Design and implementation of a portable C library for network analysis and disease simulation

Matthew Vernon

January 25, 2007

Introduction

A wide range of software exists for social network analysis. A recent review attempted to compare some of the available packages, but found it difficult to do so fairly, due to the differing emphases of the various authors (Huisman and van Duijn, 2005). That review showed, however, that relatively little social network analysis software was available for platforms other than Microsoft Windows, and that only three of the smaller, more specialised applications were free software. Furthermore, most of the available software packages only provided a graphical user interface (GUI), and so cannot be usefully automated for processing multiple networks; similarly, many of them were conceived for use with relatively small social networks, and do not handle large networks well. Two software packages, in particular, are sufficiently widely-used that it is worthwhile to discount them here. UCINET (Borgatti et al., 2002) is a commercial package for the analysis of social network data. It will only run on Microsoft Windows, and only provides a GUI, making automation difficult. Furthermore, it is only designed to deal with small networks (up to 32,767 nodes). On the other hand, Pajek (Batagelj and Mrvar, 1998) is designed to deal effectively with large graphs, and has some support for automation. It costs nothing, but is not free software (this distinction is discussed below). Additionally, it is only available for Microsoft Windows. It can be made to run on GNU/Linux systems using a Windows emulator, but that still restricts it to Intel i386-based hardware. This limits its utility on all but the most recent Apple hardware, as well as most UNIX servers; specifically, all the analysis presented in this thesis was performed on a Macintosh Powerbook G4 and/or a Sun V440, neither of which can run Pajek.

Free software is not the same as software that costs nothing to acquire. Colloquially, free software is "free" as in "free speech", not as in "free beer". More specifically, free software grants its users freedom to run the program for any purpose, to study its workings and adapt them to their needs, to redistribute copies to others, and to improve it and release such improvements to the wider community. A widely-accepted definition of free software is the Debian Free Software Guidelines[1]. Free software has several advantages that make it suitable for use for scientific computing. Free availability means that results may be verified by other authors. The availability of source code means that other authors may not only verify the results of any work, but may also verify the workings of a piece of software. The freedom to modify and redistribute free software enables enhancements of algorithms (and bug-fixes) to be disseminated to the wider scientific community, and avoids duplication of effort. For these reasons, free software should be considered the gold standard for applications used for scientific research. Accordingly, the software for network analysis and disease simulation described in this chapter (and included on a CD with this thesis) is released as free software under the GNU General Public Licence, version 2.

The software described here allows a range of different network analysis techniques to be deployed, networks to be stored in memory in three different representations, the generation of networks according to different criteria, and the simulation of disease processes upon networks. It understands, and may output, networks in a range of formats understood by other network analysis programs, is portable across a range of computing platforms, and is written in ANSI C (Kernighan and Ritchie, 1988) conforming to the latest international standard for the language, BS ISO/IEC 9899:1999 ("C 99"). It aims to combine efficient algorithms with code that is well structured and easy to read (and hence maintain).

[1] Available online at `http://www.debian.org/social_contract#guidelines`

Library structure

The library functions and data structures are defined in a series of header files, which allows the same definitions to be used by multiple source files. Many applications, however, need only `#include` the `gsalgs.h` header, as this `#includes` `gens.h`, so giving an application access to the complete set of functions available, and the general network structure interface. Applications wishing to refer to the particular network representations (rather than the general interface) may `#include` the header files `g_im.h` (for integer matrices), `g_bm.h` (for bit matrices) and `g_al.h` (for the list of adjacency lists) directly.

There are three common ways of representing a network with N nodes and E edges in memory. The first of these is an integer matrix, which is an N by N array of integers. Edges may be valued, and memory usage is order N^2. A refinement of this approach is the bit-matrix, which is a two-dimensional array of type `unsigned char` large enough to contain N by N bits. In this representation, edges may not have value, but memory use is substantially less (depending on the width of the integer type), although still of order N^2. Finally, a network may be represented by an array of adjacency lists (these being arrays of the nodes which are adjacent to the node in question); memory use is of order E, but establishing whether an edge $u \rightarrow v$ exists is slower, depending on the number of neighbours u has; on the other hand, enumerating the neighbours of a particular node is faster than with the matrix-based methods.

