Octave routines for network analysis

GB

February 20, 2015

Contents

February 20, 2015

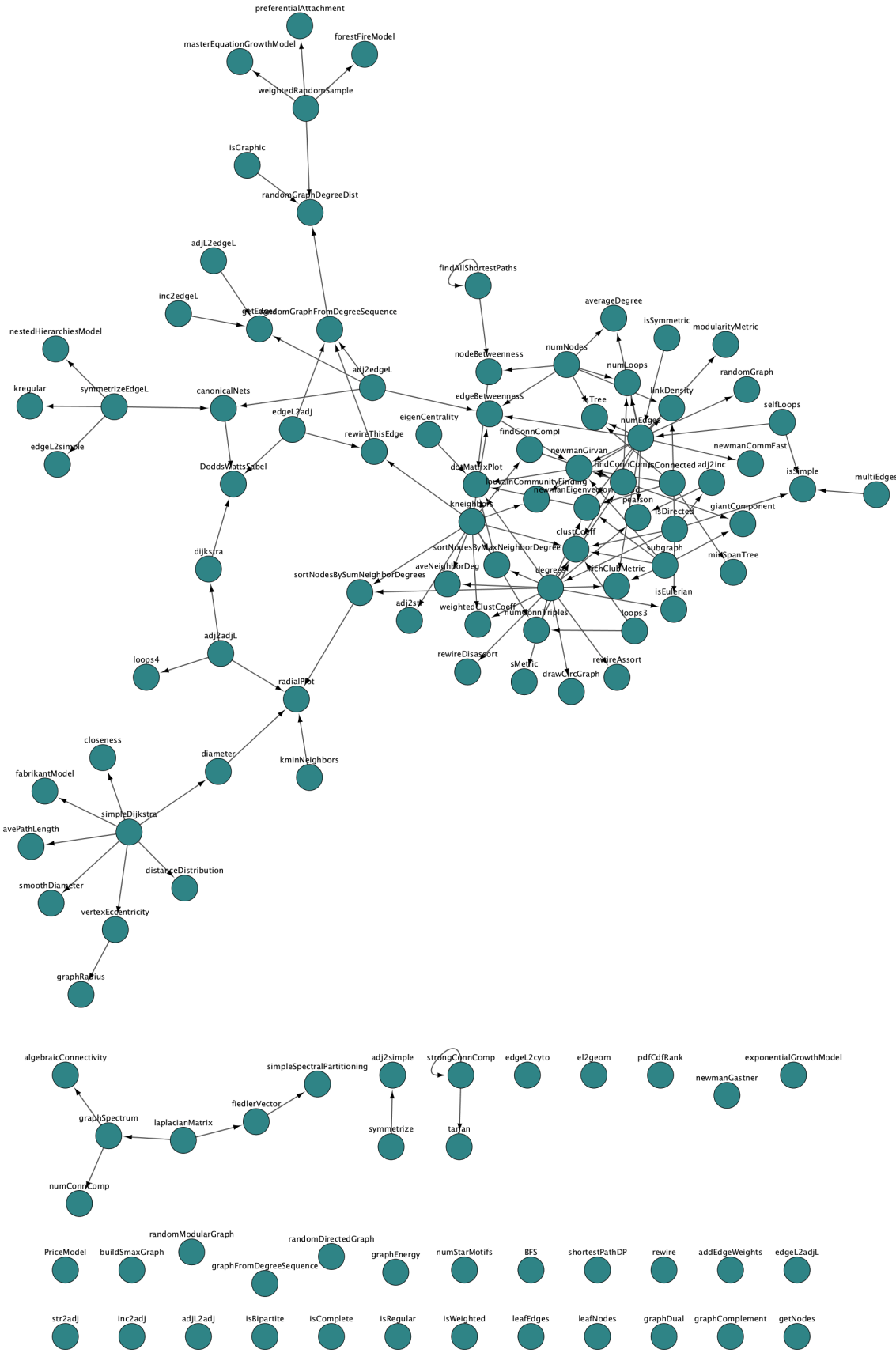


Figure 1: Graph of functional interdependencies in this toolbox. An edge points from routine A to routine B if routine A is called within routine B. For example, *isConnected.m* is called within *minSpanTree.m*

0 About this toolbox

(This is a copy of the README file.)

octave-networks-toolbox: A set of graph/networks analysis functions in Octave, 2012-2014

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:1> numNodes([0 1 1; 1 0 1; 1 1 0])
ans =  3
```

Matlab compatibility
With newer versions of Matlab, the Octave branch may not always be Matlab-compatible, for example due to syntax changes. Consider exploring forks that focus on Matlab compatibility. There is currently no plan for the Octave original version to be synchronized with Matlab.

Authorship
This code was originally written and posted by Gergana Bounova. It is undergoing continuous expansion and development. Thank you for the many comments and bug reports so far! Contributions via email are usually given tribute to in the function header. Collaborators are very welcome. Contact via github/email for comments, questions, suggestions, corrections or simply fork.

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.

Citation
If you want to refer to this code as a citation, you can use DOI: 10.5281/zenodo.10778 (https://zenodo.org/record/10778).

1 Representing graphs in octave/matlab

1.1 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 2.1), 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 1.2.

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:

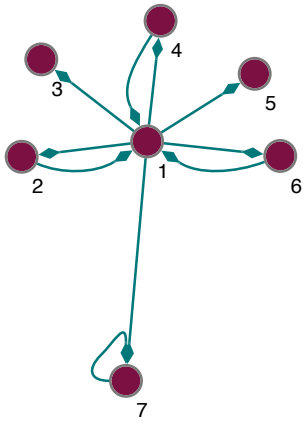
.2,.3,,.4,

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.

1.2 Routines

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



edge list =

$$\begin{array}{l} 1 \to 2 \\ 2 \to 1 \\ 1 \to 3 \\ 1 \to 4 \\ 4 \to 1 \\ 1 \to 5 \\ 1 \to 6 \\ 6 \to 1 \\ 1 \to 7 \\ 7 \to 7 \end{array}$$

adjacency =

$$\begin{array}{ccccccc} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

adj list =

$$\begin{array}{ll} \{1\} \to & 2, 3, 4, 5, 6, 7 \\ \{2\} \to & 1 \\ \{3\} \to & \emptyset \\ \{4\} \to & 1 \\ \{5\} \to & \emptyset \\ \{6\} \to & 1 \\ \{7\} \to & 7 \end{array}$$

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,**".

1.2.1 adj2adjL.m

Convert an adjacency matrix to an adjacency list. This is the inverse function of *adjL2adj.m* (1.2.2).

