the calling routine exits. The API is designed for use with the `.Call` interface, although wrappers for conversion to `.External` are in principle possible. Object references are maintained through `SEXP` pointers, as is standard for R's C language interface. Because references (rather than copies of the

[3]Required files for the **network** API are available from http://www.statnetproject.org/.

objects themselves) are passed to C via the interface, C routines may directly alter the objects with which they are called. **network** has many routines for creating and modifying `networks`, as well as for accessing object contents within compiled code. To illustrate the use of the network API in practical settings, we here provide a walk-through for a relatively simple (but non-trivial) example. Consider a C function which generates an undirected network from a homogeneous Bernoulli graph distribution, tagging each edge with random "onset" and "termination" times based on a piecewise-exponential process with fixed onset/termination hazards. Such a function might also keep track of the first and last edge times for each vertex (and for the network as a whole), storing these within the network object via appropriately named attributes.

To implement our sample function, we begin with the standard header for a `.Call` function, which both takes and receives arguments of type `SEXP` (S-expression pointers). In this case, the parameters to be passed consist of an initialized `network` object, the probability of an edge between any two vertices, and the hazards for edge onset and termination (respectively). Note that we do not need to tell the function about properties such as network size, since it can determine these itself using the API's interface methods.

```
SEXP rnbernexp_R(SEXP g, SEXP ep, SEXP oh, SEXP th)
/*
C-Language code for a simple random dynamic network generator.  Arguments are
as follows:
  g - a pre-initialized network object
  ep - the edge probability parameter
  oh - the edge onset hazard parameter
  th - the edge termination hazard parameter
*/
{
  int n, i, w;
  double u, fet, let, *vfet, *vlet, ot, tt;
  SEXP tail, head, atl, atlnam, sot, stt, ec;

  /*Verify that we were called properly, and set things up*/
  netRegisterFunctions();
  if(!netIsNetwork(g))
    error("rnbernexp_R must be called with a network object.\n");
  if(netIsDir(g))
    error("Network passed to rnbernexp_R should be undirected.\n");
  n = netNetSize(g);
  PROTECT(ep = coerceVector(ep, REALSXP));
  PROTECT(oh = coerceVector(oh, REALSXP));
  PROTECT(th = coerceVector(th, REALSXP));
  PROTECT(ec = allocVector(LGLSXP, 1));
  LOGICAL(ec)[0] = 0;
  GetRNGstate();

  /*Allocate memory for first/last edge time trackers*/
  vfet = (double *)R_alloc(n, sizeof(double));
```

```
vlet = (double *)R_alloc(n, sizeof(double));
for(i = 0; i < n; i++)
  vfet[i] = vlet[i] = NA_REAL;
fet = let = NA_REAL;
```

In order to assure that all arguments are of the appropriate type, we employ a combination of verification and coercion. After registering the **network** API functions using `netRegisterFunctions`, we use the indicators `netIsNetwork` and `netIsDir` to verify that the `g` argument is indeed a `network` object, and that it is undirected. After verifying these conditions, we can use `netNetSize` to obtain the number of vertices in the network. This quantity is saved for further use.

