NetworkX:
Network Analysis with Python

Ilias Dimitriadis – idimitriad@csd.auth.gr
DataLab – CSD

Outline

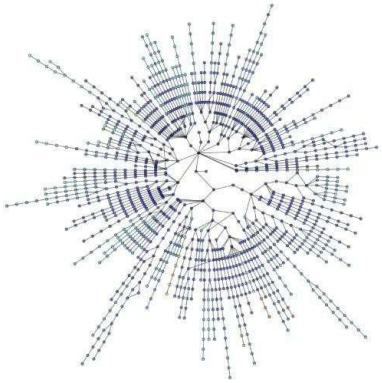
1. Introduction to NetworkX
2. Getting started with Python andNetworkX
3. Basic network analysis
4. Writing your own code
5. Ready for your own analysis!

1. Introduction to NetworkX

Introduction: networks are everywhere…

Social networks

Mobile phone networks
Web pages/citations
Internet routing



Vehicular flows

How can we analyse these networks?
Python + 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: Python's Holy Trinity

Python's primary library for **mathematical** and **statistical computing**.

Contains toolboxes for:

- Numeric optimization
- Signal processing
- Statistics, and more…

Primary data type is an **array**.

NumPy is an extension to include **multidimensional arrays** and **matrices**.

Both SciPy and NumPy rely on the Clibrary LAPACK for very fast implementation.

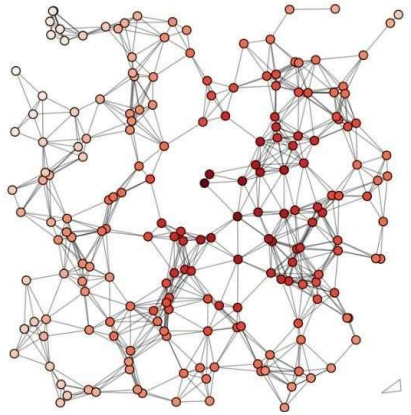
Matplotlib is the **primary plotting library** in Python.

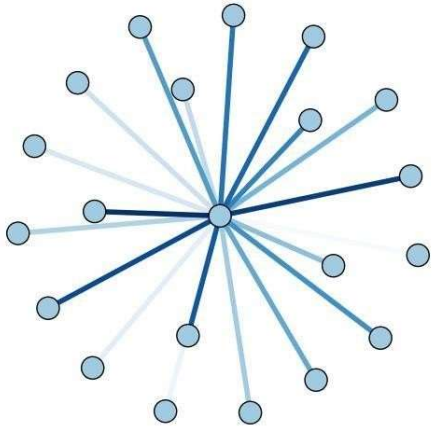
Supports 2-D and 3-D plotting. All plots are highly customisable and ready for professional publication.

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
- Numerous implementations of graph algorithms
- Multi-platform and easy-to-use





Introduction: when to useNetworkX

When touse

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

Most ofthe core algorithms relyon extremely fast legacycode

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

When toavoid

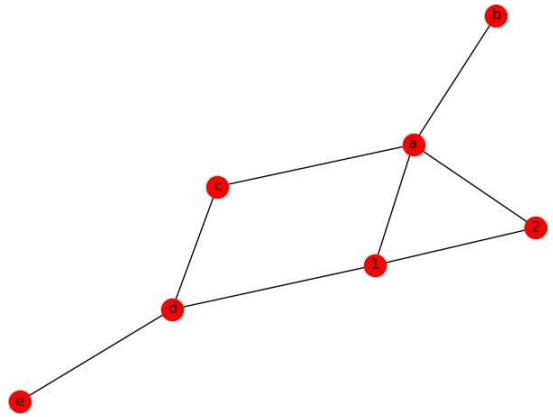
Large-scale problems that require faster approaches (i.e. massive networks with 100M/1B edges)

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

Visualization of networks is better handled by other professionaltools

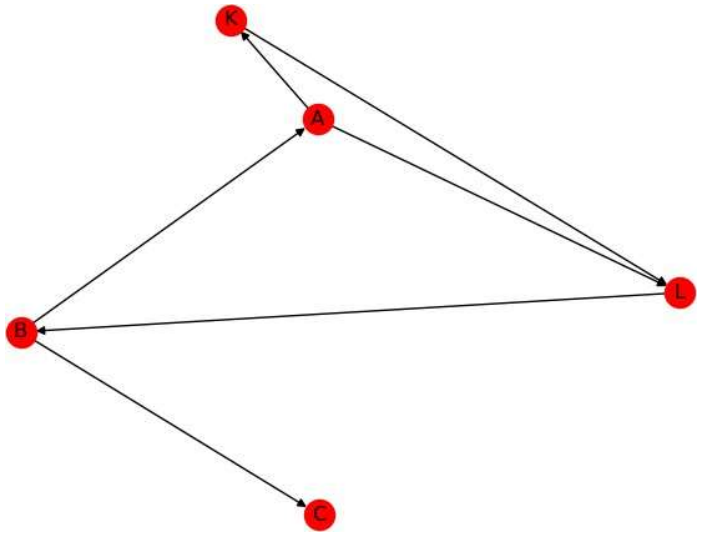
Introduction: a quick example

```
#·UNDIRECTED·NETWORK
def·create_undirected_graph():
····G·=·nx.Graph()
····G.add_node("a")
····G.add_nodes_from(["b",·"c"])
····G.add_edge(1,·2)
····edge·=·("d",·"e")
····G.add_edge(*edge)
····edge·=·("a",·"b")
····G.add_edge(*edge)
····print("Nodes·of·graph:·")
····print(G.nodes())
····print("Edges·of·graph:·")
····print(G.edges())
····#·adding·a·list·of·edges:
····G.add_edges_from([("a",·"c"),·("c",·"d"),·("a",·1),·(1,·"d"),·("a",·2)])
····print(G.edges())
····nx.draw(G,·with_labels=True,·pos=nx.spring_layout(G))
····plt.savefig("simple_undirected.png")··#·save·as·png
····plt.show()··#·display
····return·G
```



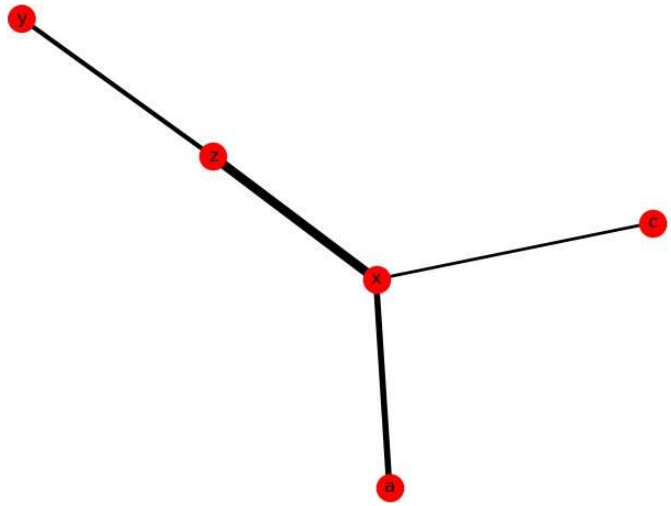
Introduction: a quick example

```
#DIRECTED GRAPH
def create_directed_graph():
    D=nx.DiGraph()
    D.add_edge('B','A')
    D.add_edge('B','C')
    D.add_edges_from([('L','B'),('A','L'),('A','K'),('K','L')])
    print("Nodes of graph: ")
    print(D.nodes())
    print("Edges of graph: ")
    print(D.edges())
    nx.draw(D, with_labels = True, pos=nx.spring_layout(D))
    plt.savefig("simple_directed.png") # save as png
    plt.show() # display
    return D
```



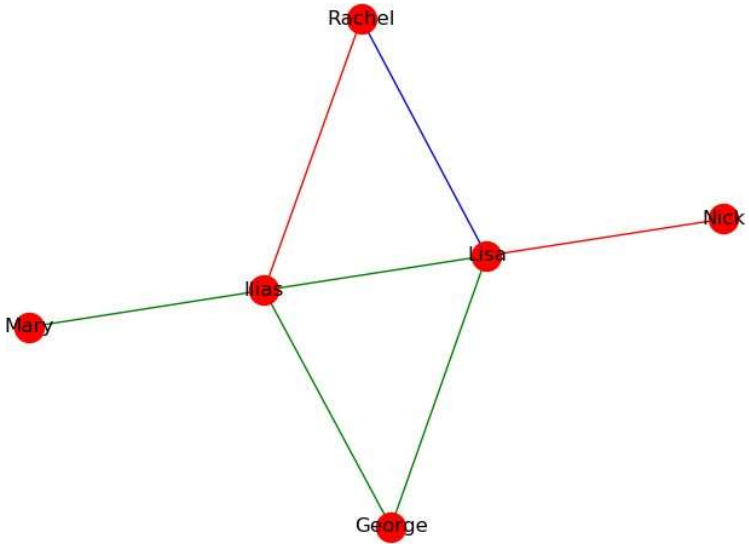
Introduction: a quick example

```
def created_weighted_graph():
    W=nx.Graph()
    W.add_edge('x','z',weight=6)
    W.add_edge('y','z',weight=3)
    W.add_edge('x','a',weight=4)
    W.add_edge('c','x',weight=2)
    print(W.edges())
    edgewidth = [ d['weight'] for (u,v,d) in W.edges(data=True)]
    print (edgewidth)
    nx.draw(W, with_labels = True, pos=nx.spring_layout(W), width=edgewidth)
    plt.savefig("simple_weighted.png") # save as png
    plt.show() # display
    return W
```



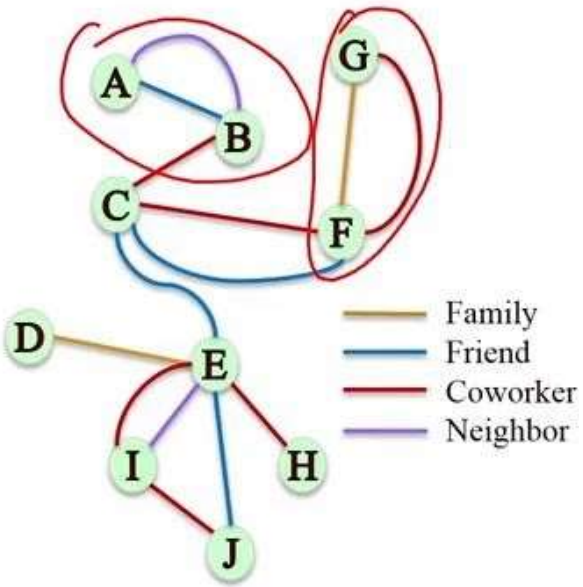
Introduction: a quick example

```python
#GRAPH WITH EDGE ATTRIBUTES
def create_graph_with_edge_attributes():
    R=nx.Graph()
    R.add_node('Ilias',role='male')
    R.add_node('George',role='male')
    R.add_node('Nick',role='male')
    R.add_node('Lisa',role='female')
    R.add_node('Rachel',role='female')
    R.add_node('Mary',role='female')
    R.add_edge('George','Lisa',relation='friend')
    R.add_edge('Ilias','Lisa',relation='friend')
    R.add_edge('Nick','Lisa',relation='couple')
    R.add_edge('Ilias','Rachel',relation='couple')
    R.add_edge('Lisa','Rachel',relation='family')
    R.add_edge('George','Ilias',relation='friend')
    R.add_edge('Ilias','Mary',relation='friend')
    colors = {'friend':'g','couple':'r','family':'b'}
    edgewidth = [ colors[d['relation']] for (u,v,d) in R.edges(data=True)]
    print (edgewidth)
    print (type(edgewidth))
    print(R.edges())
    nx.draw(R, with_labels = True, pos=nx.spring_layout(R), edge_color=edgewidth)
    plt.savefig("simple_attributed.png") # save as png
    plt.show() # display
    return R
```

