CS4423 - Networks

Prof. Götz Pfeiffer
School of Mathematics, Statistics and Applied Mathematics
NUI Galway

Lecture 3: A NetworkX Tutorial

NetworkX is a Python (https://www.python.org/) library for studying graphs and networks. NetworkX is suitable for operation on large real-world graphs: e.g., graphs in excess of 10 million nodes and 100 million edges. Using a pure-Python "dictionary of dictionaries" data structure, NetworkX is a reasonably efficient, very scalable, highly portable framework for network and social network analysis. NetworkX is free software. (Wikipedia) (https://en.wikipedia.org/wiki/NetworkX) (github.io) (http://networkx.github.io/)

In [1]:
```
import networkx as nx
```

Creating a graph

Create an **empty** graph with no nodes and no edges.

In [2]:
```
G = nx.Graph()
```

By definition, a `Graph` is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). In NetworkX, nodes can be any hashable object e.g. a text string, an image, an XML object, another Graph, a customized node object, etc. (Note: Python's `None` object should not be used as a node as it determines whether optional function arguments have been assigned in many functions.)

Nodes

The graph `G` can be grown in several ways. NetworkX includes many graph generator functions and facilities to read and write graphs in many formats. We start with simple manipulations. You can add one node at a time:

In [3]:
```
G.add_node(1)
```

Now the graph `G` has one node and, still, zero edges. This can be seen by:

In [4]:
```
G.number_of_nodes()
```

Out[4]: 1

In [5]:
```
G.order()
```

Out[5]: 1

In [6]:
```
G.number_of_edges(),  G.size()
```

Out[6]:
```
(0, 0)
```

Nodes and edges of `G` can be listed explicitly like so (both `G.nodes()` and `G.edges()` return iterator objects):

In [7]:
```
list(G.nodes())
```

Out[7]:
```
[1]
```

In [8]:
```
 list(G.edges())
```

Out[8]:
```
[]
```

You can also add a list of nodes:

In [9]:
```
G.add_nodes_from([2, 3])
```

In [10]:
```
list(G.nodes()), list(G.edges())
```

Out[10]:
```
([1, 2, 3], [])
```

With the same command you can in fact add any `nbunch` of nodes. An `nbunch` is any iterable container of nodes (e.g. a list as above, a set, another graph, a file, etc...). Let's use a generator to make a graph `H`, and then add its nodes to the graph `G`.

