Introduction to NetworkX

Franck Kalala M

AIMS SENEGAL
University of Lubumbashi

franckm@aims.ac.za | *franckm@aims-senegal.org*

April 10, 2018

Softwares

1. Networkx (Pyhton) (https://networkx.github.io/)
   1. Cross-platform python library
   2. Examples/introduction in this presentation
2. NodeXL (http://nodexl.codeplex.com/):
   1. Add-in for Microsoft Excel (Windows only)
   2. Good data collection options (Twitter, Facebook, YouTube, . . .)
   3. Basic visualization
3. igraph (Python, C, R) (http://igraph.org/redirect.html)
   1. install.packages(igraph) in R (r-project.org) (or python)
   2. Cross-platform
   3. Text driven: powerful for analysis
4. pajet http://vlado.fmf.uni-lj.si/pub/networks/pajek/
   1. http://pajek.imfm.si/doku.php?id=download
   2. Standalone, Windows (or Linux with wine)
   3. Good interactive environment for metrics and basic visualization
5. Gephi https://gephi.org/

NetworkX

1. http://networkx.github.io/
2. Package to represent networks as python objects
3. Convenient functions to add, delete, iterate nodes/edges
4. Functions to calculate network statistics (degree, clustering, etc.)
5. Easily generate comparison graphs based on statistical models
6. Visualization
7. Alternatives include igraph (available for Python and R)

Networkx

NetworkX

NetworkX Home | Documentation | Download | Developer (Github)

High-productivity software for complex networks

NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

Documentation
all documentation

Examples
using the library

Reference
all functions and methods

Features

- Python language data structures for graphs, digraphs, and multigraphs.
- Many standard graph algorithms
- Network structure and analysis measures
- Generators for classic graphs, random graphs, and synthetic networks
- Nodes can be "anything" (e.g. text, images, XML records)
- Edges can hold arbitrary data (e.g. weights, time-series)
- Open source BSD license
- Well tested: more than 1800 unit tests, >90% code coverage
- Additional benefits from Python: fast prototyping, easy to teach, multi-platform

Versions

Latest Release

networkx-1.11
30 January 2016
downloads | docs | pdf

Development

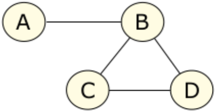
2.0dev
github | docs | pdf
build passing
coverage 94%

Contact

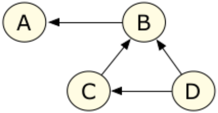
Mailing list
Issue tracker

Basic Concepts

- Graph: a way of representing the relationships among a collection of objects.
- Consists of a set of objects, called nodes, with certain pairs of these objects connected by links called edges.



Undirected Graph



Directed Graph

- Two nodes are neighbours if they are connected by an edge.
- Degree of a node is the number of edges ending at that node.
- For a directed graph, the in-degree and out-degree of a node refer to numbers of edges incoming to or outgoing from the node.

Networkx: Creating Graphs

```
>>> import networkx
```

Import library

```
>>> g = networkx.Graph()
```

Create new undirected graph

```
>>> g.add_node("John")
>>> g.add_node("Maria")
>>> g.add_node("Alex")
>>> g.add_edge("John", "Alex")
>>> g.add_edge("Maria", "Alex")
```

Add new nodes with unique IDs.

Add new edges referencing associated node IDs.