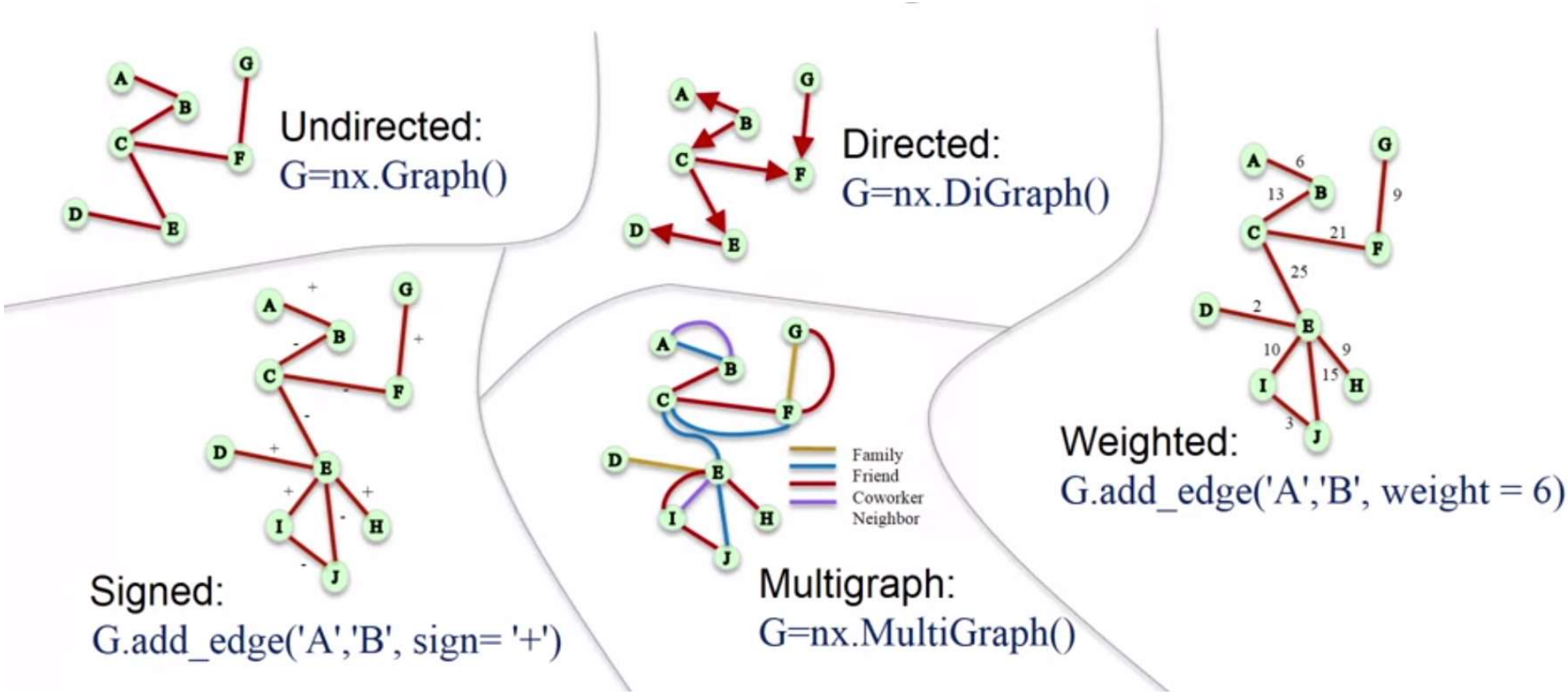


Introduction: a quick example

```
#MULTIGRAPH
def create_multigraph():
    M=nx.MultiGraph()
    M.add_edge('George','Lisa',relation='friend')
    M.add_edge('George','Lisa',relation='family')
    M.add_edge('Nick','Lisa',relation='couple')
    M.add_edge('Ilias','Rachel',relation='couple')
    M.add_edge('Nick','Rachel',relation='friend')
    M.add_edge('Lisa','Rachel',relation='family')
    M.add_edge('George','Ilias',relation='friend')
    M.add_edge('Ilias','Mary',relation='friend')
    M.add_edge('George','Mary',relation='family')
    M.add_edge('George','Mary',relation='friend')
    colors = {'friend':'g','couple':'r','family':'b'}
    edgewidth = [ colors[d['relation']] for (u,v,d) in M.edges(data=True)]
    print (edgewidth)
    print (type(edgewidth))
    print(M.edges())
    nx.draw(M, with_labels = True, pos=nx.spring_layout(M), edge_color=edgewidth)
    plt.savefig("multi_attributed.png") # save as png
    plt.show() # display
    return M
```

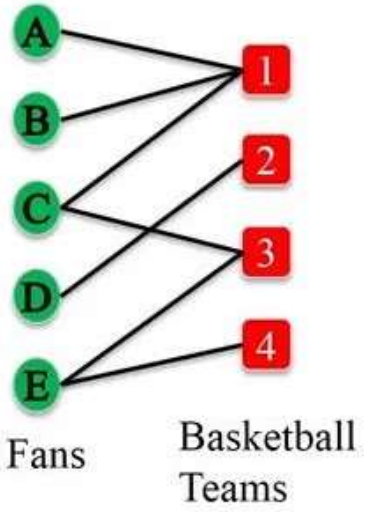


Introduction: a quick example

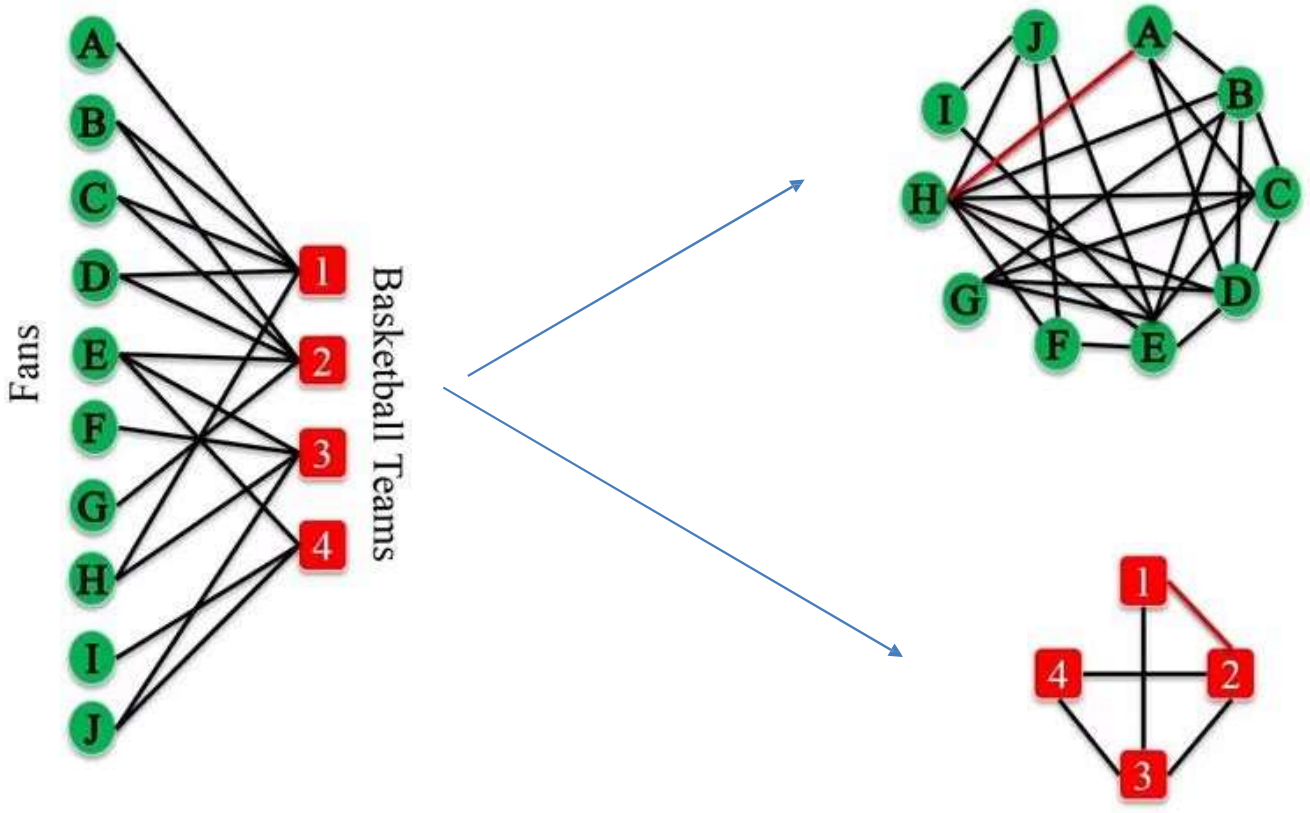


Introduction: a quick example

```
def create_bipartite_graph():
    B=nx.Graph()
    B.add_nodes_from(['A','B','C','D','E'], bipartite=0) #label one set
    B.add_nodes_from([1,2,3,4], bipartite=1) #label other set of nodes

    B.add_edges_from([('A',1),('B',1),('C',1),('D',2),('E',3),('E',4)])

    #check if a graph is bipartite!!!

    print (bipartite.is_bipartite(B))
    #let's change something
    B.add_edge('A','B')
    print (bipartite.is_bipartite(B))
    B.remove_edge('A','B')

    #check if a set of nodes is part of bipartition--
    X=set([1,2,3,4])
    print (bipartite.is_bipartite_node_set(B,X))
    return B
```



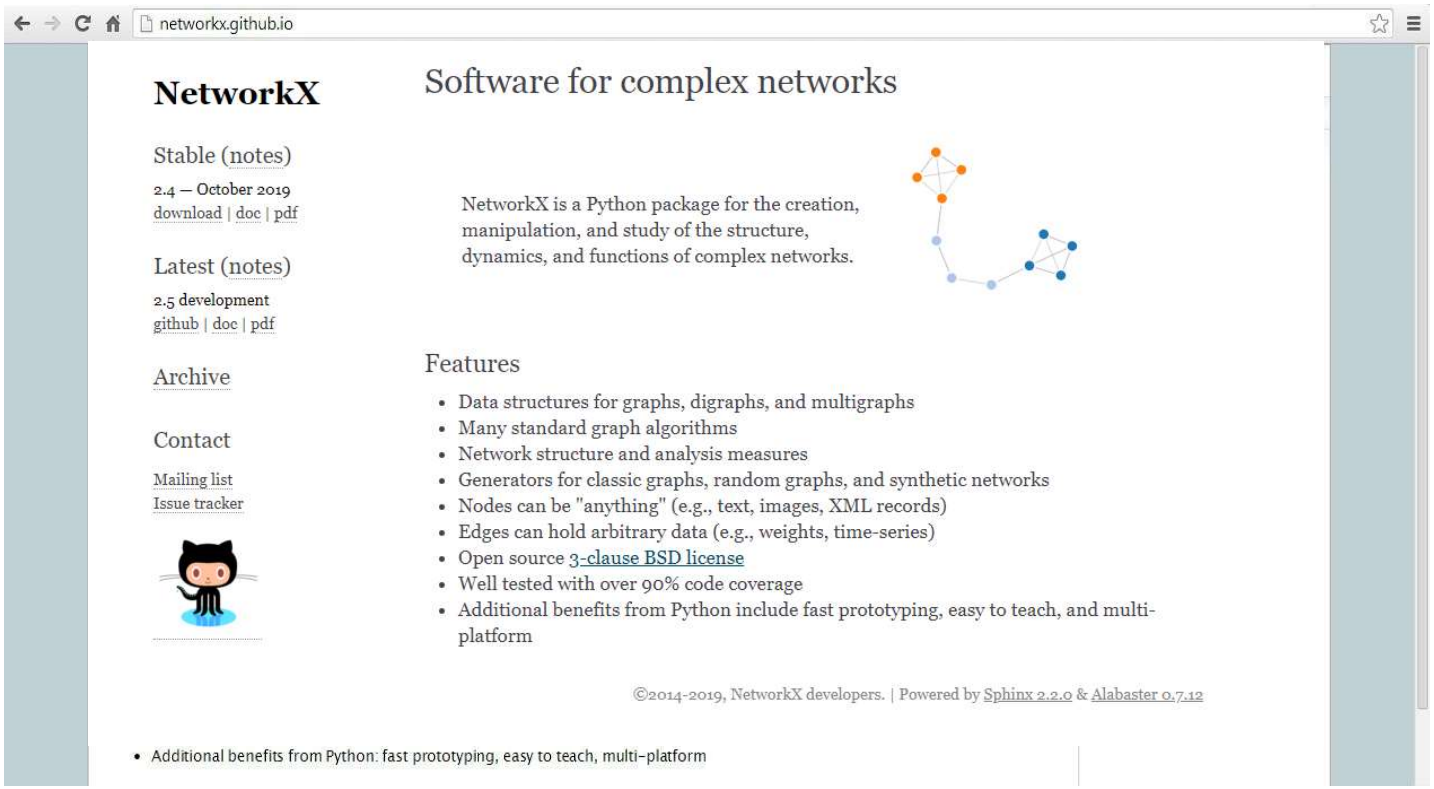
Introduction: a quick example




```
def create_projection(B):
    B.add_edges_from([('D',1),('H',1),('B',2),('
    X=set(['A','B','C','D','E','F','G','H','I','
    N=set([1,2,3,4])
    P=bipartite.projected_graph(B,X)
    print (nx.info(P))
    P2=bipartite.projected_graph(B,N)
    print (nx.info(P2))

    #if we want weights:
    P3=bipartite.weighted_projected_graph(B,N)
    print (P3.edges(data=True))
    return P,P2,P3
```

Introduction: NetworkX official website

http://networkx.github.io/



2. Getting started with Python andNetworkX

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.