In [11]:
```
H = nx.path_graph(10)
H.number_of_nodes(), H.number_of_edges()
```

Out[11]:
```
(10, 9)
```

In [12]:
```
list(H.nodes()), list(H.edges())
```

Out[12]:
```
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9)])
```

In [13]:
```
for node in H:
    print(node)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [14]:
```
G.add_nodes_from(H)
```

In [15]:
```
list(G.nodes()), list(G.edges())
```

Out[15]:
```
([1, 2, 3, 0, 4, 5, 6, 7, 8, 9], [])
```

Note that `G` now contains the nodes of `H` as its own nodes of `G`. However, `G` does not contain the edges of `H`. The graph `H` could also be added as a new node to `G`.

In [16]:
```
G.add_node(H)
```

In [17]:
```
list(G.nodes()), list(G.edges())
```

Out[17]:
```
([1,
  2,
  3,
  0,
  4,
  5,
  6,
  7,
  8,
  9,
  <networkx.classes.graph.Graph at 0x7fde825c1668>],
 [])
```

The graph `G` now contains `H` as a node. This flexibility is very powerful as it allows graphs of graphs, graphs of files, graphs of functions and much more.

Edges

Let's start again with a clean sheet.

In [18]:
```
G.clear()
list(G.nodes()), list(G.edges())
```

Out[18]:
```
([], [])
```

`G` can also be grown by adding one edge at a time,

In [19]:
```
G.add_edge(1, 2)
e = (2, 3)
G.add_edge(*e) # unpack edge tuple*
```

In [20]:
```
list(G.nodes()), list(G.edges())
```

Out[20]:
```
([1, 2, 3], [(1, 2), (2, 3)])
```

In [21]:
```
list(G.neighbors(2))  # G.neighbors(n) returns an iterator of neigboring nod
es of n
```

Out[21]:
```
[1, 3]
```

by adding a list of edges,

In [22]:
```
G.add_edges_from([(1, 2),(1, 3)])
```

In [23]:
```
list(G.nodes()), list(G.edges())
```

Out[23]:
```
([1, 2, 3], [(1, 2), (1, 3), (2, 3)])
```

or by adding any `ebunch` of edges. An `ebunch` is any iterable container of edge-tuples. An edge-tuple can be a 2-tuple of nodes or a 3-tuple with 2 nodes followed by an edge attribute dictionary, e.g. `(2, 3, {'weight' : 3.1415})` . Edge attributes are discussed further later ...

In [24]:
```
G.add_edges_from(H.edges())
```

In [25]:
```
list(G.nodes()), list(G.edges())
```

Out[25]:
```
([1, 2, 3, 0, 4, 5, 6, 7, 8, 9],
 [(1, 2),
  (1, 3),
  (1, 0),
  (2, 3),
  (3, 4),
  (4, 5),
  (5, 6),
  (6, 7),
  (7, 8),
  (8, 9)])
```

Note how nodes are automatically added if necessary.

One can demolish the graph in a similar fashion; using `Graph.remove_node` , `Graph.remove_nodes_from` , `Graph.remove_edge` and `Graph.remove_edges_from` , e.g.

In [26]:
```
G.remove_node(3)
```

In [27]:
```
list(G.nodes()), list(G.edges())
```

Out[27]:
```
([1, 2, 0, 4, 5, 6, 7, 8, 9],
 [(1, 2), (1, 0), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9)])
```

You can specify graph data in several formats.

In [28]:
```
edgelist = [(0, 1), (1, 2), (2, 3)]
```

In [29]:
```
G = nx.Graph(edgelist)
list(G.edges())
```

Out[29]:
```
[(0, 1), (1, 2), (2, 3)]
```

In [30]:
```
H = nx.DiGraph(G)  # create a DiGraph using the connections from G
```

In [31]:
```
list(H.edges())
```

Out[31]:
```
[(0, 1), (1, 0), (1, 2), (2, 1), (2, 3), (3, 2)]
```

Drawing graphs

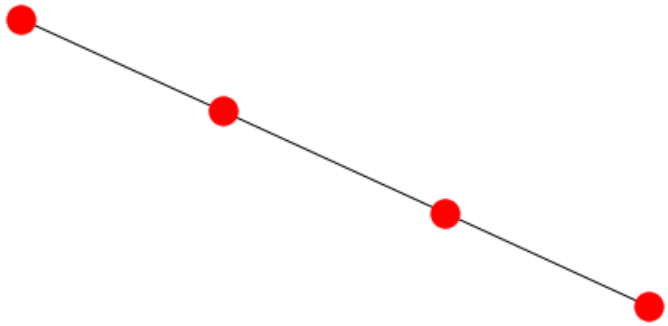
NetworkX is not primarily a graph drawing package but basic drawing with Matplotlib as well as an interface to use the open source Graphviz software package are included. These are part of the `networkx.drawing` package and will be imported if possible.

