Octave routines for network analysis

GB

June 26, 2013

Contents

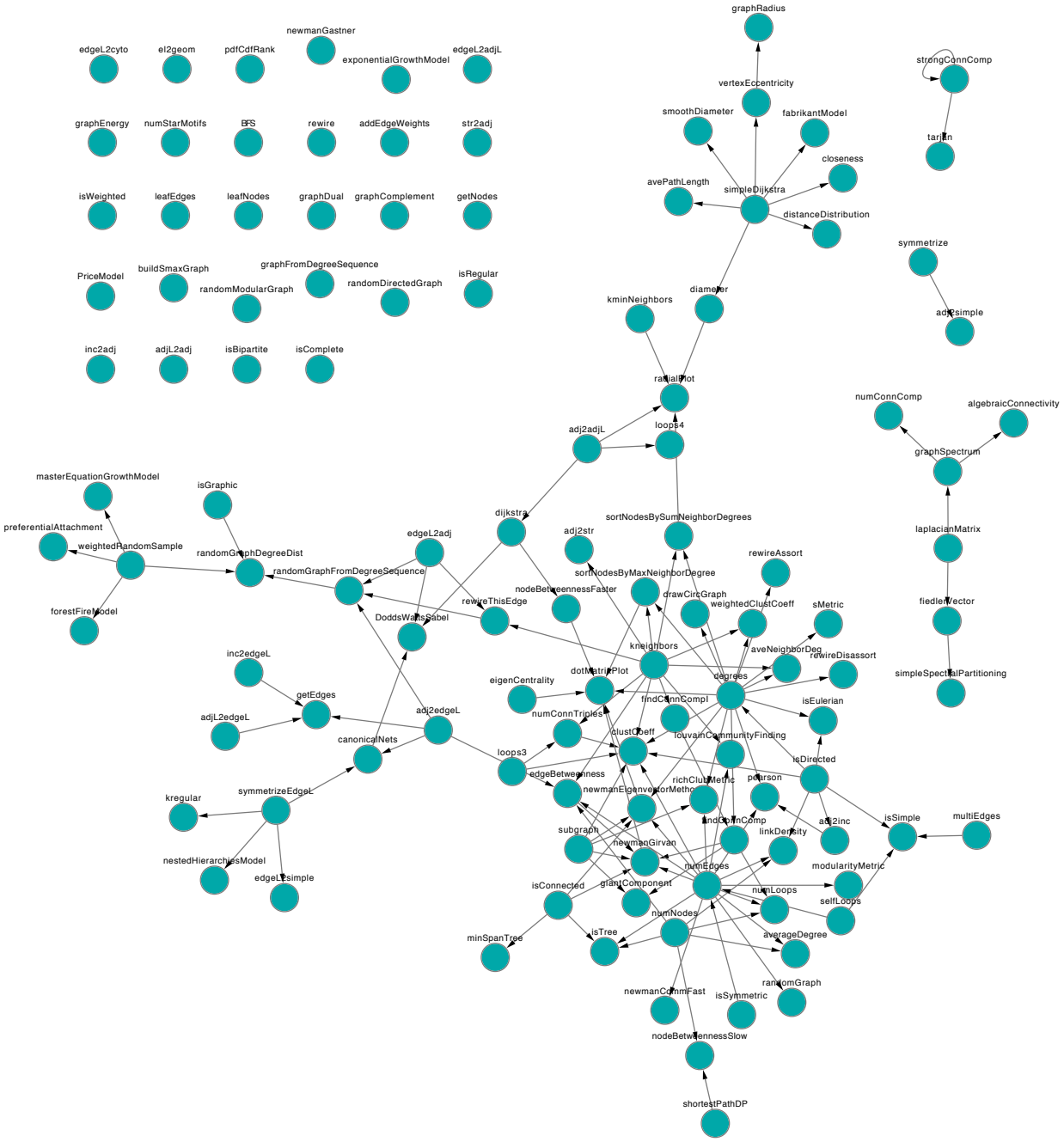


Figure 1: Graph of functional interdependencies in this toolbox. An edge points from routine A to routine B if routine A is called within routine B. For example, *isConnected.m* is called within *minSpanTree.m*

0 About this toolbox

(This is a copy of the README file.)

octave-networks-toolbox: A set of graph/networks analysis functions in Octave, 2012-2013

Quick description
This is a repository of functions relevant to network/graph analysis, organized by functionality. These routines are useful for someone who wants to start hands-on work with networks fairly quickly, explore simple graph statistics, distributions, simple visualization and compute common network theory metrics.

History
The original (2006-2011) version of these routines was written in Matlab, and is still hosted by strategic.mit.edu (http://strategic.mit.edu/downloads.php?page=matlab_networks). The octave-networks-toolbox inherits the original BSD open source license and copyright, provided at the end of this file. Many of the routines might still be compatible with Matlab. For Octave/Matlab differences, see http://en.wikibooks.org/wiki/MATLAB_Programming/Differences_between_Octave_and_MATLAB.

Installation
The code currently runs on GNU Octave Version 3.4.0 with Gnuplot 4.2.5. No specific library installation necessary. Interdependencies between functions are well-documented in the function headers. The routines can be called directly from the Octave prompt, either in the same directory or from anywhere if the toolbox folder is added to the path. For example:

```
# running numNodes.m
octave-3.4.0:1> numNodes([0 1 1; 1 0 1; 1 1 0])
ans =  3
```

Authorship
This code is written and currently maintained by Gergana Bounova. It is undergoing continuous expansion and development. Collaborators are very welcome. Contact via github for comments, questions, suggestions, corrections or simply join!

Organization
The functions are organized in 11 categories: basic routines, diagnostic routines, conversion routines, centralities, distances, simple motif routines, linear algebra functions, modularity routines, graph construction models, visualization and auxiliary. These categories reflect roles/functionality and topics in the literature, but they are arbitrary, and mostly used for documentation purposes.

Documentation
Documentation is available in this Functions Manual. The manual contains general background information, function headers, code examples, and references. For some functions, additional background, definitions or derivations are included.

License/Copyright

1 Basic network routines

1.1 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 1.2.

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 118 nodes and 125 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 0.0181. Note that equation 1 is valid for undirected graphs only.

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 2.12 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**. The giant component in Figure 1 has 80 functions. There are also one connected components of 6 functions, two 2-node components and 28 isolated nodes (functions that are independent).

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. So the two-node subgraph (*strongConnComp*,*tarjan*) is definitely weakly connected.

1.2 Routines

1.2.1 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
```

Examples:
$> getNodes([0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0], 'adj')$ % using adjacency matrix representation
$ans =$
1 2 3

$adjL = \{[2,3],[1,3],[1,2,4],[3,5,6],[4,6],[4,5]\};$ % using adjacency list representation
$> getNodes(adjL, 'adjlist')$
$ans =$
1 2 3 4 5 6

1.2.2 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
```

Examples:
$> getEdges([0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0], 'adj')$ % using adjacency matrix representation
$ans =$
1 2 1
1 3 1
2 1 1
2 3 1
3 1 1
3 2 1

$adjL = \{[2,3],[1,3],[1,2,4],[3,5,6],[4,6],[4,5]\};$ % using adjacency list representation
$> getEdges(adjL, 'adjlist')$
$ans =$
1 2 1
1 3 1
2 1 1
2 3 1
3 1 1
3 2 1
3 4 1
4 3 1
4 5 1
4 6 1
5 4 1
5 6 1
6 4 1
6 5 1

Note that the column of 1s in the output shows the edge weight for every edge. If the graph is unweighted (as in this case), this column is unnecessary and is easy to remove. In fact, from the graph representations

discussed in Section 3.1 only the *edge list* can carry edge weight information.

1.2.3 numNodes.m

Number of nodes/vertices 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);
```

Examples:
$N = randi(100);$
$adj = randomGraph(N);$
$> assert(numNodes(adj), N)$ % test whether the random graph does indeed have N nodes

$adjL = \{[2,3],[1,3],[1,2,4],[3,5,6],[4,6],[4,5]\};$ % a graph (adjacency list) with 6 nodes
$> numNodes(adjL)$
$ans = 6$

1.2.4 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
```

Examples:
$N = randi(100);$
$E = randi([1, N-1]);$
$adj = randomGraph(N, [\,], E);$
$> assert(numEdges(adj), E)$ % check that the random graph has exactly E edges

% the bowtie graph is a 6-node graph with 7 edges (shown in Figure 3)
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> numEdges(bowtie)$
$ans = 7$

1.2.5 linkDensity.m

The **density of links** of the graph. For an undirected graph the density is defined as $density = \frac{m}{n(n-1)/2}$, where n is the number of nodes and m is the number of edges. The directed graph version is the same but without the factor of 2.

```
% 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 1: The graph has to be non-trivial (more than 1 node).
% Note 2: Routine works for both directed and undirected graphs.
%
% Other routines used: numNodes.m, numEdges.m, isDirected.m
% GB: last update Sep 19, 2012
```

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$ % undirected 3-node cycle
$> linkDensity(adj)$
$ans = 1$

% the bowtie graph is a 6-node graph with 7 edges
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> linkDensity(bowtie)$
$ans = 0.46667$

1.2.6 selfLoops.m

Number of **self-loops**, i.e. number of edges that start and end at the same node.

```
% 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
```

Examples:
$> selfLoops([0\ 1; 0\ 0])$ % one directed edge
$ans = 0$

$adj = [1\ 0\ 0; 0\ 1\ 0; 0\ 0\ 1];$ % three self-loops
$> selfLoops(adj)$
$ans = 3$

1.2.7 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
```

Examples:
$one\_double\_edge = [0\ 2; 0\ 0];$ % directed double edge
$> multiEdges(one\_double\_edge)$
$ans = 2$

$double\_edge = [0\ 2; 2\ 0];$ % undirected double edge
$> multiEdges(double\_edge)$
$ans = 2$

$adj = [1\ 1\ 0; 1\ 0\ 0; 0\ 0\ 0];$ % a self-loop, an edge and a single node
$> multiEdges(adj)$
$ans = 0$

1.2.8 averageDegree.m

The **average degree** (number of links per node) across all nodes. Defined as: $\frac{2m}{n}$, where n is the number of nodes and m is the number of edges. Also, note that the link density (Section 1.2.5) is related to the average degree: $linkDensity = \frac{averageDegree}{n-1}$.

```
% 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
```

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$ % undirected 3-node cycle
$> averageDegree(adj)$
$ans = 2$

$adj = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0];$ % undirected 3-node binary tree
$> averageDegree(adj)$
$ans = 1.3333$

1.2.9 numConnComp.m

Calculating the **number of connected components** in the graph by using the eigenvalues of the Laplacian.

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

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$ % undirected 3-node cycle
$> numConnComp(adj)$
$ans = 1$

% two disconnected three-node cycles
$adj = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 0\ 0\ 0; 0\ 0\ 0\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> numConnComp(adj)$
$ans = 2$

1.2.10 findConnComp.m

findConnCompI.m: Find the **connected component** to which a given node i belongs to. This function is called within $findConnComp.m$.

```
% 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
```

Example:
% two disconnected three-node cycles
$adj = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 0\ 0\ 0; 0\ 0\ 0\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> findConnCompI(adj, 1)$
$ans =$
1 2 3

$> findConnCompI(adj, 5)$
$ans =$
4 5 6

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
```

Example:
% two disconnected three-node cycles, same as above
$adj = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 0\ 0\ 0; 0\ 0\ 0\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$comp = findConnComp(adj);$
$> comp$
$comp =$
{
$[1, 1] =$
1 2 3
$[1, 2] =$
4 5 6
}

1.2.11 giantComponent.m

The **largest connected component** in a graph. Return 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
```

Example:
$adj = [0\ 1\ 0; 1\ 0\ 0; 0\ 0\ 1];$ % an edge and a single node
$[GC, I] = giantComponent(adj);$
$> GC$
$GC =$
0 1
1 0
$> I$
$I =$
1 2

1.2.12 tarjan.m [8][9]

tarjan.m: Return 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
```

Example:
$directed\_triangle = \{\ [2], [3], [1]\ \};$ % same as $\{1:[2], 2:[3], 3:[1]\}$ (a directed 3-cycle)
$comp = tarjan(directed\_triangle);$
$> comp\{1\}$
$ans =$
1 2 3

1.2.13 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
```

Example:
$g = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0];$
$> gc = graphComplement(adj)$
$gc =$

1 0 0
0 1 1
0 1 1

1.2.14 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
```

Examples:
$triangle = \{[2,3];[1,3];[1,2]\}$ % same as $adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0]$;
$> graphDual(triangle)$
$ans =$
{
$[1,1] =$
2 3
$[1,2] =$
1 3
$[1,3] =$
1 2
}

$L = \{[2,3];[1];[1]\}$ % undirected 3-node binary tree
$> graphDual(L)$
$ans =$
{
$[1,1] = 2$
$[1,2] = 1$
}

1.2.15 subgraph.m

Return the adjacency matrix of a **subgraph**, given a subset of nodes.

```
% 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
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0]$;
$> subgraph(bowtie, [1,2,3])$
$ans =$

0 1 1
1 0 1
1 1 0

1.2.16 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
```

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0];$ % only 1 is not a leaf node, because it has degree 2
$> leafNodes(adj)$
$ans =$
2 3

$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$ % a cycle graph has no leaf nodes
$> leafNodes(adj)$
$ans = [\ ]$

1.2.17 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
```

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0];$ % a binary tree with two leaf edges/nodes
$> leafEdges(adj)$
$ans =$
1 2
1 3

$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$ % a cycle graph has no leaf edges
$> leafEdges(adj)$
$ans = [\ ]$

1.2.18 minSpanTree.m

Given a general graph, return an undirected **minimum spanning tree** of the graph, using Prim's algorithm.

```
% Prim's minimal spanning tree algorithm
% Prim's alg idea:
%  start at any node, find closest neighbor and mark edges
%  for all remaining nodes, find closest to previous cluster, mark edge
%  continue until no nodes remain
%
% INPUTS: graph defined by adjacency matrix, nxn
% OUTPUTS: matrix specifying minimum spanning tree (subgraph), nxn
%
% Other routines used: isConnected.m
% GB: Oct 7, 2012
```

Example:
% a 6-node, 7-edge graph, depicted in Figure 3
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> minSpanTree(bowtie)$
$ans =$

0	1	1	0	0	0
1	0	0	0	0	0
1	0	0	1	0	0
0	0	1	0	1	1
0	0	0	1	0	0
0	0	0	1	0	0

1.2.19 BFS.m

Breadth-first search. Returns a directed **breadth-first-search tree**, starting at given root node.

```
% Implementation of breadth-first-search of a graph.
% Returns a breadth-first-search tree.
%
% INPUTs: adjacency list (nxn), start node index
% OUTPUTs: BFS tree, in adjacency list {} format (directed)
%
% GB: last updated, Oct 7 2012
```

Example:
$L = \{[2,3],[1,3],[1,2,4],[3,5,6],[4,6],[4,5]\}$ % adjacency list representation of the *bowtie* graph
$> BFS(L,1)$
$ans =$
{
$[1,1] =$
2 3
$[2,1] = [\ ]$
$[3,1] = 4$
$[4,1] =$

5 6
[5, 1] = [ ]
[6, 1] = [ ]
}

> $BFS(L, 3)$
$ans =$
{
[1, 1] = [ ]
[2, 1] = [ ]
[3, 1] =
1 2 4
[4, 1] =
5 6
[5, 1] = [ ]
[6, 1] = [ ]
}

2 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.

2.1 Routines

2.1.1 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
```

Examples:
> $isSimple([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$ % undirected binary tree
$ans = 1$
> $isSimple([0\ 2\ 1; 2\ 0\ 0; 1\ 0\ 0])$ % a weighted graph example
$ans = 0$
> $isSimple([0\ 1\ 1; 0\ 0\ 0; 0\ 0\ 0])$ % directed graph example
$ans = 0$

2.1.2 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
```

Examples:
$> isDirected([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$
$ans = 0$
$> isDirected([0\ 1\ 1; 0\ 0\ 0; 0\ 0\ 0])$
$ans = 1$

2.1.3 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
```

Examples:
$> isSymmetric([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$
$ans = 1$
$> isSymmetric([0\ 1\ 1; 0\ 0\ 0; 0\ 0\ 0])$
$ans = 0$

2.1.4 isConnected.m

Checks whether a graph is connected. A graph is **connected** if there is a path, via edges, from any node to any other node. There are many ways to check this. Some alternatives to *isConnected.m* are listed below. The idea behind the main routine is from Matgraph's *isconnected* function.

```
% 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* 1.2.10.

Examples:
$>$ *isConnected*([0 1 1; 1 0 0; 1 0 0]) % undirected binary tree with 3 nodes
$ans = 1$

% two disconnected 3-node cycles
adj = [0 1 1 0 0 0; 1 0 1 0 0 0; 1 1 0 0 0 0; 0 0 0 0 1 1; 0 0 0 1 0 1; 0 0 0 1 1 0];
$>$ *isConnected*(adj)
$ans = 0$

2.1.5 isWeighted.m

Checks whether a graph has **weighted** links.

