Introduction to Network Models

Lecture 11

Agostino Merico

4 Decemebr 2017

Network science – Basics

We are now moving into one of the most recent developments of complex systems science: *networks*.

Network science – Basics

We are now moving into one of the most recent developments of complex systems science: *networks*.

Stimulated by two seminal papers on small-world and scale-free networks

- D. J. Watts & S. H. Strogatz, Collective dynamics of 'small-world' networks, *Nature*, 393:440–442, 1998.
- A.-L. Barabási & R. Albert, Emergence of scaling in random networks, *Science*, 286:509–512, 1999.

Network science – Basics

We are now moving into one of the most recent developments of complex systems science: *networks*.

Stimulated by two seminal papers on small-world and scale-free networks

- D. J. Watts & S. H. Strogatz, Collective dynamics of 'small-world' networks, *Nature*, 393:440–442, 1998.
- A.-L. Barabási & R. Albert, Emergence of scaling in random networks, *Science*, 286:509–512, 1999.

network science has been rapidly growing and producing novel perspectives, research questions, and analytical tools to study various kinds of systems in a number of disciplines, including biology, ecology, sociology, economics, political science, management science, engineering, medicine, and more !

Network science – Historical roots

Historical roots can be found in several disciplines:

Network science – Historical roots

Historical roots can be found in several disciplines:

Mathematics: *graph theory*, the study various properties of abstract structures called *graphs* made of *nodes* (a.k.a. vertices) and *edges* (a.k.a. links, ties);

Network science – Historical roots

Historical roots can be found in several disciplines:

Mathematics: *graph theory*, the study various properties of abstract structures called *graphs* made of *nodes* (a.k.a. vertices) and *edges* (a.k.a. links, ties);

Physics: *statistical physics*, the study of collective systems made of a large number of entities (such as phase transitions);

Network science – Historical roots

Historical roots can be found in several disciplines:

Mathematics: *graph theory*, the study various properties of abstract structures called *graphs* made of *nodes* (a.k.a. vertices) and *edges* (a.k.a. links, ties);

Physics: *statistical physics*, the study of collective systems made of a large number of entities (such as phase transitions);

Social sciences: *Social network analysis*;

Network science – Historical roots

Historical roots can be found in several disciplines:

Mathematics: *graph theory*, the study various properties of abstract structures called *graphs* made of *nodes* (a.k.a. vertices) and *edges* (a.k.a. links, ties);

Physics: *statistical physics*, the study of collective systems made of a large number of entities (such as phase transitions);

Social sciences: *Social network analysis*;

Computer science: *Artificial neural networks*.

Network science – Historical roots

Historical roots can be found in several disciplines:

Mathematics: *graph theory*, the study various properties of abstract structures called *graphs* made of *nodes* (a.k.a. vertices) and *edges* (a.k.a. links, ties);

Physics: *statistical physics*, the study of collective systems made of a large number of entities (such as phase transitions);

Social sciences: *Social network analysis*;

Computer science: *Artificial neural networks*.

Investigations based on network science focus on the connections and interactions among the components of a system, not just on each individual component.

Network science – Terminology

Network

A *network* (or *graph*) consists of a set of *nodes* (or *vertices*, *actors*) and a set of *edges* (or *links*, *ties*) that connect those nodes.

Neighbour

Node j is called a *neighbour* of node i if (and only if) node i is connected to node j.

Adjacency matrix

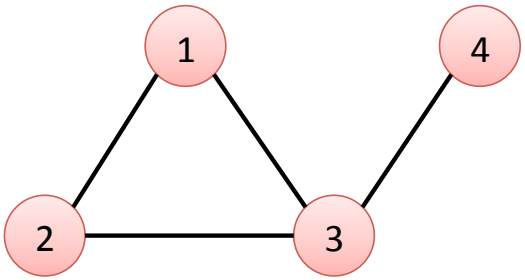
A matrix with rows and columns lab eled by nodes, whose i-th row, j-th column component a_{ij} is 1 if node i is a neighbour of node j, or 0 otherwise.

Adjacency list

A list of lists of nodes whose i-th component is the list of its neighbours.

Network science – Example

Network



Adjacency matrix

i \ j	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

Adjacency list

i	
1	→ {2, 3}
2	→ {1, 3}
3	→ {1, 2, 4}
4	→ {3}

Examples of an adjacent matrix and an adjacent list. The adjacency list offers a more compact, memory-efficient representation, especially if the network is sparse (i.e. if the network density is low–which is often the case for most rea-world networks). But the adjacency matrix also has some benefits, such as its feasibility for mathematical analysis and easy to access its specific components.

Network science – More terminology

Degree
The number of links connected to a node; the degree of node i is written as $\deg(i)$.