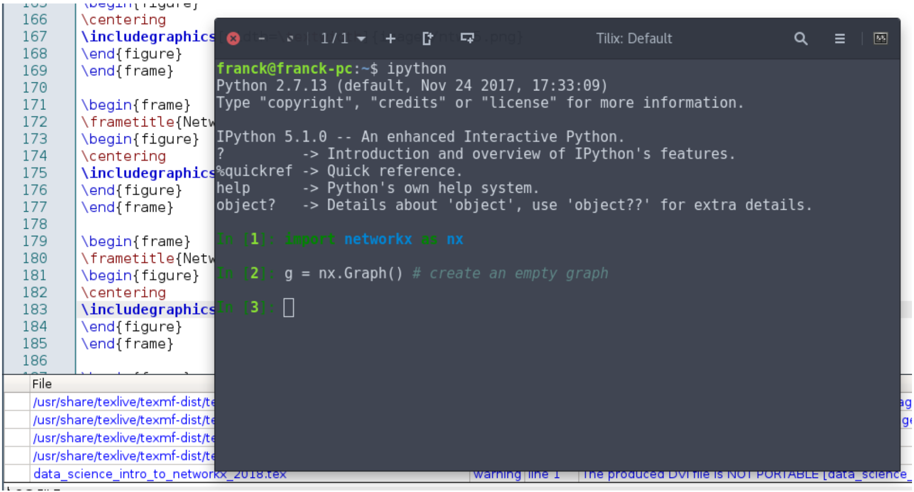
```
>>> print g.number_of_nodes()
3
>>> print g.number_of_edges()
2
>>> print g.nodes()
['John', 'Alex', 'Maria']
>>> print g.edges()
[('John', 'Alex'), ('Alex', 'Maria')]
```

Print details of our newly-created graph.

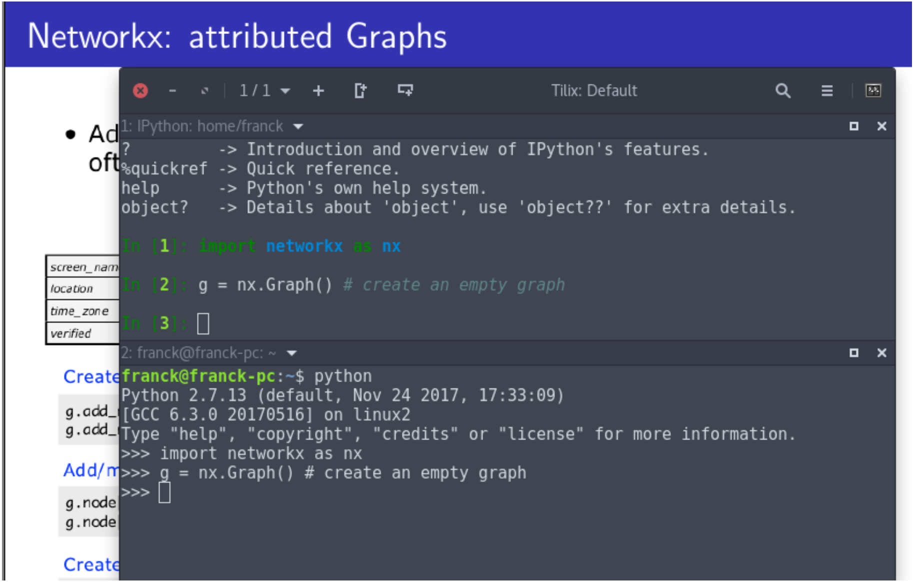
```
>>> print g.degree("John")
1
>>> print g.degree()
{'John': 1, 'Alex': 2, 'Maria': 1}
```

Calculate degree of specific node, or map of degree for all nodes.