```
% Check whether a graph is weighted, i.e edges have weights.
%
% 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
```

Examples:
eL = [
1 2 1
1 3 1
2 3 1];

$> isWeighted(eL)$
$ans = 0$

$> isWeighted([1\ 2\ 2; 2\ 1\ 2])$ % undirected double (weighted) edge
$ans = 1$

2.1.6 isRegular.m

Checks whether a graph is regular. In a **regular graph** every node has the same number of links.

```
% 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
```

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0];$ % undirected binary tree
$> isRegular(adj)$
$ans = 0$

$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$ % undirected 3-node cycle
$> isRegular(adj)$
$ans = 1$

$adj = [0\ 2\ 2; 2\ 0\ 2; 2\ 2\ 0];$ % same as above, but edges are weighted
$> isRegular(adj)$
$ans = 1$

$> isRegular([0\ 1\ 0\ 1; 1\ 0\ 1\ 0; 0\ 1\ 0\ 1; 1\ 0\ 1\ 0])$ % a 4-node cycle
$ans = 1$

2.1.7 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
```

Examples:
$> isComplete([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$
$ans = 0$

$> isComplete([0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0])$
$ans = 1$

2.1.8 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 1: Assume that the graph is connected.
% Note 2: If there is an Eulerian trail, it is reported.
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: Boolean variable, 0 or 1
%
% Other routines used: degrees.m, isDirected.m
% GB: last updated, Sep 23, 2012
```

Example:
$> isEulerian([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$ % nodes have degree (2,1,1) respectively
$ans = 0$

$> isEulerian([0\ 1\ 0\ 1; 1\ 0\ 1\ 0; 0\ 1\ 0\ 1; 1\ 0\ 1\ 0])$ % in a 4-cycle every node has degree 2
$ans = 1$

2.1.9 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
```

Examples:
$> isTree([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$
$ans = 1$
$> isTree([0\ 1\ 0\ 1; 1\ 0\ 1\ 0; 0\ 1\ 0\ 1; 1\ 0\ 1\ 0])$
$ans = 0$

2.1.10 isGraphic.m [10]

Check whether a sequence of numbers 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 graphic,
%        i.e. a graph with this degree sequence exists
% Source: Erdos, 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
```

Examples:
$> isGraphic([2\ 2\ 2])$
$ans = 1$
$> isGraphic([2\ 1\ 2])$
$ans = 0$

2.1.11 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 adjacency 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
```

Examples:
$> isBipartite(\{\ [2,3],[1],[1]\ \})$ % undirected binary tree with 3 nodes
$ans = 1$
$> isBipartite(\{\ [2,3],[1,2],[1]\ \})$ % this graph contains a self-loop (2,2)
$ans = 0$

3 Conversion routines

3.1 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 1.1), 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, that could be used for computation. These should help with understanding and using the conversion routines in Section 3.2.

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

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

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:

$I =$

	e_1	e_2	e_3
n_1	-1	0	0
n_2	1	-1	0
n_3	0	1	0
n_4	0	0	1

.

There can be other representations depending on purpose, understanding, or algorithm implementation. Suppose the graph information has to be stored as text. Here is an example **string representation** that could be easily read and 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 of 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 three-node cycle: "2.3,.1.3,.1.2,".

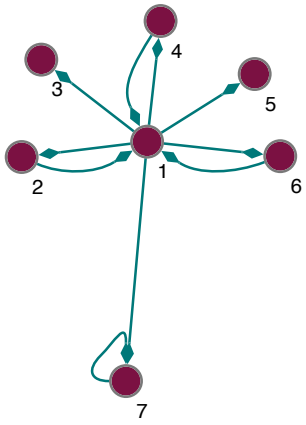
So there are many ways to write down and store a graph structure. Figure 2 shows one more example of all representations described above.

3.2 Routines

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

3.2.1 adj2adjL.m

Convert an adjacency matrix to an adjacency list.



edge list =

$1 \to 2$
$2 \to 1$
$1 \to 3$
$1 \to 4$
$4 \to 1$
$1 \to 5$
$1 \to 6$
$6 \to 1$
$1 \to 7$
$7 \to 7$

adjacency =

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

adj list =

$\{1\} \to$ 2, 3, 4, 5, 6, 7
$\{2\} \to$ 1
$\{3\} \to$ $\varnothing$
$\{4\} \to$ 1
$\{5\} \to$ $\varnothing$
$\{6\} \to$ 1
$\{7\} \to$ 7

incidence =

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}
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

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,**".

```
% Convert 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
```

Example:
$> adj2adjL([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$ % undirected binary tree with 3 nodes
$ans =$
{
$[1,1] =$
2 3
$[2,1] = 1$
$[3,1] = 1$
}

3.2.2 adjL2adj.m

Convert an adjacency list to an adjacency matrix. This is the inverse function of *adj*2*adjL.m* (3.2.1).

```
% 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
```

Example:
$aL = \{\ [2,3],[1,3],[1,2]\ \};$
$> adjL2adj(aL)$
$ans =$

0	1	1
1	0	1
1	1	0

3.2.3 adj2edgeL.m

Convert an adjacency matrix to an edge list.

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

Example:
$> adj2edgeL([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$
$ans =$

2	1	1
3	1	1
1	2	1
1	3	1

3.2.4 edgeL2adj.m

Convert edge list to adjacency matrix. This is the inverse routine of *adj*2*edgeL.m* (3.2.3).

```
% Convert edge list to adjacency matrix.
%
% INPUTS: edge list: 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
```

Example:
$> edgeL2adj([1\ 2\ 1])$ % a single directed edge
$ans =$

0 1
0 0

3.2.5 adj2inc.m

Convert an adjacency matrix to an incidence matrix.

```
% Convert 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
```

Example:
$> adj2inc([0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0])$
$ans =$
1 1
1 0
0 1

3.2.6 inc2adj.m

Convert an incidence matrix to an adjacency matrix. This is the inverse function of *adj*2*inc.m* (3.2.5).

```
% Convert 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
```

Example:
$inc = [$
1 0 1
1 1 0
0 1 1];
$> inc2adj(inc)$
$ans =$
0 1 1
1 0 1
1 1 0

3.2.7 adj2str.m

Convert an adjacency matrix to a string (text) graph representation.

```
% Convert 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
```

Example:
$>$ *adj2str*([0 1 1; 1 0 0; 1 0 0]) % undirected binary tree
ans = .2.3, .1, .1,

3.2.8 str2adj.m

This is the reverse routine of *adj2str.m* (3.2.7). Convert a string (text) graph representation to an adjacency matrix.

```
% Convert 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
```

Example:
$>$ *str2adj*(".2.3, .1.3, .1.2,") % a three-node undirected cycle
ans =

0	1	1
1	0	1
1	1	0

3.2.9 adjL2edgeL.m

Convert an adjacency list to an edge list.

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

Example:
$>$ *adjL2edgeL*({ [2, 3], [1], [1] })

$ans =$
1 2 1
1 3 1
2 1 1
3 1 1

3.2.10 edgeL2adjL.m

Convert an edge list to an adjacency list. This is the inverse routine of $adjL2edgeL.m$ (3.2.9).

```
% Convert 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
```

Example:
$> edgeL2adjL([1\ 2\ 1; 1\ 3\ 1])$
$ans =$
{
$[1,1] =$
2 3
$[2,1] = [\ ](0x0)$
$[3,1] = [\ ](0x0)$
}

3.2.11 inc2edgeL.m

Convert an incidence matrix to an edge list.

```
% Convert 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
```

Example:
$inc = [$
1 0
1 1
0 1];
$> inc2edgeL(inc)$
$ans =$
1 2 1
2 3 1
2 1 1
3 2 1

3.2.12 adj2simple.m

Remove 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.

```
% Convert 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
```

Example:
$>$ *adj*2*simple*([1 2 1; 2 0 1; 1 1 0])
ans =
0 1 1
1 0 1
1 1 0

3.2.13 edgeL2simple.m

Remove 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: edge list (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
% Other routines used: symmetrizeEdgeL.m
% GB: last updated, Sep 25, 2012
```

Example:
$>$ *edgeL*2*simple*([1 1 1; 1 2 1; 1 3 2]) % remove one self-loop and one double edge
ans =
1 2 1
1 3 1
2 1 1
3 1 1

3.2.14 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));
```

Example:
$> symmetrize([0\ 1; 0\ 0])$
$ans =$
0 1
1 0

3.2.15 symmetrizeEdgeL.m

```
% Making an edge list (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
```

Example:
$> symmetrizeEdgeL([1\ 2\ 1; 1\ 3\ 1])$
$ans =$
1 2 1
1 3 1
2 1 1
3 1 1

3.2.16 addEdgeWeights.m

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

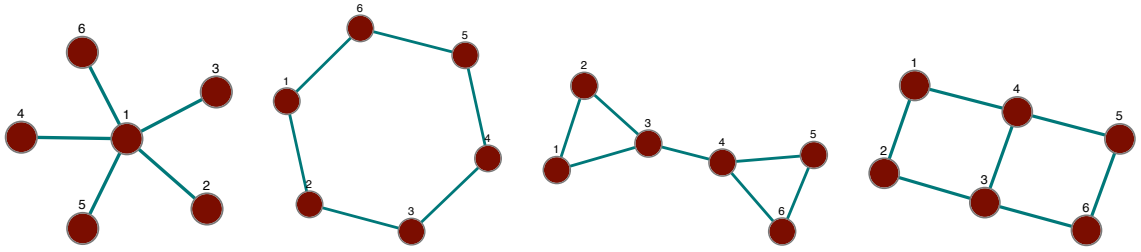


Figure 3: Simple graph examples: a star, a circle, a "bowtie" graph and a lattice graph.

```
% 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
```

Example:
$> addEdgeWeights([1\ 2\ 1; 1\ 2\ 0.5; 2\ 3\ 1; 3\ 4\ 1; 3\ 4\ 3])$
$ans =$
1 2 1.5
2 3 1.0
3 4 4.0

4 Centrality measures. Distributions

4.1 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.

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 2.1.10). 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.

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

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 three-node cycles. 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 often *measured* with 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 is a random graph, because there is no preference in which nodes attach to which. The random graph should have a degree correlation of zero (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

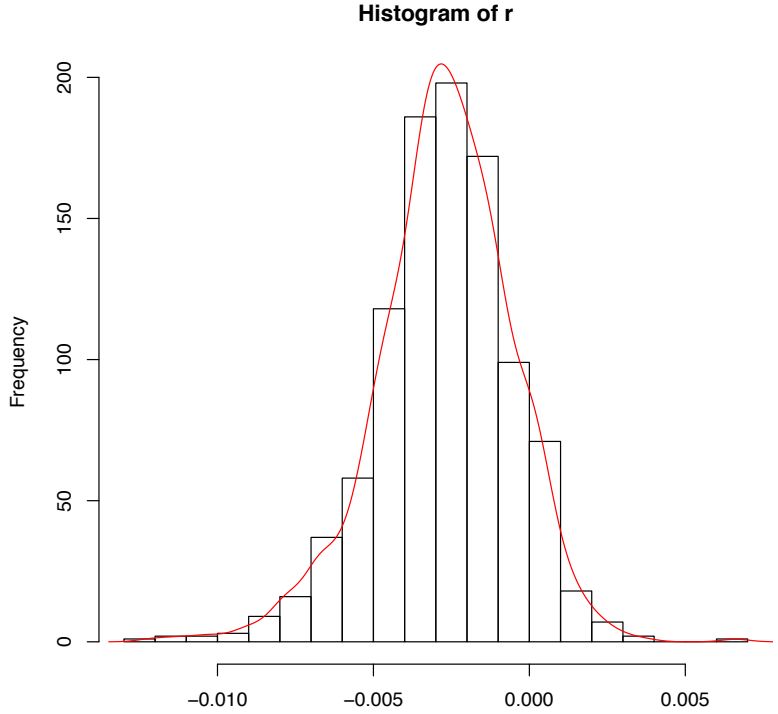


Figure 4: The histogram of degree correlations (r) of 1000 random graphs. The distribution should be centered around zero. The slight negative offset is probably due to finite-size effects.

density is computed as in Section 1.2.5. 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 (4.2.13) 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.

4.2 Routines

4.2.1 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
```

Examples:
$> degrees([0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0])$
$ans =$
$2 \quad 2 \quad 2$

$adj = [0\ 1\ 0; 0\ 0\ 1; 1\ 0\ 0];$
$[deg, indeg, outdeg] = degrees(adj);$
$> deg =$
$2 \quad 2 \quad 2$
$> indeg =$
$1 \quad 1 \quad 1$
$> outdeg =$
$1 \quad 1 \quad 1$

$adj = [0\ 1\ 1; 0\ 0\ 0; 0\ 0\ 0];$
$[deg, indeg, outdeg] = degrees(adj);$
$> deg =$
$2 \quad 1 \quad 1$
$> indeg =$
$0 \quad 1 \quad 1$
$> outdeg =$
$2 \quad 0 \quad 0$

4.2.2 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: edge list, el (mx3) and number of rewirings, k (integer)
% OUTPUTS: rewired edge list
%
% GB: last updated, Sep 26, 2012
```

Example:
$adj = randomGraph(20, 0.4);$
$elr = rewire(adj2edgeL(adj), 5);$ % rewire five edges
$adjr = edgeL2adj(elr);$
$> assert(degrees(adj), degrees(adjr))$ % check that the degrees of the two graphs are the same
$> assert(isequal(adjr, adj), false)$ % make sure that the graphs are different (edges have been rewired)

4.2.3 rewireThisEdge.m

Degree-preserving random rewiring of **one given edge**. Assumes an undirected, unweighted edge list. This is useful for rewiring *problematic* edges that, for example, can create non-simple graphs, as part of some graph construction algorithm. For an example, see Section 9.2.6.

```
% Degree-preserving rewiring of 1 given edge.
% Note 1: Assume unweighted undirected graph.
%
% INPUTS: edge list, el (mx3) and the two nodes of the edge to be rewired.
% OUTPUTS: rewired edge list, same size and same degree distribution
%
% Note: There are cases when rewiring is not possible, while
%       keeping the graph simple, so an empty edge list is returned.
%
% Other routines used: edgeL2adj.m, kneighbors.m
% GB: last updated, Oct 25, 2012
```

Example:
$bowtie\_edgeL =$

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

$> elr = rewireThisEdge(bowtie\_edgeL, 1, 3)$ % rewire the 1↔3 edge

$elr =$
1 2 1
1 6 1
2 1 1
2 3 1
6 1 1
3 2 1
3 4 1
4 3 1
4 5 1
4 6 1
5 4 1
3 5 1
6 4 1
5 3 1

$adj = edgeL2adj(bowtie\_edgeL);$
$adjr = edgeL2adj(elr);$
$> assert(degrees(adj), degrees(adjr))$ % check that the degree sequences of the two matrices are the same

4.2.4 rewireAssort.m

Degree-preserving rewiring with increasing assortativity. **Increased assortativity** here means higher Pearson coefficient (see Section 4.2.16).

```
% 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
```

Example:
$adj = randomGraph(20, 0.4);$
$elr = rewireAssort(adj2edgeL(adj), 5);$ % rewire five random edges
$adjr = edgeL2adj(elr);$
$> assert(degrees(adj), degrees(adjr))$ % check that the degree sequences stay the same
$> assert(pearson(adjr) >= pearson(adj), true)$ % verify that the Pearson coefficient has increased

4.2.5 rewireDisassort.m

Degree-preserving rewiring with **decreasing assortativity**. That means lower Pearson coefficient (4.2.16).

```
% 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
```

Example:
$adj = randomGraph(20, 0.4);$
$elr = rewireDisassort(adj2edgeL(adj), 5);$ % rewire five random edges
$adjr = edgeL2adj(elr);$
$> assert(degrees(adj), degrees(adjr))$ % check that the degree sequences stay the same
$> assert(pearson(adjr) <= pearson(adj), true)$ % verify that the Pearson coefficient has decreased

4.2.6 aveNeighborDeg.m

Computes the **average degree** of **neighboring nodes** for every vertex.

```
% Compute 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
```

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$
$> aveNeighborDeg(adj)$
$ans =$
2 2 2

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> aveNeighborDeg(bowtie)$
$ans =$
2.5000 2.5000 2.3333 2.3333 2.5000 2.5000