```
% 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:

```
% undirected binary tree with 3 nodes
octave:1> adj2adjL([0 1 1; 1 0 0; 1 0 0])
ans =
{
  [1,1] =

     2   3

  [2,1] =  1
  [3,1] =  1
}
```

1.2.2 adjL2adj.m

Convert an adjacency list to an adjacency matrix. This is the inverse function of *adj2adjL.m* (1.2.1).

```
% 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:

```
octave:48> aL = { [2,3],[1,3],[1,2]};
octave:49> adjL2adj(aL)
ans =

   0   1   1
   1   0   1
   1   1   0
```

1.2.3 adj2edgeL.m

Convert an adjacency matrix to an edge list. This is the inverse routine of *edgeL2adj.m* (1.2.4).

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

Example:

```
octave:31> adj2edgeL([0 1 1; 1 0 0; 1 0 0])
ans =
   2   1   1
   3   1   1
   1   2   1
   1   3   1
```

1.2.4 edgeL2adj.m

Convert edge list to adjacency matrix. This is the inverse routine of *adj2edgeL.m* (1.2.3).

```
% 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:

```
# a single directed edge
octave:1> edgeL2adj([1 2 1])
ans =

   0   1
   0   0
```

1.2.5 adj2inc.m

Convert an adjacency matrix to an incidence matrix. This is the inverse function of *inc2adj.m* 1.2.6.

```
% 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:

```
octave:4> adj2inc([0 1 1; 1 0 0; 1 0 0])
ans =
   1   1
   1   0
   0   1
```

1.2.6 inc2adj.m

Convert an incidence matrix to an adjacency matrix. This is the inverse function of *adj2inc.m* (1.2.5).

```
% 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:

```
octave:10> inc = [1 0 1; 1 1 0; 0 1 1];
octave:11> inc2adj(inc)
ans =
   0   1   1
   1   0   1
   1   1   0
```

1.2.7 adj2str.m

Convert an adjacency matrix to a string (text) graph representation. This is the inverse function of *str2adj.m* (1.2.8).

```
% 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:

```
% undirected binary tree
adj2str([0 1 1; 1 0 0; 1 0 0])
ans = .2.3,.1,.1,
```

1.2.8 str2adj.m

This is the reverse routine of *adj2str.m* (1.2.7). 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:

```
% a three-node undirected cycle
octave:7> str2adj(".2.3,.1.3,.1.2,")
ans =
   0   1   1
   1   0   1
   1   1   0
```

1.2.9 adjL2edgeL.m

Convert an adjacency list to an edge list. This is the inverse routine of *edgeL2adjL.m* (1.2.10).

```
% 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:

```
octave:14> adjL2edgeL({ [2,3],[1],[1] })
ans =
   1   2   1
   1   3   1
   2   1   1
   3   1   1
```

1.2.10 edgeL2adjL.m

Convert an edge list to an adjacency list. This is the inverse routine of *adjL2edgeL.m* (1.2.9).

```
% 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:

```
octave:1> edgeL2adjL([1 2 1; 1 3 1])
ans =
{
  [1,1] =

     2   3

  [2,1] = [](0x0)
  [3,1] = [](0x0)
}
```

1.2.11 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:

```
octave:2> inc = [1  0;  1  1;  0  1];
octave:3> inc2edgeL(inc)
ans =

   1   2   1
   2   3   1
   2   1   1
   3   2   1
```

1.2.12 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
```

```
%
% Other routines used: symmetrize.m
% GB: last updated, Sep 6 2014
```

Example:

```
octave:1> adj2simple([1 2 1; 2 0 1; 1 1 0])
ans =
   0   1   1
   1   0   1
   1   1   0
```

1.2.13 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:

```
octave:2> edgeL2simple([1 1 1; 1 2 1; 1 3 2])
ans =
   1   2   1
   1   3   1
   2   1   1
   3   1   1
```

1.2.14 symmetrize.m

Symmetrize a matrix. In this context, this means convert a directed graph representation to its equivalent undirected representation.

```
% 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:

```
octave:8> symmetrize([0 1; 0 0])
ans =
   0   1
   1   0
```

1.2.15 symmetrizeEdgeL.m

This function is similar to 1.2.14. For a general edge list, perhaps representing a directed graph, it checks whether the reverse edges of all edges are present. If not, they are added so the resulting graph is undirected.

```
% 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:

```
octave:6> symmetrizeEdgeL([1 2 1; 1 3 1])
ans =
   1   2   1
   1   3   1
   2   1   1
   3   1   1
```

1.2.16 addEdgeWeights.m

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

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

Example:

```
octave:11> addEdgeWeights([1 2 1; 1 2 0.5; 2 3 1; 3 4 1; 3 4 3])
ans =
   1.0000   2.0000   1.5000
   2.0000   3.0000   1.0000
   3.0000   4.0000   4.0000
```

2 Basic network routines

2.1 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] [8].

- 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 2.2.

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 do not call or are not called within other functions).

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.

2.2 Routines

2.2.1 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: 'adjacency','edgelist','adjlist','incidence'
% Note 1: only the edge list allows/returns non-consecutive node indexing
%
% Example representations of a directed 3-loop: 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] (with edge weights)
%         'inc' - [-1  0  1
%                   1 -1  0
%                   0  1 -1]
%
% GB: last updated, Jul 12 2014
```

Examples:

```
octave:1> getNodes([0 1 1; 1 0 1; 1 1 0],'adjacency')
ans =
   1   2   3

octave:2> adjL ={[2,3],[1,3],[1,2,4],[3,5,6],[4,6],[4,5]};
octave:3> getNodes(adjL,'adjlist')
ans =
   1   2   3   4   5   6
```

2.2.2 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: 'adjacency','edgelist','adjlist', 'incidence'
% Note 2: symmetric edges will appear twice, also in undirected graphs,
%         (i.e. [n1,n2] and [n2,n1])
```

```
%
% Example representations of a directed triangle: 1->2->3->1
%           'adjacency' - [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)
%           'incidence' - [-1  0  1
%                           1 -1  0
%                           0  1 -1]
%
% Other routines used: adj2edgeL.m, adjL2edgeL.m, inc2edgeL.m
% GB: last updated, Sep 18 2012
```