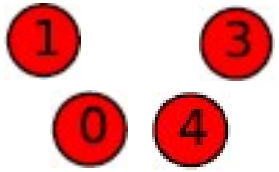
Getting started: adding nodes

```
# 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

- Anode 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)
```

```
# 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():
        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: 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: Load Graphs

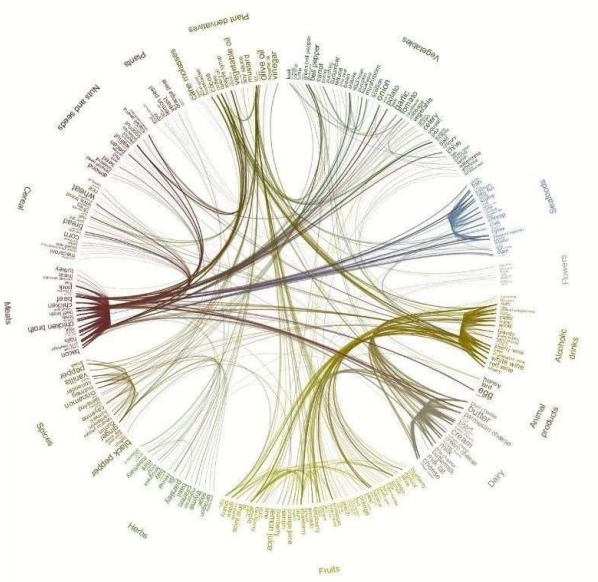
```
# from adjacency list
def load_from_adjlist(file):
    G1 = nx.read_adjlist('graphs/'+file)
    #print (nx.info(G1))
    return G1
```

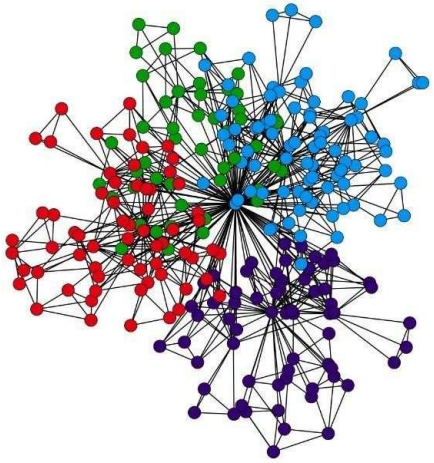
```
# from edgelist
#using an edgelist with weights
def load_from_edgelist(file):
    G2 = nx.read_edgelist('graphs/'+file, data=[('Weight', int)])
    print (nx.info(G2))
    print (G2.edges(data=True))
    return G2
```

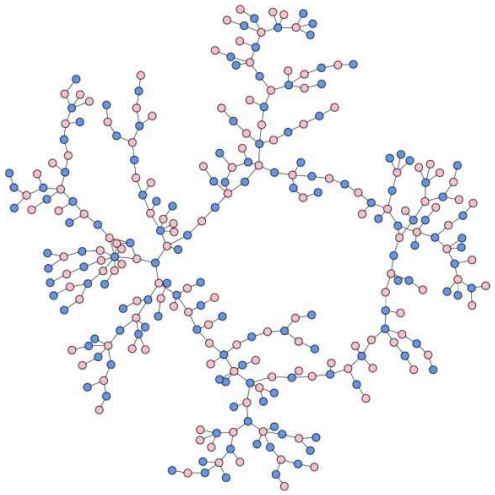
```
# from pandas   <<powerful Python data analysis toolkit>>
#using a pandas dataframe
def load_from_pandas(file):
    df=pd.read_csv('graphs/'+file,delim_whitespace=True,header=None,names=['n1','n2','weight'])
    print (df.head(3))
    G3 = nx.from_pandas_edgelist(df,'n1','n2',edge_attr='weight')
    print (nx.info(G3))
    return G3
```

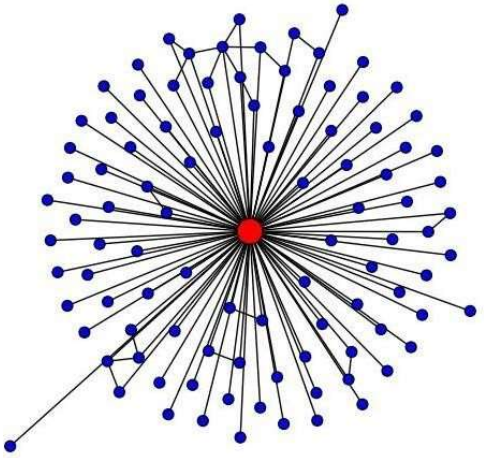
Introduction: drawing and plotting

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

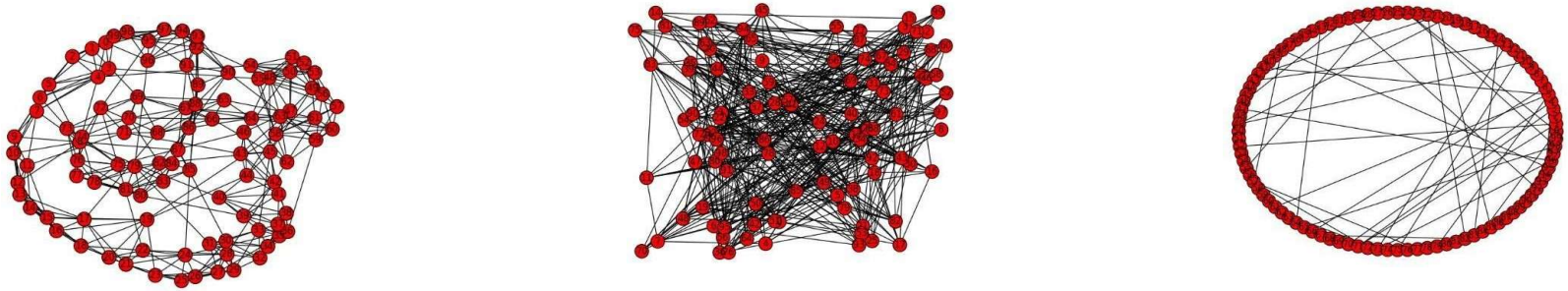




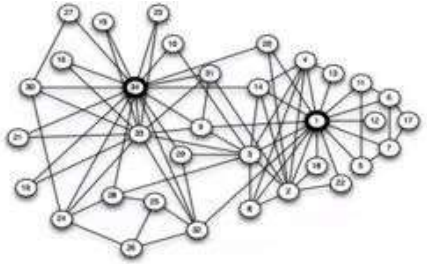




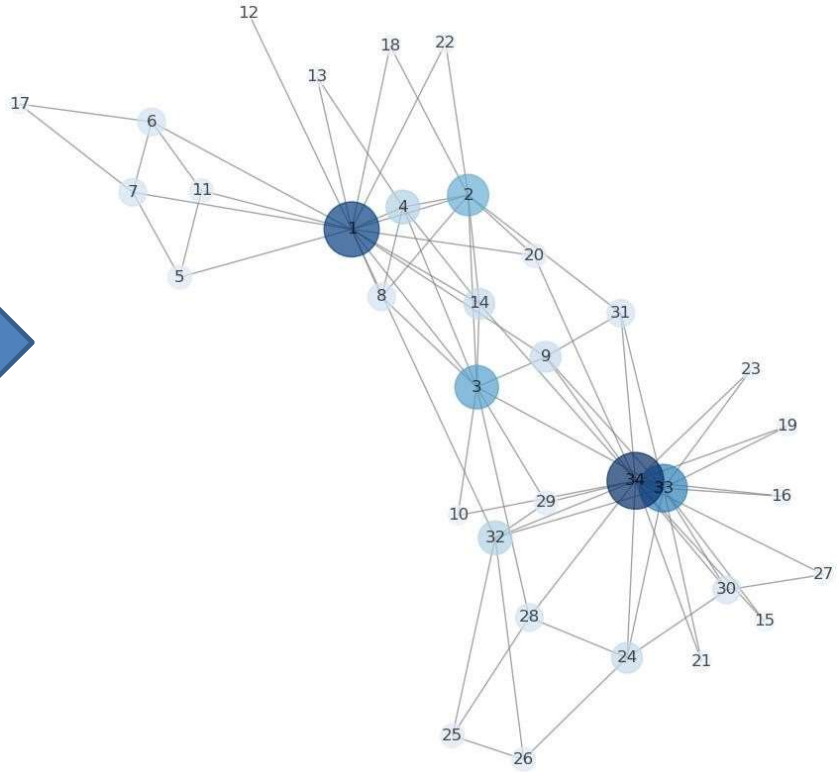
Getting started: drawing graphs



Getting started: drawing graphs



Friendship network in a 34-person karate club
[Zachary 1977]



Getting started: drawing graphs

```
def draw_graph(G):
    plt.figure(figsize=(10,9))
    node_color = [G.degree(v) for v in G]
    node_size = [G.degree(v)*100 for v in G]
    pos1 = nx.circular_layout(G)
    pos2 = nx.fruchterman_reingold_layout(G)
    pos = nx.spring_layout(G)
    nx.draw_networkx(G,pos,alpha=0.7,with_labels=True,
        edge_color='.5',node_size=node_size,node_color=node_color,cmap=plt.cm.Blues)
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

3. Basic network analysis

Clustering Coefficient

Local Clustering Coefficient

Compute the local clustering coefficient of node C:

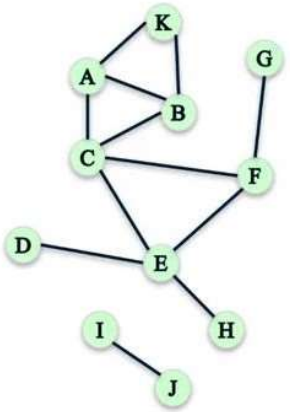
$$\frac{\#\text{ of pairs of C's friends who are friends}}{\#\text{ of pairs of C's friends}}$$

# of C's friends = $d_c = 4$ (the "degree" of C)

# of pairs of C's friends = $\frac{d_c(d_c-1)}{2} = \frac{12}{2} = 6$

# of pairs of C's friends who are friends = 2

Local clustering coefficent of C = $\frac{2}{6} = \frac{1}{3}$

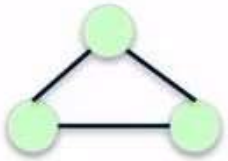



```
def clus_coef_v1(G,each_node):#Fraction of pairs of the
    if each_node==True:
        for node in G.nodes():
            print (nx.clustering(G,node))
    print ('Average with v1:',nx.average_clustering(G))
```

Transitivity

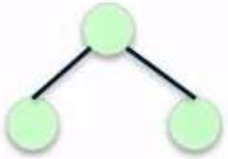
Percentage of "open triads" that are triangles in a network.

Triangles:



$$\textbf{Transitivity} = \frac{3*\text{Number of closed triads}}{\text{Number of open triads}}$$

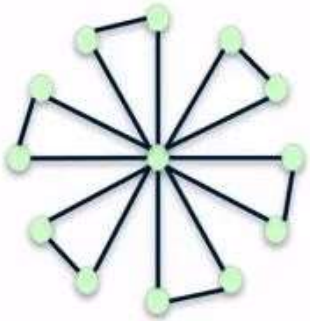
Open triads:



```
def clus_coef_v2(G): #Ratio of numbers of triangles and number of "open triads"
    print ('Average with v2: ',nx.transitivity(G))
```

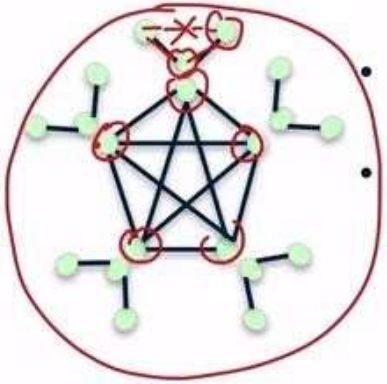
Clustering Coefficiency vs Transitivity

Both measure the tendency for edges to form triangles.
Transitivity weights nodes with large degree higher.



- Most nodes have high LCC
- The high degree node has low LCC

Ave. clustering coeff. = 0.93
Transitivity = 0.23



- Most nodes have low LCC
- High degree node have high LCC

Ave. clustering coeff. = 0.25
Transitivity = 0.86

Distance

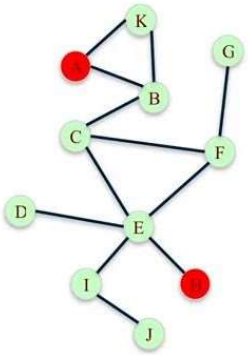
Distance

How far is node A from node H?

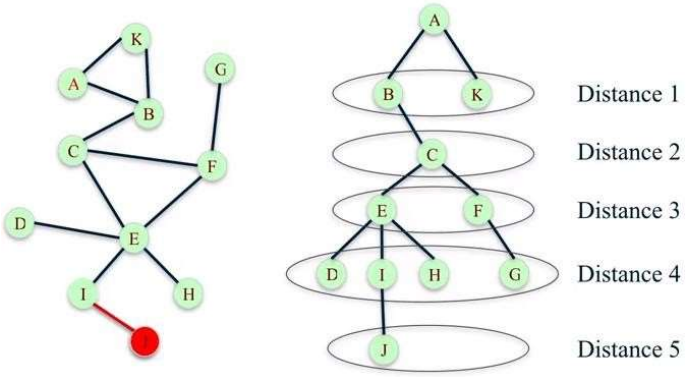
Path 1: A – B – C – E – H (4 "hops")

Path 2: A – B – C – F – E – H (5 "hops")

Path length: Number of steps it contains from beginning to end.



Breadth-First Search




```
#path: sequence of nodes connected by an edge --number of steps it contains from beginning to the end
#breadth-first search --create a tree
def create_tree(G):
    nodes=list(G.nodes())
    print (type(G.nodes()))
    T=nx.bfs_tree(G,nodes[0])
    print (T.edges())
    return T
```

Distance

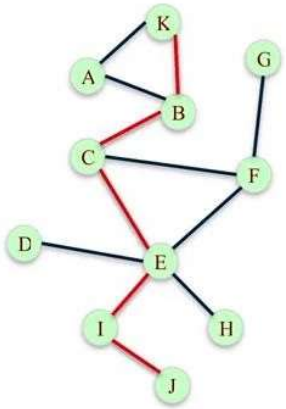
How to characterize the distance between all pairs of nodes in a graph?

Average distance between every pair of nodes.

In: nx.average_shortest_path_length(G)
Out: 2.52727272727

Diameter: maximum distance between any pair of nodes.

In: nx.diameter(G)
Out: 5




```
def find_avg_distance(G):
    try:
        avg = nx.average_shortest_path_length(G)
    except nx.NetworkXError:
        print ('Graph is not connected')
        avg=0
    print ('Average sp length:',avg)
    return avg
```

```
def find_diameter(G):#diameter: maximum distance between any pair of nodes
    try:
        d = nx.diameter(G)
    except nx.exception.NetworkXError:
        print ('Found infinite path length because the graph is not connected')
        d=0
    print ('diameter:',d)
    return (d)
```

Distance

How to summarize the distances between all pairs of nodes in a graph?

The **Eccentricity** of a node *n* is the largest distance between *n* and all other nodes.