4.2.7 sortNodesBySumNeighborDegrees.m

Return graph node indices in order of decreasing nodal degree, and where there's equality, by the sum of neighbor degrees, and sum of neighbor degrees 2 links away, and so on. Ideas from [18] and [19].

```
% Sort nodes by degree, and where there's equality,
% by sum of neighbor degrees and then neighbors' neighbors degree and so on
% Ideas from s-max algorithm by Li et al 2005 "Towards a theory of scale-free graphs"
% and Guo, Chen, Zhou, "Fingerprint for Network Topologies"
%
% INPUTS: adjacency matrix, 0s and 1s, nxn
% OUTPUTS: sorted (decreasing) sequence (nx1), where n is the number of
%                                          rows/cols of the adjacency
%
% Other routines used: degrees.m, kneighbors.m
% GB: last update, Oct 4, 2012
```

Examples:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> sortNodesBySumNeighborDegrees(bowtie)$
$ans =$
4
3
6
5
2
1

$adj = [0\ 1\ 1\ 0; 1\ 0\ 1\ 0; 1\ 1\ 0\ 1; 0\ 0\ 1\ 0];$
$> sortNodesBySumNeighborDegrees(adj)$
$ans =$
3
2
1
4

4.2.8 sortNodesByMaxNeighborDegree.m

Return nodal indices sorted nodes by degree, and where there's equality, by maximum neighbor degree.

```
% Sort nodes by degree, and where there's equality, by maximum neighbor degree
% Ideas from Guo, Chen, Zhou, "Fingerprint for Network Topologies"
%
% INPUTS: adjacency matrix, 0s and 1s, nxn
% OUTPUTS: sorted (decreasing) sequence of nodal indices (nx1)
%
% Other routines used: degrees.m, kneighbors.m
% GB: last updated, Oct 4, 2012
```

Example:
$adj = [0\ 1\ 1\ 1; 1\ 0\ 0\ 0; 1\ 0\ 0\ 0; 1\ 0\ 0\ 0];$
$> sortNodesByMaxNeighborDegree(adj)$
$ans =$
1
4
3
2

4.2.9 closeness.m

Compute the **closeness centrality** for all vertices. The closeness of a given node is defined as the inverse of the sum of distances to all other nodes.

```
% 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
```

Examples:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> closeness(bowtie)$
$ans =$
0.10000
0.10000
0.14286
0.14286
0.10000
0.10000

$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$
$> closeness(adj)$
$ans =$
0.50000
0.50000
0.50000

4.2.10 nodeBetweennessSlow.m [14]

Compute the **betweenness centrality** of all vertices [14]. Betweenness is proportional to the number of shortest paths that go through a node.

```
% 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.
% Note 2: Bug: currently the routine does not return the correct
%       betweenness values for all nodes if the graph contains an even
%       cycle. That is because an even cycle results in multiple shortest
%       paths between some of the nodes, and this function chooses one
%       shortest path for every pair of nodes. Fix TBA.
%
% INPUTS: adjacency or distances matrix (nxn)
% OUTPUTS: betweeness vector for all vertices (1xn)
%
% Other routines used: numNodes.m, shortestPathDP.m
% GB: June 4, 2013
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> nodeBetweennessSlow(bowtie)$
$ans =$
0.000 0.000 0.400 0.400 0.000 0.000

4.2.11 nodeBetweennessFaster.m [14]

A faster **node betweenness** algorithm (same input/output as 4.2.10).

```
% Betweenness centrality measure: number of shortest paths running through a vertex.
% Compute for all vertices, using Dijkstra's algorithm.
%
% Note 1: The graphs has to be connected.
% Note 2: Bug: currently the routine does not return the correct
%       betweenness values for all nodes if the graph contains an even
%       cycle. That is because an even cycle results in multiple shortest
%       paths between some of the nodes, and this function chooses one
%       shortest path for every pair of nodes. Fix TBA.
%
% INPUTS: adjacency or distances matrix, nxn
% OUTPUTS: betweeness vector for all vertices (1xn)
%
% Other routines used: dijkstra.m
% GB: June 4 2013
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> nodeBetweennessFaster(bowtie)$
$ans =$
0.000 0.000 0.400 0.400 0.000 0.000

4.2.12 edgeBetweenness.m [15]

Compute **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
```

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$ % undirected 3-node cycle: all edges should have equal betweenness
$> edgeBetweenness(adj)$
$ans =$
2 1 0.16667
3 1 0.16667
1 2 0.16667
3 2 0.16667

1 3 0.16667
2 3 0.16667

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> edgeBetweenness(bowtie)$
$ans =$
2 1 0.033333
3 1 0.133333
1 2 0.033333
3 2 0.133333
1 3 0.133333
2 3 0.133333
4 3 0.300000
3 4 0.300000
5 4 0.133333
6 4 0.133333
4 5 0.133333
6 5 0.033333
4 6 0.133333
5 6 0.033333

4.2.13 eigenCentrality.m

The **eigen-centrality vector** is the eigenvector corresponding to the largest eigenvalue of the adjacency matrix. The i^{th} component of this eigenvector gives the centrality score of the i^{th} 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
```

Examples:
$> eigenCentrality([0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0])$
$ans =$
0.57735
0.57735
0.57735

$adj = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0];$
$> eigenCentrality(adj)$
$ans =$
0.70711
0.50000
0.50000

4.2.14 clustCoeff.m

Compute two **clustering coefficients**: one based on loops, and one based on local clustering. 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].

```
% Compute 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
```

Examples:
$triangle = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0]$;
$> clustCoeff(triangle)$
$ans = 1$

$adj = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0]$;
$> clustCoeff(adj)$
$ans = 0$

4.2.15 weightedClustCoeff.m [16]

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

```
function 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
```

Examples:
$adj = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0];$
$> weightedClustCoeff(adj)$
$ans =$
0
0
0

% an arbitrary weighted (symmetric) matrix
$adj =$

0	2	1	1
2	0	3	0
1	3	0	0
1	0	0	0

$> weightedClustCoeff(adj)$
$ans =$
0.37500
1.00000
1.00000
0.00000

4.2.16 pearson.m [17]

Pearson degree correlation: the degree-degree correlation in a graph. 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
```

Examples:
$star = [0\ 1\ 1\ 1\ 1; 1\ 0\ 0\ 0\ 0; 1\ 0\ 0\ 0\ 0; 1\ 0\ 0\ 0\ 0; 1\ 0\ 0\ 0\ 0];$ % a hub node with four spokes
$> pearson(star)$
$ans = -1$

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> pearson(bowtie)$
$ans = -0.16667$

4.2.17 richClubMetric.m [12]

The **rich club metric** is defined as the density of links among nodes with nodal degree k or higher. 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
```

Examples:
$undirected\_triangle = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0]$;
$> richClubMetric(undirected\_triangle, 2)$
$ans = 1$

$> richClubMetric(undirected\_triangle, 3)$
$ans = 0$

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0]$;
$> richClubMetric(bowtie, 2)$ % should be the same as $linkDensity(bowtie)$
$ans = 0.46667$

$> richClubMetric(bowtie, 3)$
$ans = 1$ % all degree-3 nodes are connected to each other

4.2.18 sMetric.m [18]

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
```

Examples:
$undirected\_triangle = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$
$> sMetric(undirected\_triangle)$
$ans = 24 \qquad \% \sum_{edges}(2 \times 2)$

$one\_edge = [0\ 1; 0\ 0];$
$> sMetric(one\_edge)$
$ans = 1$

5 Distances

5.1 Basic concepts

Distances in graphs are interesting to many fields, from transportation, logistics to social science and media. Milgram's letters experiment [1] brought attention to the distance between people in social networks via acquaintance. The *six degrees of separation* term refers to distance.

The simplest types of distance notion in a graph is the **shortest path**. The shortest path from a node i to a node j is a path of edges that connects the two nodes, and it is the shortest possible in number of edges. If the edges have weight or cost, or there are constraints, the shortest path definition can vary. The prefered way to travel by air from Boston to Los Angeles could be the fastest - direct, or the cheapest - through various airport hubs, or a combination of the two.

Another distance-related notion is **small-world networks**. These are networks in which the distances are short with respect to the size of the network. More precisely, the diameter of the graph scales as $log(n)$, where n is the number of nodes. Figure 5 shows an example of a small-world graph.

In this section most routines use the basic shortest path algorithm, by Dijkstra (see *simpleDijkstra.m*, Section 5.2.1 and *dijkstra.m*, Section 5.2.2).

The **diameter** is the maximum shortest path over the shortest paths for all pairs of nodes.

The **average path length** is the average shortest path.

The diameter and the average path length are just two summaries of the distance distribution. The **distance distribution** is the frequency distribution of distances in the graph. Distance distributions can reveal graph structure. For example, Figure 6 shows that the Lufthansa network and a random graph with the same size and density have different distance distributions.

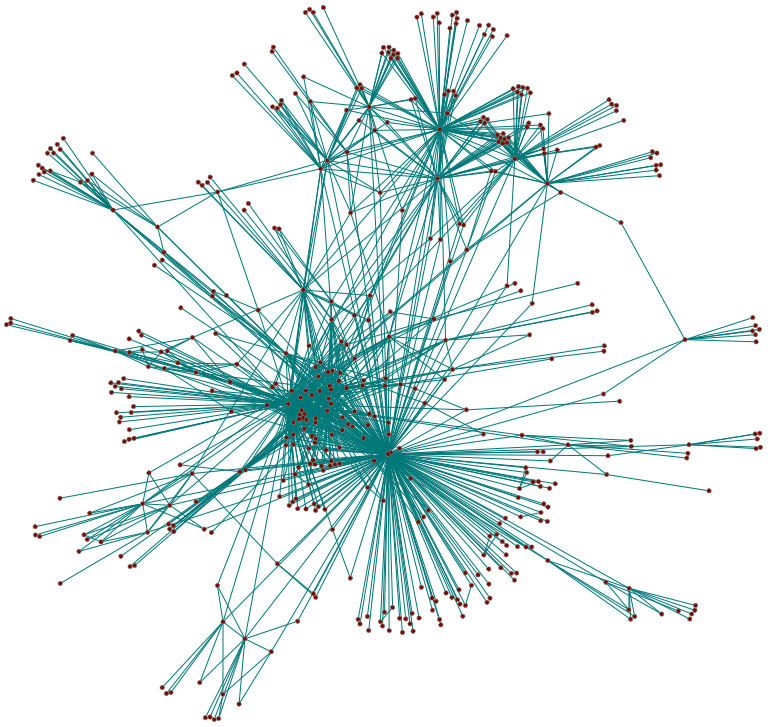


Figure 5: Small-world network example: the graph of the Lufthansa air routes from July 2006; 470 nodes (airports) and diameter of 5 ($log(470) \sim 6.15$). So 5 is the largest number of hops that need to be traveled to reach any airport from any starting location.

5.2 Routines

5.2.1 simpleDijkstra.m

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

```
% Implementation of 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 the start node to all other nodes, 1xn
%
% Note: Works for a weighted/directed graph.
% GB: last updated, September 28, 2012
```

Example:
$adj = [0\ 1; 0\ 0];$ % a single directed edge
$d = simpleDijkstra(adj, 1);$
$> d$
$d =$
$0 \quad 1$

$d = simpleDijkstra(adj, 2);$
$> d$
$Inf \quad 0$

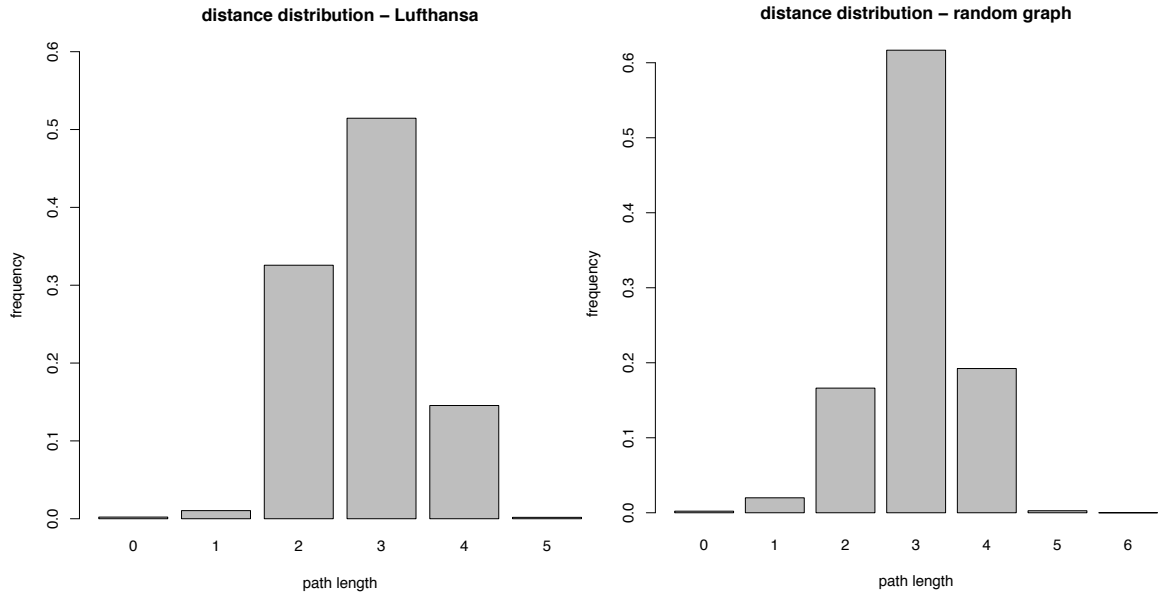


Figure 6: The distance distribution of the Lufthansa network from Figure 5 versus the distance distribution of a random graph with the same size (470 nodes) and same density (0.02).

5.2.2 dijkstra.m

Dijkstra's algorithm. This routine returns the shortest distances, as well as the paths.

```
% Dijkstra's algorithm.
%
% INPUTS: adj - adjacency matrix (nxn), s - source node, target - target node
% OUTPUTS: distance, d and path, P (from s to target)
%
% Note: if target==[], then dist and P include all distances and paths from s
% Other routines used: adj2adjL.m
% GB: last updated, Oct 5, 2012
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$[d, P] = dijkstra(bowtie, 1, 2)$ % distance and path from node 1 to node 2
$d = 1$
$P =$
1 2

$[d, P] = dijkstra(bowtie, 6, 2)$ % distance and path from node 6 to node 2
$d = 3$
$P =$
6 4 3 2

5.2.3 shortestPathDP.m [20]

Shortest path algorithm using dynamic programming. Returns the minimum weight path length and the route. Ideas from [20].

```
% Shortest path algorithm using dynamic programming.
% Note 1: Valid for directed/undirected network.
% Note 2: if links have weights, they are treated as distances.
% Source: D. P. Bertsekas, Dynamic Programming and Optimal Control,
%                              Athena Scientific, 2005 (3rd edition)
%
% INPUTs: L - (cost/path lengths matrix), s - (start/source node),
%                                      t - (end/destination node)
%                                  steps - number of arcs allowable
% OUTPUTS:
%       route - sequence of nodes on optimal path, at current stage
%       route(k,i).path - best route from "i" to destination "t" in "k" steps
%       route_st - best route from "s" to "t"
%       J_st - optimal cost function (path length) from "s" to "t"
%       J(k,i) - distance from node "i" to "t" in "k" steps
%
% GB: last updated, Oct 5 2012
```

Examples:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 0\ 1\ 1\ 0]$;
$[J\_st, route\_st, J, route] = shortestPathDP(bowtie, 1, 6, 1)$;
$> J\_st$
$J\_st = Inf$

$[J\_st, route\_st, J, route] = shortestPathDP(bowtie, 1, 6, 6)$;
$> J\_st$
$J\_st = 3$
$> route\_st$
$route\_st =$
1 3 4 6

5.2.4 kneighbors.m

Returns the list of nodes k links away from a start node i. If the graph is undirected, i will appear among the list of its own k-neighbors if k is even.

```
% Finds the number of k-neighbors (k links away) for every node
%
% INPUTS: adjacency matrix (nxn), start node index, k - number of links
% OUTPUTS: vector of k-neighbors indices
%
% GB: last updated, Oct 7 2012
```

Example: For the bowtie graph, starting at 1, the nodes two links away are: 1 ($1 \to 2 \to 1$), 2 ($1 \to 3 \to 2$), 3 ($1 \to 2 \to 3$) and 4 ($1 \to 3 \to 4$).
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 0\ 1\ 1\ 0]$;
$> kneighbors(bowtie, 1, 2)$
$ans =$
1 2 3 4

5.2.5 kminNeighbors.m

Returns the indices of nodes that are minimum k links away from a given node i.

```
% Finds the number of "kmin"-neighbors (k links away at a minimum) for every node
% If nodes are k-links away due to loops (so they appear as m-neighbours, m<k),
%                                                          they are not counted
%
% INPUTS: adjacency matrix (nxn), start node index, k - number of links
% OUTPUTS: vector of "kmin"-neighbor indices
%
% GB: last update, Oct 7 2012
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> kminNeighbors(bowtie, 2, 1)$
$ans =$
1 3

$> kminNeighbors(bowtie, 3, 2)$
$ans =$
5 6

5.2.6 diameter.m

The **diameter** is the longest shortest path in the graph.

