NetworkX:
Network Analysis with Python

Ilias Dimitriadis – idimitriad@csd.auth.gr
Vasileios Souvatzis – vasisouv@csd.auth.gr

DataLab – CSD

Outline

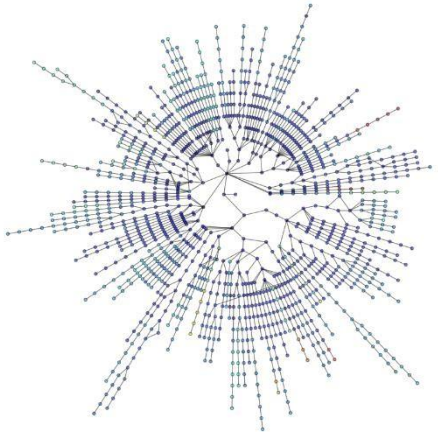
1. Introduction to NetworkX
2. Getting started with Python and NetworkX
3. Basic network analysis
4. Writing your own code
5. Ready for your own analysis!

1. Introduction to NetworkX

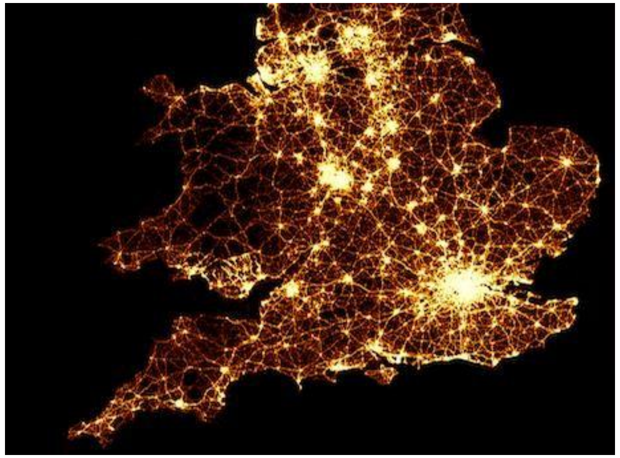
Introduction: networks are everywhere…

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 C library LAPACK for very fast implementation.

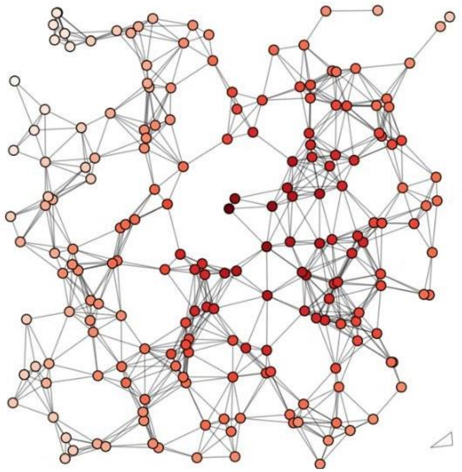
Matplotlib is the **primary plotting library** in Python.

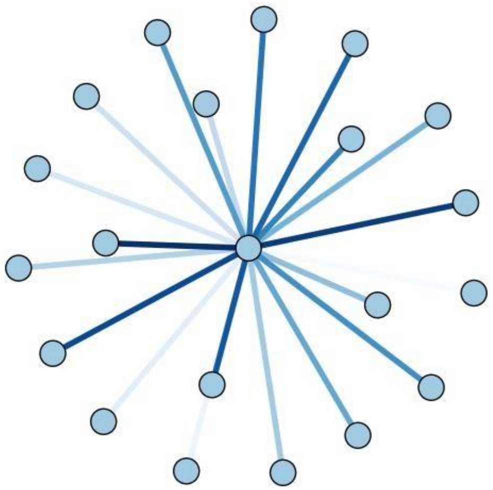
Supports 2-D and 3-D plotting. All plots are highly customisable and ready for professional publication.