These different representations are best suited to different tasks, but for many networks, the particular representation used does not especially matter, and the user may not care which is deployed. Accordingly, the library provides a largely object-oriented approach to the network representations; networks are presented to the application as a general network structure, containing a set of basic functions (described below); the available analysis and disease simulation functions all accept one of these general network structures as an argument. This provides a convenient interface for the application developer, with minimal computational overhead, as well as allowing the developer to be more specific about which representation they desire if they wish to be.

The general network structure

This structure is defined in `gens.h` as follows:

```
struct gennet{
  const struct gennetvtable *vtable;
};

typedef int gtest_fn(const struct gennet *g, const int x, const int y);
typedef int gset_fn(struct gennet *g, const int x, const int y);
typedef int gsetval_fn(struct gennet *g, const int x, const int y,
                       const int val);
typedef void gunset_fn(struct gennet *g, const int x, const int y);
typedef int giter_fn(const struct gennet *g, const int node, const int x);
typedef void gsort_fn(struct gennet *g);
typedef void gfree_fn(struct gennet *g);

struct gennetvtable{
  const net_desired_t type;
  gtest_fn *test;
  gset_fn *set;
```

```
  gsetval_fn *setval;
  gunset_fn *unset;
  giter_fn *next_neighb;
  gsort_fn *sort;
  gfree_fn *free;
};
```

The `struct gennetvtable` contains a variable describing the type of representation, and then a series of pointers to functions that perform basic operations: `test()` returns non-zero if an edge exists between `x` and `y`, `set()` creates an edge between `x` and `y` (returning 0 if the edge already existed, otherwise 1), `setval()` creates an edge of value `val` between `x` and `y` (it is an error to call this function on anything other than an integer matrix network), `unset()` removes the edge between `x` and `y`, `next_neighb()` returns the next neighbour of `node` after `x` or -1 if no such neighbour exists (this is useful for looping through the neighbours of a node), `sort()` sorts all the adjacency lists of a network (and does nothing for the two matrix representations), and `free()` frees all the memory associated with a network structure. As an example, the following code fragment illustrates the use of the test function to see if there is an edge from node 10 to node 20 in network `g`:

```
g->vtable->test(g,10,20);
```

The specific network representations

The library provides 5 specific network representations: the three described above, and two special immutable cases: an empty (or "null") network, and a complete (or "full") network. They all may be manipulated via the general network interface described above, and are implemented in a similar way. As an example, the integer matrix representation is considered. From `g_im.h`:

```
struct intmatrix{
  struct gennet general;
  int n;
  int **net;
};
```

The `net` member is the `n` by `n` integer array containing the network data (`n` being the number of nodes). The `general` member is the general network structure discussed above, and its `vtable` member contains a set of functions that understand the integer matrix representation. In C, the address of a structure is the same as the address of the first member of that structure, accordingly for a variable `i` of type `struct intmatrix`, `i.general` and `i` have the same address. This means that functions dealing with specific representations and those dealing with the general network interface can pass networks back and forth simply.

Internal and external node numbering

When a network is generated or loaded from a file, nodes are assigned numbers from 0 upwards; these internal node numbers will typically be different from the node numbers in a data file (referred to hereafter as external node numbers), unless it was generated by one of the output functions of this library. Functions requiring an input node (i.e. `dijkstra()`) require that the internal number of the desired node be given, and functions returning data about each node of a network will also use these internal numbers.

The edges_load() function can supply data to facilitate the conversion between external node numbers and internal node numbers. Specifically, the map[] array provides a convenient mapping from external to internal numbers; the value of map[x] is the internal number corresponding to the external node number x, or -1 if there is no such internal node. The revmap[] array provides an analogous mapping from internal to external numbers, with revmap[y] being the external node number corresponding to internal node y.

Available functions

Internally, functions that deal solely with the general network structure are kept in one source file, whilst those that need to access internals of the various specific representations are kept in another. For clarity, however, they are grouped in the header file gsalgs.h by the type of function they perform, and they will be described similarly here.

Generation functions

```
struct gennet *bitmatrix_create(const int n);
struct gennet *intmatrix_create(const int n);
struct gennet *adjlist_create(const int n);
struct gennet *fullgraph_create(const int n);
struct gennet *nullgraph_create(const int n);
typedef enum { G_BITM,G_INTM,G_ADJL,G_FULL,G_NULL,G_ANY } net_desired_t;
struct gennet *type_create(const int n, const net_desired_t type);
```