Matplotlib's plot interface is imported as follows.

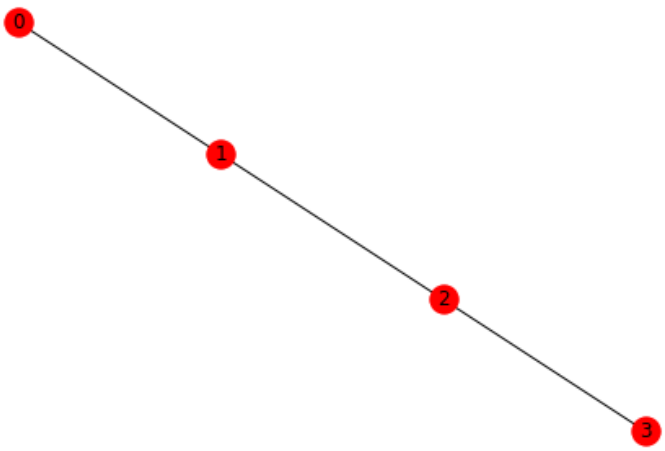
In [32]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

To test if the import of `networkx.drawing` was successful draw `G` using one of

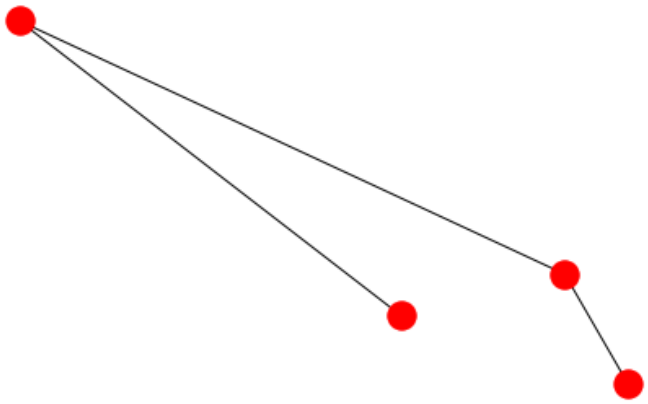
In [33]:
```
nx.draw(G)
```



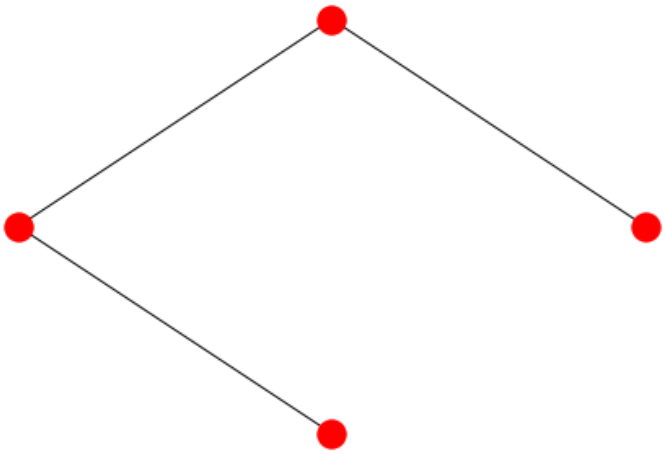
In [34]:
```
nx.draw(G, with_labels=True)
```



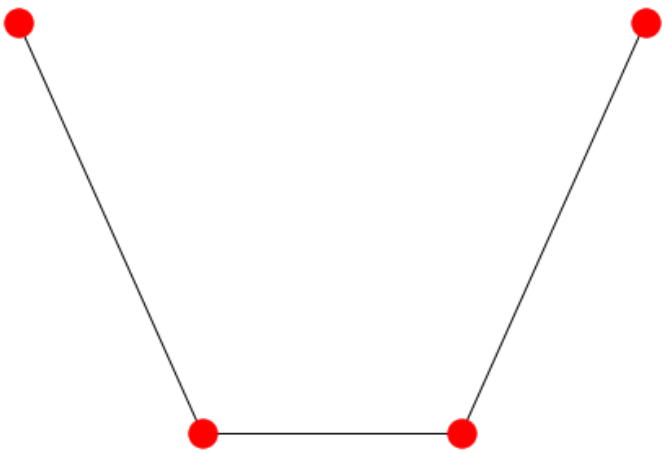
In [35]:
```
nx.draw_random(G)
```



In [36]:
```
nx.draw_circular(G)
```



In [37]:
```
nx.draw_spectral(G)
```



when drawing to an interactive display. Note that you may need to issue a Matplotlib `plt.show()` command if you are not using matplotlib in interactive mode.

What to use as nodes and edges

Nodes and edges are not specified as NetworkX objects. This leaves you free to use meaningful items as nodes and edges. The most common choices are numbers or strings, but a node can be any hashable object (except `None`), and an edge can be associated with any object x using `G.add_edge(n1, n2, object=x)`.

For example, `n1` and `n2` could be protein objects from the RCSB Protein Data Bank, and `x` could refer to an XML record of publications detailing experimental observations of their interaction.

This power can be quite useful, but its abuse can lead to unexpected surprises unless one is familiar with Python. If in doubt, consider using `convert_node_labels_to_integers` to obtain a more traditional graph with integer labels.

Accessing edges

We have seen that nodes and edges of a graph `G` can be accessed with the methods `G.nodes`, `G.edges`, and `G.neighbors`. These methods return iterators that can save you from creating large lists when you are just going to iterate through them anyway.

Fast direct access to the graph data structure is also possible using subscript notation.

Warning. Do not change the returned dict--it is part of the graph data structure and direct manipulation may leave the graph in an inconsistent state.

In [38]:
```
G[1]  # Warning: do not change the resulting dict
```

Out[38]:
```
AtlasView({0: {}, 2: {}})
```

In [39]:
```
list(G.neighbors(1))
```

Out[39]:
```
[0, 2]
```

In [40]:
```
G.add_edge(1, 3, color='blue')
```

In [41]:
```
G[1][3]
```

Out[41]:
```
{'color': 'blue'}
```

You can safely set the attributes of an edge using subscript notation if the edge already exists.

In [42]:
```
list(G.adjacency())
```

Out[42]:
```
[(0, {1: {}}),
 (1, {0: {}, 2: {}, 3: {'color': 'blue'}}),
 (2, {1: {}, 3: {}}),
 (3, {2: {}, 1: {'color': 'blue'}})]
```

Fast examination of all edges is achieved using adjacency(iterators). Note that for undirected graphs this actually looks at each edge twice.

Adding attributes to graphs, nodes, and edges

Attributes such as weights, labels, colors, or whatever Python object you like, can be attached to graphs, nodes, or edges.

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but attributes can be added or changed using `add_edge`, `add_node` or direct manipulation of the attribute dictionaries named `G.graph`, `G.node` and `G.edge` for a graph G.

Graph attributes

Assign graph attributes when creating a new graph

