NetworkX

Outline

1. Introduction to NetworkX
2. Getting started with Python and NetworkX
3. Basic network analysis
4. Writing your own code
5. Ready for your own analysis!

1. Introduction to NetworkX

Introduction: why Python?

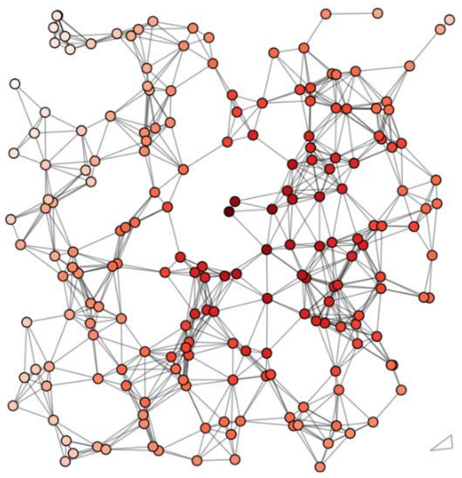
Python is an interpreted, general-purpose high-level programming language whose design philosophy emphasises code readability

Clear syntax
Multiple programming paradigms
Dynamic typing
Strong on-line community
Rich documentation
Numerous libraries
Expressive features
Fast prototyping

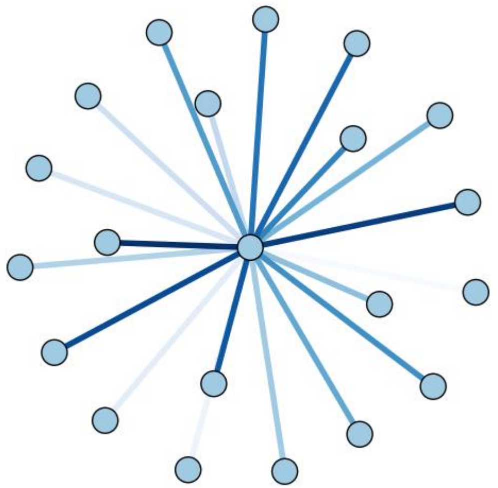
Can be slow
Beware when you are analysing very large networks

Introduction: NetworkX

A “high-productivity software for complex networks” analysis



- Data structures for representing various networks (directed, undirected, multigraphs)
- Extreme flexibility: nodes can be any hashable object in Python, edges can contain arbitrary data
- A treasure trove of graph algorithms
- Multi-platform and easy-to-use



Introduction: when to use NetworkX

When to use

Unlike many other tools, it is designed to handle data on a scale relevant to modern problems

Most of the core algorithms rely on extremely fast legacy code

Highly flexible graph implementations (a node/edge can be anything!)

When to avoid

Large-scale problems that require faster approaches (i.e. massive networks with 100M/1B edges)

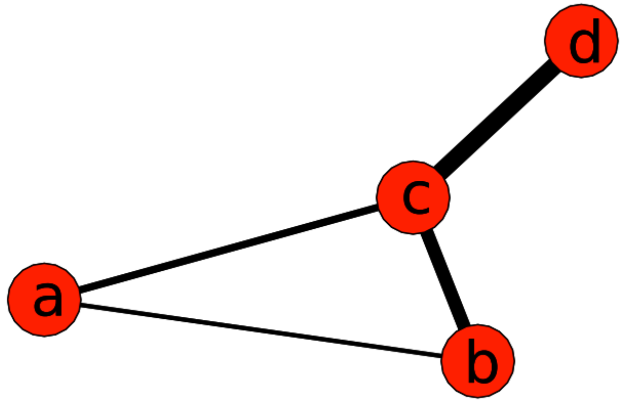
Better use of memory/threads than Python (large objects, parallel computation)