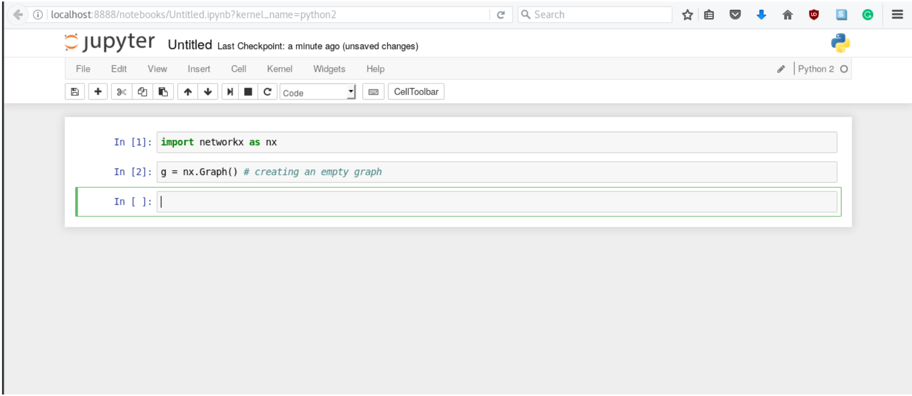
Networkx in Ipython



Networkx in python



Networkx in Jupyter Notebook



Networkx in Jupyter Notebook

I would like you get use to ipython https://ipython.org/ of python https://www.python.org/ command line before you move to jupyter notebook https://jupyter.org/.
I you impatient you can learn networkx within jupyter notebook here https://networkx.github.io/documentation/stable/tutorial.html

Networkx: Directed Graphs

```
>>> g = networkx.DiGraph()
```

Create new directed graph

```
>>> g.add_edges_from([("A","B"), ("C","A")])
```

Edges can be added in batches.

```
>>> print g.in_degree(with_labels=True)
{'A': 1, 'C': 0, 'B': 1}
>>> print g.out_degree(with_labels=True)
{'A': 1, 'C': 1, 'B': 0}
```

Nodes can be added to the graph "on the fly".

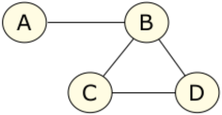
```
>>> print g.neighbors("A")
['B']
>>> print g.neighbors("B")
[]
```

```
>>> ug = g.to_undirected()
>>> print ug.neighbors("B")
['A']
```

Convert to an undirected graph

Networkx: Loading existing Graphs

- Library includes support for reading/writing graphs in a variety of file formats.



Edge List Format

```
a b
b c
b d
c d
```

Node pairs, one edge per line.

```
>>> g = networkx.read_edgelist("test.edges")
>>> print g.edges()
[('a', 'b'), ('c', 'b'), ('c', 'd'), ('b', 'd')]
```

Adjacency List Format

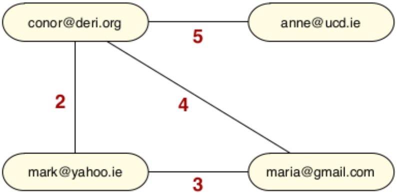
```
a b
b c d
c d
```

First label in line is the source node. Further labels in the line are considered target nodes.

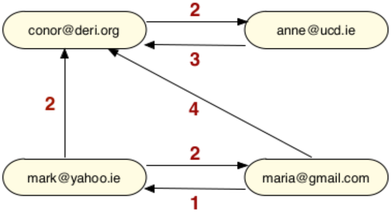
```
>>> g = networkx.read_adjlist("test_adj.txt")
>>> print edges()
[('a', 'b'), ('c', 'b'), ('c', 'd'), ('b', 'd')]
```

Networkx: Weigthed Graphs

- Weighted graph: numeric value is associated with each edge.
- Edge weights may represent a concept such as similarity, distance, or connection cost.



Undirected weighted graph



Directed weighted graph

Networkx: Weigthed Graphs

```
g = networkx.Graph()
```

```
g.add_edge("conor@deri.org", "anne@ucd.ie", weight=5)
g.add_edge("conor@deri.org", "mark@yahoo.ie", weight=2)
g.add_edge("conor@deri.org", "maria@gmail.com", weight=4)
g.add_edge("mark@yahoo.ie", "maria@gmail.com", weight=3)
```

Add weighted edges to graph.

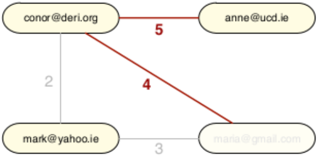
Note: nodes can be added to the graph "on the fly"

Select the subset of "strongly weighted" edges above a threshold...

```
estrong = [(u,v) for (u,v,d) in g.edges(data=True) if d["weight"] > 3]
```

```
>>> print estrong
[('conor@deri.org', 'anne@ucd.ie'), ('conor@deri.org', 'maria@gmail.com')]
```

```
>>> print g.degree("conor@deri.org", weighted=False)
3
>>> print g.degree("conor@deri.org", weighted=True)
11
```



Weighted degree given by sum of edge weights.