```
In [43]: G = nx.Graph(day="Wednesday")
```

```
In [44]: G.graph
```

```
Out[44]: {'day': 'Wednesday'}
```

Or you can modify attributes later

```
In [45]: G.graph['day'] = 'Monday'
```

```
In [46]: G.graph
```

```
Out[46]: {'day': 'Monday'}
```

Node attributes

Add node attributes using `add_node(), add_nodes_from() or G.node`

```
In [47]: G.add_node(1, time='5pm')
```

```
In [48]: G.add_nodes_from([3], time='2pm')
```

```
In [49]: G.node[1]
```

```
Out[49]: {'time': '5pm'}
```

```
In [50]: G.node[1]['room'] = 714
```

```
In [51]: list(G.nodes(data=True))
```

```
Out[51]: [(1, {'time': '5pm', 'room': 714}), (3, {'time': '2pm'})]
```

Note that adding a node to `G.node` does not add it to the graph, use `G.add_node()` to add new nodes.

Edge attributes

Add edge attributes using `add_edge()`, `add_edges_from()` or subscript notation.

In [52]:
```
G.add_edge(1, 2, weight=4.7)
```

In [53]:
```
G.add_edges_from([(3, 4), (4, 5)], color='red')
```

In [54]:
```
G.add_edges_from([(1, 2, {'color': 'blue'}), (2, 3, {'weight': 8})])
```

In [55]:
```
G[1][2]['weight'] = 4.7
```

In [56]:
```
list(G.edges(data=True))
```

Out[56]:
```
[(1, 2, {'weight': 4.7, 'color': 'blue'}),
 (3, 4, {'color': 'red'}),
 (3, 2, {'weight': 8}),
 (4, 5, {'color': 'red'})]
```

The special attribute `'weight'` should be numeric and holds values used by algorithms requiring weighted edges.

Directed Graphs

The `DiGraph` class provides additional methods specific to directed edges, e.g. `DiGraph.out_edges`, `DiGraph.in_degree`, `DiGraph.predecessors`, `DiGraph.successors` etc. To allow algorithms to work with both classes easily, the directed versions of `neighbors()` and `degree()` are equivalent to `successors()` and the sum of `in_degree()` and `out_degree()` respectively even though that may feel inconsistent at times.

In [57]:
```
DG = nx.DiGraph()
```

In [58]:
```
DG.add_weighted_edges_from([(1, 2, 0.5), (3, 1, 0.75)])
```

In [59]:
```
DG.out_degree(1, weight='weight')
```

Out[59]:
```
0.5
```

In [60]:
```
DG.degree(1, weight='weight')
```

Out[60]:
```
1.25
```

In [61]:
```
list(DG.successors(1))   # DG.successors(n) returns an iterator
```

Out[61]:
```
[2]
```

In [62]:
```
list(DG.neighbors(1))   # DG.neighbors(n) returns an iterator
```

Out[62]:
```
[2]
```

Some algorithms work only for directed graphs and others are not well defined for directed graphs. Indeed the tendency to lump directed and undirected graphs together is dangerous. If you want to treat a directed graph as undirected for some measurement you should probably convert it using `Graph.to_undirected` or with

In [63]:
```
H = nx.Graph(DG) # convert DG to undirected graph
```

Graph generators and graph operations

In addition to constructing graphs node-by-node or edge-by-edge, they can also be generated by

- Applying classic graph operations, such as:

```
 subgraph(G, nbunch)      - induce subgraph of G on nodes in nbunch
union(G1,G2)              - graph union
disjoint_union(G1,G2)     - graph union assuming all nodes are different
cartesian_product(G1,G2) - return Cartesian product graph
compose(G1,G2)            - combine graphs identifying nodes common to both
complement(G)             - graph complement
create_empty_copy(G)      - return an empty copy of the same graph class
convert_to_undirected(G) - return an undirected representation of G
convert_to_directed(G)    - return a directed representation of G
```

- Using a call to one of the classic small graphs, e.g.

In [64]:
```
petersen = nx.petersen_graph()
```

In [65]:
```
tutte = nx.tutte_graph()
```

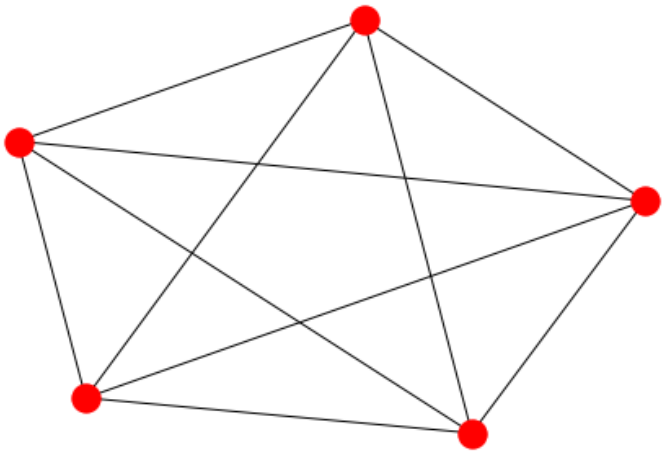
In [66]:
```
maze = nx.sedgewick_maze_graph()
```

In [67]:
```
tet = nx.tetrahedral_graph()
```

- Using a (constructive) generator for a classic graph, e.g.

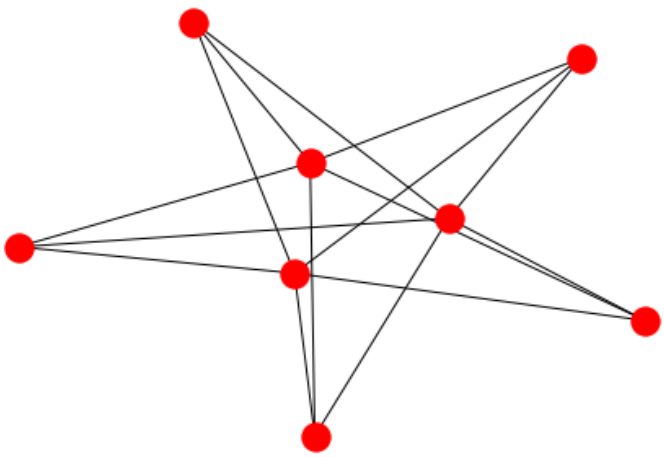
In [68]:
```
K_5 = nx.complete_graph(5)
```

In [69]:
```
nx.draw(K_5)
```



In [70]:
```
K_3_5 = nx.complete_bipartite_graph(3, 5)
```

In [71]:
```
nx.draw(K_3_5)
```



In [72]:
```
barbell = nx.barbell_graph(10, 10)
```

In [73]:
```
lollipop = nx.lollipop_graph(10, 20)
```

- Using a stochastic graph generator, e.g.

In [74]:
```
er = nx.erdos_renyi_graph(100, 0.15)
```

In [75]:
```
ws = nx.watts_strogatz_graph(30, 3, 0.1)
```

In [76]:
```
ba = nx.barabasi_albert_graph(100, 5)
```

In [77]:
```
red = nx.random_lobster(100, 0.9, 0.9)
```

- Reading a graph stored in a file using common graph formats, such as edge lists, adjacency lists, GML, GraphML, pickle, LEDA and others.