With the preliminaries out of the way, we are now in a position to draw edges. The algorithm used to generate the underlying graph is that of Batagelj & Brandes (2005), which scales well for sparse graphs (complexity is $\mathcal{O}(n+m)$). Edges themselves are added via the `netAddEdge` API function, which is analogous to `add.edge` in the R interface. Because we are operating directly on the network object, we must handle memory allocation ourselves: the `allocVector` calls in the following section are used to allocate memory for the head, tail, and attribute lists, and for the vector of attribute names. These are set accordingly, with the "OnsetTime" and "TerminationTime" attributes being created to store edge onsets and terminations, respectively. Once the edge elements are created, `netAddEdge` assures that they are placed within the `network` object; since R's garbage collection mechanism protects these elements once they are linked to `g` (which is a protected object), we can subsequently remove them from the memory protection stack using `UNPROTECT`.

```
/*Draw the network information*/
w = -1;
i = 1;
while(i < n){
  u = runif(0.0, 1.0);
  w += 1+ (int)floor(log(1.0 - u) / log(1.0 - REAL(ep)[0]));
  while((w >= i) && (i < n)){
    w -= i;
    i++;
  }
  if(i < n){                                        /*Generate an edge*/
    /*Draw and track timing information*/
    ot = rexp(REAL(oh)[0]);
    tt = ot + rexp(REAL(th)[0]);
    fet = ((ISNA(fet)) || (ot < fet)) ? ot : fet;
    let = ((ISNA(let)) || (tt > let)) ? tt : let;
    vfet[i] = ((ISNA(vfet[i])) || (ot < vfet[i])) ? ot : vfet[i];
    vlet[i] = ((ISNA(vlet[i])) || (tt > vlet[i])) ? tt : vlet[i];
    /*Allocate memory for the new edge*/
    PROTECT(tail = allocVector(INTSXP, 1));        /*Allocate head/tail*/
    PROTECT(head = allocVector(INTSXP, 1));
    INTEGER(tail)[0] = i + 1;
    INTEGER(head)[0] = w + 1;
```

```
    PROTECT(atl = allocVector(VECSXP, 2));          /*Allocate attributes*/
    PROTECT(sot = allocVector(REALSXP, 1));
    PROTECT(stt = allocVector(REALSXP, 1));
    PROTECT(atlnam = allocVector(STRSXP, 2));
    SET_STRING_ELT(atlnam, 0, mkChar("OnsetTime"));
    SET_STRING_ELT(atlnam, 1, mkChar("TerminationTime"));
    REAL(sot)[0] = ot;
    REAL(stt)[0] = tt;
    SET_VECTOR_ELT(atl, 0, sot);
    SET_VECTOR_ELT(atl, 1, stt);
    g = netAddEdge(g, tail, head, atlnam, atl, ec);    /*Add the edge*/
    UNPROTECT(6);
  }
}
```

At this point, all edges have been placed within the network. While we could stop here, it seems useful to first tabulate some basic meta-data regarding the network being produced. In particular, a function to analyze a network of this type would doubtless need to know the total time interval over which each vertex (and the network as a whole) is active. Via the **network** API, we can easily store this information in `g`'s network and vertex attribute lists before returning. To do this, we employ `netSetVertexAttrib` and `netSetNetAttrib`, API functions which are analogous to `set.vertex.attribute` and `set.network.attribute`. As with the case of edge addition, we must allocate memory for the attribute entry prior to installing it – the `netSet*` routines pass references to their arguments, rather than copying them – but these functions do handle the creation of attribute names from raw strings. After writing our metadata into the graph, we clear the protection stack and return the R object pointer.

```
/*Add network and vertex attributes*/
for(i = 0; i < n; i++){
  PROTECT(sot = allocVector(REALSXP, 1));
  PROTECT(stt = allocVector(REALSXP, 1));
  REAL(sot)[0] = vfet[i];
  REAL(stt)[0] = vlet[i];
  g = netSetVertexAttrib(g, "FirstOnsetTime", sot, i + 1);
  g = netSetVertexAttrib(g, "LastTerminationTime", stt, i + 1);
  UNPROTECT(2);
}
PROTECT(sot = allocVector(REALSXP, 1));
PROTECT(stt = allocVector(REALSXP, 1));
REAL(sot)[0] = fet;
REAL(stt)[0] = let;
g = netSetNetAttrib(g, "FirstOnsetTime", sot);
g = netSetNetAttrib(g, "LastTerminationTime", stt);

/*Clear protection stack and return*/
PutRNGstate();
```

```
  UNPROTECT(6);
  return g;
}
```

To use the `rnbernexp_R` function, it must be invoked from R using the `.Call` interface. A simple wrapper function (whose behavior is similar to R's built-in random number generation routines) might look like the following:

```
> rnbernexp <- function(n, nv, p = 0.5, onset.hazard = 1,
+     termination.hazard = 1){
+       nets <- list()
+       for(i in 1:n)
+         nets[[i]] <- .Call("rnbernexp_R", network.initialize(nv,
+           directed = FALSE), p, onset.hazard, termination.hazard,
+           PACKAGE = "networkapi.example")
+       if(i > 1)
+         nets
+       else
+         nets[[1]]
+     }
```