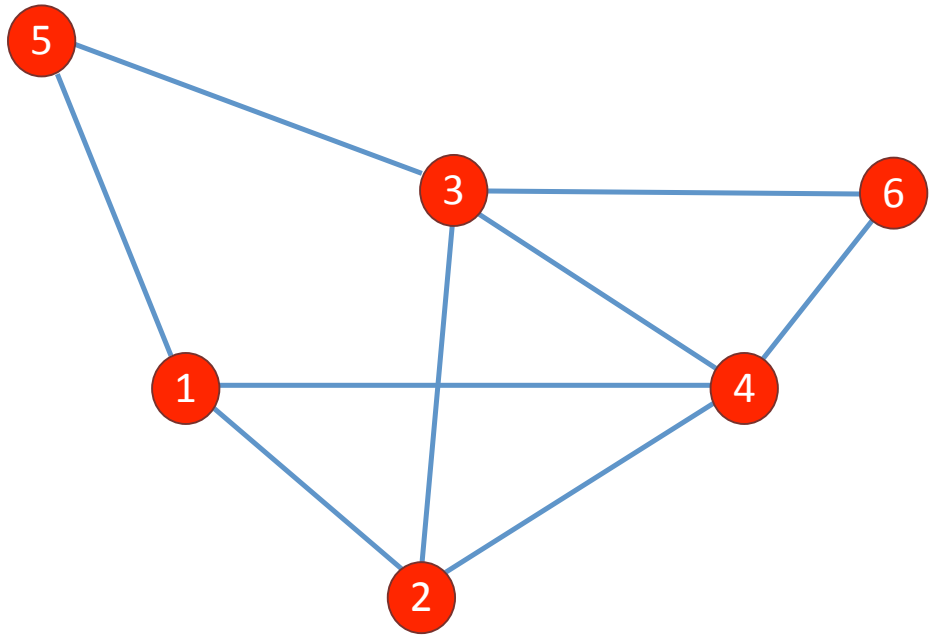
Walk
A list of links that are sequentially connected to form a continuous route on a network. In particular: *trail* is a walk that doesn't go thorugh any link more than once; *path* is a walk that doesn't go through any node (and any link) more than once; *cycle* is a walk that starts and ends at the same node without going through any node more than once on its way.

Subgraph
Part of the graph

Connected graph
A graph in which a path exists between any pair of nodes.

Connected component
A subgraph of a graph that is connected within itself but not connected to the rest of the graph.

Network science – Exercise



1. Represent the network in (a) and adjacency matrix and (b) an adjacency list;
2. Determine the degree for each node;
3. Classify the following walks as trail, path, cycle, or other.
   - $6 \to 3 \to 2 \to 4 \to 2 \to 1$
   - $1 \to 4 \to 6 \to 3 \to 2$
   - $5 \to 1 \to 2 \to 3 \to 5$
4. Identify all fully connected three-node subgraphs (i.e. triangles).

Network science – Exercise

Network science – More terminology

Complete graph
A graph in which any pair of nodes are connected;

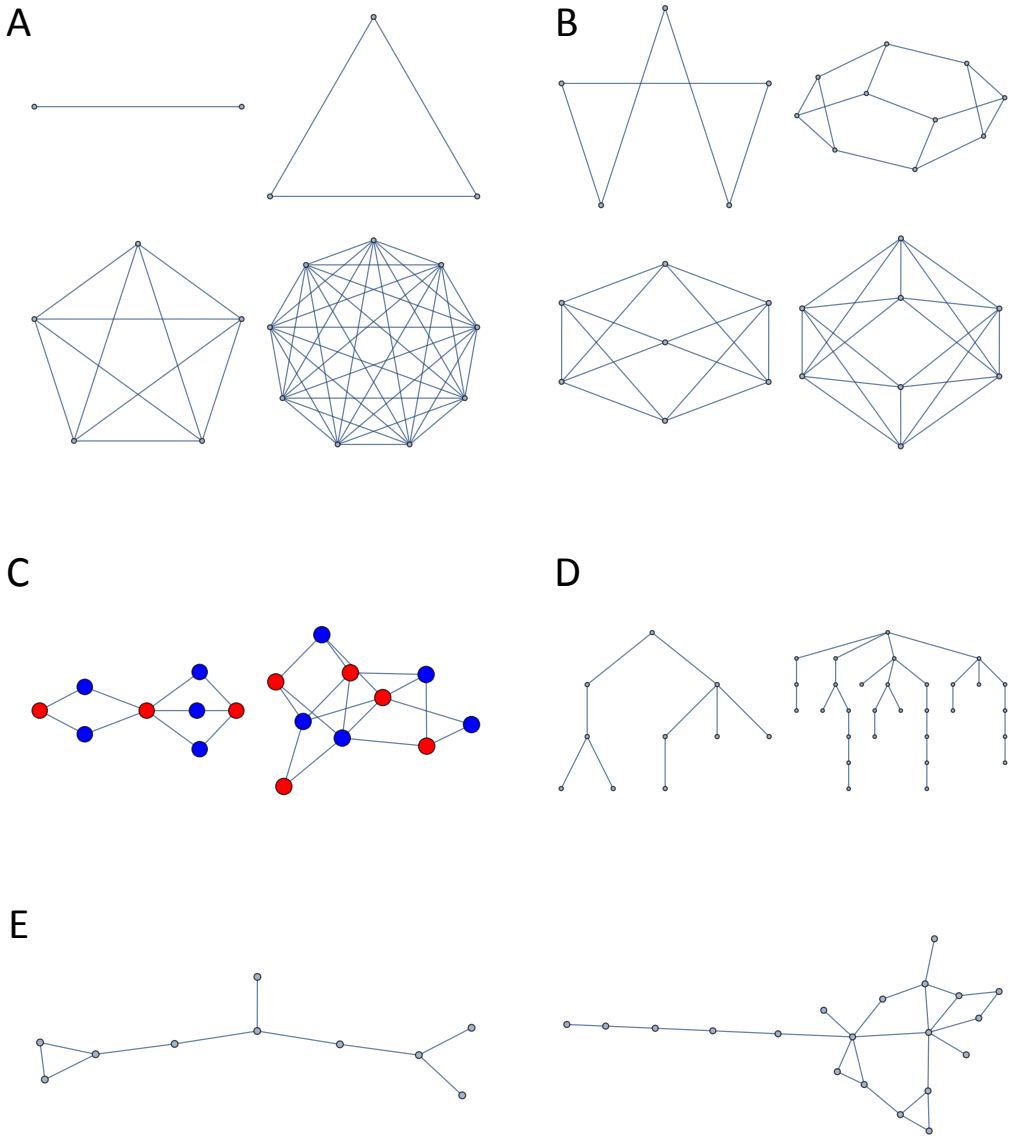
Regular graph
A graph in which all nodes have the same degree; every complete graph is regular;