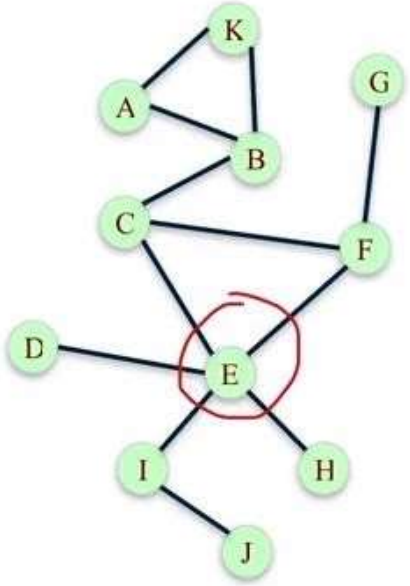
In: nx.eccentricity(G)

Out: {'A': 5, 'B': 4, 'C': 3, 'D': 4, 'E': 3, 'F': 3, 'G': 4, 'H': 4, 'I': 4, 'J': 5, 'K': 5}

The **radius** of a graph is the minimum eccentricity.

In: nx.radius(G)

Out: 3




```
def find_radius(G):
    try:
        rad = nx.radius(G)
    except nx.exception.NetworkXError:
        print('Graph is not connected')
        rad = 0
    print ('Radius: ',rad)
    return rad
```

```
def find_diameter(G): #diameter: maximum distance between any pair of nodes
    try:
        d = nx.diameter(G)
    except nx.exception.NetworkXError:
        print ('Found infinite path length because the graph is not connected')
        d=0
    print ('diameter:',d)
    return (d)
```

Distance

```
G = nx.karate_club_graph()
G = nx.convert_node_labels_to_integers(G,first_label=1)
```

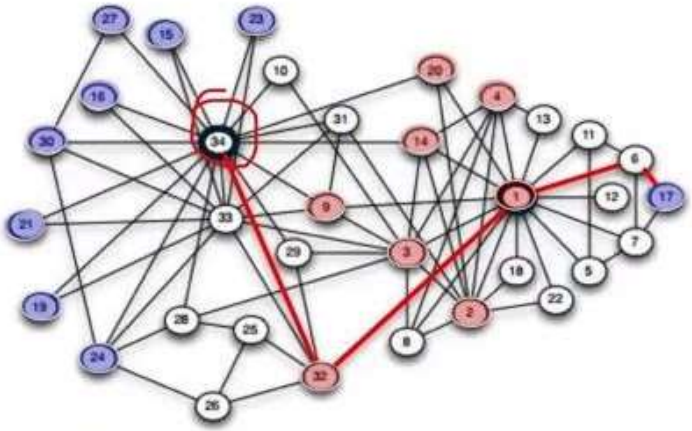
Average shortest path = 2.41

Radius = 3

Diameter = 5

Center = [1, 2, 3, 4, 9, 14, 20, 32]

Periphery: [15, 16, 17, 19, 21, 23, 24, 27, 30]



Friendship network in a 34-person karate club

Node 34 looks pretty "central". However, it has distance 4 to node 17

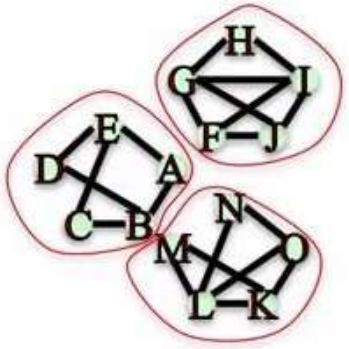
Connected Components

Undirected Graphs

Connected: for every pair nodes, there is a path between them.

Connected components

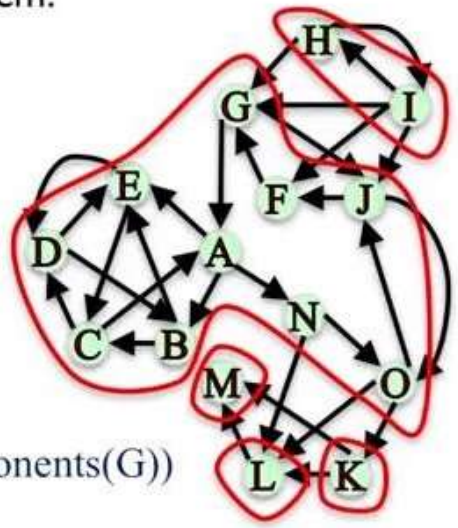
nx.connected_components(G)



Directed Graphs

Strongly connected: for every pair nodes, there is a *directed* path between them.

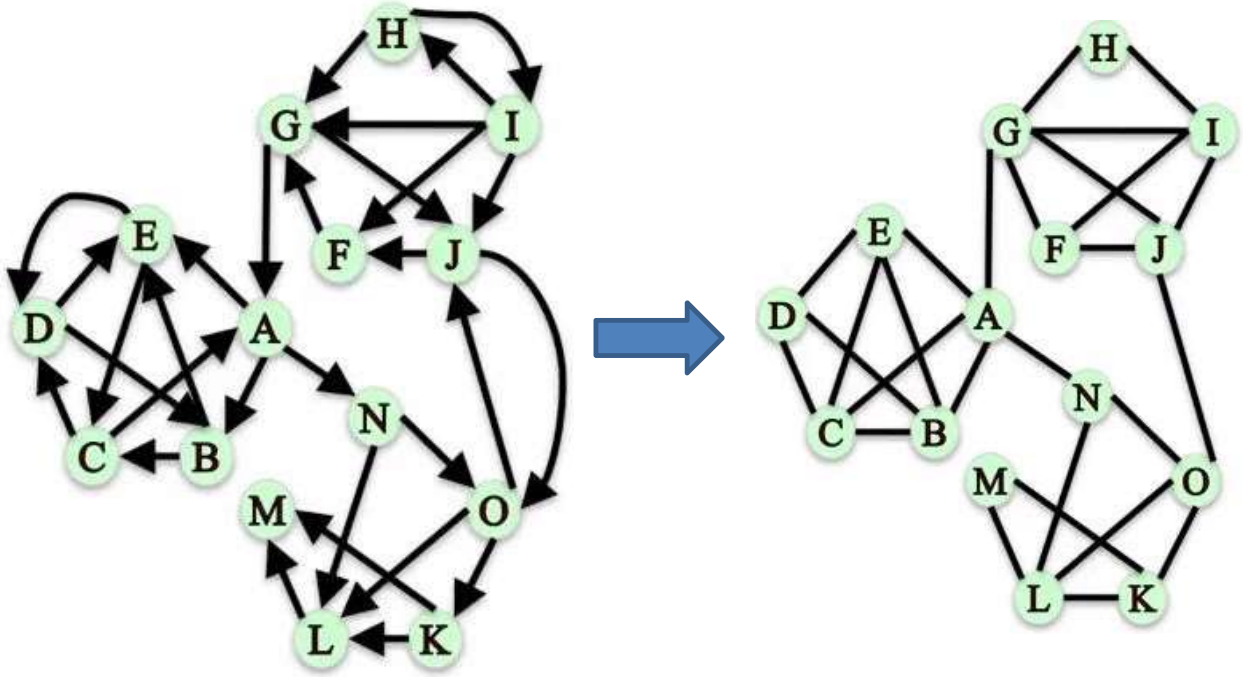
Strongly connected components



nx.strongly_connected_components(G))

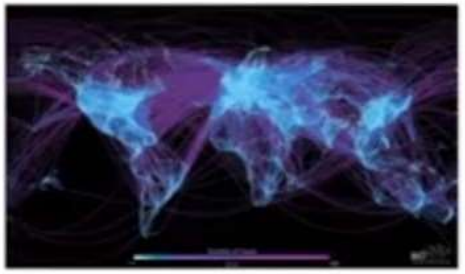
Connected Components

A directed graph is **weakly connected** if replacing all directed edges with undirected edges produces a connected undirected graph.



Network Robustness

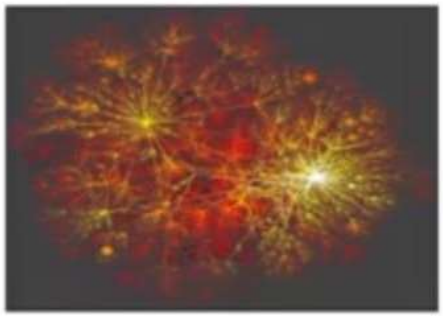
Network robustness: the ability of a network to maintain its general structural properties when it faces failures or attacks.



Network of direct flights around the world [Bio.Diaspora]

Attacks?

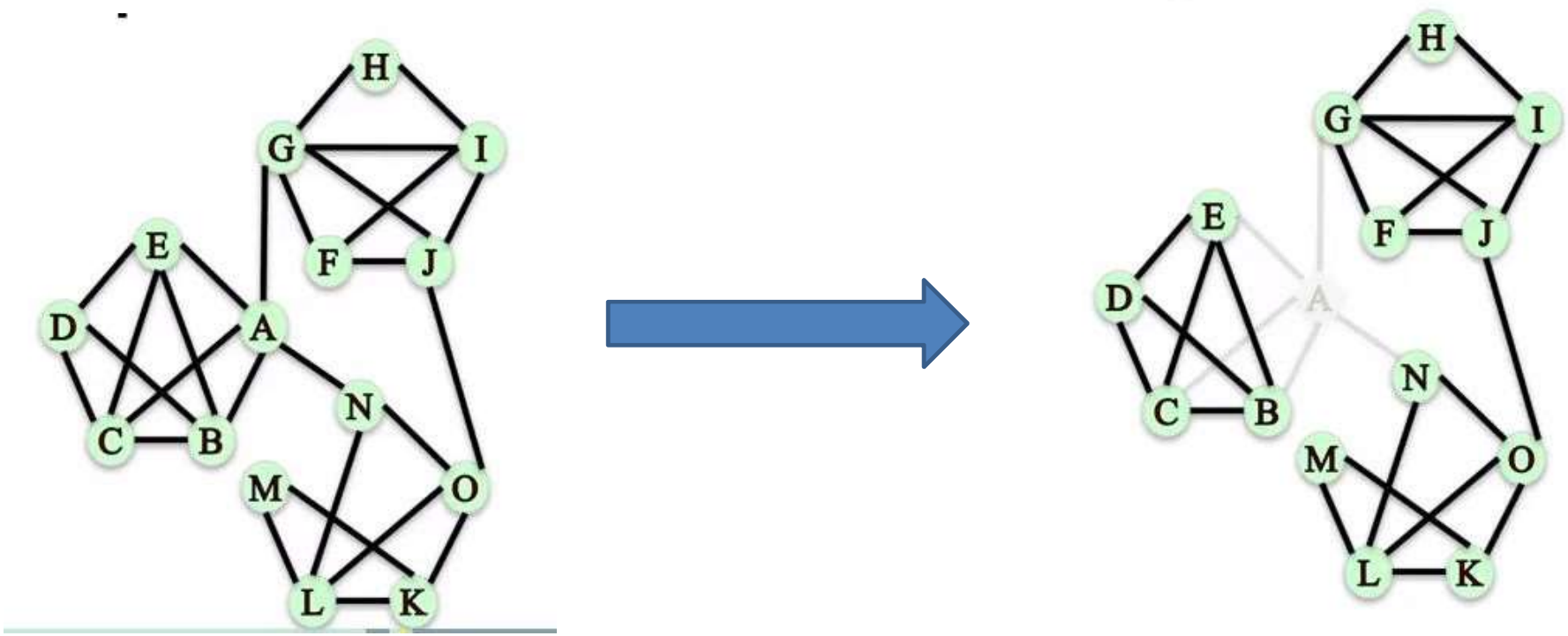
Examples: airport closures, internet router failures, power line failures.



Internet Connectivity [K. C. Claffy]

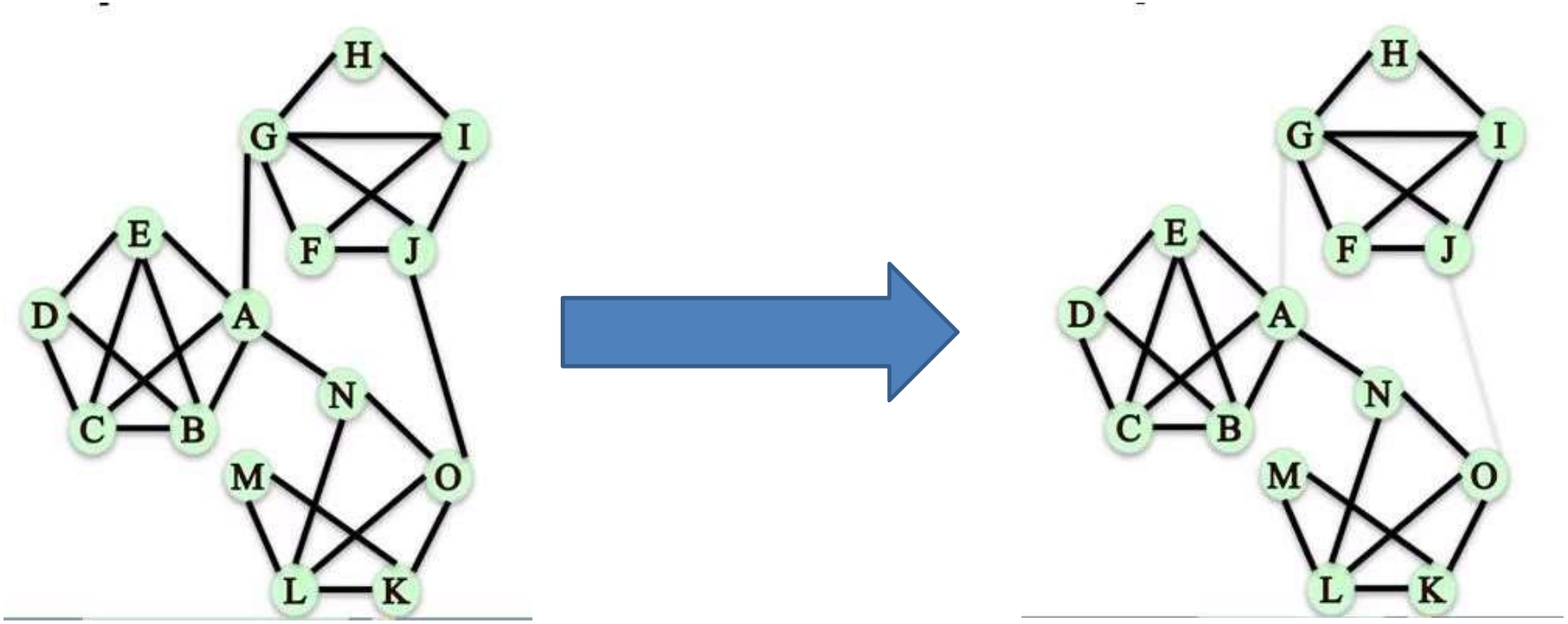
Network Robustness

```
def get_smallest_number_of_nodes_to_disconnect(G_un):
    print (nx.node_connectivity(G_un),nx.minimum_node_cut(G_un))
    return nx.node_connectivity(G_un), nx.minimum_node_cut(G_un)
```