Examples:

```
 % using adjacency matrix representation
octave:46> getEdges([0 1 1; 1 0 1; 1 1 0],'adjacency')
ans =

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

% using adjacency list representation
octave:47> adjL ={[2,3],[1,3],[1,2,4],[3,5,6],[4,6],[4,5]};
octave:48> 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 1.1 only the *edge list* can carry edge weight information.

2.2.3 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:

```
octave:2> N = randi(100);
octave:3> adj = randomGraph(N);
octave:3> % test whether the random graph does indeed have N nodes
octave:4> assert(numNodes(adj),N)
octave:4>
octave:4> % a graph (adjacency list) with 6 nodes
octave:4> adjL ={[2,3],[1,3],[1,2,4],[3,5,6],[4,6],[4,5]};
octave:5> numNodes(adjL)
ans = 6
```

2.2.4 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:

```
octave:2> N = randi(100);
octave:3> E = randi([1,N-1]);
octave:4> adj = randomGraph(N,[],E);
octave:4> % check that the random graph has exactly E edges
octave:5> assert(numEdges(adj),E)
octave:5>
octave:5> % the bowtie graph is a 6-node graph with 7 edges
octave:6> 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];
octave:7> numEdges(bowtie)
ans = 7
```

2.2.5 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, because the possible number of edges is twice as many.

See also 2.2.8.

```
% 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:

```
% undirected 3-node cycle
octave:30> adj = [0 1 1; 1 0 1; 1 1 0];
octave:31> linkDensity(adj)
ans =  1
octave:32>
octave:32>
octave:32> % the bowtie graph is a 6-node graph with 7 edges
octave:32> 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];
octave:33> linkDensity(bowtie)
ans =  0.46667
```

2.2.6 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:

```
octave:2> selfLoops([0 1; 0 0])
ans = 0

octave:3> % three self-loops
octave:3> adj = [1 0 0; 0 1 0; 0 0 1];
octave:4> selfLoops(adj)
ans =  3
```

2.2.7 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.
%
% INPUT: adjacency matrix, nxn
% OUTPUT: integer, number of multiple edges
%
% Examples: multiEdges([0 2; 2 0])=1, and multiEdges([0 0 1; 2 0 0; 0 1 0])=1
%
% Note 1: The definition of number of multi-arcs/edges (node pairs
% that have multiple edges across them) here is: mA = length(find(adj>1))
% (normalized by 2 depending on whether the graph is directed).
% Note 2: This creates a natural difference in counting for
% undirected and directed graphs.
%
% Other routines used: isSymmetric.m
% GB: last updated, Sep 26 2014
```

Examples:

```
octave:22> one_double_edge = [0 2; 0 0];
octave:23> multiEdges(one_double_edge)
ans =  1
octave:24> double_edge = [0 2; 2 0];
octave:25> multiEdges(double_edge)
ans =  1
octave:26> adj = [1 1 0; 1 0 0; 0 0 0];
octave:27> multiEdges(adj)
ans = 0
```

2.2.8 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 2.2.5) 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: float, the average degree, a number 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:

```
octave:7> % undirected 3-node cycle
octave:7> adj = [0 1 1; 1 0 1; 1 1 0];
octave:8> averageDegree(adj)
ans =  2
octave:9> % undirected 3-node binary tree
octave:9> adj = [0 1 1; 1 0 0; 1 0 0];
octave:10> averageDegree(adj)
ans =  1.3333
```

2.2.9 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:

```
octave:12> adj = [0 1 1; 1 0 1; 1 1 0];
octave:13> numConnComp(adj)
ans =  1
octave:14> 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];
```

```
octave:15> numConnComp(adj)
ans =  2
```

2.2.10 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:

```
octave:11> % two disconnected three-node cycles
octave:11> 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];
octave:12> findConnCompI(adj,1)
ans =
   1   2   3

octave:13> findConnCompI(adj,5)
ans =
   4   5   6
```

findConnComp.m: Find the **connected components** in an undirected graph. This function uses $findConnCompI.m$.

```
% 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:

```
octave:14> % two disconnected three-node cycles, same as above
octave:14> 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];
octave:15> comp = findConnComp(adj);
octave:16> comp
comp =
{
  [1,1] =

     1   2   3

  [1,2] =

     4   5   6
}
```

2.2.11 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 of the giant component
%
% Other routines used: findConnComp.m, subgraph.m
% GB: last updated: September 22, 2012
```

Example:

```
octave:23> adj = [0 1 0; 1 0 0; 0 0 1];
octave:24> [GC, I] = giantComponent(adj);
octave:25> GC
GC =

   0   1
   1   0

octave:26> I
I =

   1   2
```

2.2.12 tarjan.m [9][10]

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:

```
octave:55> % same as {1:[2],2:[3],3:[1]} (a directed 3-cycle)
octave:55> L = {[2],[3],[1]}
L =
{
  [1,1] =  2
  [1,2] =  3
  [1,3] =  1
}
octave:56> comp = tarjan(L);
octave:57> comp
comp =
{
  [1,1] =

     1   2   3
}
octave:58>
octave:58> comp = tarjan({[2,3],[],[]})   % undirected binary tree
comp =
{
  [1,1] =  2
  [1,2] =  3
  [1,3] =  1
}
```

2.2.13 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 1: Assumes no multiple edges.
% Note 2: To create a complement graph without self-loops,
%         use adjC=ones(size(adj))-adj-eye(length(adj)); instead.
% GB: last updated, October 4, 2014
```

Example:

```
octave:16> g = [0 1 1; 1 0 0; 1 0 0];
octave:17> gc = graphComplement(g)
gc =
   1   0   0
   0   1   1
   0   1   1
```

2.2.14 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:

```
octave:3> % cycle3 in adjacency matrix format is [0 1 1; 1 0 1; 1 1 0]
octave:3> cycle3 = { [2,3]; [1,3]; [1,2] };
octave:4> graphDual(cycle3)
ans =
{
  [1,1] =

     2   3

  [1,2] =

     1   3

  [1,3] =

     1   2

}
octave:5>
octave:5> % undirected 3-node binary tree
octave:6> L = { [2,3]; [1]; [1] };
octave:7> graphDual(L)
ans =
{
  [1,1] =  2
  [1,2] =  1
}
```