```
% The longest shortest path between any two nodes nodes in the network.
%
% INPUTS: adjacency matrix, nxn
% OUTPUTS: network diameter
%
% Other routines used: simpleDijkstra.m
% GB: last updated, Oct 8 2012
```

Examples:
$undirected\_triangle = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$
$> diameter(undirected\_triangle)$
$ans = 1$

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> diameter(bowtie)$
$ans = 3$

5.2.7 avePathLength.m

The **average path length** is the average shortest path.

```
% Compute average path length for a network - the average shortest path
% Note: works for directed/undirected networks
%
% INPUTS: adjacency (or weights/distances) matrix, nxn
% OUTPUTS: average path length
%
% Other routines used: simpleDijkstra.m
% GB: Oct 8, 2012
```

Examples:
$undirected\_triangle = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$
$> avePathLength(undirected\_triangle)$
$ans = 1$

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> avePathLength(bowtie)$
$ans = 1.8000$

5.2.8 smoothDiameter.m [21]

A relaxed or smoothed definition of diameter: the number d at which a threshold fraction p of pairs of nodes are at distance at most d. This diameter can be non-integer using interpolation. The definition and the idea come from [21].

```
% A relaxed/smoothed definition of diameter: the number "d" at which
% a threshold fraction "p" of pairs of nodes are at distance at most
%                        "d". Can be non-integer using interpolation.
%
% Idea: Leskovec et al, "Graphs over Time: Densication Laws,
%                      Shrinking Diameters and Possible Explanations"
%
% Input: adjacency matrix of graph and diameter threshold, p in [0,1]
% Output: relaxed or "effective" diameter
%
% Other routines used: simpleDijkstra.m
% GB: last updated, Oct 8 2012
```

Examples:
Take an undirected 3-node cycle graph, and suppose $p = 1$. For 100% of node pairs, the smooth diameter should be the same as the classic diameter (section 5.2.6).

$A = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0];$
$p = 1;$
$> smoothDiameter(A, p)$
$ans = 1$

Now consider the *bowtie* graph and $p = 0.5$. Of all node pairs, 47% are at a distance at most 1, and 73% are at a distance at most 2. For a fraction that is in between, the diameter is interpolated between 1 and 2.

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> smoothDiameter(bowtie, 0.5)$
$ans = 1.1250$

5.2.9 closeness.m

Closeness can be classified under centralities, as well as distance measures. This routine is described in Section 4.2.9.

5.2.10 vertexEccentricity.m

The vertex eccentricity (of node i) is defined as the maximum distance to any other node.

```
% Vertex eccentricity - the maximum distance to any other vertex.
%
% Input: adjacency matrix, nxn
% Output: vector of eccentricities for all nodes, 1xn
%
% Other routines used: simpleDijkstra.m
% GB: last updated, Oct 10, 2012
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> vertexEccentricity(bowtie)$
$and =$
3 3 2 2 3 3

5.2.11 graphRadius.m

The minimum vertex eccentricity is the graph radius. See Section 5.2.10.

```
% The minimum vertex eccentricity is the graph radius.
%
% Inputs: adjacency matrix (nxn)
% Outputs: graph radius
%
% Other routines used: vertexEccentricity.m
% GB: last updated, Oct 10 2012
```

Example: As seen in Section 5.2.10 above, the vector of vertex eccentricities for the *bowtie* graph is [3 3 2 2 3 3]. By the definition, the radius is expected to be 2.
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> graphRadius(bowtie)$
$ans = 2$

5.2.12 distanceDistribution.m

The distribution of distances in the graph here is defined as the fraction of pairs of nodes at a distance x, for all integer x between 1 and $n-1$. The number of pairs at some distance is divided by the total number of pairs $n(n-1)$ to obtain the fraction. This definition is used in [22].

```
% The number of pairs of nodes at a distance x,
%                       divided by the total number of pairs n(n-1)
% Source: Mahadevan et al, "Systematic Topology Analysis and
%                             Generation Using Degree Correlations"
% Note: The cumulative distance distribution (hop-plot) can be
% obtained by using  ddist(i)=length(find(dij<=i)); in line 28 instead.
%
% INPUTS: adjacency matrix, (nxn)
% OUTPUTS: distribution vector ((n-1)x1): {k_i} where k_i is the
%                 number of node pairs at a distance i, normalized
%
% Other routines used: simpleDijkstra.m
% GB: last updated, Oct 10 2012
```

Example: In the *bowtie* graph, there are 7 pairs of nodes at distance 1 (arc), 4 pairs of nodes at distance 2 and 4 pairs of nodes at distance 3. So the frequency of 1-arc paths is 7/(7+4+4) = 0.46667. For 2-arc and

3-arc paths it is 4/(7+4+4)=0.26667. There are no paths of length 4 and above.

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 0\ 1\ 1\ 0];$
$> dist = distanceDistribution(bowtie)$
$dist =$
0.46667 0.26667 0.26667 0.00000 0.00000

6 Simple Motifs

A *motif* is most generally a repeated pattern. In narrative, the motif is a recurrent element that can have a symbolic meaning and convey a mood or an underlying idea. In DNA, motifs are wide-spread repeating sequences of base pairs (usually short) that have some biological significance, say binding sites for proteins. In the networks literature motifs are usually defined as **frequently occurring subgraphs**. Whether a certain frequency is significant is usually determined by comparison with a corresponding random graph.

At the heart of the motifs question is the problem of identifying and counting subgraphs in any general graph. This is a hard problem, and the functions in this section do not come even close to addressing the general case. These are mostly simple routines that count loops and are used within other functions in this toolbox.

6.1 Routines

6.1.1 numConnTriples.m

```
% Count the number of connected triples in a graph.
% Note: works for undirected graphs only
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: integer - number of connected triples
%
% Other routines used: kneighbors.m, loops3.m
% GB: last updated, October 4, 2012
```

Examples:
$triangle = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0]$
$> numConnTriples(triangle)$
$ans = 1$

$adj = [0\ 1\ 0; 1\ 0\ 0; 0\ 0\ 0]$ % an edge and a single node
$> numConnTriples(adj)$
$ans = 0$

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 0\ 1\ 1\ 0];$
$> numConnTriples(bowtie)$
$ans = 6$

6.1.2 numLoops.m

Calculate the number of independent loops/cycles. Use $G = m - n + c$.

```
% Calculate the number of independent loops (use G=m-n+c)
%      where G = num loops, m - num edges, n - num nodes,
%                       c - number of connected components
% This is also known as the "cyclomatic number": the number of edges
% that need to be removed so that the graph doesn't have cycles.
%
% INPUTS: adjacency matrix, nxn
% OUTPUTs: number of independent loops (or cyclomatic number)
%
% Other routines used: numNodes.m, numEdges.m, findConnComp.m
% GB: last updated, Oct 5 2012
```

Examples:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> numLoops(bowtie)$
$ans = 2$

$undirected\_tree = [0\ 1\ 1; 1\ 0\ 0; 1\ 0\ 0]$
$> numLoops(undirected\_tree)$
$ans = 0$

6.1.3 loops3.m

Count all cycles of size 3 in the graph.

```
% Calculate number of loops/cycles of length 3
%
% INPUTs: adj - adjacency matrix, nxn
% OUTPUTs: L3 - number of triangles (loops of length 3)
%
% Note: Valid for an undirected network.
% GB: last updated, Oct 5, 2012

function L3 = loops3(adj)

L3 = trace(adj^3)/6;   % trace(adj^3)/3!
```

Examples:
$square = [0\ 1\ 0\ 1; 1\ 0\ 1\ 0; 0\ 1\ 0\ 1; 1\ 0\ 1\ 0];$
$> loops3(square)$
$ans = 0$

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> loops3(bowtie)$
$ans = 2$

6.1.4 loops4.m

Returns all 4-tuples of nodes that form cycles of size 4.

```
% Find cycles of length 4 in a graph;
% Note 1: Quite basic and slow.
% Note 2: Assumes undirected graph.
%
% INPUTs: adj - adjacency matrix of graph, nxn
% OUTPUTs: 4-tuples of nodes that form 4-cycles;
%          format: {"n1-n2-n3-n4","n5-n6-n7-n8",...}
%
% Other functions used: adj2adjL.m
% GB: last updated, Oct 5 2012
```

Example:
$square = [0\ 1\ 0\ 1; 1\ 0\ 1\ 0; 0\ 1\ 0\ 1; 1\ 0\ 1\ 0];$ % a cycle of size 4
$> loops4(square)$
$ans =$
{
$[1, 1] = 1 - 2 - 3 - 4$
}

6.1.5 numStarMotifs.m

Number of k-tuples that form a "star" subgraph, i.e. a hub node with $k-1$ spokes.

```
% Calculate the number of star motifs of given (subgraph) size.
% Note 1: Easily extendible to return the actual stars as k-tuples of nodes.
% Note 2: Star of size 1 is the trivial case of a single node.
%
% INPUTs: adjacency list {} (1xn), k - star motif size
% OUTPUTs: number of stars with k nodes (k-1 spokes)
%
% GB: last updated, Oct 5, 2012
```

Examples:
% a star with 3 spokes has 3 *sub-stars* with 3 nodes total.
% $adjL\{1\} = [2, 3, 4],\ k = 3,\ so\ numStarMotifs(adjL, k)$
$> s = numStarMotifs(\{\ [2, 3, 4]\ \}, 3)$
$s = 3$

$bowtieAdjL = \{[2, 3], [1, 3], [1, 2, 4], [3, 5, 6], [4, 6], [4, 5]\}$
$> s = numStarMotifs(bowtieAdjL, 4)$
$s = 2$

7 Linear Algebra Routines

These functions concern mostly the spectrum of the adjacency matrix. Most of them are auxiliary code (called within) for other functions in this toolbox.

7.1 Routines

7.1.1 laplacianMatrix.m

The **Laplacian** of the graph is defined as the degree matrix minus the adjacency.

```
  % The Laplacian matrix defined for a *simple* graph
  % Def: the difference b/w the diagonal degree and the adjacency matrices
  % Note: This is not the normalized Laplacian
  %
  % INPUTS: adjacency matrix, nxn
  % OUTPUTs: Laplacian matrix, nxn
  %
  % GB: last updated, Oct 10 2012
```

The **normalized Laplacian** can be computed as:

```
def normLaplacianMatrix(adj):

 n=length(adj);
 deg = sum(adj); % for other than simple graphs,
                 %   use [deg,~,~]=degrees(adj);

 L=zeros(n);
 edges=find(adj>0);

 for e=1:length(edges)
     [ii,jj]=ind2sub([n,n],edges(e))
     if ii==jj; L(ii,ii)=1; continue; end
     L(ii,jj)=-1/sqrt(deg(ii)*deg(jj));
 end
```

Example:
$adj = [0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 0]$ % 3×3 identity matrix
$> laplacianMatrix(adj)$
$ans =$

$$\begin{array}{rrr} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{array}$$

7.1.2 graphSpectrum.m

The **graph spectrum** is the list of eigenvalues of the Laplacian of the graph.

```
  % The eigenvalues of the Laplacian of the graph.
  %
  % INPUTs: adjacency matrix, nxn
  % OUTPUTs: laplacian eigenvalues, sorted
  %
  % Other routines used: laplacianMatrix.m
  % GB: last updated, Oct 10 2012
```

Examples:
$> graphSpectrum([0\ 1; 0\ 0])$
$ans =$
1
0

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> graphSpectrum(bowtie)$

$ans =$
$4.5616e+00$
$3.0000e+00$
$3.0000e+00$
$3.0000e+00$
$4.3845e-01$
$2.4286e-16$

7.1.3 algebraicConnectivity.m

The **second smallest eigenvalue** of the Laplacian of the adjacency matrix of the graph.

```
% The algebraic connectivity of a graph:
%                  the second smallest eigenvalue of the Laplacian
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: algebraic connectivity
%
% Other routines used: graphSpectrum.m
% GB: last updated, Oct 10 2012
```

Examples:
% connected graph
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> algebraicConnectivity(bowtie)$
$ans = 0.43845$

% disconnected graph
$> adj = [0\ 1\ 0; 1\ 0\ 0; 0\ 0\ 0];$
$> algebraicConnectivity(adj)$
$ans = 0$

7.1.4 fieldlerVector.m

Fiedler vector: the vector corresponding to the second smallest eigenvalue of the Laplacian matrix.

```
% The vector corresponding to the second smallest eigenvalue of
%                                        the Laplacian matrix
% INPUTs: adjacency matrix, nxn
% OUTPUTs: fiedler vector, nx1
%
% Other routines used: laplacianMatrix.m
% GB: last updated, Oct 10 2012
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> fiedlerVector(bowtie)$
$ans =$

-0.46471
-0.46471
-0.26096
0.26096

0.46471
0.46471

7.1.5 eigenCentrality.m

This routine can be classified both in Linear Algebra, as well as in Centralities. See Section 4.2.13.

7.1.6 graphEnergy.m [23]

The **graph energy** is defined as the sum of the absolute values of (the real components of) the eigenvalues of the adjacency matrix. This definition and more about graph energy can be found in [23].

```
% Graph energy defined as: the sum of the absolute values of the
%      real components of the eigenvalues of the adjacency matrix.
% Source: Gutman, The energy of a graph, Ber. Math. Statist.
%                  Sekt. Forsch-ungszentram Graz. 103 (1978) 1-22.
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: graph energy
%
% GB: last updated, Oct 10 2012
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$> graphEnergy(bowtie)$
$ans = 8.2925$

8 Modularity

8.1 Basic modularity notions

Modularity has many names in the networks literature: partitioning, community finding, clustering. These notions are often not exactly the same, but are flavors of the same problem: how to split a graph into *modules* or *communities* of nodes in some meaningful way. The focus of this section is quite narrow: algorithms that identify communities or subgraphs that are usually more tightly interconnected within the community than to nodes outside of its boundary. Again, this difference has to be compared to some null (random) graph model.

An example of a random graph built with four tightly-knit communities, in the sense defined above, is shown in Figure 7. A definition of a modularity metric, which measures how *good* a certain community split is, is presented in Section 8.2.5.

Most of the functions in the following section reflect Mark Newman's work on modularity. In addition to many publications on this topic, his site also contains code and some data sets.

8.2 Routines

8.2.1 simpleSpectralPartitioning.m

Using the fiedler vector (Section 7.1.4) to assign nodes to partitions of pre-determined size.

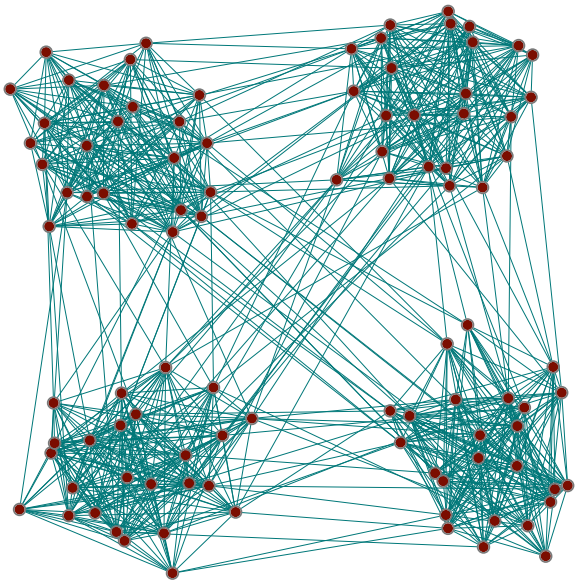


Figure 7: Example of a random modular graph with 100 nodes and 4 communities. Created with *randomModularGraph.m*, Section 9.2.8).

```
% Uses the fiedler vector to assign nodes to groups.
%
% INPUTS: adjacency matrix (nxn), k - desired number
%         of nodes in groups [n1, n2, ..], [optional].
%         The default k is 2.
% OUTPUTs: modules - [k] partitioned groups of nodes
%
% Example:
% simpleSpectralPartitioning(random_modular_graph(100,4,0.15,0.9),
%                                                  [25 25 25 25])
% Other functions used: fiedlerVector.m
% Note: To save the plot at the end of the routine, uncomment:
%                   print filename.pdf (or filename.extension)
% GB: last updated, Oct 10 2012
```

Example: Suppose G is a random graph of 100 nodes created to have 4 well-expressed clusters of more tightly connected nodes. This can be done with the routine *randomModularGraph*. See Section 9.2.8 for how to use. Also, suppose that the goal is to assign the nodes into 4 partitions. Then the input to *simpleSpectralPartitioning* looks like:

$adj = randomModularGraph(100,\ 4,\ 0.15,\ 0.9);$
$modules = simpleSpectralPartitioning(adj, [25\ 25\ 25\ 25])$

The output *modules* will be a list of four vectors of size 1x25. The routine also returns a plot, of the entries of the fiedler vector sorted, as well as a dot plot of the adjacency matrix, where the rows/columns are sorted to reflect the expected partitions. An example is shown in Figure 8.