Bipartite (n-partite) graph
A graph whose nodes can be divided into two (or n) groups so that no edge connects nodes within each group:

Tree graph
A graph in which there is no cycle; a graph made of multiple trees is called a forest graph; every tree or forest graph is bipartite;

Planar graph
A graph that can be graphically drawn in a two-dimensional plane with no edge crossings; every tree or forest graph is planar.

Network science – More terminology



A: Complete graphs

B: Regular graphs

C: Bipartite graphs
(colours show groups)

D: Tree graphs

E: Planar graphs

Network science – More terminology

Classifications of networks according to the types of their links (1 of 3).

Undirected link

A symmetric connection between nodes; if node i is connected to node j by an undirected link, then node j also recognises node i as its neighbor; a graph made of undirected linke is called an undirected graph; the adjacency matrix of an undirected graph is always symmetric.

Directed link

An asymmetric connection from one node to another; even if node i is connected to node j by a directed link, the connection isn't necessarily reciprocated from node j to node i; a graph made of directed links is called a directed graph; the adjacency matrix of a directed graph is generally asymmetric.

Unweighted link

A link without any weight value associated to it; there are only two possibilities between a pair of nodes in a network with unweighted links; whether there is a link between them or not; the adjacency matrix of such a network is made of only 0s and 1s.

Network science – More terminology

Classifications of networks according to the types of their links (2 of 3).

Weighted link
A link with a weight value associated to it; a weight is usually given by a non-negative real number, which may represent a connection strength or distance between nodes, depending on the nature of the system being modeled; the definition of the adjacency matrix can be extended to contain those link weight values for networks with weighted links; the sum of the weights of links connected to a node is often called the node strength, which corresponds to a node degree for unweighted graphs.

Multiple links
Links that share the same origin and destination; such multiple links connect two nodes more than once.

Self-loop
A link that originates and ends at the same node.

Network science – More terminology

Classifications of networks according to the types of their links (3 of 3).

Simple graph
A graph that does not contain directed, weighted, or multiple links, or self-loops; traditional graph theory mostly focuses on simple graphs.

Multigraph
A graph that may contain multiple links; many mathematicians also allow multigraphs to contain self-loops; multigraphs can be undirected or directed.

Network science – More terminology

Classifications of networks according to the types of their links (3 of 3).

Simple graph
A graph that does not contain directed, weighted, or multiple links, or self-loops; traditional graph theory mostly focuses on simple graphs.

Multigraph
A graph that may contain multiple links; many mathematicians also allow multigraphs to contain self-loops; multigraphs can be undirected or directed.

NOTE: According to these taxonomies, all the examples shown in the previous figure are simple graphs; but many real-world networks are modeled using directed, weighted, and/or multiple links.

Constructing network models with `NetworkX`

Now that we have completed a crash course on graph terminology, it is time to begin with computational modelling of networks.

There is a wonderful python module called `NetworkX` for network modelling and analysis; it is a free and widely used toolkit.

If you use Anaconda, `NetworkX` is already installed; if you use Enthought Canopy, you can easily install `NetworkX` by using the package manager.

The documentation for `NetworkX` is available online at http://networkx.github.io.

Constructing network models with `NetworkX`

Data structure used in `NetworkX`.