Network Robustness

```
def get_edges_to_remove_to_disconnect(G_un):
    print(nx.edge_connectivity(G_un), nx.minimum_edge_cut(G_un))
    return nx.edge_connectivity(G_un), nx.minimum_edge_cut(G_un)
```

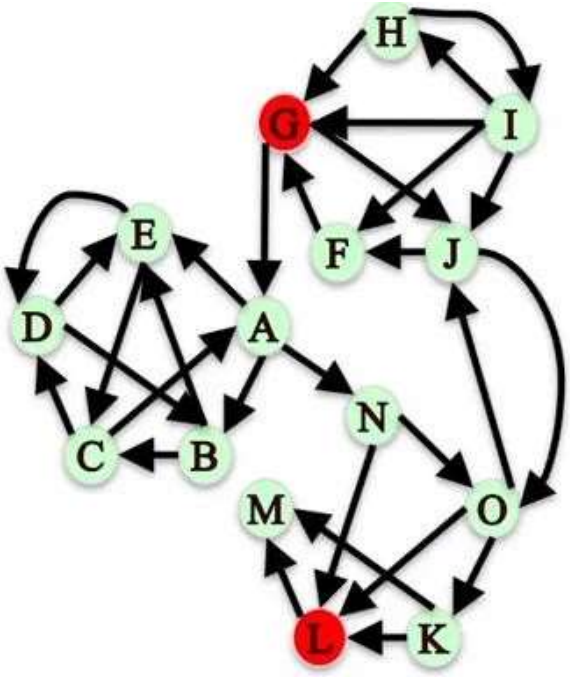


Network Robustness

Imagine node G wants to send a message to node L by passing it along to other nodes in this network.

What options does G have to deliver the message?

```
def find_simple_paths(G,n1,n2):
    print (sorted(nx.all_simple_paths(G,n1,n2)))
    return sorted(nx.all_simple_paths(G,n1,n2))

def find_which_nodes_and_edges_to_remove(G,n1,n2):
    a= nx.node_connectivity(G,n1,n2)
    b= nx.minimum_node_cut(G,n1,n2)
    c = nx.edge_connectivity(G,n1,n2)
    d = nx.minimum_edge_cut(G,n1,n2)
    print (a,b,c,d)
    return (a,b,c,d)
```



Node Importance

Network Centrality

Centrality measures identify the most important nodes in a network:

- Influential nodes in a social network.
- Nodes that disseminate information to many nodes or prevent epidemics.
- Hubs in a transportation network.
- Important pages on the Web.
- Nodes that prevent the network from breaking up.

Node Importance – Degree Centrality

Assumption: important nodes have many connections.

The most basic measure of centrality: number of neighbors.

Undirected networks: use degree

Directed networks: use in-degree or out-degree

$C_{deg}(v) = \frac{d_v}{|N|-1}$, where N is the set of nodes in the network and d_v is the degree of node v.

```python
def get_centrality_of_node(G,node):
    degCen = nx.degree_centrality(G)
    return degCen,degCen[node]

def get_centrality_of_directed(G):
    degOut = nx.out_degree_centrality(G)
    degIn = nx.in_degree_centrality(G)
    return degOut,degIn
```

Node Importance – Closeness Centrality

Assumption: important nodes are close to other nodes.

$C_{close}(v) = \frac{|N|-1}{\sum_{u \in N \setminus \{v\}} d(v,u)}$, where

N = set of nodes in the network,

$d(v,u)$ = length of shortest path from v to u.

```
def get_closeness_centrality(G,norm):
    return nx.closeness_centrality(G,wf_improved = norm)

#if we want to find the centrality of a node which is somehow disconnected - problem
#in this case we use normalized = True
```

Node Importance – Betweenness Centrality

Assumption: important nodes connect other nodes.

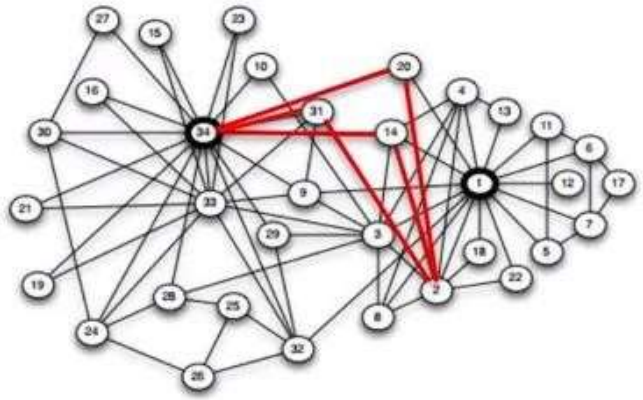
Recall: the distance between two nodes is the length of the shortest path between them.

Ex. The distance between nodes 34 and 2 is 2:

Path 1: 34 – 31 – 2

Path 2: 34 – 14 – 2

Path 3: 34 – 20 – 2



Friendship network in a 34-person karate club
[Zachary 1977]

Nodes 31, 14, and 20 are in a shortest path of between nodes 34 and 2.

$C_{btw}(v) = \sum_{s,t \in N} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$, where

$\sigma_{s,t}$ = the number of shortest paths between nodes s and t.

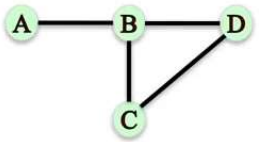
$\sigma_{s,t}(v)$ = the number shortest paths between nodes s and t *that pass through node* v.

Node v has a high Betweenness centrality if it shows up in the shortest paths of any nodes s,t

Node Importance – Betweenness Centrality

Assumption: important nodes connect other nodes.

$C_{btw}(v) = \sum_{s,t \in N} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$



Endpoints: we can either include or exclude node v as node s and t in the computation of $C_{btw}(v)$.

Ex. If we exclude node v, we have:

$C_{btw}(B) = \frac{\sigma_{A,D}(B)}{\sigma_{A,D}} + \frac{\sigma_{A,C}(B)}{\sigma_{A,C}} + \frac{\sigma_{C,D}(B)}{\sigma_{C,D}} = \frac{1}{1} + \frac{1}{1} + \frac{0}{1} = 2$

If we include node v, we have:

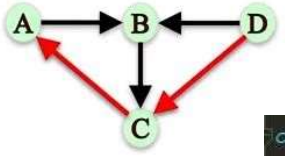
$C_{btw}(B) = \frac{\sigma_{A,B}(B)}{\sigma_{A,B}} + \frac{\sigma_{A,C}(B)}{\sigma_{A,C}} + \frac{\sigma_{A,D}(B)}{\sigma_{A,D}} + \frac{\sigma_{B,C}(B)}{\sigma_{B,C}} + \frac{\sigma_{B,D}(B)}{\sigma_{B,D}} + \frac{\sigma_{C,D}(B)}{\sigma_{C,D}} = \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{0}{1} = 5$

Normalization: betwenness centrality values will be larger in graphs with many nodes. To control for this, we divide centrality values by the number of pairs of nodes in the graph (excluding v):

$\frac{1}{2}(|N|-1)(|N|-2)$ in undirected graphs

$(|N|-1)(|N|-2)$ in directed graphs

What if not all nodes can reach each other?



Node D cannot be reached by any other node.

Hence, $\sigma_{A,D} = 0$, making the above definition undefined.

Ex. What is the betweenness centrality of node C, without including it as endpoint?

$C_{btw}(C) = \frac{\sigma_{A,B}(C)}{\sigma_{A,B}} + \frac{\sigma_{B,A}(C)}{\sigma_{B,A}} + \frac{\sigma_{D,B}(C)}{\sigma_{D,B}} + \boxed{\frac{\sigma_{D,A}(C)}{\sigma_{D,A}}} = \frac{0}{1} + \frac{1}{1} + \frac{0}{1} + \boxed{\frac{1}{1}} = 2$

```
def get_bet_centrality(G,i):
    betCen = nx.betweenness_centrality(G,normalized=True,endpoints=False,k=i)
    return betCen

def get_n_highest_bet_Cen(G,n):
    betCen = get_bet_centrality(G,34)
    return (sorted(betCen.items(), key=operator.itemgetter(1),reverse=True))[:n]
```

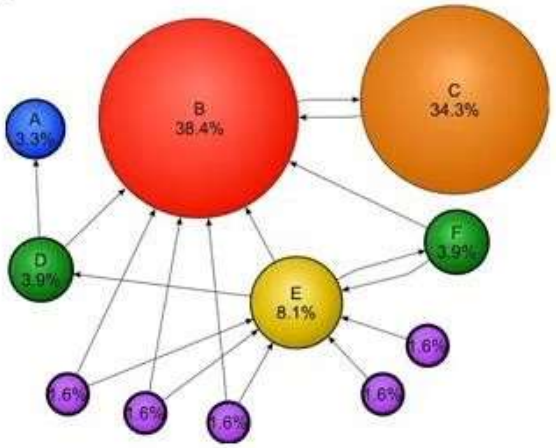
PageRank (Google Founders)

PageRank

Developed by Google founders to measure the importance of webpages from the hyperlink network structure.

PageRank assigns a score of importance to each node. Important nodes are those with many in-links from important pages.

PageRank can be used for any type of network, but it is mainly useful for directed networks.



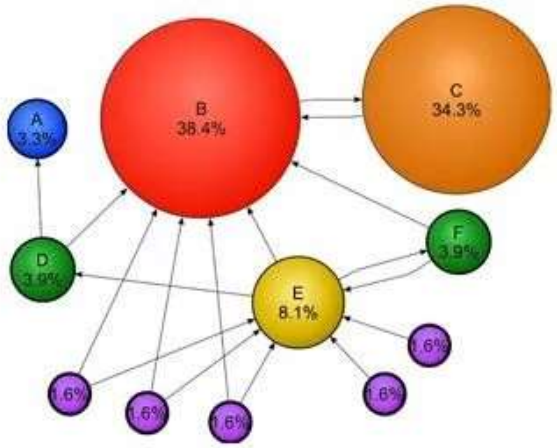
http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

PageRank – how to compute it?

PageRank

n = number of nodes in the network
k = number of steps

1. Assign all nodes a PageRank of $1/n$
2. Perform the *Basic PageRank Update Rule* k times.



Basic PageRank Update Rule: Each node gives an equal share of its current PageRank to all the nodes it links to.

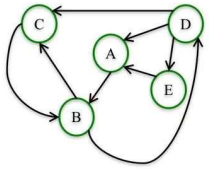
The new PageRank of each node is the sum of all the PageRank it received from other nodes.

PageRank – how to compute it?

For most networks, PageRank values converge as k gets larger ($k \to \infty$)

PageRank – Step 1

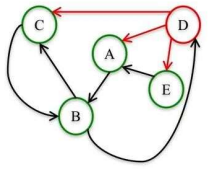
Page Rank (k = 1)					
	A	B	C	D	E
Old	1/5	1/5	1/5	1/5	1/5
New					



PageRank – Step 1

Page Rank (k = 1)					
	A	B	C	D	E
Old	1/5	1/5	1/5	1/5	1/5
New					

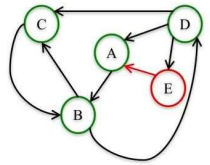
A: (1/3)*(1/5)



PageRank – Step 1

Page Rank (k = 1)					
	A	B	C	D	E
Old	1/5	1/5	1/5	1/5	1/5
New					

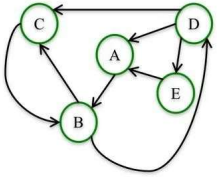
A: (1/3)*(1/5) + 1/5



PageRank – Step 1

Page Rank (k = 1)					
	A	B	C	D	E
Old	1/5	1/5	1/5	1/5	1/5
New	4/15	2/5			

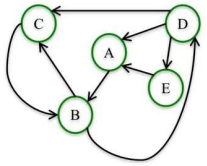
A: (1/3)*(1/5) + 1/5 = 4/15
B: 1/5 + 1/5 = 2/5



PageRank – Step 1

Page Rank (k = 1)					
	A	B	C	D	E
Old	1/5	1/5	1/5	1/5	1/5
New	4/15	2/5	1/6	1/10	1/15

A: (1/3)*(1/5) + 1/5 = 4/15
B: 1/5 + 1/5 = 2/5
C: (1/3)*(1/5) + (1/2)*(1/5) = 5/30 = 1/6
D: (1/2)*(1/5) = 1/10
E: (1/3)*(1/5) = 1/15

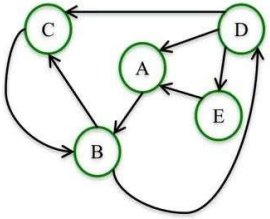


. . . . .