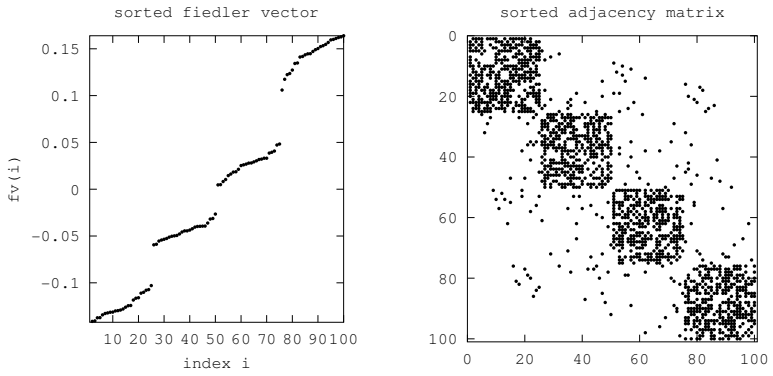


Figure 8: Example of the plot output of *simpleSpectralPartitioning.m*. The input in this case is a random modular graph with 4 clusters of nodes.

8.2.2 newmanGirvan.m [15]

This is a community finding algorithm, by Newman and Girvan, based on the notion of edge betweenness (see Section 4.2.12). It is described in [15].

```
% Newman-Girvan community finding algorithm
% Source: Newman, Girvan, "Finding and evaluating
%                          community structure in networks"
% Algorithm idea:
% 1. Calculate betweenness scores for all edges in the network.
% 2. Find the edge with the highest score and remove it from the network.
% 3. Recalculate betweenness for all remaining edges.
% 4. Repeat from step 2.
%
% INPUTs: adjacency matrix (nxn), number of modules (k)
% OUTPUTs: modules (components) and modules history -
%          each "current" module, Q - modularity metric
%
% Other routines used: edgeBetweenness.m, isConnected.m,
%                findConnComp.m, subgraph.m, numEdges.m
% GB: last updated, Oct 11 2012
```

Example: For the bowtie graph in Figure 3, with 2 desired components, the *newmanGirvan.m* routine returns:

$adj = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0]$
$modules = newmanGirvan(adj, 2);$
$> modules\{1\}$
$ans =$

1 2 3

$> modules\{2\}$
$ans =$

4 5 6

If two outputs are specified, namely,
$[modules, modules\_hist] = newmanGirvan(adj, 2)$,
then the history of consecutive partitions is returned also. In the case of the bowtie graph, the first module in the history is simply the entire graph, so:

$> modules\_hist\{1\}$
$ans =$

1 2 3 4 5 6

$> modules\_hist\{2\}$
$ans =$

1 2 3

The third output, the modularity score Q, is computed as in equation 5 in [15]. Define e_{ij} as the number of edges between module/community i and community j. Then e_{ii} is the number of edges within community i, and $\sum_j e_{ij}$ is the number of edges from community i to other communities. If $\sum_j e_{ij} = a_i$, then the modularity is computed as:

$$Q = \sum_{module\ i} (e_{ii} - a_i^2) \tag{7}$$

For the bowtie example:
$[\sim, \sim, Q] = newmanGirvan(adj, 2)$;

$> Q$
$Q = 0.20408$

8.2.3 newmanEigenvectorMethod.m [24] [25]

Find the "optimal" number of communities in a network. Algorithm described in [24] and [25].

```
% Find the "optimal" number of communities given a network using an eigenvector method
% Source: MEJ Newman: Finding community structure using the eigenvectors of matrices,
%                                                              arXiv:physics/0605087
%         Newman, "Modularity and community structure in networks",
%                                                    arxiv.org/pdf/physics/0602124v1
%
% Q=(s^T)Bs, Bij=Aij-kikj/2m
% Bij^g=Bij - delta_ij * (sum k over g)B_ik
% Bij^g=(Aij-kikj/2m)-delta_ij(sum k over g)(A(g)_ik-deg(g)_i deg(k)_j/2(m_g))
% Bij^g=(Aij-kikj/2m)-delta_ij(k_i^(g)-k_i*sum(deg^(g)/2m)
%
% STEPS:
% 1 define current modularity matrix
% 2 compute eigenvector corresp. to largest eigenvalue
% 3 separate into 2 modules based on signs in eigenvector
% terminate when max eigenvalue is 0 for all subgraphs
%
% Other functions used: numEdges.m, degrees.m, subgraph.m, isConnected.m
% GB: last modified, Oct 12, 2012
```

Example: For the bowtie graph adjacency, the *newmanEigenvectorMethod.m* routine returns the following:

$adj = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0]$
$modules = newmanEigenvectorMethod(adj);$

$> modules\{1\}$
$ans =$

1 2 3

$> modules\{2\}$
$ans =$

4 5 6

8.2.4 newmanCommFast.m [26]

A fairly fast community finding algorithm which computes the modularity metric for every possible number of communities from 1 (all nodes) to n (every node is in its own community). This is described in [26].

```
% Fast community finding algorithm by M. Newman
% Source: "Fast algorithm for detecting community
%                   structure in networks", Mark Newman
%
% INPUTs: adjacency matrix, nxn
% OUTPUTs: group (cluster) formation over time,
%          modularity metric for each cluster breakdown
%
% Other functions used: numEdges.m
% Note: To save the plot generated in this routine:
%       uncomment "print newmanCommFast_example.pdf"
%
% GB: last updated, Oct 12 2012
```

Example: For the bowtie graph of Figure 3, the output of *newmanCommFast.m* looks like:

$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0]$
$[groups\_hist, Q] = newmanCommFast(bowtie);$

$> length(groups\_hist)$
$ans =\ 6$

$> groups\_hist\{5\}$
$ans =$
$\{$
$[1,1] = [1\ \ 2\ \ 3]$
$[1,2] = [4\ \ 5\ \ 6]$
$\}$

This routine also returns a plot of number of communities versus the modularity metric. The maximum modularity metric corresponds to the *best* partition of the nodes. For example, for the graph in Figure 7, the best partition is into 4 communities. This is shown in Figure 9.

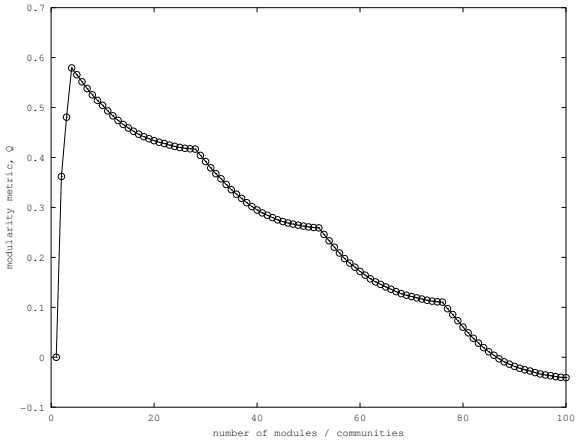


Figure 9: Example output of *newmanCommFast.m* for the random graph in Figure 7. There are 100 nodes, and the maximum modularity occurs at 4 communities.

8.2.5 modularityMetric.m [15] [26]

This is the **modularity metric** used in [15] (equation 5) and in [26]. Define e_{ij} as the number of edges between community i and community j. Then e_{ii} is the number of edges within community i, and $\sum_j e_{ij}$ is the number of edges from community i to other communities. If $\sum_j e_{ij} = a_i$, then the modularity is computed as:

$$Q = \sum_{module\ i} (e_{ii} - a_i^2)$$

This is equation 7 in Section 8.2.2. This definition makes sense for undirected graphs only.

```
% Computing the modularity for a given module/commnunity break-down
% Defined as: Q=sum_over_modules_i (eii-ai^2) (eq 5) in Newman and Girvan.
% eij = fraction of edges that connect community i to community j, ai=sum_j (eij)
%
% Source: Newman, Girvan, "Finding and evaluating community structure in networks"
%         Newman, "Fast algorithm for detecting community structure in networks"
%
% INPUTs: adjacency matrix, nxn
%         set of modules as cell array of vectors, ex: {[1,2,3],[4,5,6]}
% OUTPUTs: modularity metric, in [-1,1]
%
% Note: This computation makes sense for undirected graphs only.
% Other functions used: numEdges.m
% GB: last updated, October 16, 2012
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0]$
$Q = modularityMetric(\ \{\ [1,2,3],[4,5,6]\ \},\ bowtie\ )$
$> Q$
$Q =\ 0.20408$

Alternative to *modularityMetric.m*: First define some nomenclature.
A: adjacency matrix
m: total number of nodes

k_i: the nodal degree of node i
c_i: the community to which i belongs in some given partition
δ: the delta function, i.e. $\delta(x, y) = 1$ if $x = y$, and is 0 otherwise
e_{ss}: fraction of edges within community s (number of edges within community s divided by m)

$$
\begin{aligned}
& Q = \sum_{i,j} \frac{1}{2m}(A(i,j) - \frac{k_i k_j}{2m})\delta(c_i, c_j) && (8)\\
\Leftrightarrow\ & Q = \sum_{i,j} \frac{1}{2m} A(i,j)\delta(c_i, c_j) - \sum_{i,j} \frac{k_i k_j}{4m^2}\delta(c_i, c_j) \\
\Leftrightarrow\ & Q = \sum_{s \in modules} e_{ss} - \sum_{s \in modules} \sum_{(i,j), i \in s, j \in s} \frac{k_i k_j}{4m^2} \\
\Leftrightarrow\ & Q = \sum_{s \in modules} (e_{ss} - (\frac{\sum_{i, i \in s} k_i}{2m})^2)
\end{aligned}
$$

Equation 8 above is equation 9 in [28].

```
def modularityMetric(modules,adj):

% alternative: Q = sum_ij { 1/2m [Aij-kikj/2m]delta(ci,cj) } =
% = sum_ij Aij/2m delta(ci,cj) - sum_ij kikj/4m^2 delta(ci,cj) =
% = sum_modules e_ss - sum_modules (kikj/4m^2) =
% = sum_modules (e_ss - ((sum_i ki)/2m)^2)

n = numNodes(adj);
m = numEdges(adj);

% define the inverse of modules: node "i" <- module "c" if "i" in module "c"
mod={};
for mm=1:length(modules)
  for ii=1:length(modules{mm})
    mod{modules{mm}(ii)}=modules{mm};
  end
end

Q = 0;

for i=1:n
  for j=1:n

    if not(isequal(mod(i),mod(j))); continue; end

    Q = Q + (adj(i,j) - sum(adj(i,:))*sum(adj(j,:))/(2*m))/(2*m);

  end
end
```

8.2.6 louvainCommunityFinding.m [27]

The original paper describing this method is by Blondel et al [27]. There is also a dedicated site on the Louvain method.

In this method, the starting point is the alternative definition of the modularity from equation 8. For any given partition, i.e. set of clusters, where every node i belongs to a cluster c_i, the modularity is defined as:

$$Q = \sum_{i,j} \frac{1}{2m}(A(i,j) - \frac{k_i k_j}{2m})\delta(c_i, c_j)$$

See equation 8 for a reminder on what the notation means. The strategy then is to compare the modularity of a given partition with the *next* partition in which a node i is removed from its cluster and assigned to the cluster of a neighboring node j. The modularity gain (or loss) is $Q_1 - Q_0$, where the difference is the cluster assignment of node i.

To make this computation easier, notice that all terms in Q_0 and Q_1 are the same, except for the two clusters, which reflect the change in assignment of node i. Let c_{i0} be the cluster to which i belongs at first and c_{i1} be the next cluster of i. Moreover, for any two nodes j and k, such that $j \neq i$ and $k \neq i$, it is true that $Q_1(j,k) - Q_0(j,k) = 0$. Therefore, the only interesting terms in this difference are where i is present, and all other nodes concerned are assigned to c_{i0} or c_{i1}. Then,

$$\begin{aligned}
\Delta Q_i &= Q_1(i) - Q_0(i) \\
\Delta Q_i &= \frac{1}{2m}\sum_{j \in c_{i1}}(A(i,j) - \frac{k_i k_j}{2m}) - \frac{1}{2m}\sum_{j \in c_{i0}}(A(i,j) - \frac{k_i k_j}{2m}) \\
\Delta Q_i &= \frac{1}{2m}\left(\sum_{j \in c_{i1}} A(i,j) - \frac{k_i}{2m}\sum_{j \in c_{i1}} k_j - \sum_{j \in c_{i0}} A(i,j) + \frac{k_i}{2m}\sum_{j \in c_{i0}} k_j\right) \\
\Delta Q_i &= \frac{1}{2m}\left(\sum_{j \in c_{i1}} A(i,j) - \sum_{j \in c_{i0}} A(i,j)\right) - \frac{k_i}{4m^2}\left(\sum_{j \in c_{i1}} k_j - \sum_{j \in c_{i0}} k_j\right)
\end{aligned}$$

Notice that $\sum_{j \in c_{i1}} A(i,j)$ is the degree of i within c_{i1} and $\sum_{j \in c_{i0}} A(i,j)$ is the degree of i within c_{i0}. Also, $\sum_{j \in c_{i1}} k_j$ is twice the number of edges in c_{i1} and $\sum_{j \in c_{i0}} k_j$ is twice the number of edges in c_{i0}. Therefore, a more simplified way to write ΔQ_i is:

$$\Delta Q_i = \frac{1}{2m}(k_{i,c_{i1}} - k_{i,c_{i0}}) - \frac{k_i}{2m^2}(m_{c_{i1}} - m_{c_{i0}}) \tag{9}$$

This implementation of the Louvain method uses equation 9 at every step. Also, instead of iterating through the nodes sequentially until convergence (1 through n), this routine iterates through a random permutation of nodes at every step. This random shuffling improves the performance in practice (no theoretical justification).

```
% Implementation of a community finding algorithm by Blondel et al
% Source: "Fast unfolding of communities in large networks", July 2008
%          https://sites.google.com/site/findcommunities/
% Note 1: This is just the first step of the Louvain community finding
%         algorithm. To extract fewer communities, need to repeat with
%         the resulting modules themselves.
% Note 2: This works for undirected graphs only.
% Note 3: Permuting randomly the node order at every step helps the
%         algorithm performance. Unfortunately, node order in this
%         algorithm affects the results.
%
% INPUTs: adjancency matrix, nxn
% OUTPUTs: modules, and node community labels (inmodule)
%
% Other routines used: numEdges.m, kneighbors.m
% GB: last updated, Oct 17 2012
```

Example:
$bowtie = [0\ 1\ 1\ 0\ 0\ 0; 1\ 0\ 1\ 0\ 0\ 0; 1\ 1\ 0\ 1\ 0\ 0; 0\ 0\ 1\ 0\ 1\ 1; 0\ 0\ 0\ 1\ 0\ 1; 0\ 0\ 0\ 1\ 1\ 0];$
$[modules, inmodule] = louvainCommunityFinding(bowtie);$
$found\ \ 2\ modules$
$> modules =$
{
$[1, 1] =$
2 1 3
$[1, 2] =$
5 6 4
}

$> inmodule =$
{
$[1, 1] = 2$
$[1, 2] = 2$
$[1, 3] = 2$
$[1, 4] = 4$
$[1, 5] = 4$
$[1, 6] = 4$
}

9 Building graphs

9.1 Graph construction algorithms

The need to **construct a graph** comes from the desire to explain or mimic some existing behavior or a phenomenon in a system that can be modeled as a network. A proper model for a real-world network should have some intuitive steps as well as stochasticity built in. Of course, this is not always the case. Deterministic algorithms can also be useful, especially when looking for particular instances of graphs. The most popular graph building model is the random graph model by **Erdös and Rényi** [29]. Usually denoted as $G(n, p)$, this is a model of a graph with n nodes in which each pair of nodes is connected by an edge with probability p. The Erdös-Rényi graph construction, or random graph construction, is presented in Section 9.2.3. The *directed graph* version is in Section 9.2.4.

The **Price model** (Section 9.2.10, [2]) is an attempt to model the network of scientific citations. The **Newman-Gastner** model (Section 9.2.14, [34]) mimics *spatial distribution* networks, such as subway routes and water distribution systems.

Examples of *deterministic* algorithms are the **k-regular graph** routine (9.2.2) and the **Havel-Hakimi** algorithm (9.2.5) which builds a deterministic graph given a particular degree distribution. The *buildSmaxGraph.m* (Section 9.2.9) function constructs a deterministic graph with a maximum *s-metric* (Section 4.2.18), given the degree distribution [18].

There are many more models in the literature, including variations on the ones presented here. This section is easily extendible, and perhaps in the future can be structured differently to list the most general models and their derivatives.

9.2 Routines

9.2.1 canonicalNets.m

Building simple graphs such as trees and lattices with prescribed number of nodes and branch factor. In particular, the possible types of graphs are: trees (line, binary tree, general tree with branch factor b), cycles, lattices (triangle, square and hexagonal), hierarchies and cliques (complete graphs). Examples are shown in Figure 10.