2.2.15 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:

```
octave:8> 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];
octave:9> subgraph(bowtie,[1, 2, 3])
ans =

   0   1   1
   1   0   1
   1   1   0
```

```
octave:10> subgraph(bowtie,[3, 4, 5])
ans =

   0   1   0
   1   0   1
   0   1   0
```

2.2.16 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 given 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:

```
octave:30> % only 1 is not a leaf node, because it has degree 2
octave:30> adj = [0 1 1; 1 0 0; 1 0 0];
octave:31> leafNodes(adj)
ans =

   2   3

octave:32> % a cycle graph has no leaf nodes
octave:32> adj = [0 1 1; 1 0 1; 1 1 0];
octave:33> leafNodes(adj)
ans = [](1x0)
```

2.2.17 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 given root node.
% Note 3: Edges that are self-loops are not considered leaf edges.
% Note 4: Single floating disconnected edges are not considered to be 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:

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

   1   2
   1   3

octave:4> % a cycle has no leaf edges
octave:4> adj = [0 1 1; 1 0 1; 1 1 0];
octave:5> leafEdges(adj)
ans = [](0x0)
```

2.2.18 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:

```
octave:9> 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];
octave:10> 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
```

2.2.19 DFS.m

Depth first search. Given a graph, a *start* and an *end* node, return all possible paths from the *start* to the *end*, subject to a path length upper bound. Currently, no edge weights are implemented in the path length, but that should be straighforward to modify.

```
% Find all paths from a start to an end node,
% bounded by a constant, using depth-first-search.
%
% Source: Idea from 6.002x, Spring 2014.
% Note: Uses recursion.
%
% INPUTs: graph structure (a nxn adjacency matrix)
%         s - start node
%         e - end node
%         upperBound = 5, some constant bounding the
%                      length of the path; typically size(adj,1)-1
%         allPaths = {}, path = []; are by default empty,
```

```
%                                   serving the recursion
%
% OUTPUTs: a list {} of all shortest paths from "s" to "e",
%                                 shorter than "upperBound"
%
% Note: Uses also kneighbors.m, DFS.m.
% GB: last updated Oct 27 2014
```

Example:

```
octave:7> adj = [0 1 0 1; 1 0 1 0; 0 1 0 1; 1 0 1 0];
octave:8> allPaths = DFS(adj, 1, 3, allPaths = {}, path = [], upperBound = 3);
octave:9> allPaths
allPaths =
{
  [1,1] =

     1   2   3

  [1,2] =

     1   4   3
}
```

2.2.20 BFS.m

Simple (queue) implementation of breadth-first search. Returns a directed **breadth-first-search tree**, starting at given root node. The tree grows by adding all adjacent nodes of every expanded node. If the target node is reached, the expansion stops. If the target node is not connected to (part of) the graph, then the output tree is effectively a spanning tree.

```
% Simple implementation of breadth-first-search of a graph.
% Returns a breadth-first-search tree.
%
% INPUTs: adjacency list (nxn), "adjL"
%         start node index, "s"
%         end node index, "t"
% OUTPUTs: BFS tree, in adjacency list {} format (directed)
%          (This is the tree that the algorithm walks in
%          search of the target. If the target is not found,
%          this tree is effectively a spanning tree of the
%          entire graph.
%
% GB: last updated, Nov 8 2014
```

Examples:

```
octave:188> % adjacency list representation of the bowtie graph
octave:188> L = { [2,3],[1,3],[1,2,4],[3,5,6],[4,6],[4,5] };
octave:189> BFS(L, 1, 6)
ans =
{
  [1,1] =

     2   3

  [1,2] = [](0x0)
  [1,3] =  4
  [1,4] =
```

```
     5    6

  [1,5] = [](0x0)
  [1,6] = [](0x0)
}
octave:190> BFS(L, 5, 3)
ans =
{
  [1,1] = [](0x0)
  [1,2] = [](0x0)
  [1,3] = [](0x0)
  [1,4] =  3
  [1,5] =

     4    6

  [1,6] = [](0x0)
}
```

3 Diagnostic routines

These are functions that return boolean values when querying some property of the graph. They are often used by other algorithms in this toolbox.

3.1 Routines

3.1.1 isSimple.m

A simple graph is undirected, without self-loops, no multiple edges and no edge weights.

```
% Checks whether a graph is simple (undirected, no self-loops,
%                        no multiple edges, no weighted edges)
%
% INPUTs: adj - adjacency matrix
% 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:

```
octave:9> % undirected binary tree
octave:9> isSimple([0 1 1; 1 0 0; 1 0 0])
ans =  1
octave:10> % a weighted graph example
octave:10> isSimple([0 2 1; 2 0 0; 1 0 0])
ans = 0
octave:11> % directed graph example
octave:11> isSimple([0 1 1; 0 0 0; 0 0 0])
ans = 0
```

3.1.2 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

function S=isDirected(adj)

S = true;
if adj==transpose(adj); S = false; end
```

Examples:

```
octave:3> isDirected([0 1 1; 1 0 0; 1 0 0])
ans = 0
octave:4> isDirected([0 1 1; 0 0 0; 0 0 0])
ans =  1
```

3.1.3 isSymmetric.m

Checks whether a matrix is **symmetric**. Note then when a graph is represented with an adjacency matrix, isSymmetric.m is the same as isDirected.m (3.1.2).

```
% 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:

```
octave:7> isSymmetric([0 1 1; 1 0 0; 1 0 0])
ans =  1
octave:8> isSymmetric([0 1 1; 0 0 0; 0 0 0])
ans = 0
```

3.1.4 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* 2.2.10.