Networkx: attributed Graphs

- Additional attribute data, relating to nodes and/or edges, is often available to compliment network data.

follow_date	2011-07-07

screen_name	peter78
location	Galway
time_zone	GMT
verified	FALSE

screen_name	mark763
location	London
time_zone	GMT
verified	FALSE

Create new nodes with attribute values

```
g.add_node("318064061", screen_name="peter78", location="Galway", time_zone="GMT")
g.add_node("317756843", screen_name="mark763", location="London", time_zone="GMT")
```

Add/modify attribute values for existing nodes

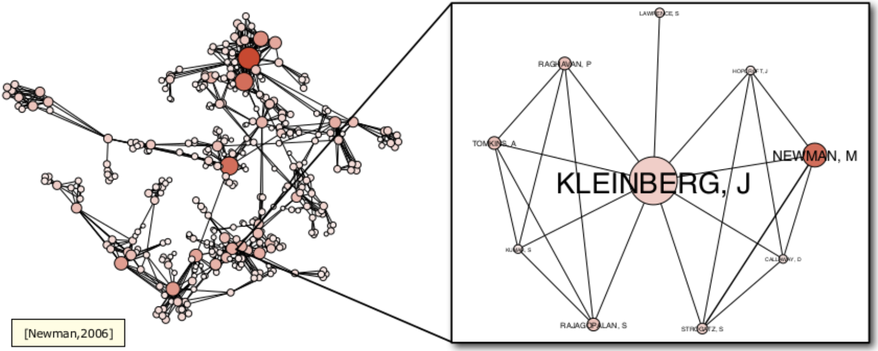
```
g.node["318064061"]["verified"] = False
g.node["317756843"]["verified"] = False
```

Create new edge with attribute values

```
g.add_edge("318064061", "317756843", follow_date=datetime.datetime.now())
```

Networkx: Ego networks

- Ego-centric methods really focus on the individual, rather than on network as a whole.
- By collecting information on the connections among the modes connected to a focal ego, we can build a picture of the **local** networks of the individual.



Networkx: Ego networks

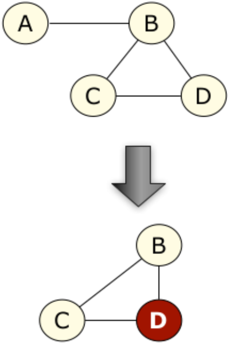
- We can readily construct an ego network subgraph from a global graph in NetworkX.

```
>>> g = networkx.read_adjlist("test.adj")
```

```
>>> ego = "d"
```

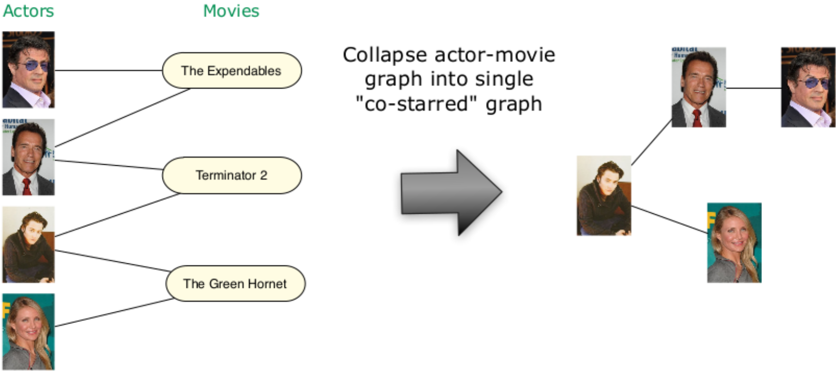
```
>>> nodes = set([ego])
>>> nodes.update(g.neighbors(ego))
>>> egonet = g.subgraph(nodes)
```

```
>>> print egonet.nodes()
['c', 'b', 'd']
>>> print egonet.edges()
[('c', 'b'), ('c', 'd'), ('b', 'd')]
```



Networkx: Bipartite Graphs

- In a bipartite graph the nodes can be divided into two disjoint sets so that no pair of nodes in the same set share an edge.



