Octave routines for network analysis

GB

October 3, 2012

Contents

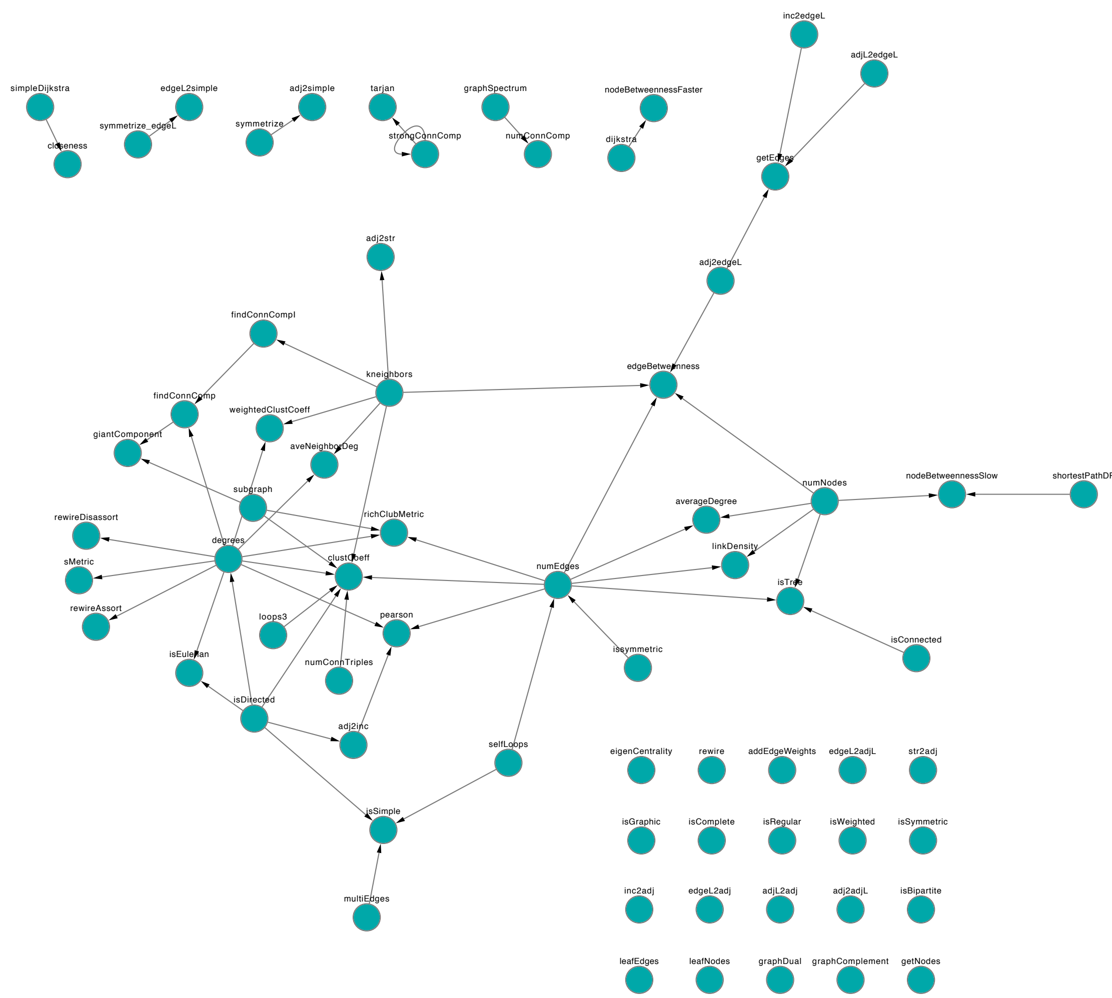


Figure 1: Graph of functions in this toolbox. An edge points from routine A to routine B if routine A is used within routine B.

0 Basic network routines

0.1 Basic network theory

A **graph** is a set of nodes, and an associated set of links between them.

Networks are instantiations of graphs. They often represent real world systems that can be modeled as a set of connected entities.

Network theory is a modern branch of **graph theory**, concerned with statistics on practical instances of mathematical graphs. Graph theory and network theory references are abundant. Social science is probably the most recent instigator of the trend to see the world as a network. In 1967, Milgram conducted his famous small world experiment [1], and found that Omahans are on average six steps away by acquaintance from Bostonians. Other prominent first sources are Price's work on the graph of scientific citations in 1965 [2] and in 1998, Watts and Strogatz's paper on dynamics of small-world networks [3].

Nowadays, there is no shortage of books and reviews on networks. Below is a non-exhaustive list of good reads [4] [5] [6] [7].

- S. Wasserman and K. Faust, Social network analysis, Cambridge University Press, 1994
- Duncan J. Watts, Six degrees: The science of a Connected Age, W. W. Norton, 2004
- M. E. J. Newman, The structure and function of complex networks, SIAM Review 45, 167-256 (2003)
- Alderson D., Catching the Network Science Bug: ..., Operations Research, Vol. 56, No. 5, Sep-Oct 2008, pp. 1047-1065

Here are some basic notions about graphs that are useful to understand the routines in Section 0.2.

Figure 1 illustrates a general **directed** graph. The nodes are functions from this toolbox. An edge points from function A to function B if *function A is called within function B*. For example, *strongConnComp* is used within *tarjan*. Notice, also that *strongConnComp* points to itself, i.e. *strongConnComp* contains a recursion. Stand-alone functions, that use no other function, are **single nodes** in the graph, such as *leafNodes*, *getEdges* and *graphDual*.

A **directed graph** is a graph in which the links have a direction. In the functions graph one function can call another, but the call is usually not reciprocated.

A **single node** is a node without any connections to other nodes. *graphDual* is an example of a single node in Figure 1.

A **self-loop** is an edge which starts and ends at the same node. (*strongConnComp*→*strongConnComp*) is an example of a self-loop.

Multiedges are two or more edges which have the same origin and destination pair of nodes. This can be useful in some graph representations. In the functions graph this is equivalent to some function being called twice inside another function.

Basic graph statistics are the **number of nodes** (n) and the **number of edges** (m). The functions graph has 69 nodes and 58 edges.

The **link density** is derived directly from the number of nodes and number of edges: it is the number of edges, divided by the maximum possible number of edges.

$$density = \frac{m}{n(n-1)/2} \tag{1}$$

For the functions graph, the link density is about 0.02.

The **average nodal degree** is the average number of links per node. This is calculated as $2m/n$ (every edge is counted twice towards the total sum of degrees).

$$average\ degree = \frac{2m}{n} \tag{2}$$

The functions graph has 1.68 links per function on average.

A graph S is a **subgraph** of graph G, if the set of nodes (and edges) of S is subset of the set of nodes (and edges) of graph G.

A **disconnected** graph is a graph in which there are two nodes between which there exists no path of edges. In the functions graph there is no path between *rewire* and *subgraph*. So the functions graph is disconnected. Disconnected graphs consist of multiple connected components. The largest connected component (in number of nodes) is usually called the **giant component**.

In the context of **directed graphs**, the notion of strong and weak connectivity is important. A **strongly connected graph** is a graph in which there is a path from every node to every other node, where paths respect link directionality. In Figure 1, for example, there is a path from *strongConnComp* to *tarjan*, but no path in reverse. Therefore, the component (*strongConnComp*,*tarjan*) is not strongly connected. If, however, link directionality is disregarded, this subgraph is certainly connected. A **weakly connected graph** or subgraph is a graph which is connected if considered as undirected, but not connected if link directionality is taken into account.