Graph For undirected simple graphs (self-loops are allowed);

DiGraph For directed simple graphs (self-loops are allowed);

MultiGraph For undirected multigraphs (self-loops and multiple links are allowed);

MultiDiGraph For directed multigraphs (self-loops and multiple links are allowed).

You can choose one of these four data types that suits your modeling purposes. We will use mostly Graph and DiGraph.

You can construct a graph of your own manually.

Constructing network models with NetworkX

```
import networkx as nx
```

Constructing network models with NetworkX

```
import networkx as nx

# creates a new empty Graph object
g = nx.Graph( )
```

Constructing network models with NetworkX

```
import networkx as nx

# creates a new empty Graph object
g = nx.Graph()

# add a node named 'John'
g.add_node('John')
```

Constructing network models with NetworkX

```
import networkx as nx

# creates a new empty Graph object
g = nx.Graph()

# add a node named 'John'
g.add_node('John')

# add a bunch of nodes at once
g.add_nodes_from(['Josh', 'Jane', 'Jesse', 'Jack'])
```

Constructing network models with NetworkX

```
import networkx as nx

# creates a new empty Graph object
g = nx.Graph()

# add a node named 'John'
g.add_node('John')

# add a bunch of nodes at once
g.add_nodes_from(['Josh', 'Jane', 'Jesse', 'Jack'])

# add a link between 'John' and 'Jane'
g.add_edge('John', 'Jane')
```

Constructing network models with NetworkX

```
import networkx as nx

# creates a new empty Graph object
g = nx.Graph()

# add a node named 'John'
g.add_node('John')

# add a bunch of nodes at once
g.add_nodes_from(['Josh', 'Jane', 'Jesse', 'Jack'])

# add a link between 'John' and 'Jane'
g.add_edge('John', 'Jane')

# add a bunch of links at once
g.add_edges_from([('Jesse', 'Josh'), ('John', 'Jack'), ('Jack', 'Jane')])
```

Constructing network models with NetworkX

```
import networkx as nx

# creates a new empty Graph object
g = nx.Graph()

# add a node named 'John'
g.add_node('John')

# add a bunch of nodes at once
g.add_nodes_from(['Josh', 'Jane', 'Jesse', 'Jack'])

# add a link between 'John' and 'Jane'
g.add_edge('John', 'Jane')

# add a bunch of links at once
g.add_edges_from([('Jesse', 'Josh'), ('John', 'Jack'), ('Jack', 'Jane')])

# add more links
# undefined nodes will be created automatically
g.add_edges_from([('Jesse', 'Jill'), ('Jill', 'Jeff'), ('Jeff', 'Jane')])
```

Constructing network models with NetworkX

```
import networkx as nx

# creates a new empty Graph object
g = nx.Graph()

# add a node named 'John'
g.add_node('John')

# add a bunch of nodes at once
g.add_nodes_from(['Josh', 'Jane', 'Jesse', 'Jack'])

# add a link between 'John' and 'Jane'
g.add_edge('John', 'Jane')

# add a bunch of links at once
g.add_edges_from([('Jesse', 'Josh'), ('John', 'Jack'), ('Jack', 'Jane')])

# add more links
# undefined nodes will be created automatically
g.add_edges_from([('Jesse', 'Jill'), ('Jill', 'Jeff'), ('Jeff', 'Jane')])

# remove the link between 'John' and 'Jane'
g.remove_edge('John', 'Jane')
```

Constructing network models with NetworkX

```
import networkx as nx

# creates a new empty Graph object
g = nx.Graph()

# add a node named 'John'
g.add_node('John')

# add a bunch of nodes at once
g.add_nodes_from(['Josh', 'Jane', 'Jesse', 'Jack'])

# add a link between 'John' and 'Jane'
g.add_edge('John', 'Jane')

# add a bunch of links at once
g.add_edges_from([('Jesse', 'Josh'), ('John', 'Jack'), ('Jack', 'Jane')])

# add more links
# undefined nodes will be created automatically
g.add_edges_from([('Jesse', 'Jill'), ('Jill', 'Jeff'), ('Jeff', 'Jane')])

# remove the link between 'John' and 'Jane'
g.remove_edge('John', 'Jane')

# remove the node 'John'
# all links connected to that node will be removed too
g.remove_node('John')
```

```
import numpy as np
```

```
import numpy as np
```

```
import numpy as np

x = 2
```

```
import numpy as np

x = 2
np.linspace(0,1)
```

```python
import numpy as np

x = 2
np.linspace(0,1)
# Hi
```

```
import numpy as np

x = 2
np.linspace(0,1)
# Hi
np.sin(x)
```

Constructing network models with `NetworkX`

In those examples, I used strings for the names of the nodes, but a node's name can be a number, a string, a list, a tuple, a dictionary, etc.

If you execute those command lines, you can see the following results:

;

Constructing network models with NetworkX

In those examples, I used strings for the names of the nodes, but a node's name can be a number, a string, a list, a tuple, a dictionary, etc.