```
% Build edge lists for simple canonical graphs, ex: trees and lattices
%
% INPUTS: number of nodes, network type, branch factor (for trees only).
%         Network types can be 'line','circle','star','btree','tree',
%            'hierarchy','trilattice','sqlattice','hexlattice', 'clique'
% OUTPUTS: edge list (mx3)
%
% Note: Produces undirected graphs, i.e. symmetric edge lists.
% Other functions used: symmetrizeEdgeL.m, adj2edgeL.m
% GB: last updated: Oct 27 2012
```

Examples:
$> canonicalNets(4,'line')$
$ans =$
1 2 1
2 3 1
3 4 1
2 1 1
3 2 1
4 3 1

$> canonicalNets(4,'tree',3)$
$ans =$
1 2 1
1 3 1
1 4 1
2 1 1
3 1 1
4 1 1

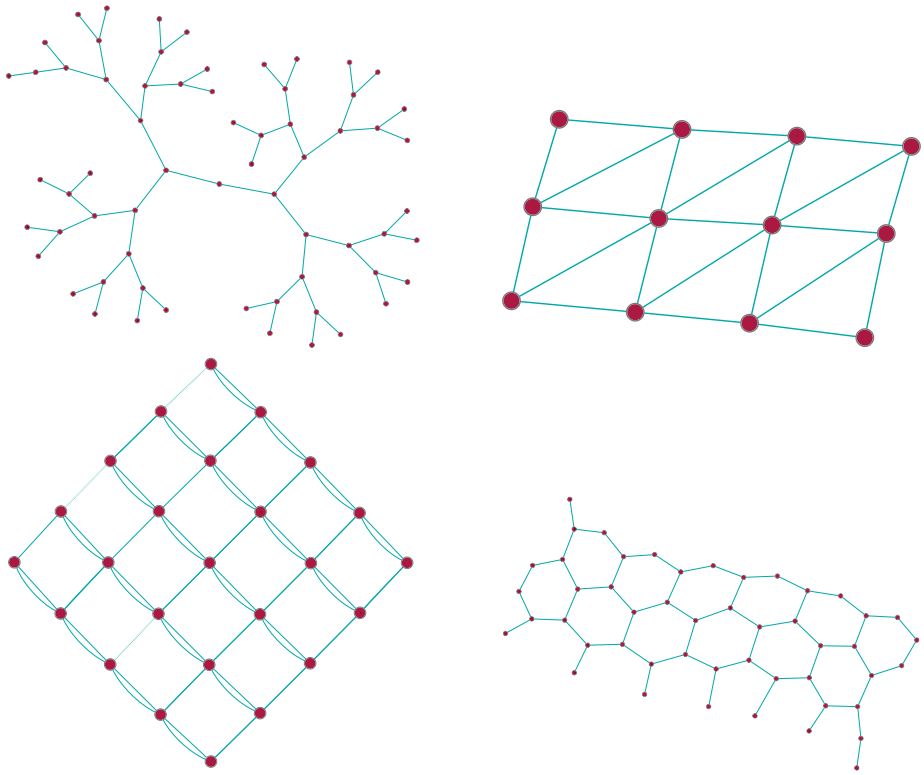


Figure 10: Examples of canonical graphs built by *canonicalNets.m*: binary tree, triangular lattice, square lattice and hexagonal lattice.

9.2.2 kregular.m

Simple routine for building k-regular graphs. In a **k-regular** graph all nodes have k links.

```
% Create a k-regular graph.
% Note: No solution for k and n both odd.
%
% INPUTs: n - # nodes, k - degree of each vertex
% OUTPUTs: el - edge list of the k-regular undirected graph
%
% Other routines used: symmetrizeEdgeL.m
% GB: last updated, Oct 28 2012
```

Example:
$n = randi(40) + 10;$
$k = randi([2, n-1]);$
$if\ mod(k,2) == 1\ \&\ mod(n,2) == 1;\ n = n-1;\ end$ % no solution for both k and n odd
$el = kregular(n,k);$
$adj = edgeL2adj(el);$
$> assert(\ degrees(adj),\ k * ones(1,n))$ % check that all degrees equal k

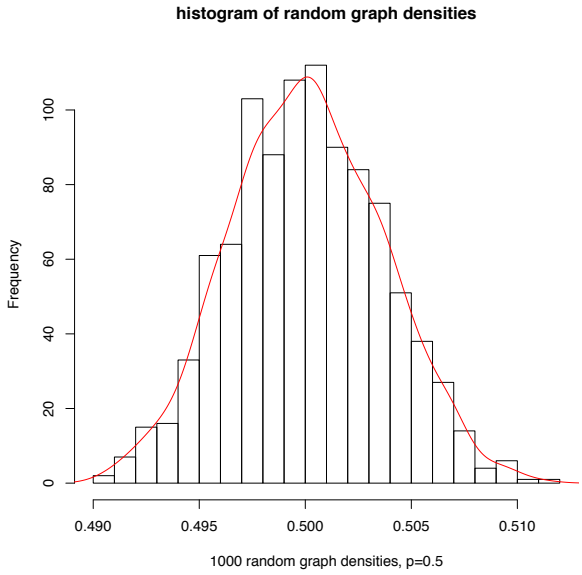


Figure 11: The distribution of link densities of 1000 random graphs with 200 nodes and $p = 0.5$.

9.2.3 randomGraph.m

The classical random graph model is by Erdös and Rényi [29]. In this model, links are added to randomly chosen pairs of nodes, starting with an initially empty graph. After the nodes are chosen uniformly at random, a link is added with some probability p. It is also possible to specify the number of edges, and keep adding edges randomly, until that number is reached.

```
% Random graph construction routine.
% Note 1: Default is Erdos-Renyi graph G(n,0.5)
% Note 2: Generates undirected, simple graphs only
%
% INPUTS:  N - number of nodes
%          p - probability, 0<=p<=1
%          E - fixed number of edges; if specified, p is irrelevant
% OUTPUTS: adj - adjacency matrix of generated graph (symmetric), nxn
%
% Other routines: numEdges.m
% GB: last updated, Oct 20, 2012
```

Example:
$> adj = randomGraph(1000, 0.4);$ % 1000 nodes and desired link density of 0.4
$> linkDensity(adj)$
$ans = 0.39956$ % the resulting density should be close to the specified density of 0.4

Figure 11 shows the distribution of link densities of 1000 random graphs with specified 0.5 probability of attachment. The densities are distributed normally around 0.5.

9.2.4 randomDirectedGraph.m

A random directed graph routine - similar to Section 9.2.3 but links are not added symmetrically.

```
% Random directed graph construction
% Note 1: if p is omitted, p=0.5 is default
% Note 2: no self-loops, no double edges
%
% Inputs:  n - number of nodes
%               p - probability, 0<=p<=1
% Output: adjacency matrix, nxn
%
% GB: last updated, Oct 21 2012
```

Example:
$adj = randomDirectedGraph(200, 0.3);$
$> linkDensity(adj)$
$ans = 0.30023$

9.2.5 graphFromDegreeSequence.m [30]

Construct a graph given the degree sequence, using the Havel-Hakimi algorithm [30]. This is a **deterministic** routine, i.e. for a given degree sequence, the resulting graph is always the same.

```
% Constructing a graph from a given degree sequence: deterministic
% Note: This is the Havel-Hakimi algorithm.
%
% Inputs: a graphic degree sequence, [d1,d2, ... dn],
%                            where di is the degree of the ith node
% Outputs: adjacency matrix, nxn
%
% GB: last updated, Oct 21 2012
```

Example:
$adj = randomGraph(100, 0.4);$
$deg = degrees(adj);$
$adjH = graphFromDegreeSequence(deg);$
$> assert(degrees(adjH), deg)$ % verify that the degree sequences of the two graphs are equal

9.2.6 randomGraphFromDegreeSequence.m [31]

Construct a **random** graph with a given degree sequence. The idea is to assign stubs equal to the degree of every node, and connect the stubs at random. This idea is usually attributed to Molloy and Reed [31].

```
% Constructing a random graph based on a given degree sequence.
% Idea source: Molloy M. & Reed, B. (1995) Random Structures and Algorithms 6, 161-179
%
% INPUTs: a graphic sequence of numbers, 1xn
% OUTPUTs: adjacency matrix of resulting graph, nxn
%
% Note: The simple version of this algorithm gets stuck about half
%       of the time, so in this implementation the last problematic
%       edge is rewired.
%
% Other routines used: adj2edgeL.m, rewireThisEdge.m, edgeL2adj.m
% GB: last updated, Oct 25 2012
```

Example:
$> randomGraphFromDegreeSequence([2\ 2\ 3\ 3\ 2\ 2])$
$ans =$
0 1 0 1 0 0

1 0 1 0 0 0
0 1 0 1 1 0
1 0 1 0 0 1
0 0 1 0 0 1
0 0 0 1 1 0

9.2.7 randomGraphDegreeDist.m

Construct a random graph given a degree distribution. Possible built-in distributions are *uniform*, *normal*, *binomial*, and *exponential*. A *custom* distribution can be specified discretely as follows. For a graph of n nodes, the possible degree for any node can be any number i from 1 to $n-1$. The custom distribution is specified as $P(any\ degree = i) = p_i$, where $\sum_{i=1}^{n-1} p_i = 1$.

```
% Construct a random graph given a degree distribution.
% The algorithm first generates the degree sequence by
%       drawing numbers from the specified distribution.
%
% INPUTs: number of nodes, n; distribution type, string
%         "distribution" can be: 'uniform', 'normal',
%         'binomial', 'exponential' and 'custom'
%         if 'custom', P has to be specified: P is a set
%         of probabilities, 1x(n-1), where P(i) is the
%         probability of a node having degree "i".
%         sum(P) = 1
% OUTPUTs: adjacency matrix of the random graph, nxn
%
% Other routines used: isGraphic.m, weightedRandomSample.m,
%                           randomGraphFromDegreeSequence.m
% GB: last updated, Oct 31 2012
```

Examples:
$adj = randomGraphDegreeDist(20,'uniform');$
$> numNodes(adj)$
$ans = 20$
$> degrees(adj)$
$ans =$
5 2 2 3 2 2 1 2 5 2 5 1 5 2 1 3 1 1 3 4

9.2.8 randomModularGraph.m

The idea in this model is to pre-assign nodes to modules and then connect them differently to nodes within the *same module* compared to nodes in *other modules* [32]. Namely, the average degree within a module is set to be r times the average degree computed only from links to "outside" nodes.

Suppose the graph has n nodes, c number of clusters/modules, overall probability of attachment p and ratio of inside to outside degree r. Then let the inside-module average degree be k_{in}, and the average outside-module degree be k_{out}. Using the average degree definition and the conventions above:

$$
\begin{aligned}
& k_{in}/k_{out} = r \\
& k_{in} + k_{out} = \bar{k} = p(n-1) \\
\Rightarrow\ & k_{in} = \frac{rp(n-1)}{r+1} \\
& k_{out} = \frac{p(n-1)}{r+1}
\end{aligned}
$$

Let p_{in} be the probability of attachment within a module, and p_{out} be the probability of attachment to nodes outside the module. These are related to the average degrees as follows:

$$k_{in} = p_{in}(n/c - 1),\ k_{out} = p_{out}(n - n/c)$$

where n/c is the number of nodes in a module, and $n - n/c$ is the number of nodes in all other modules combined.

$$
\begin{aligned}
p_{in} &= \frac{rpc(n-1)}{(r+1)(n-c)} \\
p_{out} &= \frac{pc(n-1)}{n(r+1)(c-1)}
\end{aligned}
$$

These are the p_{in} and p_{out} used in this routine.

```
% Build a random modular graph, given number of modules, and link densities
%
% INPUTs: number of nodes (n), number of modules (c), total link density (p),
%         and ratio of nodal degree to nodes within the same module
%         to the degree to nodes in other modules (r)
% OUTPUTs: adjacency matrix, modules to which the nodes are assigned
%
% GB: last updated, November 4, 2012
```

Example:
$[adj,\ modules] = randomModularGraph(100, 4, 0.1, 5);$
$> assert(numNodes(adj), 100)$
$> assert(length(modules), 4)$

Visual representation of this example is shown in Figure 12.

9.2.9 buildSmaxGraph.m [18]

Construct the graph with the **maximum possible s-metric**, given the degree sequence. The *s-metric* is the sum of products of degrees across all edges (Section 4.2.18). The algorithm for this construction is described in [18].

```
% Construct the graph with the maximum possible s-metric, given the degree
% sequence; the s-metric is the sum of products of degrees across all edges
% Source: Li et al "Towards a Theory of Scale-Free Graphs"
%
% INPUTs: degree sequence: 1xn vector of positive integers (graphic)
% OUTPUTs: edge list of the s-max graph, mx3
%
% GB: last updated, November 9 2012
```

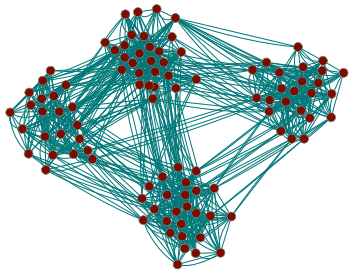


Figure 12: Visual representation of the example graph in Section 9.2.8: 100 nodes, 4 modules, 0.1 density and ratio of inside-module links to outside of 5.

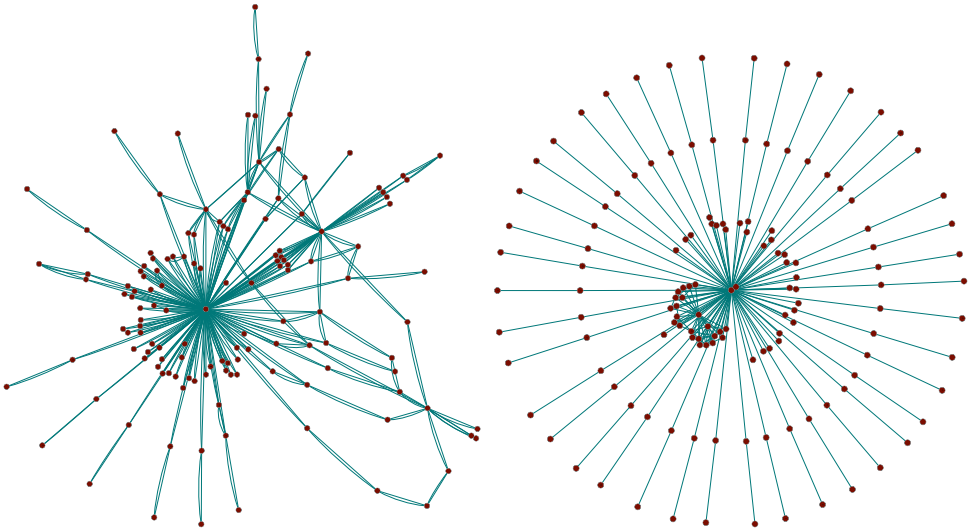


Figure 13: An original graph (left), constructed with the Price model (Section 9.2.10) and its s-max graph equivalent (right), i.e. the unique graph with the same degree distribution and maximum s-metric.

Example:
$adj = [\,];$
$while\ not(isConnected(adj)); adj = randomGraph(20, 0.1); end$
$sm = sMetric(adj);$
$elmax = buildSmaxGraph(degrees(adj));$
$adjmax = symmetrize(\ edgeL2adj(elmax)\ );$

$smax = sMetric(adjmax);$
$> assert(\ degrees(adjmax), degrees(adj)\ )$
$> assert(smax >= sm, true)$ % verify that the "s-max" graph has a higher s-metric

A visual representation of a graph and its "s-max" equivalent is shown in Figure 13.

9.2.10 PriceModel.m [2]

The **Price network growth** model for scientific citations [2][6].

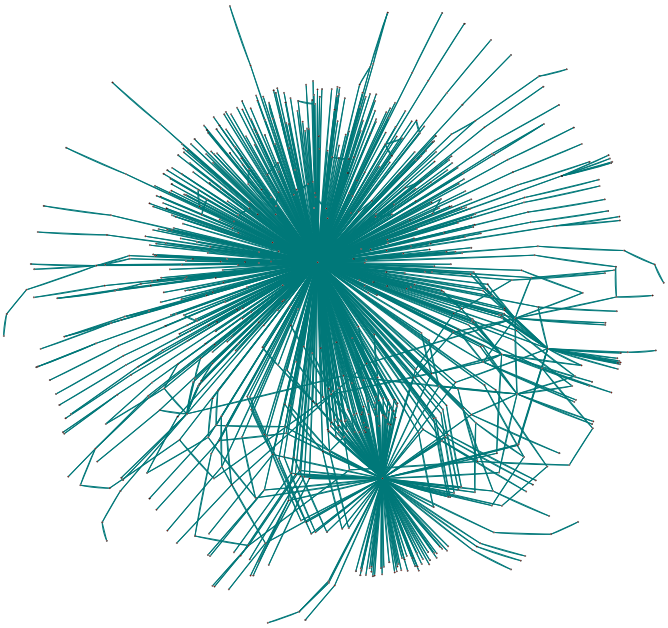


Figure 14: Visual representation of a graph constructed with $PriceModel.m$. 770 nodes.