0.2 Routines

0.2.1 getNodes.m

Returns the list of nodes for varying graph representations.

```
% Returns the list of nodes for varying graph representation types
% Inputs: graph structure (matrix or cell or struct) and type of
%                                                structure (string)
%        'type' can be: 'adj','edgelist','adjlist' (neighbor list),
%                                          'inc' (incidence matrix)
% Note 1: only the edge list allows non-consecutive node indexing
% Note 2: no build-in error check for graph structure
%
% Example representations of a directed triangle: 1->2->3->1
%           'adj' - [0 1 0; 0 0 1; 1 0 0]
%           'adjlist' - {1: [2], 2: [3], 3: [1]}
%           'edgelist' - [1 2; 2 3; 3 1] or [1 2 1; 2 3 1; 3 1 1]
%                                           (1 is the edge weight)
%           'inc' - [-1  0  1
%                     1 -1  0
%                     0  1 -1]
%
% GB: last updated, Sep 18 2012
```

0.2.2 getEdges.m

Returns the list of edges for varying graph representations.

```
% Returns the list of edges for graph varying representation types
% Inputs: graph structure (matrix or cell or struct) and type of
%                                               structure (string)
% Outputs: edge list, mx3 matrix, where the third column is
%                                                      edge weight
%
% Note 1: 'type' can be: 'adj','edgelist','adjlist' (neighbor list),
%                                          'inc' (incidence matrix)
% Note 2: symmetric edges will appear twice, also in undirected
%                                graphs, (i.e. [n1,n2] and [n2,n1])
% Other routines used: adj2edgeL.m, adjL2edgeL.m, inc2edgeL.m
%
% Example representations of a directed triangle: 1->2->3->1
%           'adj' - [0 1 0; 0 0 1; 1 0 0]
%           'adjlist' - {1: [2], 2: [3], 3: [1]}
%           'edgelist' - [1 2; 2 3; 3 1] or [1 2 1; 2 3 1; 3 1 1]
%                                           (1 is the edge weight)
%           'inc' - [-1  0  1
%                     1 -1  0
%                     0  1 -1]
%
% GB: last updated, Sep 18 2012
```

0.2.3 numNodes.m

Number of vertices/nodes in the network.

```
% Returns the number of nodes, given an adjacency list, or adjacency matrix
% INPUTs: adjacency list: {i:j_1,j_2 ..} or adjacency matrix, ex: [0 1; 1 0]
% OUTPUTs: number of nodes, integer
%
% GB: last update Sep 19, 2012

function n = numNodes(adjL)

n = length(adjL);
```

0.2.4 numEdges.m

Number of edges/links in the network.

```
% Returns the total number of edges given the adjacency matrix
% INPUTs: adjacency matrix, nxn
% OUTPUTs: m - total number of edges/links
%
% Note: Valid for both directed and undirected, simple or general graph
% Other routines used: selfloops.m, issymmetric.m
% GB, last updated Sep 19, 2012
```

0.2.5 linkDensity.m

The density of links of the graph. $Density = \frac{m}{n(n-1)/2}$ (n is the number of nodes and m is the number of edges).

```
% Computes the link density of a graph, defined as the number
%    of edges divided by number_of_nodes(number_of_nodes-1)/2
%    where the latter is the maximum possible number of edges.
%
% Inputs: adjacency matrix, nxn
% Outputs: link density, a float between 0 and 1
%
% Note: The graph has to be non-trivial (more than 1 node).
% Other routines used: numNodes.m, numEdges.m
% GB: last update Sep 19, 2012
```

0.2.6 selfLoops.m

Number of selfloops, i.e. nodes connected to themselves.

```
% Counts the number of self-loops in the graph
%
% INPUT: adjacency matrix, nxn
% OUTPUT: integer, number of self-loops
%
% Note: in the adjacency matrix representation
%    loops appear as non-zeros on the diagonal
% GB: last updated, Sep 20 2012
```

0.2.7 multiEdges.m

An edge counts towards the multi-edge total if it shares origin and destination nodes with another edge.

```
% Counts the number of multiple edges in the graph
% Multiple edges here are defined as two or more edges
%      that have the same origin and destination nodes.
% Note 1: This creates a natural difference in counting
%                   for undirected and directed graphs.
%
% INPUT: adjacency matrix, nxn
% OUTPUT: integer, number of multiple edges
%
% Examples: multiEdges([0 2; 2 0])=2, and
%           multiEdges([0 0 1; 2 0 0; 0 1 0])=2
%
% Note 2: The definition of number of multi-arcs
%     (node pairs that have multiple edges across them)
%     would be: mA = length(find(adj>1)) (normalized by
%      2 depending on whether the graph is directed)
%
% GB: last updated, Sep 20 2012
```

0.2.8 averageDegree.m

The average degree (# links) across all nodes. Defined as: $\frac{2m}{n}$, where n is the number of nodes and m is the number of edges. Also, $linkDensity = \frac{averageDegree}{n-1}$.

```
% Computes the average degree of a node in a graph, defined as
%          2 times the number of edges divided by the number of
%               nodes (every edge is counted in degrees twice).
%
% Inputs: adjacency matrix, nxn
% Outputs: the average degree, a float between 0 and max(sum(adj))
%
% Note: The average degree is related to the link density, namely:
%  link_density = ave_degree/(n-1), where n is the number of nodes
%
% Other routines used: numNodes.m, numEdges.m
% GB: last update, September 20, 2012
```

0.2.9 numConnComp.m

Calculating the number of connected components in the graph by using the algebraic connectivity.

```
% Calculate the number of connected components
%              using the Laplacian eigenvalues
%               - counting the number of zeros
%
% INPUTS: adjacency matrix, nxn
% OUTPUTs: the number of connected components
%
% Other routines used: graph_spectrum.m
% GB: last updated: September 22, 2012
```

0.2.10 findConnComp.m

findConnCompI.m: Finds the connected component to which node "i" belongs to.

```
% Find the connected component to which node "i" belongs to
%
% INPUTS: adjacency matrix and index of the key node
% OUTPUTS: all node indices of the nodes in the same group
%          to which "i" belongs to (including "i")
%
% Note: Only works for undirected graphs.
% Other functions used: kneighbors.m
% GB: last updated, Sep 22 2012
```

findConnComp.m: Find the connected components in an undirected graph.

```
% Algorithm for finding connected components in a graph
% Note: Valid for undirected graphs only
%
% INPUTS: adj - adjacency matrix, nxn
% OUTPUTS: a list of the components comp{i}=[j1,j2,...jk]
%
```

```
% Other routines used: findConnCompI.m, degrees.m
% GB: last updated, September 22, 2012
```

0.2.11 giantComponent.m

The largest connected component in a graph. Returns the set of nodes in the largest component, as well as its adjacency matrix.

```
% Extract the giant component of a graph;
% The giant component is the largest connected component.
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: giant component matrix and node indices
%
% Other routines used: findConnComp.m, subgraph.m
% GB: last updated: September 22, 2012
```