Visualization of networks is better handled by other professional tools

Introduction: a quick example

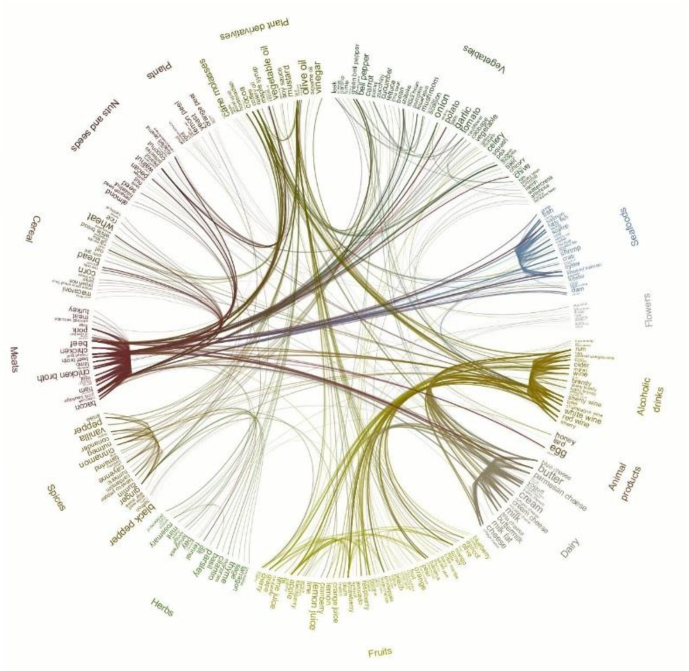
- Use Dijkstra's algorithm to find the shortest path in a weighted and unweighted network.

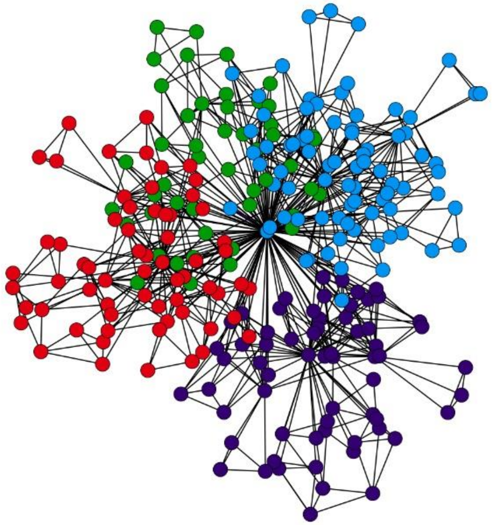
```
>>> import networkx as nx
>>> g = nx.Graph()
>>> g.add_edge('a', 'b', weight=0.1)
>>> g.add_edge('b', 'c', weight=1.5)
>>> g.add_edge('a', 'c', weight=1.0)
>>> g.add_edge('c', 'd', weight=2.2)
>>> print nx.shortest_path(g, 'b', 'd')
['b', 'c', 'd']
>>> print nx.shortest_path(g, 'b', 'd', weight='weight')
['b', 'a', 'c', 'd']
```