```
% Routine implementing the Price model for network growth
% Notes:
%   p_k - fraction of vertices with degree k
%   probability a new vertex attaches to any of the degree-k vertices is
%   (k+1)p_k/(m+1), where m - mean number of new citations per vertex
% Source: "The Structure and Function of Complex Networks", M.E.J. Newman
%
% INPUTs: n - final number of vertices
% OUTPUTs: adjacency matrix, directed
%
% GB: last modified, November 9, 2012
```

Example:
$adj = PriceModel(100);$ % visual representation of an example with 770 nodes in Figure 14
$> assert(\ numNodes(adj), 100\ )$

9.2.11 preferentialAttachment.m

Implement simple **preferential attachment** with one node arriving at a time. Ideas from [6].

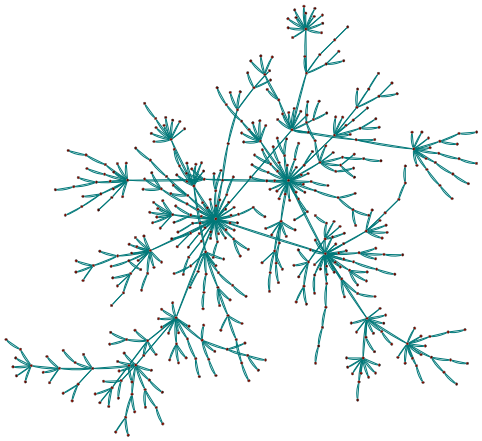


Figure 15: Visual representation of a graph constructed with *preferentialAttachment.m*. 500 nodes.

```
% Routine implementing simple preferential attachment for network growth.
% The probability that a new vertex attaches to a given old vertex
%                           is proportional to the (total) vertex degree.
% Note 1: Vertices arrive one at a time.
% Note 2: Assume undirected simple graph.
% Source: Newman, "The Structure and Function of Complex Networks"
%         B-A., "Emergence of Scaling in Random Networks"
%
% INPUTs: n - final (desired) number of vertices,
%         m - # edges to attach at every step
% OUTPUTs: edge list, [number of edges x 3]
%
% Other routines used: weightedRandomSample.m
% GB: last updated, November 9, 2012
```

Example:
$el = preferentialAttachment(100, 1);$ % visual representation of an example with 500 nodes in Figure 15
$adj = edgeL2adj(el);$
$> numNodes(adj)$
$ans = 100$
$> assert(\ isSimple(adj), true\ )$

9.2.12 exponentialGrowthModel.m

Construct a graph with an **exponential degree distribution**.

```
% Construct a graph with an exponential degree distribution.
% Probability of node s having k links at time t:
%    p(k,s,t)=1/t*p(k-1,s,t-1)+(1-1/t)*p(k,s,t-1)
%
% INPUTS: number of time steps, t
% OUTPUTs: edge list, mx3
%
% GB, last updated: Nov 11, 2012
```

Example:
$el = exponentialGrowthModel(99);$
$adj = edgeL2adj(el);$

$> assert(\ numNodes(adj), 99\ )$
$> assert(\ isSimple(adj), true\ )$

9.2.13 masterEquationGrowthModel.m

The **master equation** model is a generalization of the nodal-degree based growth algorithms. This implementation follows an algorithm description provided in [33]. In this generalized model, one node arrives at every time step t, and is connected to already existing nodes via m links. The probability of attaching the new arrival to a node with degree k at time step t is given in equation 10 (a is a constant).

$$p_{k,t}(k) = \frac{q(k,t) + ma}{(1+a)mt} \tag{10}$$

Notice that for $a = 0$ the master equation approach is equivalent to the simple preferential attachment model (Section 9.2.11).

```
% "Master equation" growth model, as presented in
%                   "Evolution of Networks" by Dorogovtsev, Mendez
% Note: probability of attachment: (q(i)+ma)/((1+a)mt),
%             q(i)- in-degree of i, a=const, t - time step (# nodes)
%
% INPUTS: number of nodes n, m - # links to add at each step, a=constant
% OUTPUTS: adjacency matrix, nxn
%
% Other routines used: weightedRandomSample.m
% GB: last updated, Nov 11, 2012
```

Examples:
$adj = masterEquationGrowthModel(100, 1, 0);$ % simple preferential attachment case
$> isTree(adj)$
$ans = 1$

$adj = masterEquationGrowthModel(99, 2, 1);$
$> isSimple(adj)$
$ans = 1$
$> numNodes(adj)$
$ans = 99$

9.2.14 newmanGastner.m [34]

A **spatial distribution** growth model by Gastner and Newman [34]. A new point (node) attaches to an old node by minimizing a weighted (β) sum of the distance to the node and its distance to a *root* node.

```
% Implements the Newman-Gastner model for spatially distributed networks.
% Source: Newman, Gastner, "Shape and efficiency in spatial distribution networks"
% Note 1: minimize: wij = dij + beta x (dj0)
% Note 2: To save output plot, use "print filename.ext" (see line 65)
%
% Inputs: n - number of points/nodes, beta - parameter in [0,1],
%         points: point coordinates (nx2) or empty; plot - 'on' or 'off'
% Outputs: graph (edge list), point coordinates and plot [optional]
%
% GB: last updated: November 11 2012
```

Example:
$N = randi(100) + 10;$ % a random number of nodes, minimum 11, maximum 110

Figure 16: Visual representation of a graph constructed with *newmanGastner.m*. 4186 points, β=0.1

$el = newmanGastner(N, rand, [\,], 'off');$ % no plot, random point coordinates
$adj = symmetrize(edgeL2adj(el));$ % convert to an undirected graph
$> assert(\ numNodes(adj), N\ )$ % verify that the graph has the right number of nodes
$> assert(\ isSimple(adj), true\ )$

A visual example of the *newmanGastner.m* plot output is shown in Figure 16.

9.2.15 fabrikantModel.m [35]

This is another **spatial distribution** growth model, similar to the Newman-Gastner idea (Section 9.2.14). The algorithm is described in [35].

```
% Implements the network growth model from: Fabrikant et al,
%                  "Heuristically Optimized Trade-offs: A New Paradigm
%                                       for Power Laws in the Internet"
% Note: Assumes the central point (root) to be the one closest to (0,0)
%
% INPUTS: n - number of points, parameter alpha, [0,inf),
%                     plt='on'/'off', if [], then 'off' is default
% OUTPUTS: generated network (adjacency matrix) and plot [optional]
%
% Other functions used: simpleDijkstra.m
% GB: last updated: November 14, 2012
```

Example:
$[adj, p] = fabrikantModel(1000, 30, 'on');$ % produces Figure 17

9.2.16 DoddsWattsSabel.m[36]

Construct a **randomized hierarchy**: a graph with a hierarchical backbone and additional randomized cross-links.

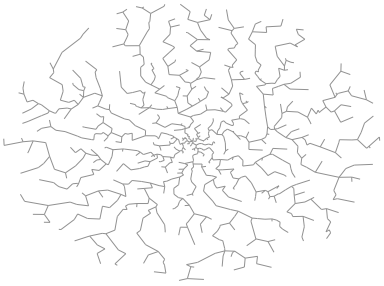


Figure 17: Visual representation of a graph constructed with $fabrikantModel.m$. 1000 points, α=30

```
% Add random cross-links on top of a perfect hierarchy.
% Non-backbone edges are added with probability
%                              P(i,j)=e^(-Dij/lambda)*e^(-xij/ksi),
%            where Dij is the level of the lowest common ancestor
%                          and xij is the "organizational" distance
% Source: Dodds, Watts, Sabel, "Information exchange and the
% robustness of organizational networks", PNAS 100 (21): 12516-12521
%
% INPUTs: number of nodes (N), tree branch factor (b),
%               m - number of additional edges to add,
%           parameters lambda (lam) and ksi in [0,inf)
% OUTPUTs: adjacency matrix of the randomized hierarchy, NxN
%
% Other routines used: edgeL2adj.m, canonicalNets.m, dijkstra.m
% GB: last updated, November 23 2012
```

Example:
$adj = DoddsWattsSabel(50, 3, 10, 15, 15);$
$> assert(numEdges(adj), 10 + 50 - 1)$
$> isSimple(adj)$
$ans = 1$

9.2.17 nestedHierarchiesModel.m [37]

This model is developed in [37]. The exact details can be found in section 2.2 (Hierarchically Nested Random Graphs) in the Supplementary Information of their paper.

```
% Based on: Sales-Pardo et al, "Extracting the hierarchical organization
%                of complex systems", PNAS, Sep 25, 2007; vol.104; no.39
% Supplementary material:
% http://www.pnas.org/content/suppl/2008/02/27/0703740104.DC1/07-03740SItext.pdf
%
% INPUTs: N: number of nodes; L: number of hierarchy levels;
%         [G1,G2,..,GL]: number of nodes in each group in each level
%         kbar: average degree,
%         rho [optional]: ratio between average degrees at different levels
%                                                  (see supplementary material)
%         Example inputs (from paper): N=640, L=3, G=[10,40,160], kbar=16, rho=1
% OUTPUTs: edge list, in mx2 or mx3 format, where m = number of edges
%
% Other routines used: symmetrizeEdgeL.m
% GB: last updated, November 24 2012
```

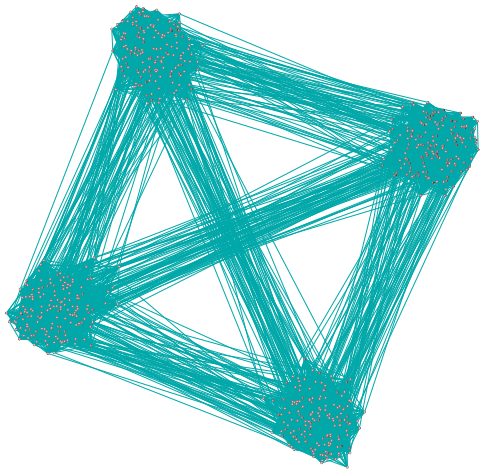


Figure 18: A visualization of the graph produced by $nestedHierarchiesModel.m$, $N = 640, L = 3, \bar{k} = 16, \rho = 1$. Each of the four 160-node communities in this picture contains another nested hierarchy of four 40-node sub-communities.

Example: % produces the graph in Figure 18
$> el = nestedHierarchiesModel(640, 3, [10, 20, 40], 16, 1);$

9.2.18 forestFireModel.m [38]

This model was proposed by Leskovec et al [38]. Below is the graph construction procedure directly quoted from the paper.

> To begin with, we will need two parameters, a forward burning probability p, and a backward burning ratio r, whose roles will be described below. Consider a node v joining the network at time $t > 1$, and let G_t be the graph constructed thus far. (G_1 will consist of just a single node.) Node v forms out-links to nodes in G_t according to the following process.
>
> (i) v first chooses an *ambassador node* w uniformly at random, and forms a link to w.
> (ii) We generate two random numbers: x and y that are geometrically distributed with means $p/(1-p)$ and $rp/(1-rp)$ respectively. Node v selects x out-links and y in-links of w incident to nodes that were not yet visited. Let $w_1, w_2, \ldots, w_{x+y}$ denote the other ends of these selected links. If not enough in- or out-links are available, v selects as many as it can.
> (iii) v forms out-links to $w_1, w_2, \ldots, w_{x+y}$, and then applies step (ii) recursively to each of $w_1, w_2, ..., w_{x+y}$. As the process continues, nodes cannot be visited a second time, preventing the construction from cycling.

```
% Implementation of the forest fire model by Leskovec et al
%                 "Graphs over Time: Densification Laws, Shrinking
%                             Diameters and Possible Explanations"
%
% Inputs: forward burning probability p in [0,1],
%         backward burning ratio r, in [0,inf),
%         T - number of nodes
% Outputs: adjacency list of the constructed (directed) graph
%
% Other routines used: weightedRandomSample.m
% GB: last updated, November 28, 2012
```

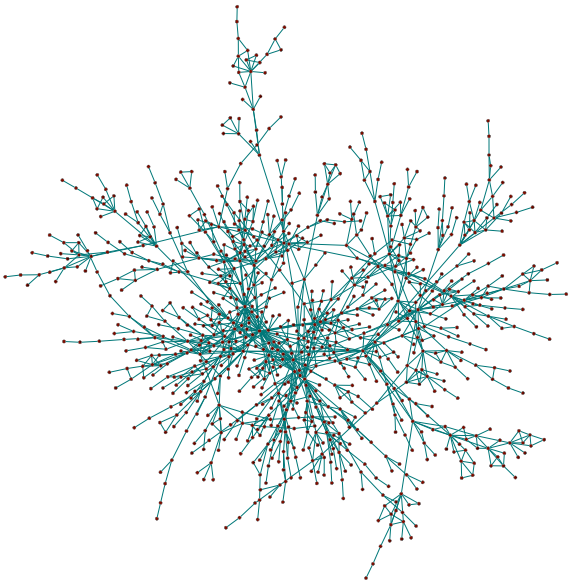


Figure 19: Visual representation of a graph created with $forestFireModel.m$, 1000 nodes, $p = 0.2$, $\rho = 0.3$.

Example:
$L = forestFireModel(200, 0.2, 4)$;
$adj = adjL2adj(L)$;
$adj = symmetrize(adj)$;
$> numNodes(adj)$
$ans = 200$
$> isSimple(adj)$
$ans = 1$

A visual example of a graph created with $forestFireModel.m$ is shown in Figure 19.

10 Visualizing graphs

10.1 On visualization

Visualization of graphs or networks is a huge field. There are many libraries, software packages and interactive tools that do a great job in creating quick plots of any graph, using various methods. The Wikipedia article on Graph Drawing lists the various *layout methods*, as well as some popular software used for visualization. One of the most popular layouts is the *spring energy* layout. In this method, nodes are massless particles, and edges are modeled as springs. The goal is to position the nodes so that the overall energy of the system is minimized. Different algorithms under this category employ different tricks to make the plots cleaner, such as minimizing crossing lines.

This section contains functions that plot the degree distributions, as well as other simple representations of the adjacency matrix, such as a sparsity plot. The only true graph drawing algorithm is *radialPlot.m* (Section 10.2.4), which is best for trees and perhaps for very sparse graphs. The last routine, *edgeL2cyto.m* (Section 10.2.6), shows an example of exporting a graph structure to text format in the input syntax of Cytoscape (one of software packages mentioned above). Many of the plots in this manual have been created using *edgeL2cyto.m* and Cytoscape.

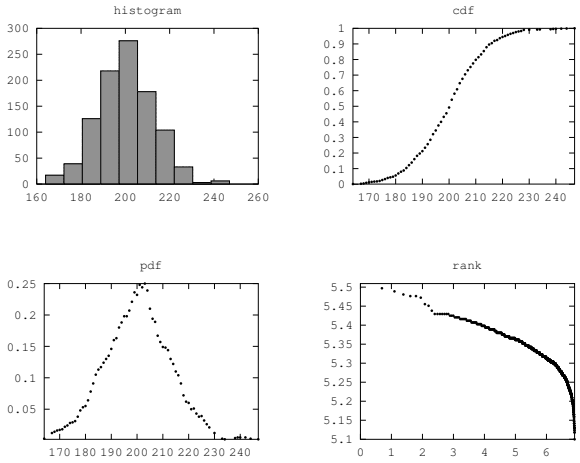


Figure 20: The visual output of *pdfCdfRank.m* for a sequence of 1000 numbers.

10.2 Routines

10.2.1 pdfCdfRank.m

Probability density, cumulative density and rank distributions of a sequence of numbers. That sequence of numbers in this context is usually the degree sequence of some graph.