Networkx: Bipartite Graphs

- NetworkX does not have a custom bipartite graph class.

➡ A standard graph can be used to represent a bipartite graph.

```
import networkx
from networkx.algorithms import bipartite
```

Import package for handling bipartite graphs

```
g = networkx.Graph()
```

Create standard graph, and add edges.

```
g.add_edges_from([("Stallone","Expendables"), ("Schwarzenegger","Expendables")])
g.add_edges_from([("Schwarzenegger","Terminator 2"), ("Furlong","Terminator 2")])
g.add_edges_from([("Furlong","Green Hornet"), ("Diaz","Green Hornet")])
```

```
>>> print bipartite.is_bipartite(g)
True
>>> print bipartite.bipartite_sets(g)
(set(['Stallone', 'Diaz', 'Schwarzenegger', 'Furlong']),
set(['Terminator 2', 'Green Hornet', 'Expendables']))
```

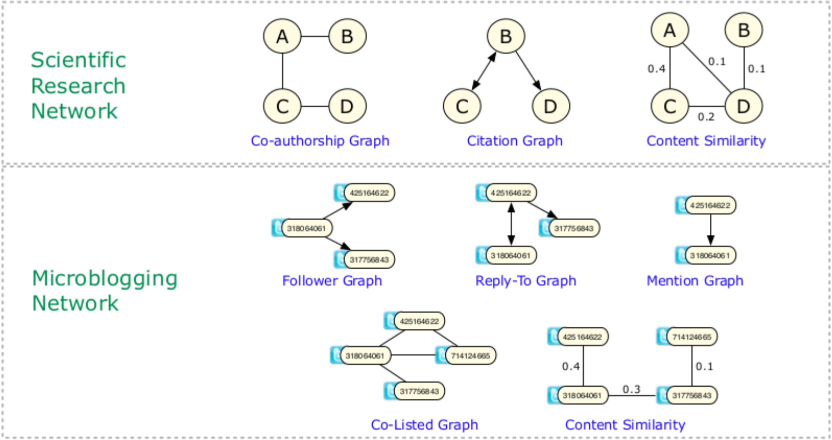
Verify our graph is bipartite, with two disjoint node sets.

```
>>> g.add_edge("Schwarzenegger","Stallone")
>>> print bipartite.is_bipartite(g)
False
```

Graph is no longer bipartite!

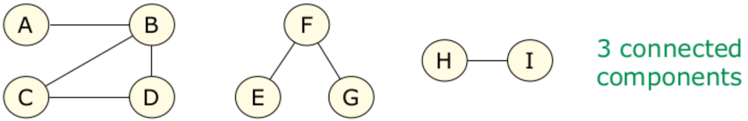
Networkx: Multi-relational networks

- In many SNA applications there will be multiple kinds of relations between nodes. Nodes may be closely-linked in one relational network, but distant in another.



Networkx: Graph Connectivity - Components

- A graph is connected if there is a path between every pair of nodes in the graph.
- A connected component is a subset of the nodes where:
  1. A path exists between every pair in the subset.
  2. The subset is not part of a larger set with the above property.

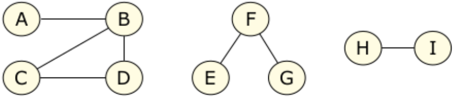


- In many empirical social networks a larger proportion of all nodes will belong to a single giant component.

Networkx: Graph Connectivity - Components

```
g = networkx.Graph()
g.add_edges_from([("a","b"),("b","c"),("b","d"),("c","d")])
g.add_edges_from([("e","f"),("f","g"),("h","i")])
```

Build undirected graph.




```
>>> print networkx.is_connected(g)
False
>>> print networkx.number_connected_components(g)
3
```

Is the graph just a single component?