PageRank – Step 2

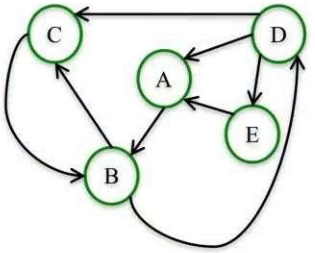
Page Rank (k = 2)					
	A	B	C	D	E
Old	4/15	2/5	1/6	1/10	1/15
New	1/10	13/30	7/30	2/10	1/30

A: (1/3)*(1/10) + 1/15 = 1/10
B: 1/6 + 4/15 = 13/30
C: (1/3)*(1/10) + (1/2)*(2/5) = 7/30
D: (1/2)*(2/5) = 2/10
E: (1/3)*(1/10) = 1/30



PageRank

	Page Rank				
	A	B	C	D	E
k=2	1/10	13/30	7/30	2/10	1/30
k=2	.1	.43	.23	.20	.03
k=3	.1	.33	.28	.22	.06

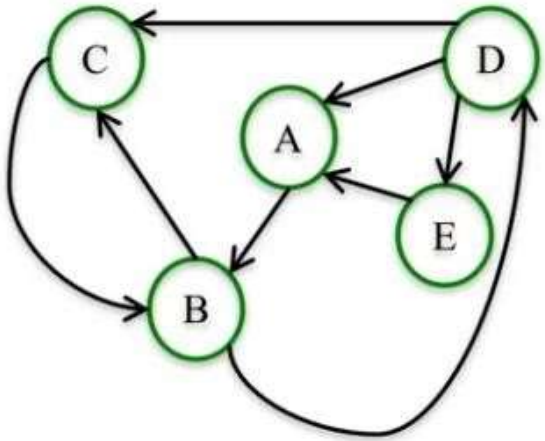


PageRank – more steps…

For most networks, PageRank values converge as k gets larger.

PageRank

	Page Rank				
	A	B	C	D	E
k=2	1/10	13/30	7/30	2/10	1/30
k=2	.1	.43	.23	.20	.03
k=3	.1	.33	.28	.22	.06
k=∞	.12	.38	.25	.19	.06



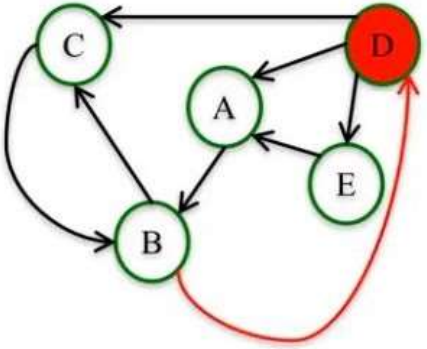
What if continue with k = 4,5,6,...?

Scaled PageRank

The PageRank of a node at step k is the probability that a ***random walker*** lands on the node after taking k steps.

Random walk of k steps: Start on a random node. Then choose an outgoing edge at random and follow it to the next node. Repeat k times.

For example, a random walk of 5 steps on this graph looks like this:

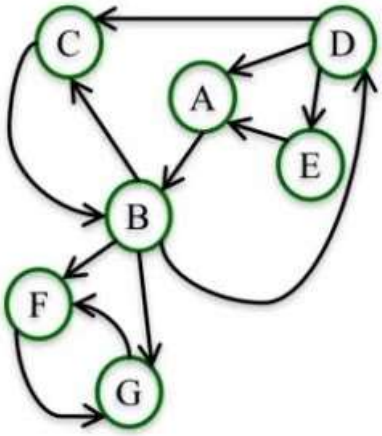


PageRank Problem

What's the PageRank of the nodes in this network?
[Hint: think about the random walk interpretation]

For a large enough k: F and G each have PageRank of ½ and all the other nodes have PageRank 0.

Why? Imagine a random walk on this network. Whenever the walk lands on F or G, it is "stuck" on F and G.



Scaled PageRank

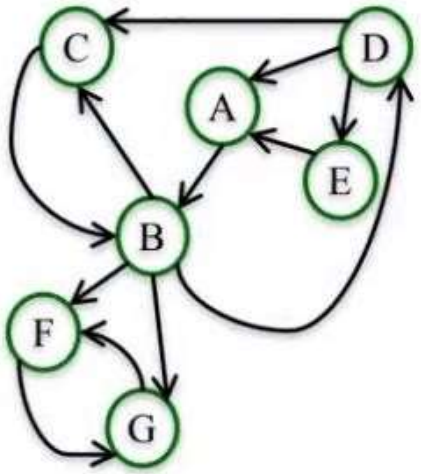
PageRank Problem

To fix this, we introduce a "damping parameter" α.

Random walk of k steps with damping parameter α: Start on a random node. Then:

- With probability α: choose an outgoing edge at random and follow it to the next node.
- With probability $1 - \alpha$: choose a node at random and go to it.

Repeat k times.

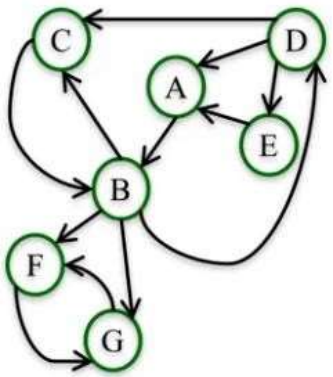


The random walk is no longer "stuck" on nodes F and G.

Scaled PageRank

The **Scaled PageRank** of k steps and damping factor α of a node n is the probability that a random walk with damping factor α lands on a n after k steps.

For most networks, as k gets larger, Scaled PageRank converges to a unique value, which depends on α.



Scaled PageRank ($\alpha = .8, k$ large)						
A	B	C	D	E	F	G
.08	.17	.1	.08	.05	.27	.25

F and G still have high PageRank, but not all the PageRank.

Damping factor works better in very large networks like the Web or large social networks.

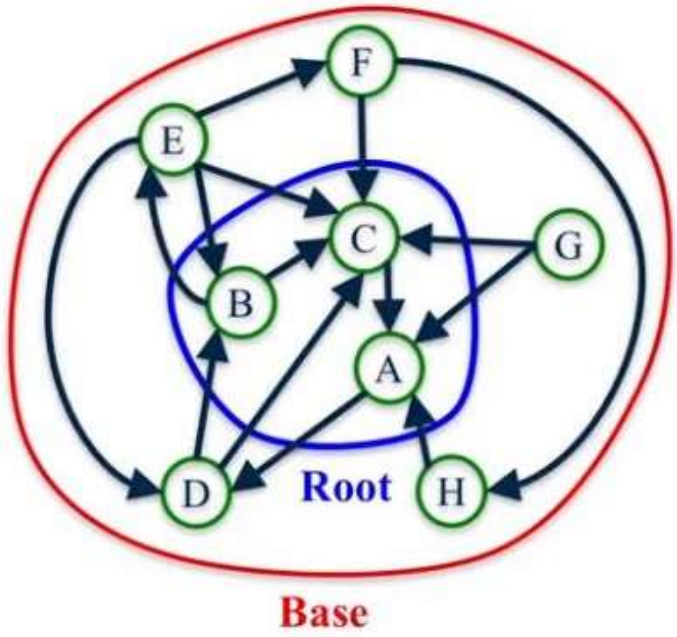
You can use NetworkX function pagerank(G, alpha=0.8) to compute Scaled PageRank of network G with damping parameter alpha.

```
def get_pagerank(G, dampingFactor):
    pRank = nx.pagerank(G, alpha=dampingFactor)
    return pRank
```

Hubs and Authorities

Given a query to a search engine:

- **Root**: set of highly relevant web pages (e.g. pages that contain the query string) – potential *authorities*.
- Find all pages that link to a page in root – potential *hubs*.
- **Base**: root nodes and any node that links to a node in root.
- Consider all edges connecting nodes in the base set.

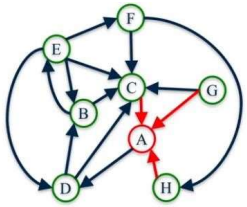


HITS Algorithm

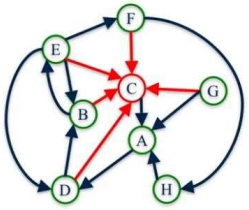
Computing k iterations of the HITS algorithm to assign an *authority score* and *hub score* to each node.

1. Assign each node an authority and hub score of 1.
2. Apply the ***Authority Update Rule***: each node's *authority* score is the sum of *hub* scores of each node that *points to it*.
3. Apply the ***Hub Update Rule***: each node's hub score is the sum of *authority* scores of each node that *it points to*.
4. **Nomalize** Authority and Hub scores: $\text{auth}(j) = \frac{\text{auth}(j)}{\sum_{i \in N} \text{auth}(i)}$

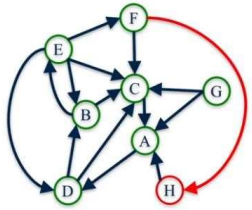
	Old Auth	Old Hub	New Auth	New Hub
A	1	1		
B	1	1		
C	1	1		
D	1	1		
E	1	1		
F	1	1		
G	1	1		
H	1	1		



	Old Auth	Old Hub	New Auth	New Hub
A	1	1	3	
B	1	1	2	
C	1	1	5	
D	1	1		
E	1	1		
F	1	1		
G	1	1		
H	1	1		



	Old Auth	Old Hub	New Auth	New Hub
A	1	1	3	
B	1	1	2	
C	1	1	5	
D	1	1	2	
E	1	1	1	
F	1	1	1	
G	1	1	0	
H	1	1	1	



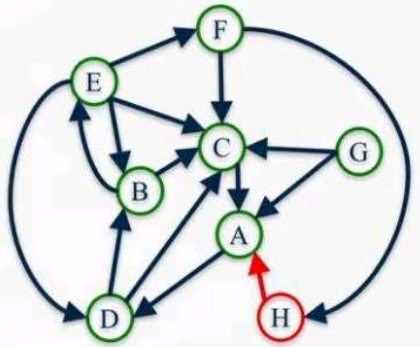
HITS Algorithm

Computing k iterations of the HITS algorithm to assign an *authority score* and *hub score* to each node.

1. Assign each node an authority and hub score of 1.
2. Apply the ***Authority Update Rule***: each node's *authority* score is the sum of *hub* scores of each node that *points to it*.
3. Apply the ***Hub Update Rule***: each node's hub score is the sum of authority scores of each node that *it points to*.
4. **Nomalize** Authority and Hub scores: $\text{auth}(j) = \frac{\text{auth}(j)}{\sum_{i \in N} \text{auth}(i)}$

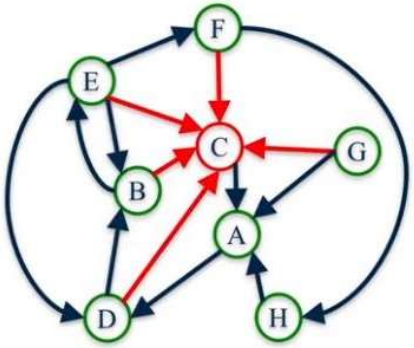
For the hub update rule, we compute the out-degree of each node.

	Old Auth	Old Hub	New Auth	New Hub
A	1	1	3	1
B	1	1	2	2
C	1	1	5	1
D	1	1	2	2
E	1	1	1	4
F	1	1	1	2
G	1	1	0	2
H	1	1	1	1



HITS Algorithm

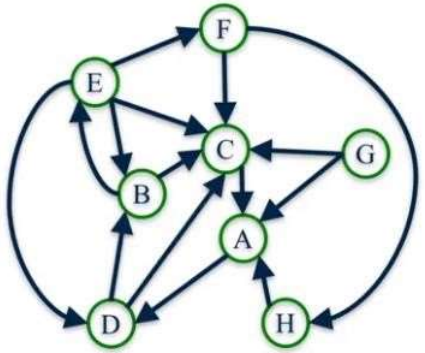
	Old Auth	Old Hub	New Auth	New Hub
A	1/5	1/15	4/15	
B	2/15	**2/15**	6/15	
C	1/3	1/15	**12/15**	
D	2/15	**2/15**		
E	1/15	**4/15**		
F	1/15	**2/15**		
G	0	**2/15**		
H	1/15	1/15		



2/15 + 2/15 + 4/15 + 2/15 + 2/15 = 12/15

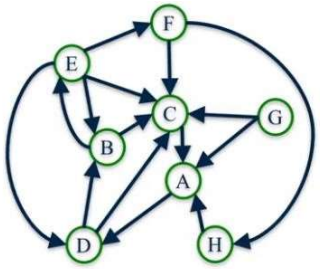
.....

	Old Auth	Old Hub	New Auth	New Hub
A	1/5	1/15	4/15	2/15
B	2/15	2/15	6/15	2/5
C	1/3	1/15	12/15	1/5
D	2/15	2/15	1/3	7/15
E	1/15	4/15	2/15	**2/3**
F	1/15	2/15	4/15	**2/5**
G	0	2/15	0	**8/15**
H	1/15	1/15	2/15	**1/5**



After Normalisation . . .