Examples:

```
octave:17> % undirected binary tree with 3 nodes
octave:17> isConnected([0 1 1; 1 0 0; 1 0 0])
ans =  1
octave:18> % two disconnected 3-node cycles
octave:18> 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];
octave:19> isConnected(adj)
ans = 0
```

3.1.5 isWeighted.m

Checks whether a graph has **weighted** links. Uses the edge list representation.

```
% 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:

```
octave:16> eL =[1 2 1; 1 3 1; 2 3 1];
octave:17> isWeighted(eL)
ans = 0
octave:18> % undirected double (weighted) edge
octave:18> isWeighted([1 2 2; 2 1 2])
```

```
ans =  1
```

3.1.6 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:

```
octave:3> % undirected binary tree
octave:3> adj = [0 1 1; 1 0 0; 1 0 0];
octave:4> isRegular(adj)
ans = 0
octave:5> % undirected 3-node cycle
octave:5> adj = [0 1 1; 1 0 1; 1 1 0];
octave:6> isRegular(adj)
ans =  1
octave:7> % same as above, but edges are weighted
octave:7> adj = [0 2 2; 2 0 2; 2 2 0];
octave:8> isRegular(adj)
ans =  1
octave:9> % a 4-node cycle
octave:9> isRegular([0 1 0 1; 1 0 1 0; 0 1 0 1; 1 0 1 0])
ans =  1
```

3.1.7 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:

```
octave:3> isComplete([0 1 1; 1 0 0; 1 0 0])
ans = 0
octave:4> isComplete([0 1 1; 1 0 1; 1 1 0])
ans =  1
```

3.1.8 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:

```
octave:2> % nodes have degree (2,1,1) respectively
octave:2> isEulerian([0 1 1; 1 0 0; 1 0 0])
isEulerian.m: There is an Eulerian trail from node  2 to node  3
ans = 0
octave:3> % in a 4-cycle every node has degree 2
octave:3> isEulerian([0 1 0 1; 1 0 1 0; 0 1 0 1; 1 0 1 0])
ans =  1
```

3.1.9 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:

```
octave:57> isTree([0 1 1; 1 0 0; 1 0 0])
ans =  1
octave:58> isTree([0 1 0 1; 1 0 1 0; 0 1 0 1; 1 0 1 0])
ans = 0
```

3.1.10 isGraphic.m [11]

Check whether a sequence of numbers is graphic. A sequence of numbers is **graphic** if a graph exists with the same degree sequence [11].

```
% 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:

```
octave:9> isGraphic([2 2 2])
ans = 1
octave:10> isGraphic([2 1 2])
ans = 0
```

3.1.11 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:

```
octave:37> % undirected binary tree with 3 nodes
octave:37> isBipartite({ [2,3],[1],[1] })
ans =  1
octave:38> % this graph contains a self-loop (2,2)
octave:38> isBipartite( { [2,3],[1,2],[1] } )
ans = 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 they are 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 will 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 3.1.10). The degree sequence of the star graph in Figure 3 (a) is $[5, 1, 1, 1, 1, 1]$, whereas the degree sequence of the bowtie graph (c) is $[2, 2, 3, 3, 2, 2]$.

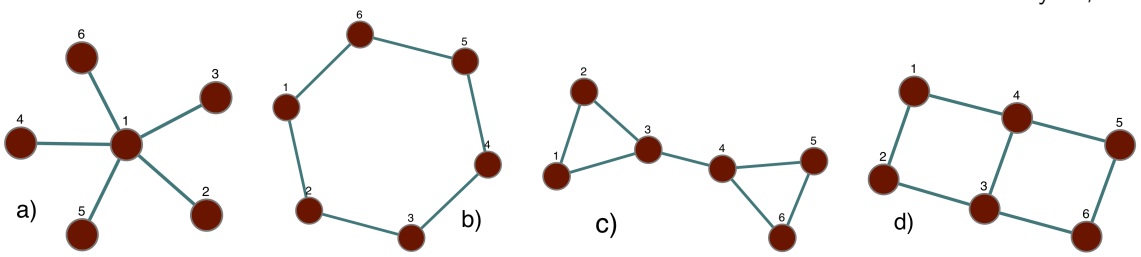


Figure 3: Simple graph examples: a star (a), a circle (b), a "bowtie" graph (c) and a lattice graph (d).

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.

The **clustering coefficient** was first discussed as presented here by Watts and Strogatz [3], but has also long been present in the social networks literature. The clustering coefficient represents the frequency with which the adjacent nodes (neighbors) of a given node are also connected. The definition is given in equation (3).

$$C_i = \frac{\sum_{j,k \in L(i), j<k} A(j,k)}{\binom{|L(i)|}{2}} \qquad C = \frac{1}{n}\sum_{i=1}^{n} C_i \tag{3}$$

C_i is the clustering coefficient for every node and C is the average of C_i over the entire graph. L is the adjacency list representation of the graph, and A is the adjacency matrix. $L(i)$ is the list of neighbors of i.

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

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

A measure very similar to the clustering coefficient is the **transitivity** ([6]). Again, the goal is to measure how close-knit triples of connected nodes are. The definition of transitivity is given in equation 4:

$$C = \frac{number\ of\ cycles\ of\ size\ 3}{number\ of\ connected\ triples} \tag{4}$$

Among the example graphs in Figure 3 the star graph (a), the circle (b) and the lattice graph (d) all have zero clustering coefficients and zero transitivities. That is easy to see. Since none of them have three-node cycles, the numerator in equation 4 is zero. And, for any node, no two of its neighbors are connected, so the first sum in equation 3 is zero. The bowtie graph (c), however, has a positive values for both. The transitivity, according to equation (4), is $C = 2/6 = 1/3$. The clustering coefficient, according to equation 3, is $C = \frac{1}{6}(1+1+1/3+1/3+1+1) = 7/9$.

Even though notionally related, the clustering coefficient and the transitivity produce different results for the connectivity of triples of nodes. Schank and Wagner [7] explore the relationship between the two, and find equivalence when the definition of clustering coefficient is extended to include node weights.

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 (a) 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 (b) 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).

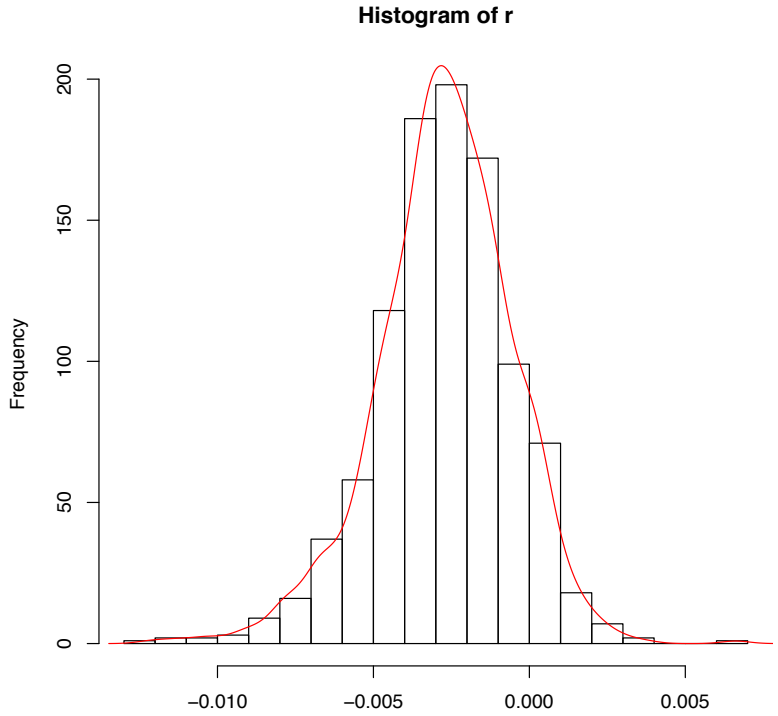


Figure 4: The histogram of degree correlations (r) of 1000 random graphs. The distribution should be centered around zero. The slight negative offset is probably due to finite-size effects.

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 in this context, it is 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 that there is a huge diversity of graphs with the same degree sequence [12]. The lattice graph in Figure 3 (d) is a rewired version of the bowtie graph (c). Indeed, both graphs are connected and both have degree sequence $[2, 2, 3, 3, 2, 2]$.

The **rich club metric** [13] measures the density of links among nodes which have a degree higher than some threshold degree. Intuitively, if all the highly-connected individuals are connected, then there's a *rich club*. If k is the threshold degree, and N_k is the set of nodes with $deg(i) \geq k$, then the rich club metric is defined as $\phi_k = linkDensity(G(N_k))$, where $G(N_k)$ is the subgraph of G defined by the set of nodes N_k. The link density is computed as in Section 2.2.5. 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 [13].

The *eigenCentrality* routine (4.2.12) is an implementation of **eigenvector centrality**. Eigenvector centrality reflects how important are all neighboring nodes. High-centrality nodes are adjacent to other high-scoring nodes. Eigenvector centrality is defined in [14]. If a node i's score x_i is the sum of scores 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 [15] 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 same framework is extensible to edges. **Edge betweenness** is proportional to the number of shortest paths that go through a given edge. An edge betweenness algorithm is given in [16]. 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:

```
octave:40> degrees([0 1 1; 1 0 1; 1 1 0])
ans =

   2   2   2

octave:41> adj = [0 1 0; 0 0 1; 1 0 0];
octave:42> [deg,indeg,outdeg] = degrees(adj);
octave:43> deg
deg =

   2   2   2

octave:44> indeg
indeg =

   1   1   1

octave:45> outdeg
outdeg =
```

```
   1   1   1

octave:46> adj = [0 1 1; 0 0 0; 0 0 0];
octave:47> [deg,indeg,outdeg] = degrees(adj);
octave:48> deg
deg =

   2   1   1

octave:49> indeg
indeg =

   0   1   1

octave:50> outdeg
outdeg =

   2   0   0
```

4.2.2 rewire.m

Degree-preserving rewiring: A graph is rewired k number of times (edges are swaped k times), in such a way that the degree of every node stays the same.

```
% 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:

```
octave:59> adj = randomGraph(20,0.4);
octave:60> % rewire five edges
octave:60> elr = rewire(adj2edgeL(adj),5);
octave:61> adjr = edgeL2adj(elr);
octave:62> assert(degrees(adj), degrees(adjr))
octave:63> % make sure that the graphs are different (edges have been rewired)
octave:63> assert( isequal(adjr, adj), false)
```

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 **??**.

```
% 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 with simultaneously
%       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:

```
octave:4> 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];
octave:5>
octave:5> % rewire the 1 -> 3 edge
octave:5> elr = rewireThisEdge(bowtie_edgeL,1,3)
elr =

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

octave:6> adj = edgeL2adj(bowtie_edgeL);
octave:7> adjr = edgeL2adj(elr);
octave:8> % check that the degree sequences of the two matrices are the same
octave:8> assert(degrees(adj),degrees(adjr))
```

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:

```
octave:17> adj = randomGraph(20,0.4);
octave:18> % rewire five random edges
octave:18> elr = rewireAssort(adj2edgeL(adj),5);
octave:19> adjr = edgeL2adj(elr);
octave:20> % check that the degree sequences stay the same
octave:20> assert(degrees(adj), degrees(adjr))
octave:21> % verify that the Pearson coefficient has increased
octave:21> assert(pearson(adjr)>=pearson(adj), true)
```

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:

```
octave:22> adj = randomGraph(20,0.4);
octave:23> % rewire five random edges
octave:23> elr = rewireDisassort(adj2edgeL(adj),5);
octave:24> adjr = edgeL2adj(elr);
octave:25> % check that the degree sequences stay the same
octave:25> assert(degrees(adj), degrees(adjr))
octave:26> % verify that the Pearson coefficient has decreased
octave:26> assert(pearson(adjr)<=pearson(adj), true)
```

4.2.6 aveNeighborDeg.m

Computes the **average degree** of **adjacent 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:

```
octave:2> adj = [0 1 1; 1 0 1; 1 1 0];
octave:3> aveNeighborDeg(adj)
ans =

   2   2   2

octave: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 1 1 0];
octave:5> 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 [19] and [20].

```
% 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:

```
octave:26> 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];
octave:27> sortNodesBySumNeighborDegrees(bowtie)
ans =

   4
   3
   6
   5
   2
   1

octave:28> adj = [0 1 1 0; 1 0 1 0; 1 1 0 1; 0 0 1 0];
octave:29> 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)
%
% Note: Works for undirected graphs only.
% Other routines used: degrees.m, kneighbors.m
% GB: last updated, Nov 24, 2014
```

Example:

```
octave:25> adj = [0 1 1 1; 1 0 0 0; 1 0 0 0; 1 0 0 0];
octave:26> 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:

```
octave: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];
octave:3> closeness(bowtie)
ans =

   0.10000
   0.10000
   0.14286
   0.14286
   0.10000
   0.10000

octave:4> adj = [0 1 1; 1 0 1; 1 1 0];
octave:5> closeness(adj)
ans =

   0.50000
   0.50000
   0.50000
```

4.2.10 nodeBetweenness.m [15]

Compute the **betweenness centrality** of all vertices [15]. 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 (multiple shortest paths possible).
%
% INPUTS: adjacency or distances matrix (nxn)
% OUTPUTS: betweeness vector for all vertices (1xn)
%
% Other routines used: numNodes.m, findAllShortestPaths.m
% GB: July 3, 2014
```

Example:

```
octave:17> 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];
octave:18> nodeBetweenness(bowtie)

ans =
```

```
   0.00000    0.00000    0.40000    0.40000    0.00000    0.00000
```

4.2.11 edgeBetweenness.m [16]

Compute **edge betweenness**. Analogous to node betweenness, edge betweenness is proportional to the number of shortest paths going through an edge. Algorithm described in [16].