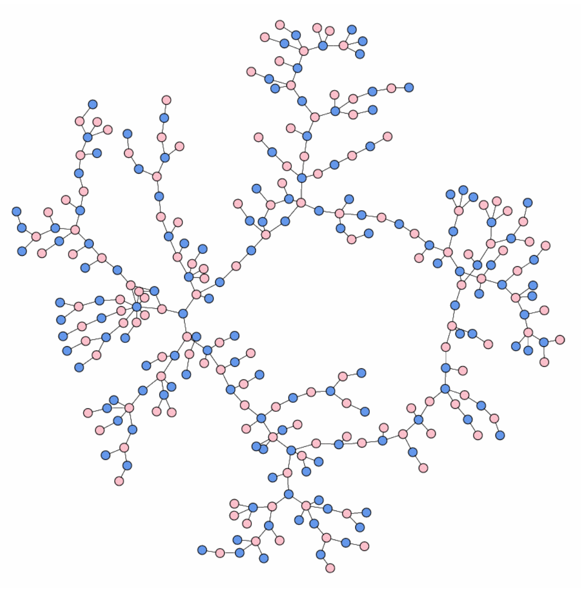


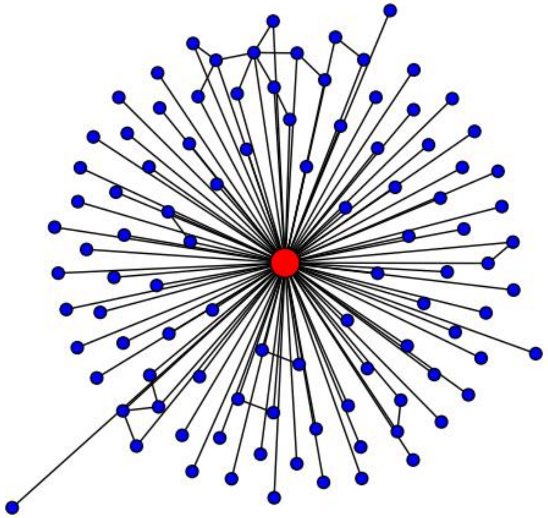
Introduction: drawing and plotting

- It is possible to draw small graphs with NetworkX. You can export network data and draw with other programs (GraphViz, Gephi, etc.).









2. Getting started with Python and NetworkX

Getting started: the environment

- Start Python (interactive or script mode) and import NetworkX

```
$ python
>>> import networkx as nx
```

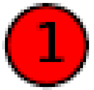
- Different classes exist for directed and undirected networks. Let’s create a basic undirected Graph:

```
>>> g = nx.Graph() # empty graph
```

- The graph **g** can be grown in several ways. NetworkX provides many generator functions and facilities to read and write graphs in many formats.

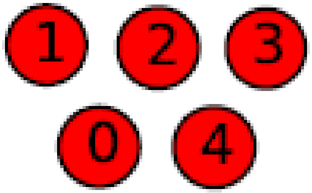
Getting started: adding nodes

```
# One node at a time
>>> g.add_node(1)
```

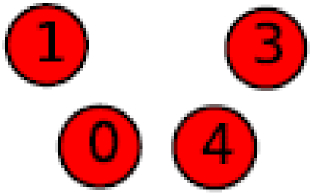



```
# A list of nodes
>>> g.add_nodes_from([2, 3])
```


```
# A container of nodes
>>> h = nx.path_graph(5)
>>> g.add_nodes_from(h)
```




```
# You can also remove any node of the graph
>>> g.remove_node(2)
```



Getting started: node objects

- A node can be **any hashable object** such as a string, a function, a file and more.

```
>>> import math
>>> g.add_node('string')
>>> g.add_node(math.cos) # cosine function
>>> f = open('temp.txt', 'w') # file handle
>>> g.add_node(f)
>>> print g.nodes()
['string', <open file 'temp.txt', mode 'w' at
0x000000000589C5D0>, <built-in function cos>]
```

Getting started: adding edges

```
# Single edge
>>> g.add_edge(1, 2)
>>> e = (2, 3)
>>> g.add_edge(*e) # unpack tuple
```

```
# List of edges
>>> g.add_edges_from([(1, 2), (1, 3)])
```

```
# A container of edges
>>> g.add_edges_from(h.edges())
```

```
# You can also remove any edge
>>> g.remove_edge(1, 2)
```

Getting started: accessing nodes and edges

```
>>> g.add_edges_from([(1, 2), (1, 3)])
>>> g.add_node('a')
>>> g.number_of_nodes() # also g.order()
4
>>> g.number_of_edges() # also g.size()
2
>>> g.nodes()
['a', 1, 2, 3]
>>> g.edges()
[(1, 2), (1, 3)]
>>> g.neighbors(1)
[2, 3]
>>> g.degree(1)
2
```

Getting started: Python dictionaries

- NetworkX takes advantage of Python dictionaries to store node and edge measures. The **dict** type is a data structure that represents a key-value mapping.

```
# Keys and values can be of any data type
>>> fruit_dict = {'apple': 1, 'orange': [0.12, 0.02], 42: True}

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
['orange', 42, 'apple']

# Or (key, value) tuples
>>> fruit_dict.items()
[('orange', [0.12, 0.02]), (42, True), ('apple', 1)]
# This becomes especially useful when you master Python list
comprehension
```