0.2.12 tarjan.m [8][9]

tarjan.m: Returns the strongly connected components in a directed graph.

```
% Find the stronly connected components in a directed graph
% Source: Tarjan, "Depth-first search and linear graph algorithms",
%                   SIAM Journal on Computing 1 (2): 146-160, 1972
% Wikipedia description:
% http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm
%
% Input: graph, set of nodes and edges, in adjacency list format,
%        example: L{1}=[2], L{2}=[1] is a single (1,2) edge
% Outputs: set of strongly connected components, in cell array format
%
% Other routines used: strongConnComp.m
% GB: last updated, Sep 22, 2012
```

strongConnComp.m: Support function for tarjan.m.

```
% Support function for tarjan.m
% "Performs a single depth-first search of the graph, finding all
% successors from the node vi, and reporting all strongly connected
% components of that subgraph."
% See: http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm
%
% INPUTs: start node, vi;
%         graph structure (list), L
%         tarjan.m variables to update: S, ind, v, GSCC
% OUTPUTs: updated tarjan.m variables: S, ind, v, GSCC
%
% Note: Contains recursion.
% GB: last updated, Sep 22 2012
```

0.2.13 graphComplement.m

A graph with the same nodes, but "flipped" edges: where the original graph has an edge, the complement graph doesn't, and where the original graph doesn't have an edge, the complement graph does.

```
% Returns the complement of a graph.
% The complement graph has the same nodes, but edges
% where the original graph doesn't and vice versa.
%
% INPUTs: adj - original graph adjacency matrix, nxn
% OUTPUTs: complement graph adjacency matrix, nxn
%
% Note: Assumes no multiple edges
% GB: last updated, September 23, 2012
```

0.2.14 graphDual.m

The graph dual is the inverted nodes-edges graph.

```
% Finds the dual of a graph; a dual is the inverted nodes-edges graph.
% This is also called the line graph, adjoint graph or the edges adjacency.
%
% INPUTs: adjacency (neighbor) list representation of the graph (see adj2adjL.m)
% OUTPUTs: adj (neighbor) list of the corresponding dual graph and cell array of edges
%
% Note: This routine only works for undirected, simple graphs.
% GB: last updated, Sep 23, 2012
```

0.2.15 subgraph.m

```
% This function outputs the adjacency matrix of a subgraph
%      given the supergraph and the node set of the subgraph.
%
% INPUTs: adj - supergraph adjacency matrix (nxn), S - vector of subgraph node indices
% OUTPUTs: adj_sub - adjacency matrix of the subgraph (length(S) x length(S))
%
% GB: last update, September 23, 2012
```

0.2.16 leafNodes.m

Leaf nodes are nodes connected to only one other node.

```
% Return the indices of the leaf nodes of the graph, i.e. all nodes of degree 1
%
% Note 1: For a directed graph, leaf nodes are those with a single incoming edge
% Note 2: There could be other definitions of leaves, for example:
%                                                  farthest away from a root node
% Note 3: Nodes with self-loops are not considered leaf nodes.
%
% Input: adjacency matrix, nxn
% Output: indices of leaf nodes
%
% GB: last updated, Sep 23, 2012
```

0.2.17 leafEdges.m

Leaf edges are edges with only one adjacent edge.

```
% Returns the leaf edges of the graph: edges with one adjacent edge only.
%
% Note 1: For a directed graph, leaf edges are those that "flow into" a leaf node.
% Note 2: There could be other definitions of leaves, for example:
%                                                 farthest away from a root node.
% Note 3: Edges that are self-loops are not considered leaf edges.
% Note 4: Single floating disconnected edges are not considered leaf edges.
%
% Input: adjacency matrix, nxn
% Output: set of leaf edges: a (num edges x 2) matrix
%         where every row contains the leaf edge nodal indices
%
% GB: last updated, Sep 23, 2012
```

1 Diagnostic routines

These are functions that return boolean values depending on some property of the graph. They are often used by other algorithms whose output may vary with different graph types.

1.1 Routines

1.1.1 isSimple.m

A **simple graph** is a graph which contains no self-loops and no multiple edges, no directed and no weighted edges.

```
% Checks whether a graph is simple
% (undirected, no self-loops, no multiple edges, no weighted edges)
%
% INPUTs: adj - adjacency matrix, nxn
% OUTPUTs: S - a Boolean variable; true (1) or false (0)
%
% Other routines used: selfLoops.m, multiEdges.m, isDirected.m
% GB: last updated, September 23, 2012
```

1.1.2 isDirected.m

This routine checks whether a graph is directed or not.

```
% Checks whether the graph is directed, using the matrix transpose function
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: boolean variable, 0 or 1
%
% Note: one-liner alternative: S=not(issymmetric(adj));
% GB: last updated, Sep 23, 2012
```

1.1.3 isSymmetric.m

Checks whether a matrix is symmetric.

```
% Checks whether a matrix is symmetric (has to be square)
%
```

```
% INPUTS: adjacency matrix, nxn
% OUTPUTS: boolean variable, {0,1}
%
% GB: last update, Sep 23, 2012
```

1.1.4 isConnected.m

Checks whether a graph is connected.

```
% Determine if a graph is connected
% Idea by Ed Scheinerman, circa 2006,
%      source: http://www.ams.jhu.edu/~ers/matgraph/
%      routine: matgraph/@graph/isconnected.m
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: Boolean variable, 0 or 1
%
% Note: This function works only for undirected graphs.
% GB: last updated, Sep 23 2012
```

Alternative 1 to isConnected.m
If the algebraic connectivity is >0 then the graph is connected.
$a = algebraic\_connectivity(adj);$
$S = false;\ if\ a > 0;\ S = true;\ end$

Alternative 2 to isConnected.m
Uses the fact that multiplying the adjacency matrix to itself k times give the number of ways to get from i to j in k steps. If the end of the multiplication in the sum of all matrices there are 0 entries then the graph is disconnected. Computationally intensive, but can be sped up by the fact that in practice the diameter is very short compared to n, so it will terminate in order of log(n)? steps.

```
 function S=isconnected(el):

     S=false;

     adj=edgeL2adj(el);
     n=numnodes(adj); % number of nodes
     adjn=zeros(n);

     adji=adj;
     for i=1:n
         adjn=adjn+adji;
         adji=adji*adj;

         if length(find(adjn==0))==0
             S=true;
             return
         end
     end
```

Alternative 3 to isConnected.m
Find all connected components, if their number is 1, the graph is connected. Use $findConnComp.m$ 0.2.10.

1.1.5 isWeighted.m

Checks whether a graph has weighted links.

```
% Check whether a graph is weighted,
%                    i.e not all edges are 0,1.
%
% INPUTS: edge list, m x 3, m: number of edges,
%                 [node 1, node 2, edge weight]
% OUTPUTS: Boolean variable, 0 or 1
%
% GB: last updated, Sep 23, 2012
```

1.1.6 isRegular.m

A **regular graph** is a graph in which every node has the same number of links. *isRegular* checks whether a graph is regular.

```
% Checks whether a graph is regular, i.e.
% whether every node has the same degree.
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: Boolean, 0 or 1
%
% Note: Defined for unweighted graphs only.
% GB: last updated, Sep 23, 2012
```

1.1.7 isComplete.m

A **complete graph** is a graph in which all nodes are connected to all other nodes.

```
% Check whether a graph is complete, i.e.
% whether every node is linked to every other node.
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: Boolean variable, true/false
%
% Note: Only defined for unweighted graphs.
% GB: last updated, Sep 23, 2012
```

1.1.8 isEulerian.m

Find out whether a graph is **Eulerian**.
A connected undirected graph is Eulerian if and only if every graph vertex has an even degree.
A connected directed graph is Eulerian if and only if every graph vertex has equal in- and out- degree.

```
% Check if a graph is Eulerian, i.e. it has an Eulerian circuit
% "A connected undirected graph is Eulerian if and only if
%                        every graph vertex has an even degree."
% "A connected directed graph is Eulerian if and only if
%             every graph vertex has equal in- and out- degree."
% Note: Assume that the graph is connected.
%
```

```
% INPUTS: adjacency matrix, nxn
% OUTPUTS: Boolean variable, 0 or 1
%
% Other routines used: degrees.m, isDirected.m
% GB: last updated, Sep 23, 2012
```