These functions are used to create new network structures. In all cases, n is the number of nodes in the resulting network. No edges are created by any of these functions, excepting fullgraph_create(), since the network resulting from this function always returns 1 when its test() function is called. The function type_create() creates a network of the type specified in the type argument, which may take one of the 6 values specified in the declaration of the net_desired_t type, except G_ANY (G_ANY may be passed to other functions discussed below to indicate that the caller has no preference as to which type of network is generated).

```
struct gennet *poisson_gen(const int n, const int e, const int sym,
                           const net_desired_t type);
struct gennet *barabasi_gen(const int x_zero, const int x,
                            const int t, const net_desired_t type);
struct gennet *dd_gen(const int e, const struct degdist *d,
                      const net_desired_t type);
```

These functions generate random graphs according to three different models. For each function, type specifies the type of representation to be used (and can take any of the 6 legal values for this type, including G_ANY, although specifying G_FULL or G_NULL is meaningless here, as neither of these network types can be usefully modified); pseudo-random numbers are generated with the drand48() function, so srand48() should be used to seed the pseudo-random number generator before calling these functions.

The function poisson_gen() generates a Poisson or Erdős-Rényi random graph with n nodes and e edges; if sym is non-zero, then the resulting edges are symmetrical (i.e. both $a \to b$ and $b \to a$ are generated), which has the effect of creating an undirected graph. The algorithm used is naïve: two nodes are chosen at random, and an edge created between them if same does

not already exist; this process is repeated until `e` edges have been generated. This approach performs as well in practice as more complex algorithms (Batagelj and Brandes, 2005).

The function `barabasi_gen()` generates a scale-free network, using a preferential attachment model. Initially, there are `x_zero` nodes and no edges. For `t` iterations, a node is added, and `x` (which must be less than or equal to `x_zero`) edges made between it and the existing nodes with a probability based on the degree of those nodes (Barabási and Albert, 1999). This results in an undirected graph with `x_zero` + `t` nodes and `x` × `t` edges. The parameters of this function are discussed in more detail on pages **??**–**??**.

The function `dd_gen()` generates a network with a two-dimensional degree distribution described by the structure `d` (the structure of which is not relevant here), which results in a network with `e` edges. The algorithm used is DEGDIST-GEN, which is described on page **??**.

```
struct gennet *dyad_rewire(struct gennet *g, const int n,
                           const dyad_count *dc);
```

This function rewires the network `g` to match the dyad census `dc`, using the algorithm REWIRE-STEP discussed on page **??**. This preserves the two-dimensional degree distribution, and number of nodes and edges in the original graph. The resulting network is in the adjacency lists array representation; if the original network is not, then it is left untouched, and a new copy in the adjacency lists array form generated; the caller must remember to `free()` the original network if necessary in this case.

Simulation function

```
void sir_net(struct gennet *g, const int n, const int istart,
             const double risk, const int remain, const int t,
             FILE *out);
```

This function performs a discrete-time stochastic SIR simulation of disease upon the network `g`, which has `n` nodes. Initially, `istart` nodes are infected, and at each time point, every edge from an infectious to a susceptible node has a chance `risk` of transmitting infection. Nodes remain infectious for `remain` iterations, and the simulation runs for `t` iterations, or until the infection has died out, whichever is sooner. Output is written to `out`. The algorithm employed is SIMULATE, discussed in detail on page **??**.

I/O functions

```
struct gennet *edges_load(FILE *f, int *size, int **map, int *maplen,
                          int **revmap, int *edges, int *dup,
                          const int bi, const int sort,
                          const net_desired_t type);
struct gennet *dl_load(FILE *f, int *size, const int binary,
                       const net_desired_t type);
```

These two functions read in network data from `f`, and if `size` is non-NULL, set the integer pointed to by `size` to be the number of nodes. As above, `type` specifies which representation to use.