If you execute those command lines, you can see the following results:

```
>>> g
<networkx.classes.graph.Graph object at 0x1819cdfe10>
```

The first output (<networkx.classes. ... >) shows that g is a Graph object;

;

Constructing network models with NetworkX

In those examples, I used strings for the names of the nodes, but a node's name can be a number, a string, a list, a tuple, a dictionary, etc.

If you execute those command lines, you can see the following results:

```
>>> g
<networkx.classes.graph.Graph object at 0x1819cdfe10>

>>> print(g.nodes)
['John', 'Josh', 'Jane', 'Jesse', 'Jack', 'Jill', 'Jeff']
```

The first output (`<networkx.classes. ... >`) shows that g is a Graph object;

`print(g.nodes)` returns a list of all nodes in the network;

;

Constructing network models with NetworkX

In those examples, I used strings for the names of the nodes, but a node's name can be a number, a string, a list, a tuple, a dictionary, etc.

If you execute those command lines, you can see the following results:

```
>>> g
<networkx.classes.graph.Graph object at 0x1819cdfe10>

>>> print(g.nodes)
['John', 'Josh', 'Jane', 'Jesse', 'Jack', 'Jill', 'Jeff']

>>> print(g.edges)
[('Jane', 'Jeff'), ('Jesse', 'Jill'), ('Jill', 'Jeff')]
```

The first output (`<networkx.classes. ... >`) shows that g is a Graph object;

`print(g.nodes)` returns a list of all nodes in the network;

`print(g.edges)` returns a list of all links;

Constructing network models with NetworkX

In those examples, I used strings for the names of the nodes, but a node's name can be a number, a string, a list, a tuple, a dictionary, etc.

If you execute those command lines, you can see the following results:

```
>>> g
<networkx.classes.graph.Graph object at 0x1819cdfe10>

>>> print(g.nodes)
['John', 'Josh', 'Jane', 'Jesse', 'Jack', 'Jill', 'Jeff']

>>> print(g.edges)
[('Jane', 'Jeff'), ('Jesse', 'Jill'), ('Jill', 'Jeff')]

>>> print(g.adj)
{'John': {}, 'Josh': {}, 'Jane': {'Jeff': {}}, 'Jess': {'Jill': {}}, 'Jack': {},
 'Jill': {'Jess': {}, 'Jeff': {}}, 'Jeff': {'Jill': {}, 'Jane': {}}}
```

The first output (`<networkx.classes. ... >`) shows that g is a Graph object;

`print(g.nodes)` returns a list of all nodes in the network;

`print(g.edges)` returns a list of all links;

`print(g.adj)` provides a view of the adjacency list data structure (the links of each node) by the dictionary-like object; **note**: the property of each node are also stored (they are initially empty).

Constructing network models with NetworkX

Any node property can be dynamically added or modified as follows:

```
>>> g.node['Jeff']['job'] = 'student'
>>> g.node['Jeff']['age'] = 20
>>> g.node['Jeff']
{'age':  20, 'job':  'student'}
```

Constructing network models with NetworkX

In the examples above, we manually constructed network models by adding or removing nodes and links. But NetworkX also has some built-in functions that can generate networks of specific shapes.

```
import networkx as nx

# complete graph made of 5 nodes
g1 = nx.complete_graph(5)

# complete (fully connected) bipartite graph
# made of group of 3 nodes and group of 4 nodes
g2 = nx.complete_bipartite_graph(3, 4)

# Zachary's Karate Club graph
g3 = nx.karate_club_graph()
```

The last example (Zachary's Karate Club graph) is a famous classic example of social networks reported by Wayne Zachary in the '70s; it is a network of friendships among 34 members of a karate club at a U.S. university.

Constructing network models with NetworkX

The lists of links of the networks above are as follows:

```
>>> print(g1.edges)
[(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]

>>> print(g2.edges)
[(0, 3), (0, 4), (0, 5), (0, 6), (1, 3), (1, 4), (1, 5), (1, 6), (2, 3), (2, 4),
(2, 5), (2, 6)]

>>> print(g3.edges)
[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 10), (0, 11),
(0, 12), (0, 13), (0, 17), (0, 19), (0, 21), (0, 31), (1, 2), (1, 3), (1, 7), (1, 13),
(1, 17), (1, 19), (1, 21), (1, 30), (2, 3), (2, 32), (2, 7), (2, 8), (2, 9), (2, 13),
(2, 27), (2, 28), (3, 7), (3, 12), (3, 13), (4, 10), (4, 6),(5, 16), (5, 10), (5, 6),
(6, 16), (8, 32), (8, 30), (8, 33), (9, 33), (13, 33), (14, 32), (14, 33), (15, 32),
(15, 33), (18, 32), (18, 33), (19, 33), (20, 32), (20, 33), (22, 32), (22, 33), (23, 32),
(23, 25), (23, 27), (23, 29), (23, 33), (24, 25), (24, 27), (24, 31), (25, 31), (26, 33),
(26, 29), (27, 33), (28, 33), (28, 31), (29, 32), (29, 33), (30, 33), (30, 32), (31, 33),
(31, 32), (32, 33)]
```