1.1.9 isTree.m

Check whether a graph is a tree. A **tree** is a connected graph with n nodes and $(n-1)$ edges.

```
% Check whether a graph is a tree
% A tree is a connected graph with n nodes and (n-1) edges.
% Source: "Intro to Graph Theory" by Bela Bollobas
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: Boolean variable, 0 or 1
%
% Other routines used: isConnected.m, numEdges.m, numNodes.m
% GB: last updated, Sep 24, 2012
```

1.1.10 isGraphic.m

Check whether a sequence of number is graphic. A sequence of numbers is **graphic** if a graph exists with the same degree sequence [10].

```
% Check whether a sequence of number is graphical,
%    i.e. a graph with this degree sequence exists
% Source: Erds, P. and Gallai, T.
%      "Graphs with Prescribed Degrees of Vertices"
%       [Hungarian]. Mat. Lapok. 11, 264-274, 1960.
%
% INPUTs: a sequence (vector) of numbers
% OUTPUTs: boolean, true or false
%
% Note: Not generalized to directed graph degree sequences.
% GB: last updated, Sep 24, 2012
```

1.1.11 isBipartite.m

Check whether a graph is bipartite. A **bipartite graph** is a graph for which the nodes can be split into two sets, A and B, such that any given edge connects a node from A to a node from B.

```
% Test whether a graph is bipartite. If so, return the two vertex sets.
% A bipartite graph is a graph for which the nodes can be split in two
% sets A and B, such that there are no edges that connect nodes within
%                                                          A or within B.
%
% Inputs: graph in the form of adjancency list (neighbor list, see adj2adjL.m)
% Outputs: true/false (boolean), empty set (if false) or two sets of vertices
%
% Note: This only works for undirected graphs.
% GB: last updated, Sep 24, 2012
```

2 Conversion routines

2.1 Graph representations

Most succintly, a graph is a set of edges. For example, $\{(n_1, n_2), (n_2, n_3), (n_4, n_4)\}$ is a representation which stands for a 4-node graph with 3 edges, one of which is a self-loop. It is also easy to see that this graph is directed and disconnected, and it has a 3-node weakly connected component (see 0.1), namely $\{n_1, n_2, n_3\}$.

For larger graphs, text or visual representation does not suffice to answer even simple questions about the graph. Below are the definitions of some common graph representations, used to represent graphs computationally. These should help with understanding and using the conversion routines in Section 2.2.

For the following discussion, assume that **n** is the number of nodes in a given graph, and **m** is the number of edges.

An **adjacency matrix** is a $n \times n$ matrix A, such that $A(i, j) = 1$ if i is connected to j and $A(i, j) = 0$, otherwise. The 1s in the matrix stand for the edges. If the graph is undirected, then the matrix is symmetric, because $A(i, j) = A(j, i)$ for any i and any j. While usually this is a 0-1 matrix, sometimes edge weights can be indicated by using other numbers, so most generally the adjacency matrix has zeros and positive entries.

An **edge list** is a matrix representation of the set of edges. For the toy example $\{(n_1, n_2), (n_2, n_3), (n_4, n_4)\}$, the edge list representation would be $[n_1\ n_2; n_2\ n_3; n_4\ n_4]$. Edge lists can have weights too, for example

$$\textit{edge list} = \begin{bmatrix} n_1 & n_2 & 0.5 \\ n_2 & n_3 & 1 \\ n_4 & n_4 & 2 \end{bmatrix}.$$

The **adjacency list** is the sparsest graph representation. For every node, only its list of neighbors is recorded. In Octave, one can use the cell structure to represent the adjacency list. In other languages this is known as a dictionary. The adjacency list representation of the 4-node example above is: $adjList\{n_1\} = [n_2]$, $adjList\{n_2\} = [n_3]$, $adjList\{n_4\} = [n_4]$.

The **incidence matrix** I is a table of nodes (n) versus edges (m). In other words, the rows are node indices and columns correspond to edges. So if edge e connects nodes i and j, then $I(i, e) = 1$ and $I(j, e) = 1$. For directed graphs $I(i, e) = -1$ and $I(j, e) = 1$, if i is the source node and j the target. For the above example:

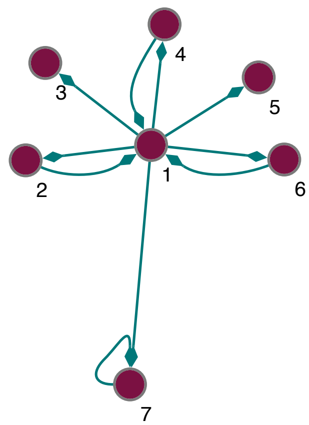
$$I = \begin{array}{c|ccc} & e_1 & e_2 & e_3 \\ \hline n_1 & -1 & 0 & 0 \\ n_2 & 1 & -1 & 0 \\ n_3 & 0 & 1 & 0 \\ n_4 & 0 & 0 & 1 \end{array}.$$

There can be other representations depending on purpose, understanding, or algorithm implementation. Suppose that there is a reason to store graph information as text. Here is an example **string representation** that could be easily read and easily stored in a text file. It is essentially the adjacency list, with some string nomenclature. Nodes are indexed from 1 to n, and every node has a list neighbors (could be empty). Nodes and their lists are separated by commas (,), new neighbors by dots (.). Of course, this is arbitrary, but it is quite clear. The toy example representation is:

.2,.3,,.4,

Four commas mean four nodes. Node 1 has one neighbor, namely node 2. Node 2 connects to node 3, node 3 has no neighbors (adjacent commas), and node 4 connects to itself. As an additional example, here is the representation of an undirected triangle: “2.3,.1.3,.1.2,”.

So there are many ways to write down and store a graph structure. Figure 2 shows one more example of all structures described above.



$$\text{edge list} = \begin{array}{c} 1 \to 2 \\ 2 \to 1 \\ 1 \to 3 \\ 1 \to 4 \\ 4 \to 1 \\ 1 \to 5 \\ 1 \to 6 \\ 6 \to 1 \\ 1 \to 7 \\ 7 \to 7 \end{array} \quad \text{adjacency} = \begin{array}{ccccccc} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

$$\text{adj list} = \begin{array}{ll} \{1\} \to & 2, 3, 4, 5, 6, 7 \\ \{2\} \to & 1 \\ \{3\} \to & \varnothing \\ \{4\} \to & 1 \\ \{5\} \to & \varnothing \\ \{6\} \to & 1 \\ \{7\} \to & 7 \end{array} \quad \text{incidence} = \begin{array}{c|cccccccccc} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 & e_{10} \\ \hline n_1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 & 0 \\ n_2 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ n_3 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ n_4 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ n_5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ n_6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ n_7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Figure 2: Most common graph representations: edge list, adjacency matrix, adjacency list and incidence matrix. Example of 7-node directed graph, with one self-loop. The string representation of this graph is “**.2.3.4.5.6.7,.1,,.1,,.1,.7,**”.

2.2 Routines

The functions in this section are conversion routines from one graph representation to another.