The function `edges_load()` reads in a network expressed as a list of edges. Formally, each line should consist of two integers separated by any quantity of whitespace, but should not consist of more than 1023 characters. If `map` is non-NULL, then a pointer to an array describing the mapping from node numbers in the input file to the internal node numbers is placed there (the application is then responsible for freeing the associated memory), if `maplen` is non-NULL, then the integer it points to is set to the length of the `map` array, and if `revmap` is non-NULL, then a pointer to an array describing the mapping from internal node numbers to the node numbers in the input file is placed there (and again the application is then responsible for freeing the associated memory). If `edges` and `dup` are non-NULL, then the integers they point to will be set to the total number of edges read in, and the number of those which are duplicates, respectively. The number of edges in the resulting network may be calculated as `edges` $-$ `dup`. If `bi` is non-zero, then the edges are bidirectionalised, equivalent to generating an undirected network. Note that in this case, the edges $a \to b$ and $b \to a$ will be considered duplicates of each other. If `sort` is non-zero, then the adjacency lists will be sorted (if another network representation is specified, this argument has no effect); this is desirable in nearly all circumstances, as many other library functions perform better if the network has been sorted. If `type` is `G_ANY`, then the resulting network will use the list of adjacency lists representation. The following pseudocode describes the key part of the algorithm used:

```
EDGES-LOAD(file)
 1  max ← −1
 2  min ← 0
 3  n ← 1
 4  for line in file
 5       do PARSE(line,from,to)
 6          if from = to or −1 = from or −1 = to
 7            then continue
 8          big ← maximise(from, to)
 9          small ← minimise(from, to)
10          if small < min
11            then map[small .. min] ← −1
12                 min ← small
13          if big > max
14            then grow(map, max)
15                 if min > max
16                   then map[min .. big] ← −1
17                        min ← small
18                   else map[max .. big] ← −1
19          if map[to] = −1
20            then map[to] ← n
21                 ADD-NODE(G, n)
22                 n ← n + 1
23          if map[from] = −1
24            then map[from] ← n
25                 ADD-NODE(G, n)
26                 n ← n + 1
27          G[map[from]][map[to]] ← 1
```

Essentially, an array map[] is maintained, such that the value of $map[x]$ is the internal node number corresponding to node number x in the input, or -1 if that input node has yet to be allocated an internal node number. The complication with this approach is that whilst allocating a large array is quite a quick operation, assigning -1 to every member is time-consuming; an optimisation is therefore employed, where the highest and lowest node numbers yet observed in the input file are stored, and new areas of the map[] array allocated and assigned to only where necessary. This substantially increased the speed of loading input files from RADAR, where the individual node ids are high.

The first three lines set initial state. Then, the following procedure is followed for every line of the input file. It is parsed, and if it represents a self-loop, or a movement to or from location -1 (the unknown location), it is abandoned, and the next line processed (lines 5–7). Then the two nodes involved (to and $from$) are compared with the highest and lowest nodes yet encountered. If a lower node number than previously encountered is found, then all elements of the mapping array between the new minimum node number and the previous minimum are assigned the value -1 (lines 10–12); if a larger node number is encountered, then the map[] array is grown to encompass this, and then the elements between the old and new maxima are assigned the value -1 (lines 13–18). Lines 16–17 are a special case to deal with the initial conditions (when min will be the smaller node number from the first input line, and max will be -1); they assign -1 to all the elements between the two node values from the first input line, and set min to the value of $small$. Then both ends of the new edge are looked up in the

mapping array, and if they have not been encountered before (i.e. the corresponding $map[]$ element is -1), then new nodes are added to the network G, and $map[]$ updated (lines 19–26). Finally, the new edge is added to the network (line 27). Some details of the algorithm (such as error checking, and keeping a count of edges and duplicates) have been elided for clarity, but they are routine.

The function `dl_load()` is a simple parser of UCINET's "DL" file format (Borgatti et al., 2002), specifically it requires the "whole network" format, with labels removed. If `binary` is non-zero the function assumes that the input data is dichotomous (i.e. that edges in the input network are unweighted). The function parses the header line from the DL file, establishes the number of nodes, and from that calculates the size of each subsequent line. It then reads in each line, and sets the edge values indicated. If `type` is `G_ANY`, an integer matrix representation is generated.

```
int dl_output(const struct gennet *g, const int n, FILE *f);
```

This function outputs a network in UCINET's "DL" file format, specifically the "whole network" format, with labels removed, to `f`. The argument `n` is the number of nodes in the network. It returns -1 if any of its `fprintf()` calls fail; the caller should inspect `errno` for the reason why.

```
int dd_output(const struct degdist *d, FILE *f);
```

This function outputs a two-dimensional degree distribution described in `d` to `f`; each output line contains the indegree and outdegree of a node (as integers), separated by a single space character. Return value is as `dl_output()`. Note that the ordering of the nodes is by internal number - the first line corresponds to node 0, the second to node 1, and so on.