If not, how many components are there?

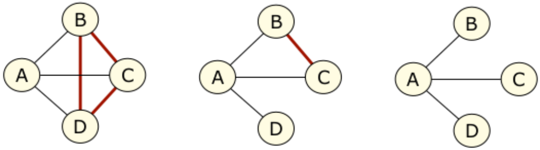
```
>>> comps = networkx.connected_component_subgraphs(g)
>>> print comps[0].nodes()
['a', 'c', 'b', 'd']
>>> print comps[1].nodes()
['e', 'g', 'f']
>>> print comps[2].nodes()
['i', 'h']
```

Find list of all connected components.

Each component is a subgraph with its own set of nodes and edges.

Networkx: Clustering Coefficients

- The neighbourhood of a node is set of nodes connected to it by an edge, not including itself.
- The clustering coefficient of a node is the fraction of pairs of its neighbours that have edges between one another.

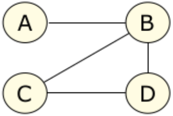


Node A: $CC = \frac{3}{3}$ $CC = \frac{1}{3}$ $CC = \frac{0}{3}$

- Locally indicates how concentrated the neighbourhood of a node is, globally indicates level of clustering in a graph.
- Global score is average over all nodes: $\bar{CC} = \frac{1}{n}\sum_{i=1}^{n} CC(v_i)$

Networkx: Clustering Coefficients

```
g = networkx.Graph()
g.add_edges_from([("a","b"),("b","c"),("b","d"),("c","d")])
```




```
>>> print networkx.neighbors(g, "b")
['a', 'c', 'd']
```

Get list of neighbours for a specific node.

```
>>> print networkx.clustering(g, "b")
0.333333333333
```

Calculate coefficient for specific node.

```
>>> print networkx.clustering(g, with_labels=True)
{'a': 0.0, 'c': 1.0, 'b': 0.33333333333333331, 'd': 1.0}
```

Build a map of coefficients for all nodes.

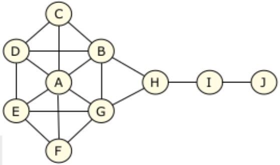
```
>>> ccs = networkx.clustering(g)
>>> print ccs
[0.0, 1.0, 0.33333333333333331, 1.0]
>>> print sum(ccs)/len(ccs)
0.583333333333
```

Calculate global clustering coefficient.

Networkx: Measure of Centrality

```
import networkx
from operator import itemgetter
```

```
g = networkx.read_adjlist("centrality.edges")
```




```
dc = networkx.degree_centrality(g)
print sorted(dc.items(), key=itemgetter(1), reverse=True)
```

```
[('a', 0.66666666666666663), ('b', 0.55555555555555558), ('g', 0.55555555555555558), ('c', 0.33333333333333331),
('e', 0.33333333333333331), ('d', 0.33333333333333331), ('f', 0.33333333333333331), ('h', 0.33333333333333331),
('i', 0.22222222222222221), ('j', 0.1111111111111111)]
```

```
bc = networkx.betweenness_centrality(g)
print sorted(bc.items(), key=itemgetter(1), reverse=True)
```

```
[('h', 0.38888888888888884), ('b', 0.2361111111111111), ('g', 0.2361111111111111), ('i', 0.22222222222222221),
('a', 0.16666666666666666), ('c', 0.0), ('e', 0.0), ('d', 0.0), ('f', 0.0), ('j', 0.0)]
```

```
bc = networkx.eigenvector_centrality(g)
print sorted(bc.items(), key=itemgetter(1), reverse=True)
```

```
[('a', 0.17589997921479006), ('b', 0.14995290497083508), ('g', 0.14995290497083508), ('c', 0.10520440827586457),
('e', 0.10520440827586457), ('d', 0.10520440827586457), ('f', 0.10520440827586457), ('h', 0.078145778134411939),
('i', 0.020280613919932109), ('j', 0.0049501856857375875)]
```

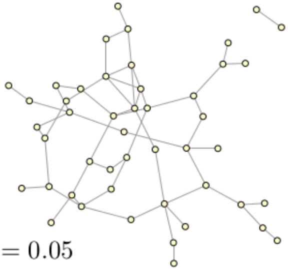
Networkx: Random Networks

- Erdős–Rényi random graph model:
  - Start with a collection of n disconnected nodes.
  - Create an edge between each pair of nodes with a probability p, independently of every other edge.