Visualising networks with NetworkX

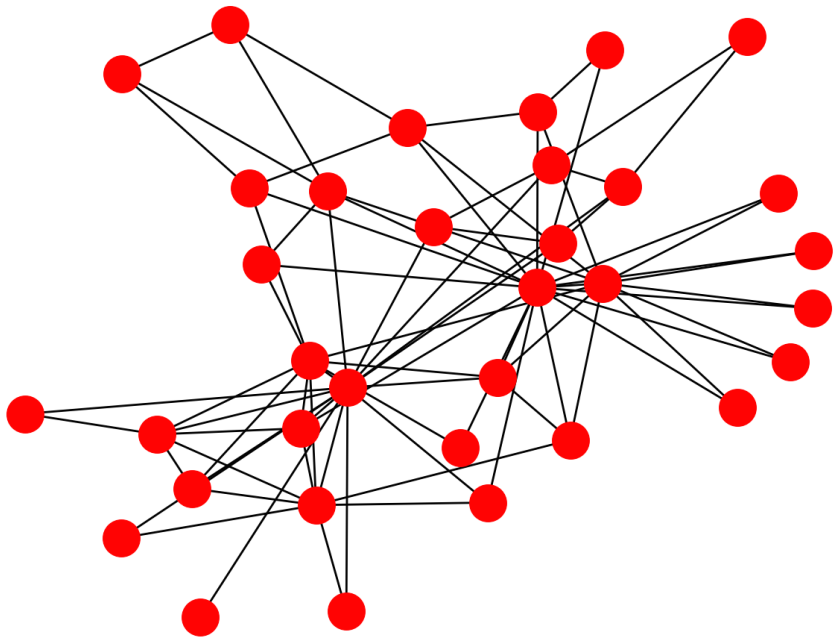
NetworkX also provides functions for visualizing networks; they are not as powerful as other more specialised software, but still quite handy and useful, especially for small- to mid-sized network visualization; those visualization functions depend on the functions defined in matplotlib (pylab), so we need to import it before visualizing networks.

The simplest way is to use the NetworkX's `draw` function, as follows:

```
from pylab import *
import networkx as nx

g = nx.karate_club_graph()
nx.draw(g)
show()
```

Visualising networks with `NetworkX`

The layout of the nodes and links is automatically determined by the *Fruchterman-Reingold force-directed* algorithm (called "spring layout" in NetworkX); this heuristic algorithm tends to bring groups of well-connected nodes closer to each other, making the result of visualization more meaningful and aesthetically more pleasing.



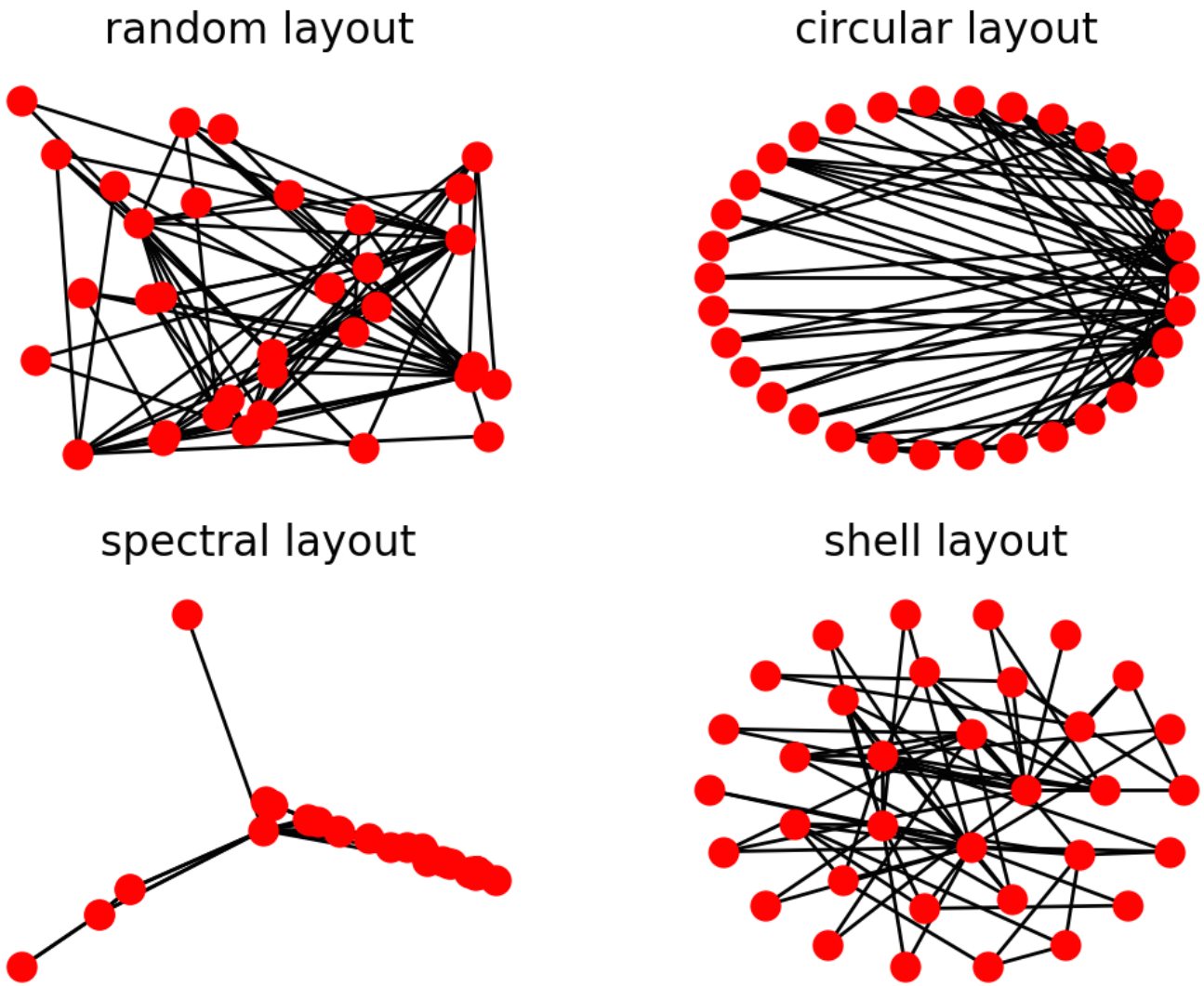
Visual output of the Zachary's Karate Club network. Your result may not look like this because the spring layout algorithm uses random initial positions.