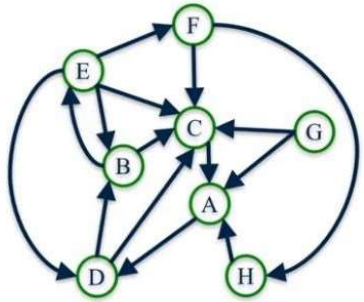
	Old Auth	Old Hub	New Auth	New Hub
A	1/5	1/15	4/15	2/15
B	2/15	2/15	6/15	2/5
C	1/3	1/15	12/15	1/5
D	2/15	2/15	1/3	7/15
E	1/15	4/15	2/15	2/3
F	1/15	2/15	4/15	2/5
G	0	2/15	0	8/15
H	1/15	1/15	2/15	1/5



Normalize:

$\sum_{i \in N} \text{auth}(i) = {}^{35}/_{15}$

	Old Auth	Old Hub	New Auth	New Hub
A	1/5	1/15	4/35	2/15
B	2/15	2/15	6/35	2/5
C	1/3	1/15	12/35	1/5
D	2/15	2/15	1/7	7/15
E	1/15	4/15	2/35	2/3
F	1/15	2/15	4/35	2/5
G	0	2/15	0	8/15
H	1/15	1/15	2/35	1/5

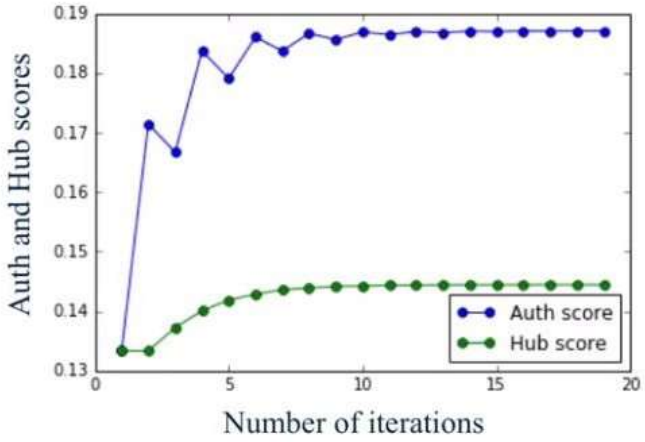


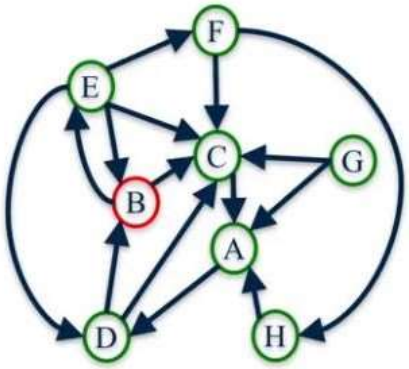
Normalize:

$\sum_{i \in N} \text{hub}(i) = {}^{45}/_{15} = 3$

HITS Algorithm

HITS Algorithm Convergence

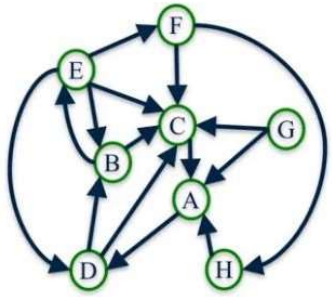




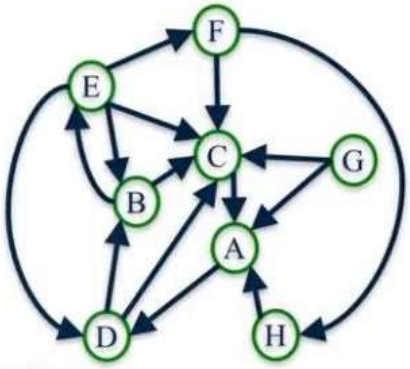
For most networks, as k gets larger, authority and hub scores converge to a unique value.

As $k \longrightarrow \infty$ the hub and authority scores approach:

	A	B	C	D	E	F	G	H
Auth	.08	.19	.40	.13	.06	.11	0	.06
Hub	.04	.14	.03	.19	.27	.14	.15	.03



- The HITS algorithm starts by constructing a *root* set of relevant web pages and expanding it to a *base set.*
- HITS then assigns an authority and hub score to each node in the network.
- Nodes that have incoming edges from *good hubs* are *good authorities*, and nodes that have outgoing edges to *good authorities* are *good hubs.*
- Authority and hub scores converge for most networks.

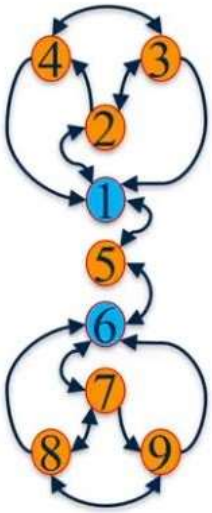



```
def get_hubs_authorities(G):
    h,a = nx.hits(G)
    return h,a
```

Centrality Comparison

Comparing Centrality Measures

In-Deg	Closeness	Betweenness	PageRank	Auth	Hub
1	5	5	1	1	2
6	1	1	6	6	5
2	6	6	5	4	7
3	2	2	2	9	3
4	3	7	7	3	8
5	7	3	3	8	4
7	8	8	8	2	9
8	4	4	4	7	1
9	9	9	9	5	6



- In this example, no pair of centrality measures produces the exact same ranking of nodes, but they have some commonalities.
- Centrality measures make different assumptions about what it means to be a "central" node. Thus, they produce different rankings.
- The best centrality measure depends on the context of the network one is analyzing.
- When identifying central nodes, it is usually best to use multiple centrality measures instead of relying on a single one.

Degree Distributions

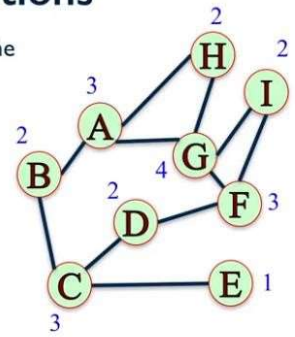
Degree Distributions

The **degree** of a node in an undirected graph is the number of neighbors it has.

The **degree distribution** of a graph is the probability distribution of the degrees over the entire network.

The degree distribution, $P(k)$, of this network has the following values:

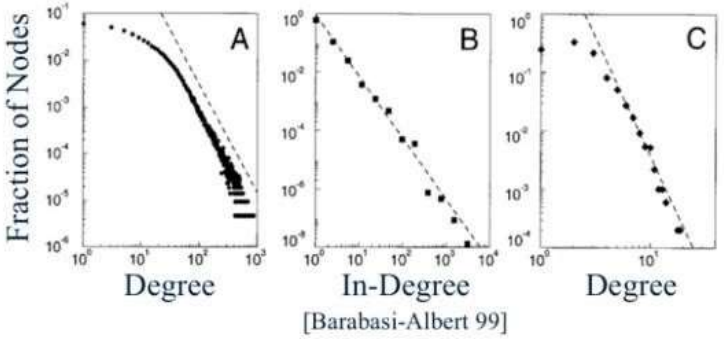
$$P(1) = \frac{1}{9}, P(2) = \frac{4}{9}, P(3) = \frac{1}{3}, P(4) = \frac{1}{9}$$



A – **Actors**: network of 225,000 actors connected when they appear in a movie together.

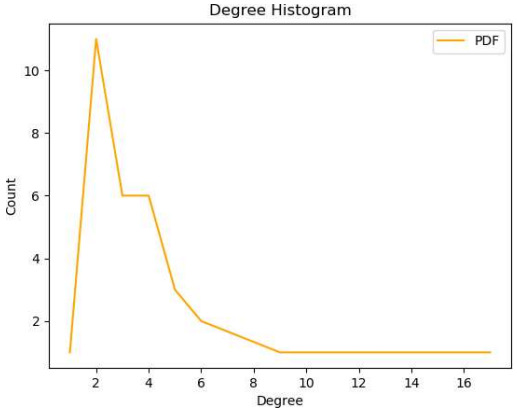
B – **The Web**: network of 325,000 documents on the WWW connected by URLs.

C – **US Power Grid**: network of 4,941 generators connected by transmission lines.



[Barabasi-Albert 99]

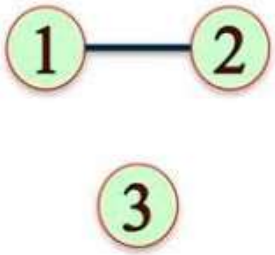
Degree distribution looks like a straight line when on a log-log scale. **Power law**: $P(k) = Ck^{-\alpha}$, where α and C are constants. α-values – A: 2.3, B:2.1, C:4.

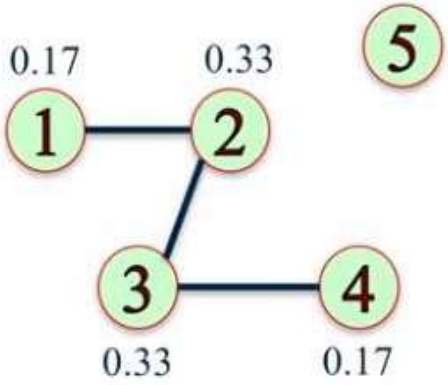


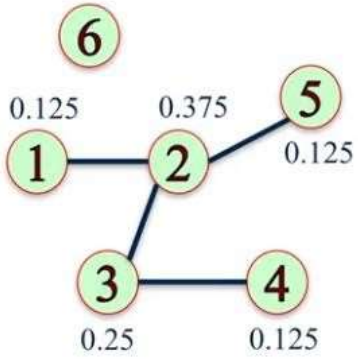
http://www.cs.cornell.edu/home/kleinber/networks-book/networks-book-ch18.pdf

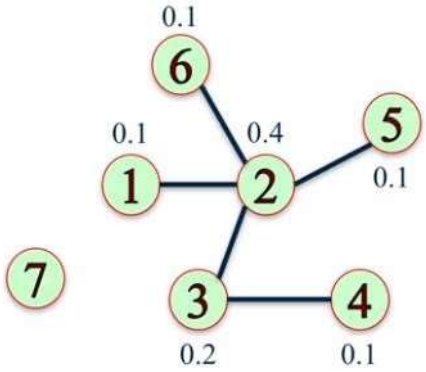
Preferential Attachment Model

- Start with two nodes connected by an edge.
- At each time step, add a new node with an edge connecting it to an existing node.
- Choose the node to connect to at random with probability proportional to each node's degree.
- The probability of connecting to a node u of degree k_u is $k_u / \sum_j k_j$.







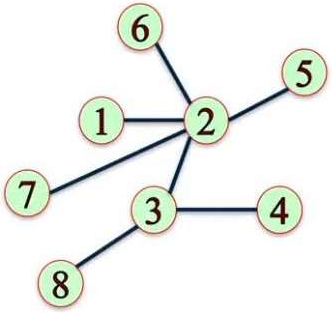


Preferential Attachment Model

Rich get Richer Phenomena

As the number of nodes increases, the degree distribution of the network under the preferential attachment model approaches the power law $P(k) = Ck^{-3}$ with constant C.

The preferential attachment model produces networks with degree distributions similar to real networks.



barabasi_albert_graph(n, m) returns a network with *n* nodes. Each new node attaches to *m* existing nodes according to the Preferential Attachment model.

```
G = nx.barabasi_albert_graph(1000000,1)
```

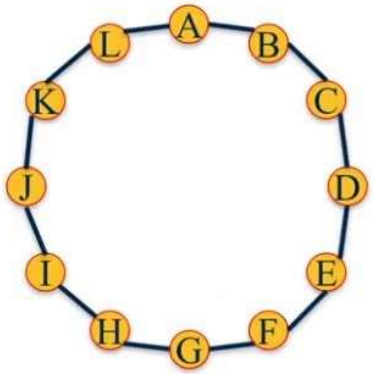
- The degree distribution of a graph is the probability distribution of the degrees over the entire network.
- Many real networks have degree distributions that look like power laws ($P(k) = Ck^{-\alpha}$).
- Models of network generation allow us to identify mechanisms that give rise to observed patterns in real data.
- The Preferential Attachment Model produces networks with a power law degree distribution.

Small World Networks

Motivation: Real networks exhibit high clustering coefficient and small average shortest paths. Can we think of a model that achieves both of these properties?

Small-world model:

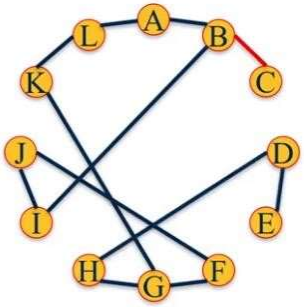
- Start with a ring of n nodes, where each node is connected to its k nearest neighbors.
- Fix a parameter $p \in [0,1]$
- Consider each edge (u, v). With probability p, select a node w at random and rewire the edge (u, v) so it becomes (u, w).



Small-world model:

- Start with a ring of n nodes, where each node is connected to its k nearest neighbors.
- Fix a parameter $p \in [0,1]$
- Consider each edge (u, v). With probability p, select a node w at random and rewire the edge (u, v) so it becomes (u, w).

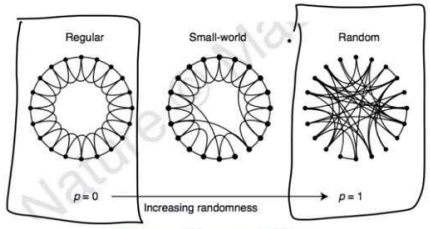
Example: $k = 2, p = 0.4$



Regular Lattice ($p = 0$): no edge is rewired.

Random Network ($p = 1$): all edges are rewired.

Small World Network ($0 < p < 1$): Some edges are rewired. Network conserves some local structure but has some randomness.