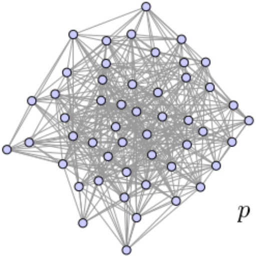
```
g1 = networkx.erdos_renyi_graph(50, 0.05)
```

```
g2 = networkx.erdos_renyi_graph(50, 0.3)
```

Specify number of nodes to create, and connection probability p.



$p = 0.05$



$p = 0.3$

Networkx: Samllworld networks

Milgram's Small World Experiment:

- Route a package to a stockbroker in Boston by sending them to random people in Nebraska and requesting them to forward to someone who might know the stockbroker.

➡ Although most nodes are not directly connected, each node can be reached from another via a relatively small number of hops.

Six Degrees of Kevin Bacon

- Examine the actor-actor "co-starred" graph from IMDB.
- The Bacon Number of an actor is the number of degrees of separation he/she has from Bacon, via the shortest path.

$\Rightarrow$ Bacon Number $= 2$

http://oracleofbacon.org

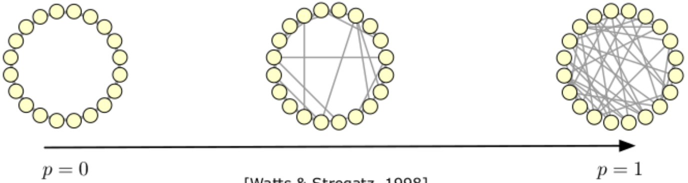
- Take a connected graph with a high diameter, randomly add a small number of edges, then the diameter tends to drop drastically.
- Small-world network has many local links and few long range “shortcuts”.

Typical properties:
- High clustering coefficient.
- Short average path length.
- Over-abundance of hub nodes.

[Watts & Strogatz, 1998]

Generating Small World Networks:

1. Create ring of n nodes, each connected to its k nearest neighbours.
2. With probability p, rewire each edge to an existing destination node.



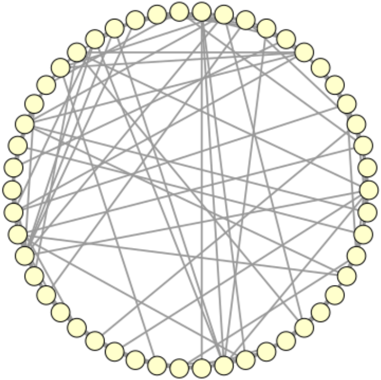
[Watts & Strogatz, 1998]

Networkx: Samllworld networks

- NetworkX includes functions to generate graphs according to a variety of well-known models:

```
n = 50
k = 6
p = 0.3
g = networkx.watts_strogatz_graph(n, k, p)
```

```
>>> networkx.average_shortest_path_length(g)
2.4506122448979597
```



Networkx: social network

```
import networkx as nx

g=nx.Graph() #A new (empty) undirected graph
g.add_node("Alan") #Add one new node
g.add_nodes_from(["Bob","Carol","Denise"])#Add three new nodes

#Nodes can have attributes
g.node["Alan"]["gender"]="M"
g.node["Bob"]["gender"]="M"
g.node["Carol"]["gender"]="F"
g.node["Denise"]["gender"]="F"

for n in g:
    print("{0} has gender {1}".format(n,g.node[n]["gender"]))
```