```
% 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:

```
octave:4> % undirected 3-node cycle: all edges should have equal betweenness
octave:4> adj = [0 1 1; 1 0 1; 1 1 0];
octave:5> edgeBetweenness(adj)
ans =

   2.00000    1.00000    0.16667
   3.00000    1.00000    0.16667
   1.00000    2.00000    0.16667
   3.00000    2.00000    0.16667
   1.00000    3.00000    0.16667
   2.00000    3.00000    0.16667

octave:6> 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];
octave:7> edgeBetweenness(bowtie)
ans =

   2.000000    1.000000    0.033333
   3.000000    1.000000    0.133333
   1.000000    2.000000    0.033333
   3.000000    2.000000    0.133333
   1.000000    3.000000    0.133333
   2.000000    3.000000    0.133333
   4.000000    3.000000    0.300000
   3.000000    4.000000    0.300000
   5.000000    4.000000    0.133333
   6.000000    4.000000    0.133333
   4.000000    5.000000    0.133333
   6.000000    5.000000    0.033333
   4.000000    6.000000    0.133333
   5.000000    6.000000    0.033333
```

4.2.12 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:

```
octave:64> eigenCentrality([0 1 1; 1 0 1; 1 1 0])
ans =

   0.57735
   0.57735
   0.57735

octave:65> adj = [0 1 1; 1 0 0; 1 0 0];
octave:66> eigenCentrality(adj)
ans =

   0.70711
   0.50000
   0.50000
```

4.2.13 clustCoeff.m

The **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] and in [7].

```
% Compute the clustering coefficient per node.
% Ci = 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: the average clustering coefficient (aveC) and the
%         clustering coefficient vector C per node (where mean(C) = aveC)
%
% Other routines used: degrees.m, isDirected.m, kneighbors.m, numEdges.m, subgraph.m
% GB, Last updated: February 7, 2015
```

Examples:

```
octave:21> adj = [0 1 1; 1 0 1; 1 1 0];
octave:22> clustCoeff(adj)
ans =  1
octave:23>
octave:23> adj = [0 1 1; 1 0 0; 1 0 0];
octave:24> clustCoeff(adj)
ans = 0
octave:25>
```

4.2.14 transitivity.m

The **transitivity** is defined as the number of cycles of size 3 divided by the number of connected triples.

```
% Calculate the transitivity.
% C = number of triangle loops (3-cycles) / number of connected triples
```

```
% Ref: M. E. J. Newman, "The structure and function of complex networks"
% Note: Valid for directed and undirected graphs
%
% INPUT: adjacency matrix, nxn
% OUTPUT: The transitivity, C
%
% Other routines used: loops3.m, numConnTriples.m
% GB, Last updated: February 6, 2015
% Input/corrections by Dimitris Maniadakis.

function [C] = transitivity(adj)

C=3*loops3(adj)/(numConnTriples(adj)+2*loops3(adj));
```

Examples:

```
octave:28> adj = [0 1 1; 1 0 1; 1 1 0];
octave:29> transitivity(adj)
ans =  1
octave:30>
octave:30>
octave:30> adj = [0 1 1; 1 0 0; 1 0 0];
octave:31> transitivity(adj)
ans = 0
```

4.2.15 weightedClustCoeff.m [17]

Clustering coefficient for (edge-)weighted graphs. Definition from [17]. Weighted clustering coefficient with node weights is discussed in [7].

```
% Weighted clustering coefficient (edge-weights).
% 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:

```
octave:18> adj = [0 2 1; 2 0 0; 1 0 0];
octave:19> weightedClustCoeff(adj)
ans =

   0
   0
   0


octave:21> % an arbitrary weighted (symmetric) matrix
octave:21> adj = [ 0  2  1  1;  2  0  3  0;  1  3  0  0;  1  0  0  0];
octave:22> weightedClustCoeff(adj)
ans =

   0.37500
   1.00000
   1.00000
   0.00000
```

4.2.16 pearson.m [18]

Pearson degree correlation: the degree-degree correlation in a graph. Algorithm and ideas from [18].

```
% 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
% See also pearsonW.m
% GB: last updated, October 1, 2012
```

An alternative routine, using matrix algebra entirely is given in *pearsonW.m*, and is reproduced below.

```
function prs = pearsonW(M)

%calculates pearson degree correlation of M
[rows,colms]=size(M);
won=ones(rows,1);
k=won'*M;
ksum=won'*k';
ksqsum=k*k';
xbar=ksqsum/ksum;
num=(won'*M*won'*xbar)*M*(M*won-xbar*won);
M*(M*won-xbar*won);
kkk=(k'-xbar*won).*(k'.^.5);
denom=kkk'*kkk;

prs=num/denom;
```

Examples:

```
octave:14> adj = [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];
octave:15> pearson(adj)
ans = -1
```

```
octave:16> pearsonW(adj)
ans = -1

octave:17> 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];
octave:18> pearson(bowtie)
ans = -0.16667
octave:19> pearsonW(bowtie)
ans = -0.16667
```

4.2.17 richClubMetric.m [13]

The **rich club metric** is defined as the density of links among nodes with nodal degree k or higher. Algorithm and ideas from [13].