Getting started: graph attributes

- Any NetworkX graph behaves like a Python dictionary with nodes as primary keys

(**for access only!**)

```
>>> g.add_node(1, time='10am')
>>> g.node[1]['time']
10am
>>> g.node[1] # Python dictionary
{'time': '10am'}
```

- The special edge attribute **weight** should always be numeric and holds values used by algorithms requiring weighted edges.

```
>>> g.add_edge(1, 2, weight=4.0)
>>> g[1][2]['weight'] = 5.0 # edge already added
>>> g[1][2]
{'weight': 5.0}
```

Getting started: node and edge iterators

- Node iteration

```
>>> g.add_edge(1, 2)
>>> for node in g.nodes(): # or node in g.nodes_iter():
        print node, g.degree(node)
1 1
2 1
```

- Edge iteration

```
>>> g.add_edge(1, 3, weight=2.5)
>>> g.add_edge(1, 2, weight=1.5)
>>> for n1, n2, attr in g.edges(data=True): # unpacking
        print n1, n2, attr['weight']
1 2 1.5
1 3 2.5
```

Getting started: directed graphs

```
>>> dg = nx.DiGraph()
>>> dg.add_weighted_edges_from([(1, 4, 0.5), (3, 1, 0.75)])
>>> dg.out_degree(1, weight='weight')
0.5
>>> dg.degree(1, weight='weight')
1.25
>>> dg.successors(1)
[4]
>>> dg.predecessors(1)
[3]
```

- Some algorithms work only for undirected graphs and others are not well defined for directed graphs. If you want to treat a directed graph as undirected for some measurement you should probably convert it using **Graph.to_undirected()**

Getting started: graph operators

- **subgraph(G, nbunch)** - induce subgraph of G on nodes in nbunch
- **union(G1, G2)** - graph union, G1 and G2 must be disjoint
- **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

Getting started: graph generators

```
# small famous graphs
>>> petersen = nx.petersen_graph()
>>> tutte = nx.tutte_graph()
>>> maze = nx.sedgewick_maze_graph()
>>> tet = nx.tetrahedral_graph()
```

```
# classic graphs
>>> K_5 = nx.complete_graph(5)
>>> K_3_5 = nx.complete_bipartite_graph(3, 5)
>>> barbell = nx.barbell_graph(10, 10)
>>> lollipop = nx.lollipop_graph(10, 20)
```

```
# random graphs
>>> er = nx.erdos_renyi_graph(100, 0.15)
>>> ws = nx.watts_strogatz_graph(30, 3, 0.1)
>>> ba = nx.barabasi_albert_graph(100, 5)
>>> red = nx.random_lobster(100, 0.9, 0.9)
```

Getting started: graph input/output

- General read/write

```
>>> g = nx.read_<format>('path/to/file.txt',...options...)
>>> nx.write_<format>(g,'path/to/file.txt',...options...)
```

- Read and write edge lists

```
>>> g = nx.read_edgelist(path, comments='#', create_using=None,
delimiter=' ', nodetype=None, data=True, edgetype=None,
encoding='utf-8')
>>> nx.write_edgelist(g, path, comments='#', delimiter=' ',
data=True, encoding='utf-8')
```

- Data formats
  - Node pairs with no data: 1 2
  - Python dictionaries as data: 1 2 {'weight':7, 'color':'green'}
  - Arbitrary data: 1 2 7 green

Getting started: drawing graphs

- NetworkX is not primarily a graph drawing package but it provides basic drawing capabilities by using **matplotlib**. For more complex visualization techniques it provides an interface to use the open source **GraphViz** software package.

```
>>> import pylab as plt #import Matplotlib plotting interface
>>> g = nx.watts_strogatz_graph(100, 8, 0.1)
>>> nx.draw(g)
>>> nx.draw_random(g)
>>> nx.draw_circular(g)
>>> nx.draw_spectral(g)
>>> plt.savefig('graph.png')
```