2.2.1 adj2adjL.m

Convert an adjacency matrix to an adjacency list.

```
% Converts an adjacency graph representation to an adjacency list.
% Note 1: Valid for a general (directed, not simple) graph.
% Note 2: Edge weights (if any) get lost in the conversion.
%
% INPUT: an adjacency matrix, nxn
% OUTPUT: cell structure for adjacency list: x{i_1}=[j_1,j_2 ...]
%
% GB: last updated, September 24 2012
```

2.2.2 adjL2adj.m

Convert an adjacency list to an adjacency matrix. This is the inverse function of *adj2adjL.m* 2.2.1

```
% Convert an adjacency list to an adjacency matrix.
%
% INPUTS: adjacency list: length n, where L{i_1}=[j_1,j_2,...]
% OUTPUTS: adjacency matrix nxn
%
% Note: Assume that if node i has no neighbours, then L{i}=[];
% GB: last updated, Sep 25 2012
```

2.2.3 adj2edgeL.m

Convert an adjacency matrix to an edge list.

```
% Converts adjacency matrix (nxn) to edge list (mx3)
%
% INPUTS: adjacency matrix: nxn
% OUTPUTS: edge list: mx3
%
% GB: last updated, Sep 24, 2012
```

2.2.4 edgeL2adj.m

Converts edge list to adjacency matrix. This is the inverse routine of *adj2edgeL.m* 2.2.3.

```
% Converts edge list to adjacency matrix.
%
% INPUTS: edgelist: mx3, m - number of edges
% OUTPUTS: adjacency matrix nxn, n - number of nodes
%
% Note: information about nodes is lost: indices only (i1,...in) remain
% GB: last updated, Sep 25, 2012
```

2.2.5 adj2inc.m

Converts adjacency matrix to incidence matrix.

```
% Converts adjacency matrix to an incidence matrix
% Note: Valid for directed/undirected, simple/not simple graphs
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: incidence matrix: n x m (number of edges)
%
% Other routines used: isDirected.m
% GB: last updated, Sep 25 2012
```

2.2.6 inc2adj.m

Converts incidence matrix to adjacency matrix. This is the inverse function of *adj2inc.m* 2.2.5.

```
% Converts an incidence matrix representation to an
% adjacency matrix representation for an arbitrary graph.
%
% INPUTs: incidence matrix, nxm (num nodes x num edges)
% OUTPUTs: adjacency matrix, nxn
%
% GB: last updated, Sep 25, 2012
```

2.2.7 adj2str.m

Converts adjacency matrix to a string graph representation.

```
% Converts an adjacency matrix to a one-line string representation of a graph.
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: string
%
% Note 1: The nomenclature used to construct the string is arbitrary.
%                            Here we use   .i1.j1.k1,.i2.j2.k2,....
%                            In '.i1.j1.k1,.i2.j2.k2,....',
%                            "dot" signifies new neighbor, "comma" next node
% Note 2: Edge weights are not reflected in the string representation.
% Example: [0 1 1; 0 0 0; 0 0 0] <=> .2.3,,,
%
% Other routines used: kneighbors.m
% GB: last updated, Sep 25 2012
```

2.2.8 str2adj.m

This is the reverse routine of *adj2str.m* 2.2.7. Converts a string graph representation to an adjacency matrix.

```
% Converts a string graph representation to an adjacency matrix
%                                          (see also adj2str.m)
%
% INPUTs: string graph representation: .i1.j1.k1,.i2.j2.k2,....
% OUTPUTs: adjacency matrix, nxn, n - number of nodes
%
% Note 1: Valid for a general graph.
% Note 2: This is the reverse routine for adj2str.m.
% Note 3: The string nomenclature is arbitrarily chosen.
%
% GB: last updated, Sep 25, 2012
```

2.2.9 adjL2edgeL.m

Converts adjacency list to edge list.

```
% Converts adjacency list to an edge list.
%
% INPUTS: adjacency list
% OUTPUTS: edge list, mx3 (m - number of edges)
%
% GB: last updated, Sep 25 2012
```

2.2.10 edgeL2adjL.m

Converts an edge list to an adjacency list. This is the inverse routine of *adjL2edgeL.m* 2.2.9.

```
% Converts an edge list to an adjacency list.
%
% INPUTS: edge list, mx3, m - number of edges
% OUTPUTS: adjacency list
%
% Note: Information about edge weights (if any) is lost.
% GB: last updated, September 25, 2012
```

2.2.11 inc2edgeL.m

Converts incidence matrix to an edge list.

```
% Converts an incidence matrix to an edge list.
%
% Inputs: inc - incidence matrix nxm (number of nodes x number of edges)
% Outputs: edge list - mx3, m x (node 1, node 2, edge weight)
%
% Example: [-1; 1] <=> [1,2,1], one directed (1->2) edge
% GB: last updated, Sep 25 2012
```

2.2.12 adj2simple.m

Removes self-loops and multi-edges from an adjacency matrix. Also symmetrizes the matrix and removes edge weights to produce the matrix of the corresponding simple graph.

```
% Converts an adjacency matrix of a general graph to the adjacency matrix of
%          a simple graph (symmetric, no loops, no double edges, no weights)
%
% INPUTS: adjacency matrix, nxn
% OUTPUTs: adjacency matrix (nxn) of the corresponding simple graph
%
% GB: last updated, Sep 25 2012
```

2.2.13 edgeL2simple.m

Removes self-loops and multi-edges from an edge list. Also symmetrizes the edge list and removes edge weights to produce the edge list of the corresponding simple graph.

```
% Convert an edge list of a general graph to the edge list of a
% simple graph (no loops, no double edges, no edge weights, symmetric)
%
% INPUTS: edgelist (mx3), m - number of edges
% OUTPUTs: edge list of the corresponding simple graph
%
% Note: Assumes all node pairs [n1,n2,x] occur once;
%                     if else see addEdgeWeights.m
% GB: last updated, Sep 25, 2012
```

2.2.14 symmetrize.m

```
% Symmetrize a non-symmetric matrix,
% i.e. returns the undirected version of a directed graph.
% Note: Where mat(i,j)~=mat(j,i), the larger (nonzero) value is chosen
%
% INPUTS: a matrix - nxn
% OUTPUT: corresponding symmetric matrix - nxn
%
% GB: last updated: October 3, 2012

function adj_sym = symmetrize(adj)

adj_sym = max(adj,transpose(adj));
```

2.2.15 symmetrizeEdgeL.m

```
% Making an edgelist (representation of a graph) symmetric,
% i.e. if [n1,n2] is in the edge list, so is [n2,n1].
%
% INPUTs: edge list, mx3
% OUTPUTs: symmetrized edge list, mx3
%
% GB: last updated, October 3, 2012
```

Alternative to *symmetrizeEdgeL.m* using *edgeL2adj.m*, *symmetrize.m* and *adj2edgeL.m*.

```
def symmetrizeEdgeL(el):
    adj=edgeL2adj(el);
    adj=symmetrize(adj);
    el=adj2edgeL(adj);

    return el
```

2.2.16 addEdgeWeights.m

Adding edges that occur multiple times in an edge list; summing weights.

```
% Add multiple edges in an edge list
%
% INPUTS: original (non-compact) edge list
% OUTPUTS: final compact edge list (no row repetitions)
%
% Example: [1 2 2; 2 2 1; 4 5 1] -> [1 2 3; 4 5 1]
% GB: last updated, Sep 25 2012
```

3 Centrality measures. Distributions

3.1 Centrality, distributions over the nodes/edges

Node centrality refers to the place of nodes in the network, that is how are they connected to all other nodes in a local or global sense. Generally, there are centralities based on the number of links per node, or based on the number of paths that go through a node. These two are not necessarily unrelated, but over the entire network, there could be nodes that do not score high in all centrality measures. The choice of measure usually depends on the question.