Watts and Strogatz, 1999

Small World Models

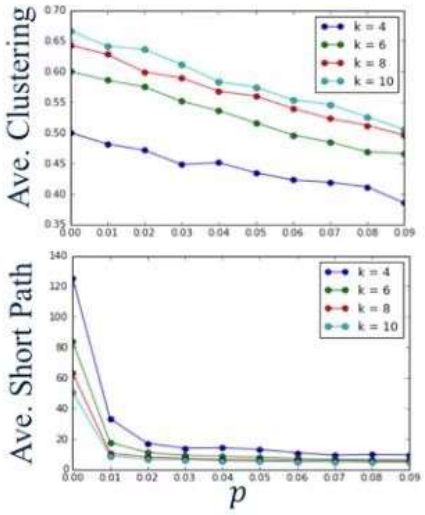
What is the average clustering coefficient and shortest path of a small world network?
It depends on parameters k and p.

As p increases from 0 to 0.1:

- average shortest path decreases rapidly.
- average clustering coefficient deceases slowly.

An instance of a network of 1000 nodes, $k = 6$, and $p = 0.04$ has:

- 8.99 average shortest path.
- 0.53 average clustering coefficient.




```
def get_watts_strogatz(n,k,p):
    G = nx.watts_strogatz_graph(n,k,p)
```

- Real social networks appear to have small shortest paths between nodes and high clustering coefficient.
- The preferential attachment model produces networks with small shortest paths but very small clustering coefficient.
- The small world model starts with a ring lattice with nodes connected to k nearest neighbors (high local clustering), and it rewires edges with probability p.
- For small values of p, small world networks have small average shortest path and high clustering coefficient, matching what we observe in real networks.
- However, the degree distribution of small world networks is not a power law.

Link prediction

Given a fixed network, can we predict how it will look in the future?

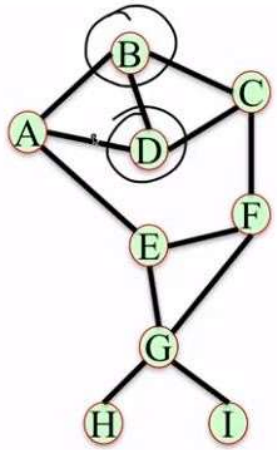
Measure 1: Common Neighbors

The number of common neighbors of nodes X and Y is

$$\text{comm\_neigh}(X, Y) = |N(X) \cap N(Y)|,$$

where $N(X)$ is the set of neighbors of node X

$$\text{comm\_neigh}(A, C) = |\{B, D\}| = 2$$



Link prediction

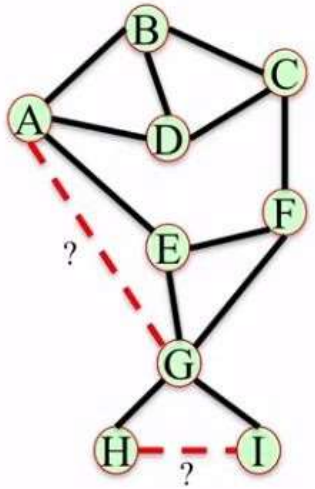
Measure 2: Jaccard Coefficient

Number of common neighbors normalized by the total number of neighbors.

The Jaccard coefficient of nodes X and Y is

$$\text{jacc\_coeff}(X,Y) = \frac{|N(X) \cap N(Y)|}{|N(X) \cup N(Y)|}$$

$$\text{jacc\_coeff}(A,C) = \frac{|\{B,D\}|}{|\{B,D,E,F\}|} = \frac{2}{4} = \frac{1}{2}$$




```
In: L = list(nx.jaccard_coefficient(G))
In: L.sort(key=operator.itemgetter(2), reverse = True)
In: print(L)
Out: [('I', 'H', 1.0), ('A', 'C', 0.5), ('E', 'I', 0.3333333333333333), ('E', 'H',
0.3333333333333333), ('F', 'I', 0.3333333333333333), ('F', 'H',
0.3333333333333333), ('A', 'F', 0.2), ('C', 'E', 0.2), ('B', 'E', 0.2), ('B', 'F',
0.2), ('E', 'D', 0.2), ('D', 'F', 0.2), ('A', 'G', 0.16666666666666666), ('C', 'G',
0.16666666666666666), ('A', 'I', 0.0), ('A', 'H', 0.0), ('C', 'I', 0.0), ('C', 'H',
0.0), ('B', 'I', 0.0), ('B', 'H', 0.0), ('B', 'G', 0.0), ('D', 'I', 0.0), ('D', 'H', 0.0),
('D', 'G', 0.0)]
```

Link prediction

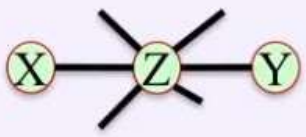
Measure 3: Resource Allocation

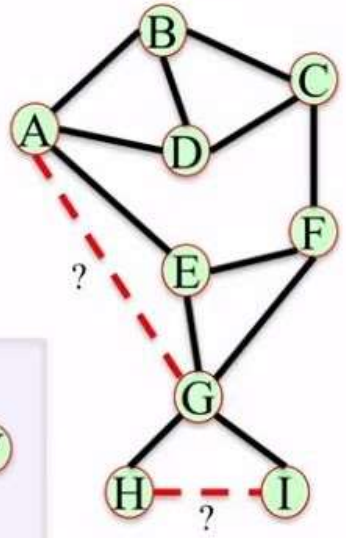
Fraction of a "resource" that a node can send to another through their common neighbors.

The Resource Allocation index of nodes X and Y is

$$\text{res\_alloc}(X, Y) = \sum_{u \in N(X) \cap N(Y)} \frac{1}{|N(u)|}$$

Z has n neighbors
X sends 1 unit to Z, Z distributes the unit evenly among all neighbors
→ Y receives $1/n$ of the unit.





$$\text{res\_alloc}(A, C) = \frac{1}{3} + \frac{1}{3} = \frac{2}{3}$$

```
In: L = list(nx.resource_allocation_index(G))
In: L.sort(key=operator.itemgetter(2), reverse = True)
In: print(L)
Out: [('A', 'C', 0.6666666666666666), ('A', 'G',
0.3333333333333333), ('A', 'F', 0.3333333333333333), ('C', 'E',
0.3333333333333333), ('C', 'G', 0.3333333333333333), ('B', 'E',
0.3333333333333333), ('B', 'F', 0.3333333333333333), ('E', 'D',
0.3333333333333333), ('D', 'F', 0.3333333333333333), ('E', 'I', 0.25),
('E', 'H', 0.25), ('F', 'I', 0.25), ('F', 'H', 0.25), ('I', 'H', 0.25), ('A', 'I', 0),
('A', 'H', 0), ('C', 'I', 0), ('C', 'H', 0), ('B', 'I', 0), ('B', 'H', 0), ('B', 'G', 0),
('D', 'I', 0), ('D', 'H', 0), ('D', 'G', 0)]
```

Link prediction

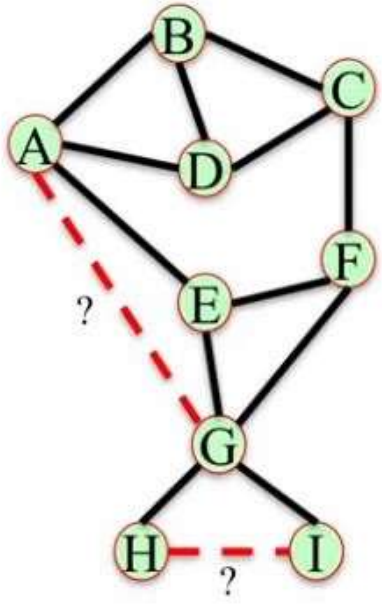
Measure 4: Adamic-Adar Index

Similar to resource allocation index, but with log in the denominator.

The Adamic-Adar index of nodes X and Y is

$$\text{adamic\_adar}(X, Y) = \sum_{u \in N(X) \cap N(Y)} \frac{1}{\log(|N(u)|)}$$

$$\text{adamic\_adar}(A, C) = \frac{1}{\log(3)} + \frac{1}{\log(3)} = 1.82$$



Link prediction

Measure 5: Pref. Attachment

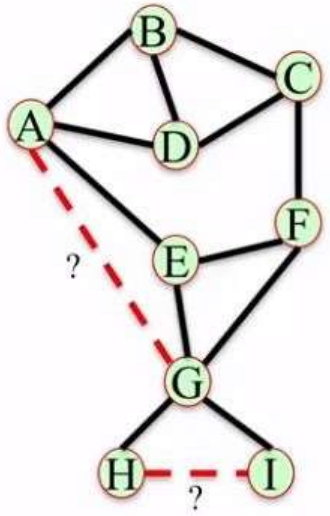
In the preferential attachment model, nodes with high degree get more neighbors.

Product of the nodes' degree.

The preferential attachment score of nodes X and Y is

$$\text{pref\_attach}(X, Y) = |N(X)||N(Y)|$$

$$\text{pref\_attach}(A, C) = 3 * 3 = 9$$



Product of the nodes' degree.

```
In: L = list(nx.preferential_attachment(G))
In: L.sort(key=operator.itemgetter(2), reverse = True)
In: print(L)
Out: [('A', 'G', 12), ('C', 'G', 12), ('B', 'G', 12), ('D', 'G', 12), ('A', 'C',
9), ('A', 'F', 9), ('C', 'E', 9), ('B', 'E', 9), ('B', 'F', 9), ('E', 'D', 9), ('D', 'F',
9), ('A', 'I', 3), ('A', 'H', 3), ('C', 'I', 3), ('C', 'H', 3), ('B', 'I', 3), ('B', 'H',
3), ('E', 'I', 3), ('E', 'H', 3), ('D', 'I', 3), ('D', 'H', 3), ('F', 'I', 3), ('F', 'H',
3), ('I', 'H', 1)]
```

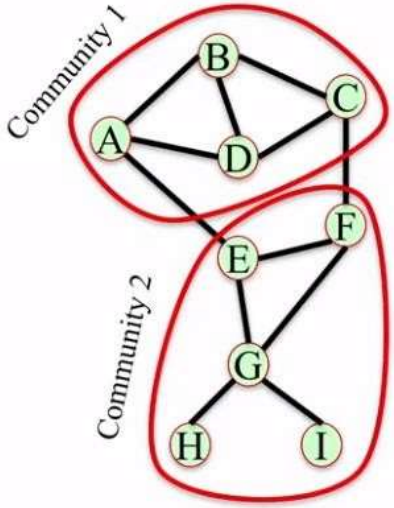
Link prediction

Community Structure

Some measures consider the community structure of the network for link prediction.

Assume the nodes in this network belong to different communities (sets of nodes).

Pairs of nodes who belong to the same community and have many common neighbors in their community are likely to form an edge.



Number of common neighbors with bonus for neighbors in same community.

$$\text{cn\_soundarajan\_hopcroft}(A, C) = 2 + 2 = 4$$
$$\text{cn\_soundarajan\_hopcroft}(E, I) = 1 + 1 = 2$$
$$\text{cn\_soundarajan\_hopcroft}(A, G) = 1 + 0 = 1$$

Assign nodes to communities with attribute node "community"

```
G.node['A']['community'] = 0
G.node['B']['community'] = 0
G.node['C']['community'] = 0
G.node['D']['community'] = 0
G.node['E']['community'] = 1
G.node['F']['community'] = 1
G.node['G']['community'] = 1
G.node['H']['community'] = 1
G.node['I']['community'] = 1
```

Link Prediction

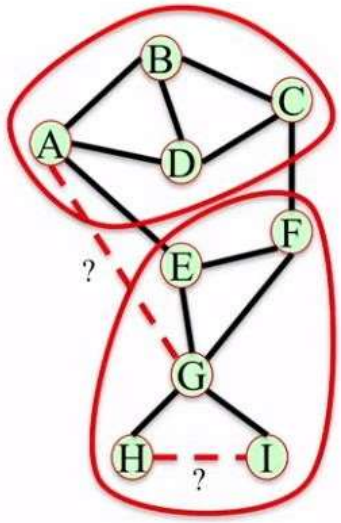
Measure 7: Community Resource Allocation

Similar to resource allocation index, but only considering nodes in the same community

$$\text{ra\_soundarajan\_hopcroft}(A, C) = \frac{1}{3} + \frac{1}{3} = \frac{2}{3}$$

$$\text{ra\_soundarajan\_hopcroft}(E, I) = \frac{1}{4}$$

$$\text{ra\_soundarajan\_hopcroft}(A, G) = 0$$




```python
def get_community_resource_allocation(G):
    cra = list(nx.ra_index_soundarajan_hopcroft(G))
    cra.sort(key=operator.itemgetter(2), reverse=True)
    return cra
```

Out: [('A', 'C', 0.6666666666666666), ('E', 'I', 0.25), ('E', 'H', 0.25), ('F', 'I', 0.25), ('F', 'H', 0.25), **('I', 'H', 0.25)**, ('A', 'I', 0), ('A', 'H', 0), **('A', 'G', 0)**, ('A', 'F', 0), ('C', 'I', 0), ('C', 'H', 0), ('C', 'E', 0), ('C', 'G', 0), ('B', 'I', 0), ('B', 'H', 0), ('B', 'E', 0), ('B', 'G', 0), ('B', 'F', 0), ('E', 'D', 0), ('D', 'I', 0), ('D', 'H', 0), ('D', 'G', 0), ('D', 'F', 0)]

Link Prediction

Summary

- Link prediction problem: Given a network, predict which edges will be formed in the future.
- 5 basic measures:
  - Number of Common Neighbors
  - Jaccard Coefficient
  - Resource Allocation Index
  - Adamic-Adar Index
  - Preferential Attachment Score
- 2 measures that require community information:
  - Common Neighbor Soundarajan-Hopcroft Score

Communities in NetworkX

https://media.readthedocs.org/pdf/python-louvain/latest/python-louvain.pdf

References

https://www.coursera.org/learn/python-social-network-analysis