```
% Compute the pdf, cdf and rank distributions for a sequence of numbers
%
% INPUTS: sequence of values: x, (1xn), 'plot' - 'on' or 'off'
% OUTPUTS: pdf, cdf and rank distribution values, plot is optional
%
% Note: pdf = frequency distribution, cdf = cumulative frequency,
%            rank = log-log scale of the sorted sequence
% GB: last updated, November 24 2012
```

Example: % produces the plot in Figure 20.
$adj = randomGraph(1000, 0.2)$;
$> pdfCdfRank(degrees(adj), "on")$;

10.2.2 dotMatrixPlot.m

Sparsity plot for a given matrix, where the nodes are arranged in six different ways: sorted by nodal degree, betweenness, eigen-centrality, closeness and two types of module (community) membership. The modules are computed with the Newman eigenvector method (8.2.3) and the Newman-Girvan method (8.2.2).

To create the sparsity plot of a matrix, without re-ordering the nodes, simply use: "$> spy(A)$", where A is the adjacency matrix $(n \times n)$.

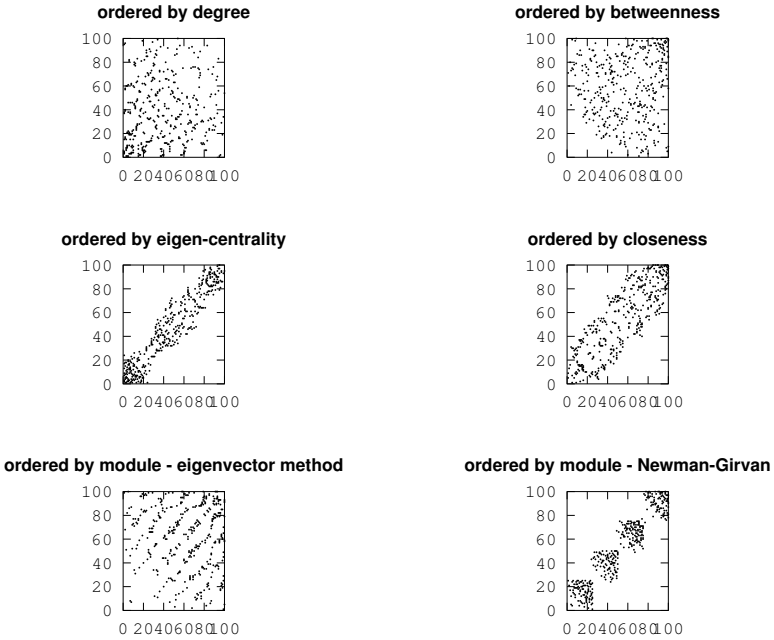


Figure 21: Example plot produced by *dotMatrixPlot.m*.

```
% Draws the matrix as a column/row sorted square sparsity pattern
%
% INPUTs: adj (nxn) - adjacency matrix representation of the graph
% OUTPUTs: plot
%
% Note 1: Change colors and marker types in lines 48, 55, 62 and 69
% Note 2: Easy to add/remove different node orderings to/from the plot
%
% Other routines used: degrees.m, sortNodesByMaxNeighborDegree.m,
%                      eigenCentrality.m, newmanEigenvectorMethod.m,
%                      nodeBetweennessFaster.m
% GB: last updated, November 25 2012
```

Example: % produces the plot in Figure 21
$adj = randomModularGraph(100, 4, 0.1, 5);$
$> dotMatrixPlot(adj)$

10.2.3 drawCircGraph.m

Visual representation in which nodes are ordered by degree and placed in a **circular** configuration.

```
% Draw a circular representation of a graph with the nodes ordered by degree
% Strategy: position vertices in a regular n-polygon
%
% INPUTs: adj, nxn - adjacency matrix
% OUTPUTs: plot
%
% Other routines used: degrees.m
% GB: last updated, Nov 29 2012
```

Example: % produces the plot in Figure 22

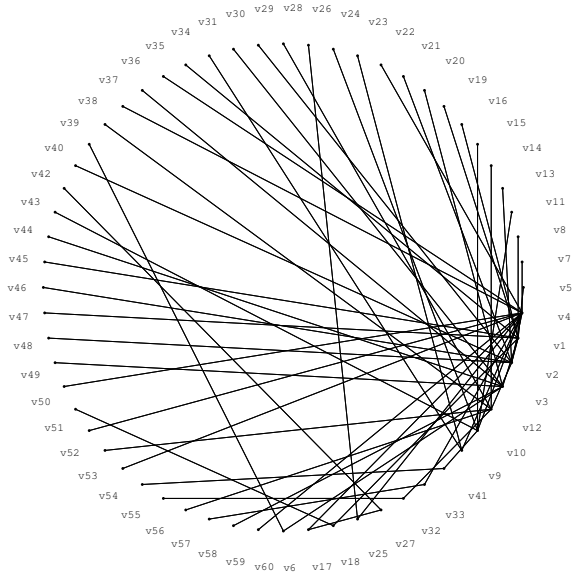


Figure 22: The plot output of *drawCircGraph.m*. The input is a tree graph with 60 nodes.

$el = preferentialAttachment(60, 1);$
$adj = edgeL2adj(el);$
$> drawCircGraph(adj)$

10.2.4 radialPlot.m

```
% Plot nodes radially out from a given center (node). Equidistant nodes
% have the same radius, but different polar angles. Works best as a quick
% visualization for trees, or very sparse graphs.
%
% Note 1: If a center node is not specified, the nodes are ordered by
%   sum of neighbor degrees, and the node with highest sum is plotted
%   in the center.
% Note 2: The graph has to be connected.
% Note 3: To change the color scheme, modify lines: 91, 98, 104 and 118
%
% Inputs: adjacency matrix (nxn), and center node (optional)
% Outputs: plot
%
% Other routines used: sortNodesBySumNeighborDegrees.m,
%                      adj2adjL.m, diameter.m, kminNeighbors.m
% GB: last updated, December 6 2012
```

Example: % create plots such as the ones in Figure 23
$el = preferentialAttachment(300, 1);$
$adj = edgeL2adj(el);$
$> isTree(adj)$
$ans = 1$
$> radialPlot(adj)$
$> radialPlot(adj, 4)$ % specifying a root (center) node

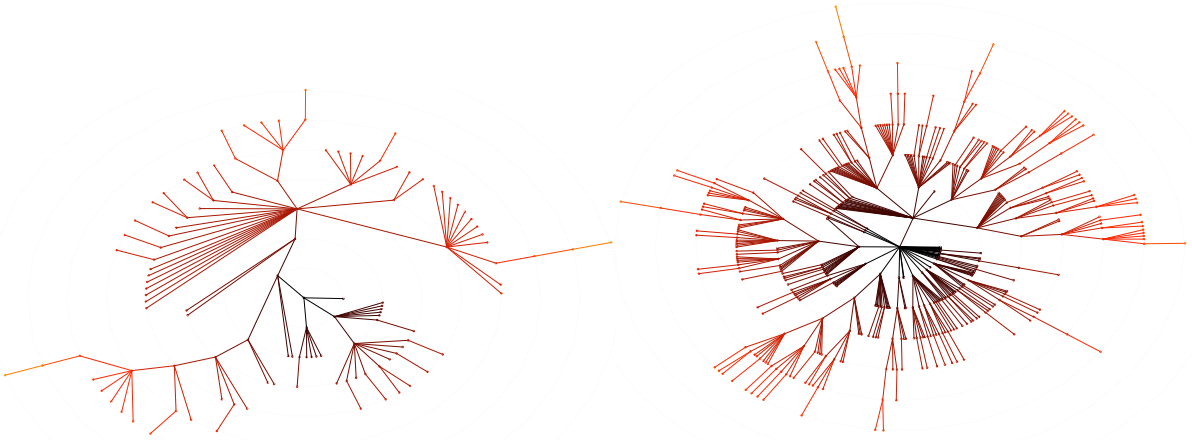


Figure 23: Example plot output of *radialPlot.m*. Input adjacency matrix created with *preferentialAttachment.m*. In the left plot the root node is chosen manually. In the right plot the nodes are ordered by sum of neighbor degrees, and the node with highest sum is plotted in the center.

10.2.5 el2geom.m

Plot an edge list geographically with color-coding of edge weights. Assumes that point (node) coordinates are known.

```
% Plot a graph for which nodes have given coordinates. Apply color
%           scheme and thicker lines if edges have varying weights.
%
% INPUTS: extended edge list el[i,:]=[n1 n2 m x1 y1 x2 y2]
% OUTPUTS: geometry plot, higher-weight links are thicker and lighter in color
%
% Note 1: m - edge weight; (x1,y1) are the Euclidean coordinates of n1, (x2,y2) - n2
% Note 2: Easy to change colors and corresponding edge weight coloring
%
% GB: last updated: December 8, 2012
```

Example: % Example plots in Figure 24
$[el, p] = newmanGastner(3000, 0.1, [\,])$; % point coordinates are stored in p
$elnew = [\,]$;
$for\ e = 1 : size(el, 1)$
$elnew = [elnew;\ el(e, 1), el(e, 2), randi(8), p(el(e, 1), 1), p(el(e, 1), 2), p(el(e, 2), 1), p(el(e, 2), 2)]$;
end
$el2geom(elnew)$

10.2.6 edgeL2cyto.m

Convert an edge list to Cytoscape text format. A good example of how to save a graph representation as a text file. Various plotting programs have different syntax conventions for their text file input.

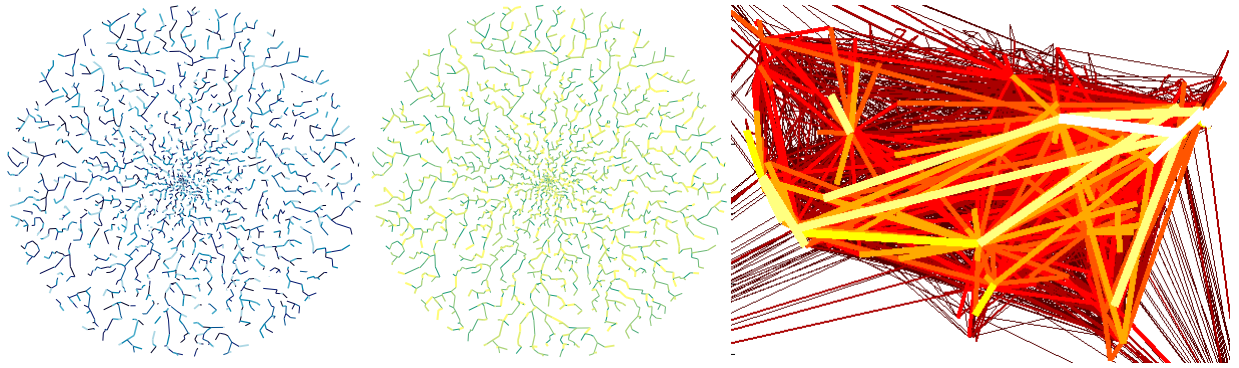


Figure 24: The two left-most plots are produced with the example code in Section 10.2.5, but with different Octave colormaps (*ocean*, *summer*). The right-most is also produced with *el2geom.m* but represents the density of commercial flights in the United States in 2007 (using the *hot* colormap)

```
% Convert an edge list structure m x [node 1, node 2, link] to
%    Cytoscape input format (.txt or any text extension works)
%
% Note: In Cytoscape the column separator option is semi-colon ";".
%             If desired, this is easy to change below in line 20.
%
% INPUTs: edge list - mx3 matrix, m = number of edges, file name text string
% OUTPUTs: text file in Cytoscape format with a semicolon column separator
```

Example:
$edgeList = canonicalNets(100,'trilattice');$
$>$ *edgeL2cyto*(*edgeList*, *'trilattice.txt'*)

11 Auxiliary routines

11.1 Routines

11.1.1 weightedRandomSample.m

Weighted random sampling: drawing a set of n numbers from a distribution P according to some normalized weights (probabilities) W. This is used within *randomGraphDegreeDist.m* (Section 9.2.7).

```
% Weighted random sampling.
%
% INPUTs: number of draws from a discrete distribution (n)
%         possible values to pick from, (P)
%         set of normalized weights/probabilities, (W)
% OUTPUTs: s - set of n numbers drawn from P
%              according to the weights in W
%
% GB: last updated, Oct 31 2012
```

Example:
$> s = weightedRandomSample(10, [1,\ 2,\ 3], [1/4,\ 1/2,\ 1/4])$
$ans =$
2 3 1 2 2 2 3 1 2 1

12 Open bugs

Betweenness bug

Currently, there is an open problem in the betweenness routines, Section 4.2.10 and Section 4.2.11. The code does not return the correct betweenness values for all nodes if the graph contains an even cycle. That is because an even cycle results in multiple shortest paths between some of the nodes. This function chooses one shortest path for every pair of nodes and iterates over nodes on this path only. The fix should include a shortest path finding routine which returns all possible shortest paths, and then iterates over all of them to calculate the correct betweenness for all nodes.

13 Links

Links referenced in this manual

- Edward Scheinerman's Matgraph.
- Degree-preserving rewiring code by Sergei Maslov.
- Louvain method: Finding communities in large networks
- Cytoscape
- Mark Newman: publications, code, data sets

Other relevant links

- SBEToolbox
- graphviz4matlab

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

[19] Guo, Chen, Zhou, Fingerprint for Network Topologies, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, ISSN1867-8211, Vol 5,Part 1, Springer Berlin Heidelberg 2009

[20] Bertsekas, Dynamic Programming and Optimal Control, Athena Scientific, 2005 (3rd edition)

[21] Leskovec, Kleinberg, Faloutsos, Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations, KDD'05, 2005, Chicago, IL

[22] Mahadevan, Krioukov, Fall, Vahdat, Systematic Topology Analysis and Generation Using Degree Correlations, SIGCOMM '06 Proceedings

[23] I. Gutman, The energy of a graph, Ber. Math. Statist. Sekt. Forsch-ungszentram Graz. 103 (1978) 1-22.

[24] M. E. J. Newman, Finding community structure using the eigenvectors of matrices, Phys. Rev. E 74, 036104 (2006)

[25] M. E. J. Newman, Modularity and community structure in networks, PNAS June 6, 2006, vol. 103, no. 23, 8577-8582

[26] M. E. J. Newman, Fast algorithm for detecting community structure in networks, Phys. Rev. E 69, 066133 (2004)

[27] Blondel, Guillaume, Lambiotte, Lefebvre, Fast unfolding of communities in large networks, J. Stat. Mech. (2008) P10008

[28] M. E. J. Newman, Analysis of weighted networks, Phys. Rev. E 70, 056131 (2004)

[29] Erdös, Paul; A. Rényi, On the evolution of random graphs, Publications of the Mathematical Institute of the Hungarian Academy of Sciences 5: 17-61, 1960

[30] S. L. Hakimi, On Realizability of a Set of Integers as Degrees of the Vertices of a Linear Graph. I, Journal of the Society for Industrial and Applied Mathematics Vol. 10, No. 3 (Sep., 1962), pp. 496-506

[31] Molloy M., Reed, B. A Critical Point for Random Graphs with a Given Degree Sequence, Random Structures and Algorithms 6 , 161-179, 1995

[32] Clauset, Finding local community structure in networks, Phys. Rev. E 72, 026132 (2005)

[33] Dorogovtsev, Mendes, Evolution of Networks, Advances in Physics 2002, Vol. 51, No. 4, 1079-1198

[34] Gastner, Newman, Shape and efficiency in spatial distribution networks, J. Stat. Mech. (2006) P01015

[35] Fabrikant, Koutsoupias, Papadimitriou, Heuristically Optimized Trade-offs: A New Paradigm for Power Laws in the Internet, Automata, Languages and Programming, Vol. 2380, 2002

[36] Dodds, Watts, Sabel, Information exchange and the robustness of organizational networks, PNAS, vol. 100, no. 21, 12516-12521, October 14 2003

[37] Sales-Pardo, Guimerà, Moreira, Amaral, Extracting the hierarchical organization of complex systems, PNAS, vol. 104, no. 39, 15224-15229, September 25 2007

[38] Leskovec, Kleinberg, Faloutsos, Graph Evolution: Densification and Shrinking Diameters, ACM Transactions on Knowledge Discovery from Data (ACM TKDD), 1(1), 2007