Analysis functions

```
int *dijkstra(const struct gennet *g, const int n, const int start);
long double msp(const struct gennet *g, const int n);
```

These functions relate to the calculation of shortest paths. The function `dijkstra()` calculates the shortest path to every node in a network `g` with `n` nodes, from an initial node `start` (which must be an internal node number), using Dijkstra's shortest-path algorithm, modified to use a Fibonacci heap, giving it a running time of $O(N \log N + E)$ (Cormen et al., 1989). The return value is an array of n elements, containing the distance from `start` to each node in the network (or `INT_MAX` if the nodes are disconnected), the nodes being ordered according to their internal node numbers. The function `msp()` calculates the mean shortest path between all pairs of nodes on a network, making use of `dijkstra()`. In the case of a disconnected network, it ignores pairs of nodes which are not connected (i.e. the number of pairs of nodes considered when calculating the mean is reduced by one).

```
double *between(const struct gennet *g, const int n, const int normalise);
```

This function calculates betweenness centrality on an unweighted network `g`; if `normalise` is non-zero, then the resulting value is scaled to enable comparison between networks (Freeman, 1979). The return value is an array of n elements, containing the betweenness centrality value for each node in the network, ordered according to their internal node numbers. The algorithm used runs in $O(NE)$ time (Brandes, 2001). If the input network is undirected, then the result needs to be divided by 2.

```
double cluster(const struct gennet *g, const int n, const int exclude);
```

This function calculates the clustering coefficient over the network `g` with `n` nodes; it may be used on either directed or undirected networks. Considering each node in turn, it calculates the number of edges between the neighbours of that node; the clustering coefficient for that node is then calculated, and added to a running total. Finally, the running total is divided by the number of nodes. The `exclude` argument determines how the edge case of nodes with only one neighbour is handled. If it has value 0, then such nodes are treated as having a clustering coefficient of zero, so they contribute nothing to the running total, which will be divided by the total number of nodes in the network, as per the definition of clustering coefficient (Watts, 1999). If it has a non-zero value, however, then such nodes are excluded from the final division (i.e. the running total is divided only by the number of nodes in the network with more than one neighbour). This behaviour is solely provided to mimic the incorrect behaviour of UCINET, a popular piece of social network analysis software (Borgatti et al., 2002).

```
struct degdist *dd(const struct gennet *g,const int n);
```

This function returns the two-dimensional degree distribution of the network `g` with `n` nodes, by simply iterating through all the edges in the network.

```
void dyad_census(struct gennet *g, const int n, dyad_count *dc);
void triad_census(struct gennet *g, const int n, triad_count tc);
```

These functions calculate the dyad and triad census of the network `g` with `n` nodes, and store it in `dc` and `tc` respectively. Both require a list of adjacency lists representation, and will convert `g` to this representation if necessary internally (`g` is left unchanged).

The function `dyad_census()` calculates the dyad census; rather than simply iterating over every pair of nodes, it uses in-lists and out-lists for each node (i.e. the list of nodes which send an edge to that node, and which that node sends an edge to). Starting with the second node in the network, and considering only the members of the in- and out-lists of a node which are smaller numbered nodes, M the number of mutual dyads the node participates in and A the number of asymmetric dyads the node participates in can be calculated:

$$\begin{aligned} M &= |\, in \cap out \,| \\ A &= |\, in \triangle out \,| \end{aligned}$$

that is, M is the number of elements common to both in and out, and A is the number of elements in one of in or out, but not in both. The `dyad_count` structure is defined in `census.h`; it contains three elements which are, in order, the number of mutual, asymmetric and null dyads, as 64-bit integers.

The function `triad_census()` uses an $O(E)$ algorithm, so may not be suitable for very dense graphs (Batagelj and Mrvar, 2001), since there exist $O(N^2)$ algorithms (Moody, 1998). The `triad_count` structure is defined in `census.h`; it is an array of unsigned 64-bit integers, each member containing the count of a triad type. The `triad_types` enumeration type (also defined in `census.h`) describes the order of elements in this array, for example the element at position `T003` corresponds to the count of triad type 003 (three null dyads).

```
void strong_component_count(const struct gennet *g, const int n, FILE *out);
void weak_component_count(const struct gennet *g, const int n, FILE *out);
struct gennet *bidirectionalise(const struct gennet *g, const int n);
```

The functions `strong_component_count()` and `weak_component_count()` calculate the size of each strong or weak component in the network `g` with `n` nodes, and write them out, one component to a line (its size, as an integer) to `out`. The function `bidirectionalise()` returns a version of `g` with every edge made bidirectional; counting weak components is equivalent to finding strong components in a graph that has been made undirected, so `weak_component_count()` calls `bidirectionalise()` on its input, and passes the result to `strong_component_count()`. The function `strong_component_count()` uses an improved version of Tarjan's algorithm, which deals better with sparse graphs (Nuutila and Soisalon-Soininen, 1993). It is highly recursive, so on some operating systems it may be necessary to allow the stack to grow substantially for this function to work on large graphs.