Networkx: Edges

```
#Interesting graphs have edges
g.add_edge("Alan","Bob") #Add one new edge
#Add two new edges
g.add_edges_from([["Carol","Denise"],["Carol","Bob"]])


#Edge attributes
g.edge["Alan"]["Bob"]["relationship"]="Friends"
g.edge["Carol"]["Denise"]["relationship"]="Friends"
g.edge["Carol"]["Bob"]["relationship"]="Married"

#New edge with an attribute
g.add_edges_from([["Carol","Alan",
    {"relationship":"Friends"}]])
```

Networkx: Edges

```
for e in g.edges_iter():
    n1=e[0]
    n2=e[1]
    print("{0} and {1} are {2}".format(n1,n2,
        g.edge[n1][n2]["relationship"]))
```

Networkx: Edges

IP[y]: Notebook python and data (unsaved changes)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

In [1]:
```
import networkx as nx
g=nx.Graph() #A new (empty) undirected graph
g.add_node("Alan") #Add one new node
g.add_nodes_from(["Bob","Carol","Denise"])
```

In [2]:
```
#Nodes can have attributes
g.node["Alan"]["gender"]="M"
g.node["Bob"]["gender"]="M"
g.node["Carol"]["gender"]="F"
g.node["Denise"]["gender"]="F"
```

In [3]:
```
for n in g:
    print("{0} has gender {1}".format(n,g.node[n]["gender"]))
```

```
Denise has gender F
Bob has gender M
Carol has gender F
Alan has gender M
```

In [4]:
```
#Interesting graphs have edges
g.add_edge("Alan","Bob") #Add one new edge
#Add two new edges
g.add_edges_from([["Carol","Denise"],["Carol","Bob"]])
```

In [5]:
```
g.edges()
```

Out[5]:
```
[('Denise', 'Carol'), ('Bob', 'Carol'), ('Bob', 'Alan')]
```

Networkx: Edges

```
In [3]: for n in g:
            print("{0} has gender {1}".format(n,g.node[n]["gender"]))

        Denise has gender F
        Bob has gender M
        Carol has gender F
        Alan has gender M

In [4]: #Interesting graphs have edges
        g.add_edge("Alan","Bob") #Add one new edge
        #Add two new edges
        g.add_edges_from([["Carol","Denise"],["Carol","Bob"]])

In [5]: g.edges()

Out[5]: [('Denise', 'Carol'), ('Bob', 'Carol'), ('Bob', 'Alan')]

In [6]: #Edge attributes
        g.edge["Alan"]["Bob"]["relationship"]="Friends"
        g.edge["Carol"]["Denise"]["relationship"]="Friends"
        g.edge["Carol"]["Bob"]["relationship"]="Married"

In [21]: g.edge["Alan"]["Bob"]["relationship"]

Out[21]: 'Friends'

In [19]: g.node["Bob"]

Out[19]: {'gender': 'M'}
```

Networkx: Measures

```
g.number_of_nodes()
g.nodes(data=True)

g.number_of_edges()
g.edges(data=True)

nx.info(g)
nx.density(g)
nx.number_connected_components(g)

nx.degree_histogram(g)
nx.betweenness_centrality(g)

nx.clustering(g)
nx.clustering(g, nodes=["Bob"])
```

Networkx: Vizualise and save

```
#Save g to the file my_graph.graphml in graphml format
#prettyprint will make it nice for a human to read
nx.write_graphml(g,"my_graph.graphml",prettyprint=True)

#Layout g with the Fruchterman-Reingold force-directed
#algorithm and save the result to my_graph.png
#with_labels will label each node with its id
import matplotlib.pyplot as plt

nx.draw_spring(g,with_labels=True)
plt.savefig("my_graph.png")
plt.clf() #Clear plot
```

Networkx: Vizualise and save

```
#Read a graph from the file my_graph.graphml in graphml format
g=nx.read_graphml("my_graph.graphml")

#Create a (empty) directed graph
g=nx.DiGraph()
```

See `http://networkx.github.io/documentation/latest/reference/index.html` for many more commands. Note that some commands are only available on directed or undirected graphs.

The End