Most centrality notions were originally coined in the social networks literature [4]. Newman also provides a good review of centrality measures and distributions in [6]. Below follow basic definitions of the most popular centrality measures. Literature sources, where relevant, are cited in the text. The simple graphs in Figure 3 are used as examples.

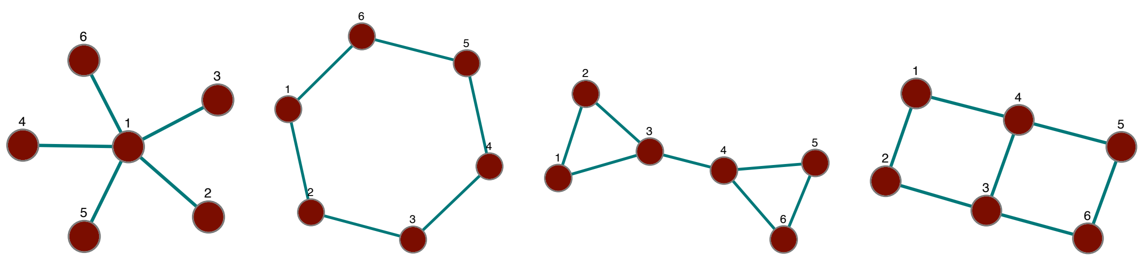


Figure 3: Simple graph examples: a star, a circle, a "bowtie" graph and a lattice graph.

The **degree** of a node is the number of links adjacent to that node. The **total degree** is the sum total of in- and out-degrees. For an undirected graph, the *total degree* is usually just called the *degree*. The **degree sequence** is the list of degrees of all nodes. Not all sequences of non-negative numbers correspond to the degree sequence of a graph (see 1.1.10). The degree sequence of the star graph in Figure 3 is $[5, 1, 1, 1, 1, 1]$, whereas the degree sequence of the bowtie graph is $[2, 2, 3, 3, 2, 2]$.

The **degree distribution** $P(k)$ is defined as the fraction of nodes with degree k. So if n_k nodes have degree k, then $P(k) = n_k/n$. The degree distribution of the bowtie graph is $P(2) = 2/3, P(3) = 1/3$. Often the **cumulative degree distribution** is used: $P(k)$ is the fraction of nodes with degree greater than or equal to k.

There are two definitions of **clustering coefficient** [6]. In both, the goal is to measure how close-knit triples of connected nodes are. The first definition (eq 3) has a *global* perspective:

$$C = \frac{number\ of\ loops\ of\ size\ 3}{number\ of\ connected\ triples} \tag{3}$$

The second definition (eq 4) computes the clustering coefficient for every node and then takes the average over the entire graph. Let L be the adjacency list representation of the graph, and A be the adjacency matrix. Then $L(i)$ is the list of neighbors of i.

$$C_i = \frac{\sum_{j,k \in L(i), j<k} A(j,k)}{\binom{|L(i)|}{2}} \qquad\qquad C = \frac{1}{n}\sum_{i=1}^{n} C_i \tag{4}$$

Another way to write this on one line, using only the adjacency A is:

$$C = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{(\sum_{j=1}^{n} A(i,j))(\sum_{j=1}^{n} A(i,j) - 1)} \sum_{j=1}^{n}\sum_{k=1}^{n} A(i,j)A(i,k)A(j,k)$$

Among the example graphs in Figure 3 the star graph, the circle and the lattice graph all have zero clustering coefficients. That is easy to see, as none of them have triangle loops. So the clustering coefficient according to the first definition is automatically zero. In the second definition, for any node, no two of its neighbors are connected, so the first sum in equation 4 is zero. The bowtie graph, however, has a positive clustering coefficient. According to the first definition (eq 3), $C = 2/6 = 1/3$. And according to the second (eq 4),

$C = \frac{1}{6}(1 + 1 + 1/3 + 1/3 + 1 + 1) = 7/9$. Therefore, the two definitions give different results between 0 and 1 (with some exceptions).

Assortativity deals with the question of whether nodes with similar degree connect to each other. It is essentially the degree-degree correlation across edges. The star example in Figure 3 is an example of the most disassortative graph: all lowest-degree nodes connect to the highest-degree node. The degree-degree correlation of this graph is -1. On the other hand, the circle graph in Figure 3 is most assortative: all nodes connect to nodes of the same degree. A *neutral* graph in terms of assortativity should be a random graph - because there is no preference in which nodes attach to which. The random graph should have a degree correlation of 0 (see Figure 4).

The **pearson degree correlation** is used to measure assortativity. It is defined as

$$r = \frac{\sum(x-\bar{x})(y-\bar{y})}{\sqrt{\sum(x-\bar{x})^2\sum(y-\bar{y})^2}} \tag{5}$$

where the sums are over the edges, and x and y are such that x_i and y_i are the degrees of the nodes at the ends of edge i. Note that for these x and y, it is always true that $\bar{x} = \bar{y}$.

Rewiring means moving edges, while keeping the nodes the same. Usually, rewiring experiments are **degree-preserving rewiring** experiments. That means that no matter how the ends of edges are relabeled, the degree of every node remains the same. This is done to determine to what extent graph topology depends on the degree distribution. It turns out, however, that there is a huge diversity of graphs with the same degree sequence [11]. The lattice graph in Figure 3 is a rewired version of the bowtie graph. Indeed, both graphs are connected and both have degree sequence $[2, 2, 3, 3, 2, 2]$.

The **rich club metric** [12] measures the density of links among nodes which have a degree higher than some threshold degree. Intuitively, if all the highly-connected individuals are connected, then there's a *rich club*. If k is the threshold degree, and N_k is the set of nodes with $deg(i) \geq k$, then the rich club metric is defined as $\phi_k = linkDensity(G(N_k))$, where $G(N_k)$ is the subgraph of G defined by the set of nodes N_k. The link density is computed as in Section 0.2.5. The same can be written as $\phi_k = numEdges(G(N_k))/\binom{|N_k|}{2}$. For the bowtie graph $\phi_2 = 7/15$, $\phi_3 = 1$.

The **rich club distribution** is simply the rich club metric computed at different threshold degrees. The threshold k can vary from 0 to $n-1$, where n is the number of nodes. Often the rich club metric or distribution is normalized by the corresponding values for a random graph with the same degree distribution [12].

The *eigenCentrality* routine 3.2.10 is an implementation of **eigenvector centrality**. Eigenvector centrality reflects how important are all neighboring nodes. So high centrality will have nodes adjacent to high-scoring nodes. Eigenvector centrality is defined in [13]. If a node i's score x_i is the sum of score of all its neighboring nodes, then: $x_i = \frac{1}{\lambda}\sum_{j,\ s.t.\ A(i,j)=1} x_j = \frac{1}{\lambda}\sum_j A(i,j)x_j$. In vector form, $x = \frac{1}{\lambda}Ax \Rightarrow Ax = \lambda x$. The positive solution is given by the eigenvector corresponding to the largest eigenvalue (by the Perron-Frobenius theorem). Therefore, the eigenvector centrality is defined as the eigenvector corresponding to the largest eigenvalue. PageRank is a version of this idea.