Visualising networks with NetworkX

There are several other layout algorithms; also, there are many options you can use to customise visualization results; check out the NetworkX's online documentation to learn more about what you can do. For example:

```
import pylab as pl
import networkx as nx

g = nx.karate_club_graph()
pl.subplot(2,2,1)
nx.draw_random(g, node_size=60)
pl.title('random layout)

pl.subplot(2, 2, 2)
nx.draw_circular(g, node_size=60)
pl.title('circular layout')

pl.subplot(2, 2, 3)
nx.draw_spectral(g, node_size=60)
pl.title('spectral layout')

pl.subplot(2, 2, 4)
shells = [[0, 1, 2, 32, 33],
   [3, 5, 6, 7, 8, 13, 23, 27, 29, 30, 31],
   [4, 9, 10, 11, 12, 14, 15, 16, 17, 18,
   19, 20, 21, 22, 24, 25, 26, 28]]
nx.draw_shell(g, , node_size=60, nlist = shells)
pl.title('shell layout')
pl.show()
```

Visualising networks with `NetworkX`

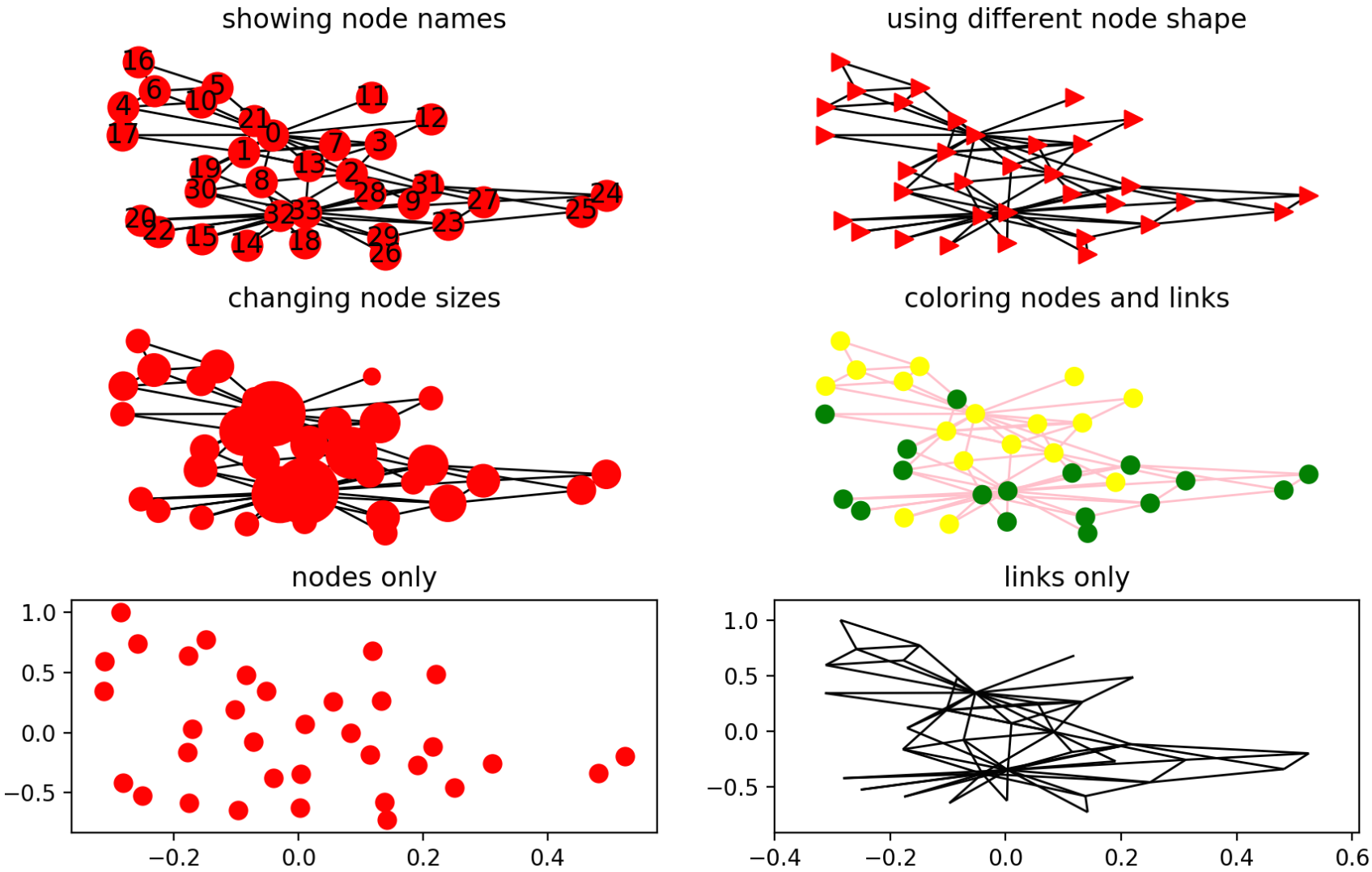


Visual output showing examples of network layouts available in NetworkX.

Visualising networks with NetworkX

```
import pylab as pl
import networkx as nx

g = nx.karate_club_graph()

positions = nx.spring_layout(g)

pl.subplot(3, 2, 1)
nx.draw(g, positions, node_size=180, with_labels=True)
pl.title('showing node names')

pl.subplot(3, 2, 2)
nx.draw(g, positions, node_size=60, node_shape='>')
pl.title('using different node shape')

pl.subplot(3, 2, 3)
nx.draw(g, positions, node_size=[g.degree(i)*50 for i in g.nodes()])
pl.title('changing node sizes')

pl.subplot(3, 2, 4)
nx.draw(g, positions, node_size=60, edge_color='pink',
        node_color=['yellow' if i<17 else 'green' for i in g.nodes()])
pl.title('coloring nodes and links')

pl.subplot(3, 2, 5)
nx.draw_networkx_nodes(g, positions, node_size=60)
pl.title('nodes only')

pl.subplot(3, 2, 6)
nx.draw_networkx_edges(g, positions)
pl.title('links only')

pl.show()
```

Visualising networks with `NetworkX`



Examples of drawing options available in NetworkX; the same node positions are used in all panels; the last two examples show the axes because they are generated using different drawing functions; to suppress the axes, use `axis('off')` right after the network drawing.

Dynamical network models

