Octave routines for network analysis

GB

September 21, 2014

Contents

September 21, 2014

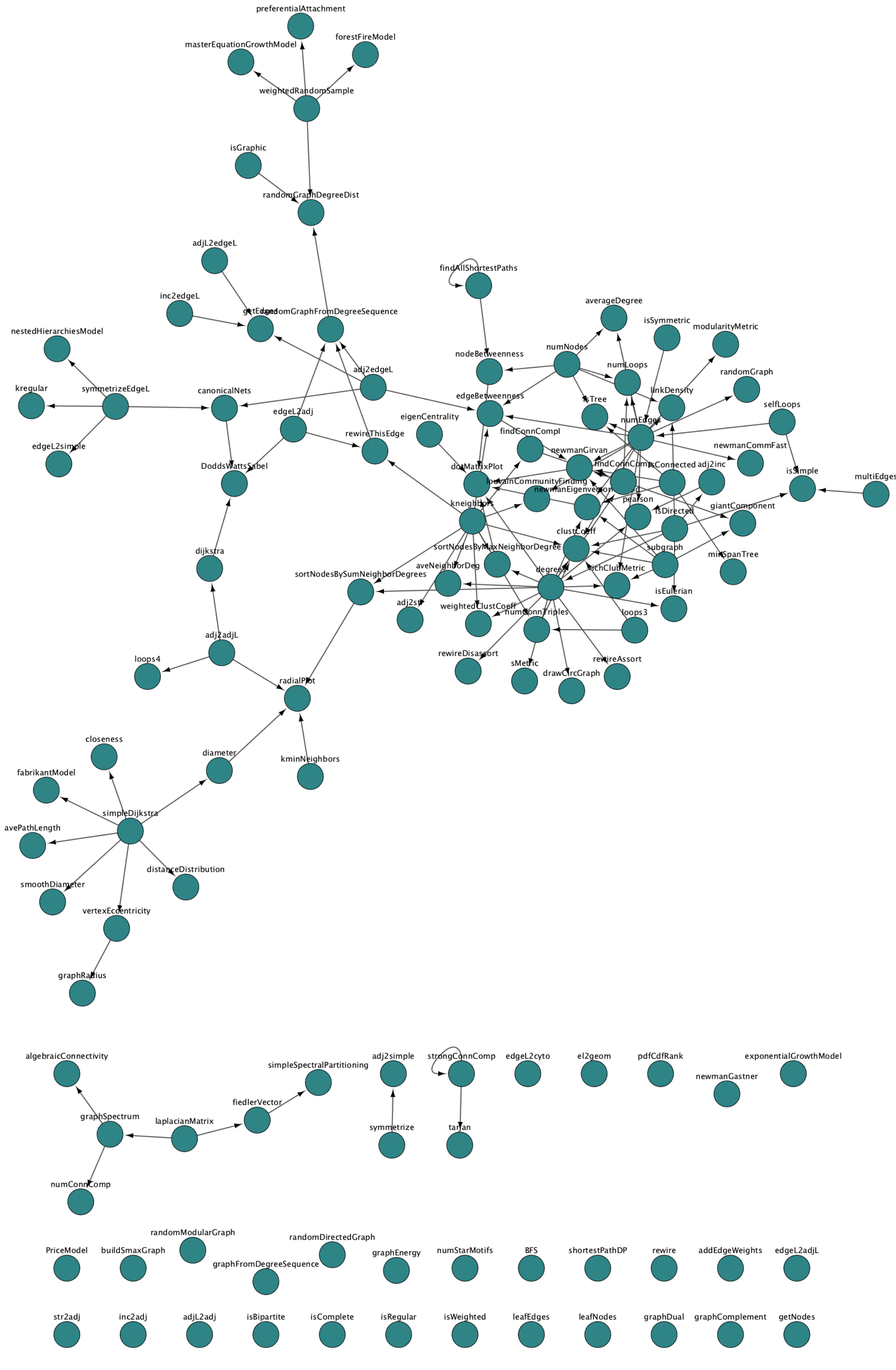


Figure 1: Graph of functional interdependencies in this toolbox. An edge points from routine A to routine B if routine A is called within routine B. For example, *isConnected.m* is called within *minSpanTree.m*

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).

License/Copyright

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:

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

There can be other representations depending on purpose, understanding, or algorithm implementation. Suppose 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:

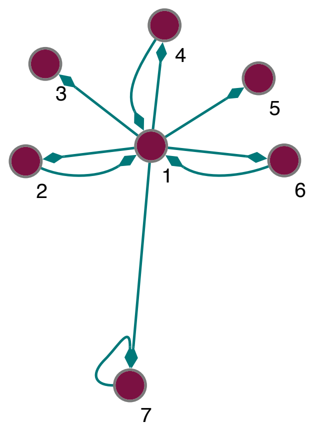
.2,.3,,.4,

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.



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

$$\text{adj list} = \begin{array}{cc} \{1\} \to & 2, 3, 4, 5, 6, 7 \\ \{2\} \to & 1 \\ \{3\} \to & \emptyset \\ \{4\} \to & 1 \\ \{5\} \to & \emptyset \\ \{6\} \to & 1 \\ \{7\} \to & 7 \end{array}$$

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] [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 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
```

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