Portability

This software library has been written in ANSI C, according to the latest standard (BS ISO/IEC 9899:1999); it is expected that it should therefore be portable to any platform with a relatively modern C implementation. Specifically, it has been tested on Solaris v9, Mac OS X, and Debian GNU/Linux.

Validation

In addition to checking the output of the various routines by hand to ensure they are behaving correctly, the clustering coefficient and betweenness centrality routines were validated against an existing standard piece of social network analysis software, UCINET (Borgatti et al., 2002).

One hundred random Poisson graphs with 45 nodes and 490 edges (these numbers are based on an unpublished study of communication within the department) were generated. AutoIt, a software package for automating graphical user interfaces (Bennett et al., 2004) was used to make UCINET calculate the clustering coefficient and betweenness centrality values for each of the hundred networks. The same measures were calculated using the software library described herein. A `bash` script was used to convert the outputs from the two pieces of software into a common format, and then `diff` was used to compare the two sets of results. A PC running Windows XP was used to run UCINET, whilst the software described here was run on both a Sun V440 running Solaris v9, and a Macintosh Powerbook G4 running Mac OS X.

The software based on this library produced the same answers on both the V440 and the Powerbook. The automation of UCINET was not robust, however, requiring significant manual intervention to successfully analyse all one hundred graphs. As noted above, UCINET incorrectly calculates the clustering coefficient; the `cluster()` function produces the same answers as UCINET when its `exclude` parameter is set to 1. The results for betweenness centrality were identical between the two sets of software, with the exception that UCINET scales its results between 0 and 100, rather than 0 and 1, so its answers were 100-fold greater than those produced by the `between()` function.

Discussion

Despite the difficulties in automating UCINET (limiting the latter's usability somewhat), validating some of the functions written against their equivalents in UCINET was a valuable exercise; it illustrated one of the problems of proprietary software as a scientific tool — that you cannot

be sure exactly what it is doing, particularly regarding "edge cases" such as how to treat nodes with only one neighbour when calculating the clustering coefficient.

The drive behind development of the library described here was the research work that is presented in this thesis. The `edges_load()` function was used to import network data from RADAR, and `poisson_gen()`, `barabasi_gen()`, and `dd_gen()` (the latter followed by `dyad_rewire()`) were then used to generate model networks with similar properties to the RADAR network; `sir_net()` was then used to run disease simulations upon the generated networks. The functions `dd()` and `dd_output()` as well as `dyad_census()` were used to analyse various of the networks generated. These uses (and the results generated thereby) are discussed more fully in subsequent chapters.

The software library presented here is a robust, portable library for scientists intending to write social network analysis programs, particularly if they wish to tie this analysis into disease simulations. It is to be released as free software along with this thesis; this will maximise its utility to the scientific community, as well as enabling effective peer-review of the algorithms deployed. It is designed to be readily extensible, which should facilitate its use in future research programmes.

Bibliography

Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286:509–512.

Batagelj, V. and Brandes, U. (2005). Efficient generation of large random networks. *Physical Review E*, 71(036113).

Batagelj, V. and Mrvar, A. (1998). Pajek - program for large network analysis. *Connections*, 21:47–57.

Batagelj, V. and Mrvar, A. (2001). A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23:237–243.

Bennett, J. et al. (2004). AutoIt v3. http://www.autoitscript.com/autoit3.

Borgatti, S. P., Everett, M. G., and Freeman, L. C. (2002). *Ucinet for Windows: Software for Social Network Analysis*. Analytic Technologies, Harvard, MA.

Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1989). *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts.

Freeman, L. C. (1979). Centrality in social networks: Conceptual clarification. *Social Networks*, 1:215–239.

Huisman, M. and van Duijn, M. A. J. (2005). *Software for Social Network Analysis*, chapter 13. Number 27 in Structural Analysis in the Social Sciences. Cambridge University Press.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall PTR, second edition.

Moody, J. (1998). Matrix methods for calculating the triad census. *Social Networks*, 20:291–299.

Nuutila, E. and Soisalon-Soininen, E. (1993). On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49:9–14.

Watts, D. J. (1999). *Small Worlds*. Princeton University Press, Princeton, New Jersey.