There are different categories of dynamical network models. We will look at the following three:

Models for "dynamics *on* networks".
These models consider how the states of components (i.e. the nodes) change over time through their interactions with other nodes that are connected to them. Network topology is fixed throughout time. Cellular automata, boolean networks, and artificial neural networks (without learning) all belong to this class.

Models for "dynamics *of* networks".
These are the models that consider changes in network topology, for various purposes: to understand mechanisms that bring particular network topologies, to evaluate robustness and vulnerability of networks, to design procedures for improving certain properties of networks, etc.

Models for "adaptive networks".
These are models that describe the co-evolution of dynamics on and of networks, where node states and network topologies dynamically change . Adaptive network models provide a generalised modeling framework for complex systems.

Dynamical network models

Because NetworkX adopts dictionaries as main data structure, we can easily add states to nodes (and links) and dynamically update those states iteratively. This is a simulation of dynamics on networks.

Many real-world dynamical networks fall into this category, including:

- Regulatory relationships among genes and proteins within a cell, where nodes are genes and/or proteins and the node states are their expression levels.
- Ecological interactions among species in an ecosystem, where nodes are species and the node states are their populations.
- Disease infection on social networks, where nodes are individuals and the node states are their epidemiological states (e.g., susceptible, infected, recovered, immunised, etc.).
- Information/culture propagation on organizational/social networks, where nodes are individuals or communities and the node states are their informational/cultural states.

Further reading

Hiroki Sayama 2015
Introduction to the Modeling and Analysis of Complex Systems
Open SUNY Textbooks

NetworkX Reference 2.0
https://networkx.github.io/documentation/stable/_downloads/networkx_reference.pdf
Sep 20, 2017