```
% 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:

```
octave:2> cycle3 = [0 1 1; 1 0 1; 1 1 0];
octave:3> richClubMetric(cycle3, 2)
ans =  1

octave: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 1 1 0];
octave:5> richClubMetric(bowtie,2)
ans =  0.46667
```

4.2.18 sMetric.m [19]

S-metric: the sum of products of nodal degrees across all edges. Definition and applications described in [19].

```
% 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*:

```
% [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:

```
octave:2> cycle3= [0 1 1; 1 0 1; 1 1 0];
octave:3> sMetric( cycle3 )
ans =  24
octave:4>
octave:4> one_edge = [0 1; 0 0];
octave:5> 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* phrase 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.

The term **small-world networks** also refers to distance. "Small-world" networks 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.

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
```

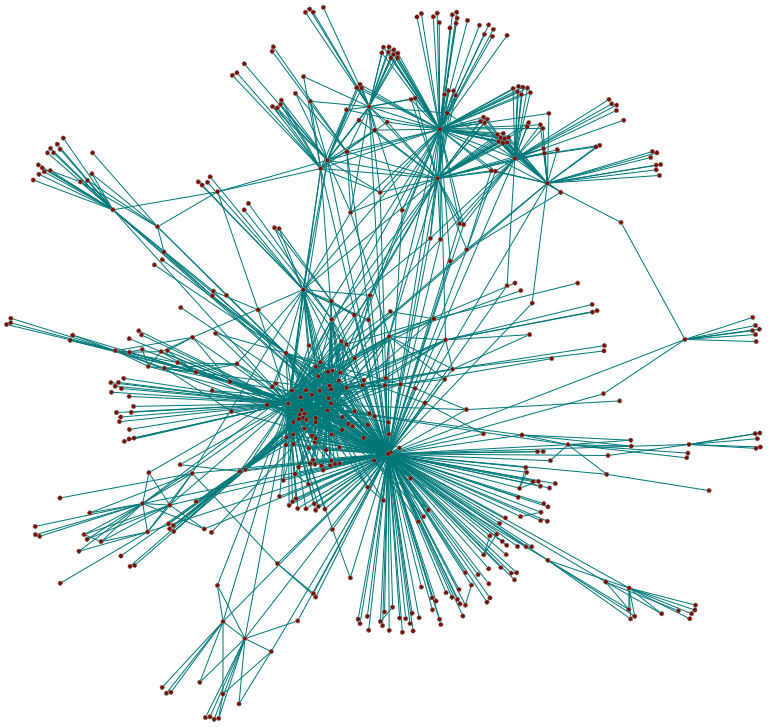


Figure 5: Small-world network example: the graph of the Lufthansa air routes from July 2006; 470 nodes (airports) and diameter of 5 ($log(470) \sim 6.15$). So 5 is the largest number of hops that need to be traveled to reach any airport from any starting location.

```
%
% Note: Works for a weighted/directed graph.
% GB: last updated, September 28, 2012
```

Examples:

```
octave:2> % a single directed edge
octave:2> adj = [0 1; 0 0];
octave:3> d = simpleDijkstra(adj,1)
d =

   0   1

octave:4> d = simpleDijkstra(adj,2)
d =

   Inf     0
```

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
```

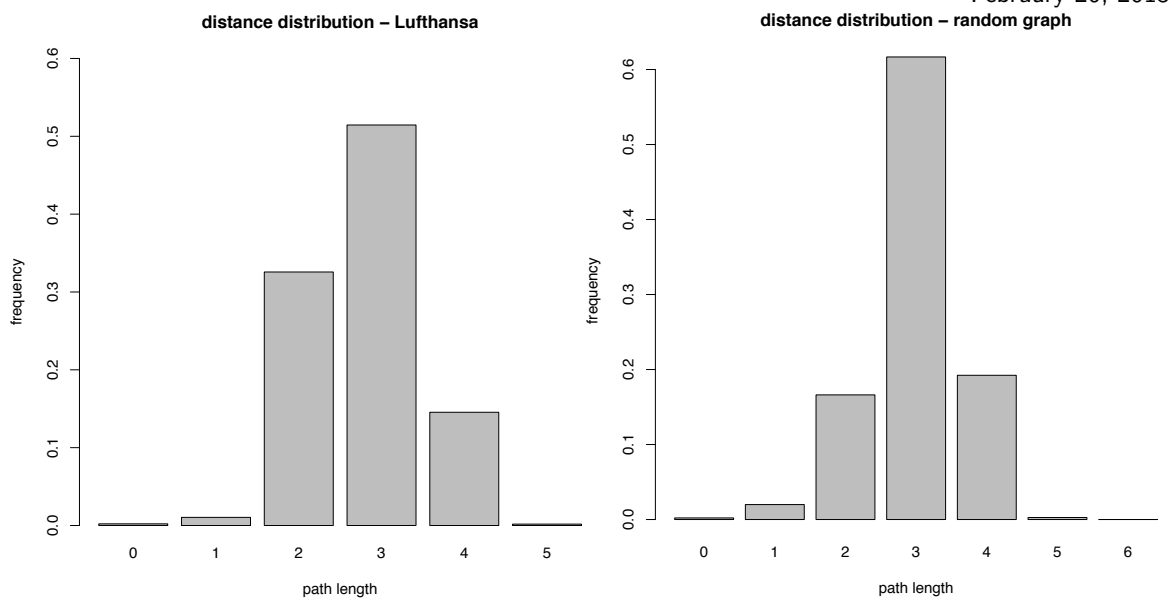


Figure 6: The distance distribution of the Lufthansa network from Figure 5 versus the distance distribution of a random graph with the same size (470 nodes) and same density (0.02).

```
%          all distances and paths from s
% Other routines used: adj2adjL.m
% GB: last updated, Oct 5, 2012
```

Examples:

```
octave: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];
octave:4> % distance and path from node 1 to node 2
octave:4> [d,P]=dijkstra(bowtie,1,2)
d =  1
P =

   1   2

octave:5> % distance and path from node 6 to node 2
octave:5> [d,P]=dijkstra(bowtie,6,2)
d =  3
P =

   6   4   3   2
```

5.2.3 shortestPathDP.m [21]

Shortest path algorithm using dynamic programming. Returns the minimum weight path length and the route. Ideas from [21].

```
% 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(1,i) - distance from node "i" to "t" in "k" steps
%
% GB: last updated, Oct 5 2012
```

Examples:

```
octave:5> [J_st, route_st, J, route]=shortestPathDP(bowtie,1,6,1);
octave:6> J_st
J_st = Inf
octave:7> route_st
route_st =

   1   6

octave:8> [J_st, route_st, J, route]=shortestPathDP(bowtie,1,6,6);
octave:9> J_st
J_st =  3
octave:10>
octave:10> route_st
route_st =

   1   3   4   6
```

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] Thomas Schank, Dorothea Wagner, Approximating Clustering-Coefficient and Transitivity, Journal of Graph Algorithms and Applications, Vol 9, 2005

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

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

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

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

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

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

[14] 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

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

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

[17] 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

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

[19] 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

[20] 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

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

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

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

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

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

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

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

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

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

[30] 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

[31] 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

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

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

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

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

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

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

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

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