Betweenness centrality [14] is a centrality measure for a node which reflects how many paths go through that node. More precisely, if node k sits on a shortest path between some nodes i and j, then this path counts towards the *betweenness* of k. Suppose σ_{ij} is the number of shortest paths between i and j and $\sigma_{ij}(k)$ is the number of shortest paths between i and j that go through k. Then the betweenness of node k is defined as:

$$nodeBetw(k) = \sum_{\substack{all\ i,j\ s.t.\\ i \neq k \neq j}} \frac{\sigma_{ij}(k)}{\sigma_{ij}} \tag{6}$$

In practice, the betweenness is normalized by the number of node pairs $\binom{n}{2}$ (for directed graphs $2\binom{n}{2}$).

The above definion of betweenness is really about **node betweenness**. The same framework is extensible to edges, and hence **edge betweenness** is proportional to the number of shortest paths that go through a given edge. An edge betweenness algorithm is given in [15]. The highest betweenness edge in the bowtie graph is the 3↔4 edge.

3.2 Routines

3.2.1 degrees.m

Returns the total degree, and in- and out-degree sequence of an arbitrary adjacency matrix. The **total degree** of a node is the number of all links adjacent to that node. The **in-degree** is the number of incoming links, and the **out-degree** is the number of outdoing links.

```
% Compute the total degree, in-degree and out-degree
%          of a graph based on the adjacency matrix;
% Note: Returns weighted degrees, if the input matrix
%                                        is weighted
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: degree (1xn), in-degree (1xn) and out-degree
%                                     (1xn) sequences
%
% Other routines used: isDirected.m
% GB: last updated, Sep 26, 2012
```

3.2.2 rewire.m

Degree-preserving rewiring. A graph is rewired k number of times (edges are moved k times), while the degree of every node stays the same.

Other code on random rewiring by Maslov is available here http://www.cmth.bnl.gov/~maslov/matlab.htm.

```
% Degree-preserving random rewiring.
% Note 1: Assume unweighted undirected graph.
%
% INPUTS: edgelist, el (mx3) and number of rewirings, k (integer)
% OUTPUTS: rewired edgelist
%
% GB: last updated, Sep 26, 2012
```

3.2.3 rewireAssort.m

Degree-preserving rewiring with increasing assortativity.

```
% Degree-preserving random rewiring
% Every rewiring increases the assortativity (pearson coefficient)
%
% Note 1: There are rare cases of neutral rewiring
%                    (coeff stays the same within numerical error)
% Note 2: Assume unweighted undirected graph
%
% INPUTS: edge list, el (mx3) and number of rewirings, k
% OUTPUTS: rewired edge list
%
% Other routines used: degrees.m
% GB: last updated, Sep 27 2012
```

3.2.4 rewireDisassort.m

Degree-preserving rewiring with decreasing assortativity.

```
% Degree-preserving random rewiring.
% Every rewiring decreases the assortativity (pearson coefficient).
%
% Note 1: There are rare cases of neutral rewiring
%      (pearson coefficient stays the same within numerical error).
% Note 2: Assume unweighted undirected graph.
%
% INPUTS: edge list, el and number of rewirings, k (integer)
% OUTPUTS: rewired edge list
% GB: last updated, Sep 27 2012
```

3.2.5 aveNeighborDeg.m

Computes the average degree of neighboring nodes for every vertex.

```
% Computes the average degree of neighboring nodes for every vertex.
% Note: Works for weighted degrees (graphs) also.
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: average neighbor degree vector, 1xn
%
% Other routines used: degrees.m, kneighbors.m
% GB: last updated, Sep 28, 2012
```

3.2.6 closeness.m

Computes the closeness centrality for all vertices.

```
% Computes the closeness centrality for every vertex:
%                      1/sum(dist to all other nodes)
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: vector of closeness centralities, nx1
%
% Source: social networks literature (example:
%       Wasserman, Faust, "Social Networks Analysis")
```

```
%
% Other routines used: simpleDijkstra.m
% GB: last updated, Sep 28, 2012
```

3.2.7 nodeBetweennessSlow.m

Returns the betweenness centrality of all vertices [14].

```
% This function returns the betweenness measure of all vertices.
% Betweenness centrality measure: number of shortest paths running through a vertex.
% Note 1: Valid for a general graph.
% Node 2: Using 'number of shortest paths through a node' definition.
%
% INPUTS: adjacency or distances matrix (nxn)
% OUTPUTS: betweeness vector for all vertices (1xn)
%
% Other routines used: numNodes.m, shortestPathDP.m
% GB: Sep 28, 2012
```

3.2.8 nodeBetweennessFaster.m

A faster node betweenness algorithm (same input/output as 3.2.7).

```
% Betweenness centrality measure: number of shortest paths running through a vertex.
% Compute for all vertices, using Dijkstra's algorithm,
%                     and the 'number of shortest paths through a node' definition.
% Note: Valid for a general connected graph.
%
% INPUTS: adjacency or distances matrix, nxn
% OUTPUTS: betweeness vector for all vertices (1xn)
%
% Other routines used: dijkstra.m
% GB: Sep 29 2012
```

3.2.9 edgeBetweenness.m

Computes edge betweenness. Analogous to node betweenness, edge betweenness is proportional to the number of shortest paths going through an edge. Algorithm described in [15].

```
% Edge betweenness routine, based on shortest paths.
% Source: Newman, Girvan, "Finding and evaluating community structure in networks"
% Note: Valid for undirected graphs only.
%
% INPUTs: edge list, mx3, m - number of edges
% OUTPUTs: w - betweenness per edge, mx3
%
% Other routines used: adj2edgeL.m, numNodes.m, numEdges.m, kneighbors.m
% GB: last modified, Sep 29, 2012
```

3.2.10 eigenCentrality.m

The eigenCentrality vector is essentially the eigenvector corresponding to the largest eigenvalue of the adjacency matrix. The ith component of this eigenvector gives the centrality score of the ith node in the network.

```
% The ith component of the eigenvector corresponding to the greatest
% eigenvalue gives the centrality score of the ith node in the network.
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: eigen(-centrality) vector, nx1
%
% GB: last updated, Sep 29, 2012
```

3.2.11 clustCoeff.m

Compute two clustering coefficients: one based on loops, and one based on local clustering (triangles). The first clustering coefficient is defined as the number of 3-loops (triangles), divided by the number of connected triples. The second clustering coefficient is node-centric. For all neighbor nodes of node i it measures what fraction connect with each other. A good review of clustering coefficients can be found in [6].

```
% Computes two clustering coefficients,
%                              based on triangle motifs count and local clustering
% C1 = number of triangle loops / number of connected triples
% C2 = the average local clustering, where Ci = (number of triangles connected to i)
%                                                / (number of triples centered on i)
% Ref: M. E. J. Newman, "The structure and function of complex networks"
% Note: Valid for directed and undirected graphs
%
% INPUT: adjacency matrix, nxn
% OUTPUT: two graph average clustering coefficients (C1, C2)
%         and clustering coefficient vector C (where mean(C) = C2)
%
% Other routines used: degrees.m, isDirected.m, kneighbors.m, numEdges.m,
%                                subgraph.m, loops3.m, numConnTriples.m
% GB: Sep 29, 2012
```

3.2.12 weightedClustCoeff.m

Clustering coefficient for weighted graphs. Definition from [16].

```
% Weighted clustering coefficient.
% Source: Barrat et al, The architecture of complex weighted networks
%
% INPUTS: weighted adjacency matrix, nxn
% OUTPUTs: vector of node weighted clustering coefficients, nx1
%
% Other routines used: degrees.m, kneighbors.m
% GB: last updated, Sep 30 2012
```