In actual use, the `PACKAGE` setting would be changed to the name of the shared object file in which the `rnbernexp_R` symbol resides. (This file would need to be linked against the `networkapi` file, and dynamically loaded after **network** is in memory. Linking against the entire **network** library is not required, however.) Although the specific distribution simulated is too simplistic to serve as a very good model of social dynamics, it nevertheless illustrates how the **network** API can be used to efficiently simulate and store the results of non-trivial processes within compiled code.

5. Final comments

For several decades, tools for social network analysis were essentially isolated from those supporting conventional statistical analyses. A major reason for this isolation was the difficulty in manipulating – or even representing – relational data within standard statistical packages. In recent years, the emergence of flexible statistical computing environments such as R have helped to change this situation. Platforms like R allow for the creation of the complex data structures needed to represent rich relational data, while also facilitating the development of tools to make such structures accessible to the end user. The **network** package represents one attempt to leverage these capabilities in order to create a low-level infrastructure for the analysis of relational data. Together with packages like **sna**, **ergm**, and the rest of the **statnet** suite, it is hoped that **network** will provide a useful resource for scientists both inside and outside of the social network community.

Acknowledgments

The author gratefully acknowledges the input of present and past **statnet** collaborators, including Mark Handcock, David Hunter, Daniel Westreich, Martina Morris, Steve Goodreau,

Pavel Krivitsky, and Krista Gile. 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

Batagelj V, Brandes U (2005). "Efficient Generation of Large Random Networks." *Physical Review E*, 71(3), 036113, 1-5. doi:10.1103/PhysRevE.71.036113.

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/.

Butts CT (2002). "Memory Structures for Relational Data in R: Classes and Interfaces." *Unpublished manuscript*, University of California, Irvine.

Butts CT (2007). *sna: Tools for Social Network Analysis*. Statnet Project http://statnetproject.org/, Seattle, WA. R package version 1.5, URL http://CRAN.R-project.org/package=sna.

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.

Carey VJ, Long L, Gentleman R (2007). *RBGL: R Interface to Boost C++ Graph Library*. R package version 1.14.0, URL http://www.bioconductor.org/.

Chambers JM (1998). *Programming with Data*. Springer-Verlag, New York. ISBN 0-387-98503-4.

Csardi G, Nepusz T (2006). "The igraph Software Package for Complex Network Re- search." *InterJournal, Complex Systems*, 1695. URL http://www.interjournal.org/manuscript_abstract.php?361100992.

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

Drabek TE, Tamminga HL, Kilijanek TS, Adams CR (1981). *Managing Multiorganizational Emergency Responses: Emergent Search and Rescue Networks in Natural Disaster and Remote Area Settings*. Number Monograph 33 in Program on Technology, Environment, and Man. Institute of Behavioral Sciences, University of Colorado, Boulder, CO.

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

Gentleman R, Whalen E, Huber W, Falcon S (2007). *graph: A Package to Handle Graph Data Structures*. R package version 1.14.2, URL http://CRAN.R-project.org/package=graph.

Gentry J, Long L, Gentleman R, Falcon S (2007). *Rgraphviz: Plotting Capabilities for R Graph Objects*. R package version 1.16.0, URL http://CRAN.R-project.org/package=Rgraphviz.

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.

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

Killworth PD, Bernard HR (1976). "Informant Accuracy in Social Network Data." *Human Organization*, 35(8), 269-286.

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

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

Mayhew BH, Levinger RL (1976). "Size and Density of Interaction in Human Aggregates." *American Journal of Sociology*, 82, 86-110.

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/.

Venables WN, Ripley BD (2000). *S Programming*. Springer-Verlag, New York. ISBN 0-387-98966-8.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. Springer-Verlag, New York, fourth edition. ISBN 0-387-95457-0.

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