Alternative to weightedClustCoeff.m

```
  functopn wC = weightedClustCoeff(adj):

  wadj=adj;
  adj=adj>0;

  [wdeg,~,~]=degrees(wadj);
```

```
  [deg,~,~]=degrees(adj);
  n=size(adj,1); % number of nodes
  wC=zeros(n,1);

  for i=1:n
      if deg(i)<2; continue; end

      s=0;
      for ii=1:n
          for jj=1:n
              s=s+adj(i,ii)*adj(i,jj)*adj(ii,jj)*(wadj(i,ii)+wadj(i,jj))/2;
          end
      end

      wC(i)=s/(wdeg(i)*(deg(i)-1));
  end
```

3.2.13 pearson.m

Pearson degree correlation. Algorithm and ideas from [17].

```
% Calculating the Pearson coefficient for a degree sequence.
% Source: "Assortative Mixing in Networks", M.E.J. Newman, Phys Rev Let 2002
%
% INPUTs: M - (adjacency) matrix, nxn (square)
% OUTPUTs: r - Pearson coefficient
%
% Other routines used: degrees.m, numEdges.m, adj2inc.m
% GB: last updated, October 1, 2012
```

3.2.14 richClubMetric.m

Rich club metric. Algorithm and ideas from [12].

```
% Compute the rich club metric for a graph.
% Source: Colizza, Flammini, Serrano, Vespignani,
% "Detecting rich-club ordering in complex networks",
%                     Nature Physics, vol 2, Feb 2006
%
% INPUTs: adjacency matrix, nxn, k - threshold number of links
% OUTPUTs: rich club metric
%
% Other routines used: degrees.m, subgraph.m, numEdges.m
% GB: last updated, October 1, 2012
```

3.2.15 sMetric.m

S-metric: the sum of products of nodal degrees across all edges. Definition and applications described in [18].

```
% The sum of products of degrees across all edges.
% Source: "Towards a Theory of Scale-Free Graphs:
%        Definition, Properties, and Implications",
```

```
%                  by Li, Alderson, Doyle, Willinger
% Note: The total degree is used regardless of
%         whether the graph is directed or not.
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: s-metric
%
% Other routines used: degrees.m
% GB: last updated, Oct 1 2012
```

Alternative to sMetric.m:

```
def sMetric(adj):

  [deg,~,~]=degrees(adj);
  el=adj2edgeL(adj);

  s=0;
  for e=1:size(el,1)
      if el(e,1)==el(e,2)
          % count self-loops twice
          s=s+deg(el(e,1))*deg(el(e,2))*el(e,3)*2;
      else
          % multiply by the weight for edges with weights
          s=s+deg(el(e,1))*deg(el(e,2))*el(e,3);
      end
  end
```

4 Distances

4.1 Routines

4.1.1 simpleDijkstra.m

Computing distances from a given node to all other nodes in the graph, without remembering the paths.

```
% Implements a simple version of the Dijkstra shortest path algorithm
% Returns the distances from a single vertex to all others, doesn't save the path
%
% INPUTS: adjacency matrix, adj (nxn), start node s (index between 1 and n)
% OUTPUTS: shortest path length from start node to all other nodes, 1xn
%
% Note: Works for a weighted/directed graph.
% GB: last updated, September 28, 2012
```

5 Links

- Edward Scheinerman's Matgraph.
- Degree-preserving rewiring code by Sergei Maslov.

References

[1] Milgram's small world experiment; source: http://en.wikipedia.org/wiki/Small_world_experiment, last accessed: Sep 23, 2012

[2] D.J. de S. Price, Networks of scientific papers, Science, 149, 1965

[3] D. Watts and S. Strogatz, Collective dynamics of 'small-world' networks, Nature 393, 1998

[4] S. Wasserman and K. Faust, Social network analysis, Cambridge University Press, 1994

[5] Duncan J. Watts, Six degrees: The science of a Connected Age, W. W. Norton, 2004

[6] M. E. J. Newman, The structure and function of complex networks, SIAM Review 45, 167-256 (2003)

[7] Alderson D., Catching the Network Science Bug: ..., Operations Research, Vol. 56, No. 5, Sep-Oct 2008, pp. 1047-1065

[8] Tarjan, R. E. , Depth-first search and linear graph algorithms, SIAM Journal on Computing 1 (2): 146-160, 1972

[9] Wikipedia description of Tarjan's algorithm; source: http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm, last accessed: Sep 23, 2012

[10] Erdös, P. and Gallai, T. Graphs with Prescribed Degrees of Vertices [Hungarian]. Mat. Lapok. 11, 264-274, 1960.

[11] Alderson, Li, Diversity of graphs with highly variable connectivity, Phys. Rev. E 75, 046102 (2007)

[12] Vittoria Colizza, Alessandro Flammini, M. Angeles Serrano, Alessandro Vespignani, Detecting rich-club ordering in complex networks, Nature Physics 2, 110-115 (2006)

[13] Wikipedia entry on eigenvector centrality; source: http://en.wikipedia.org/wiki/Centrality#Using_the_adjacency_matrix_to_find_eigenvector_centrality, last accessed: October 2, 2012

[14] Wikipedia article on node betweenness; source: http://en.wikipedia.org/wiki/Betweenness_centrality, last accessed: September 28, 2012

[15] M. E. J. Newman, M. Girvan, Finding and evaluating community structure in networks, Phys. Rev. E 69, 026113 (2004)

[16] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, The architecture of complex weighted networks, PNAS March 16, 2004 vol. 101 no. 11, 3747-3752

[17] M. E. J. Newman, Assortative mixing in networks, Phys. Rev. Lett. 89, 208701 (2002)

[18] Lun Li, David Alderson, Reiko Tanaka, John C. Doyle, Walter Willinger, Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications, Internet Math. Volume 2, Number 4 (2005), 431-523

A Additional Figures

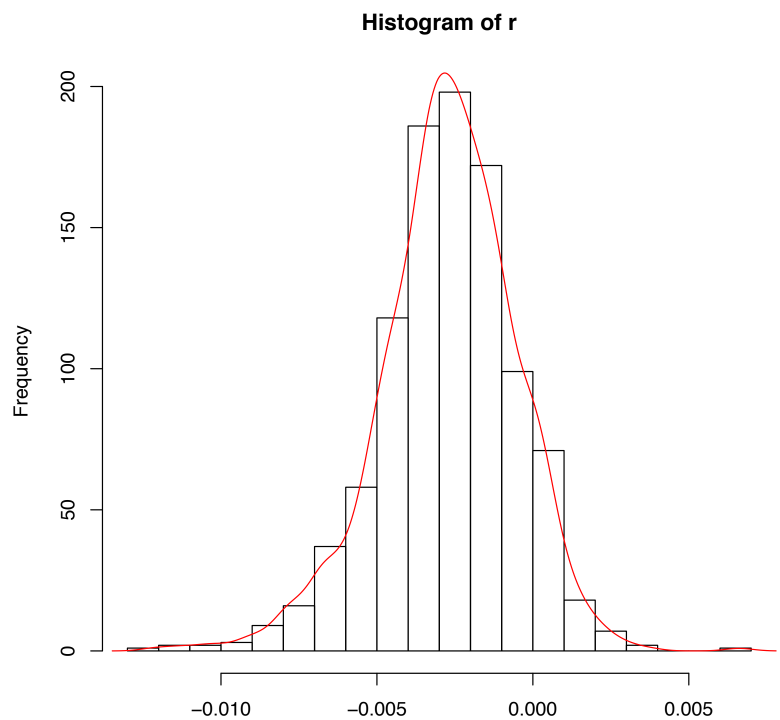


Figure 4: The histogram of degree correlations (r) of 1000 random graphs.