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 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

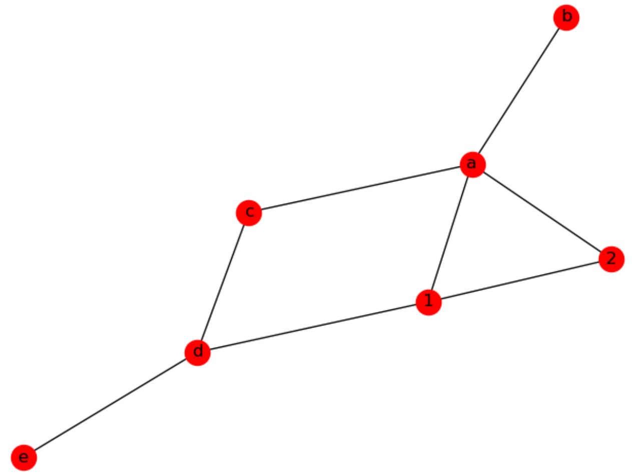
Large-scale problems that require faster approaches (i.e. massive networks with 100M/1B edges)

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

Visualization of networks is better handled by other professional tools

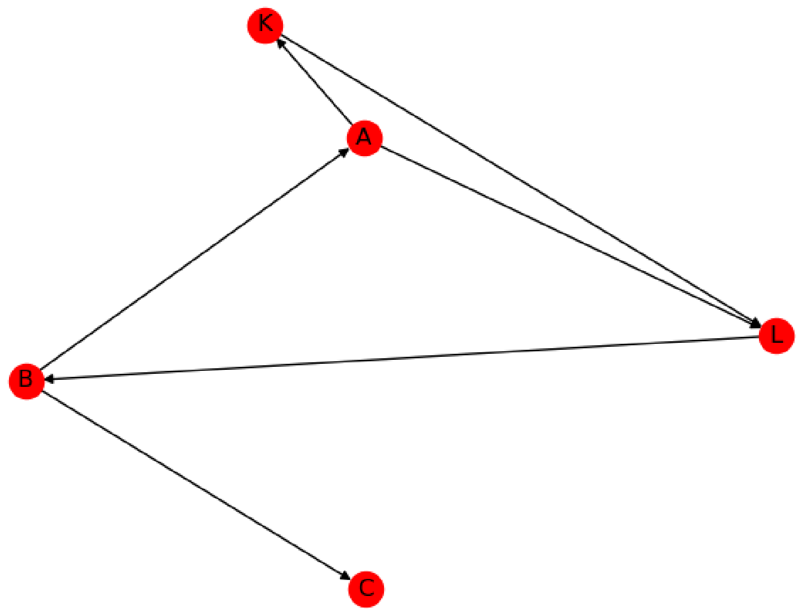
Introduction: a quick example

```
# 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
```



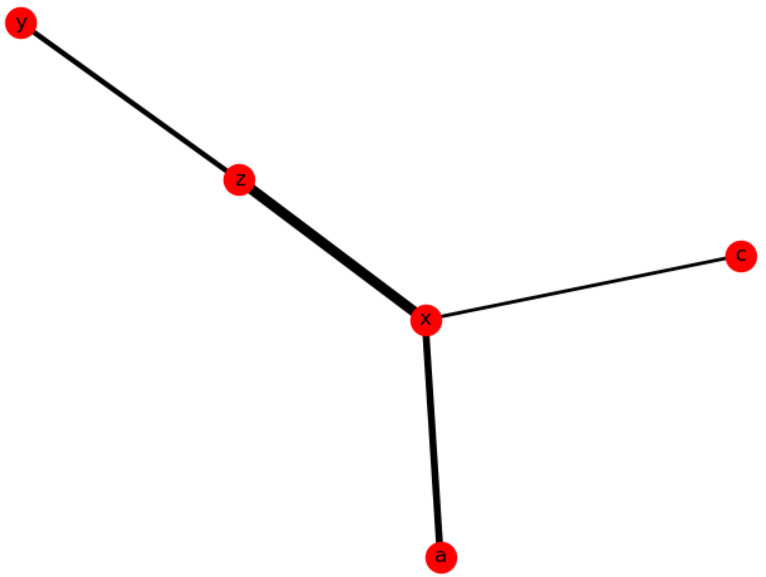
Introduction: a quick example

```
#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
```



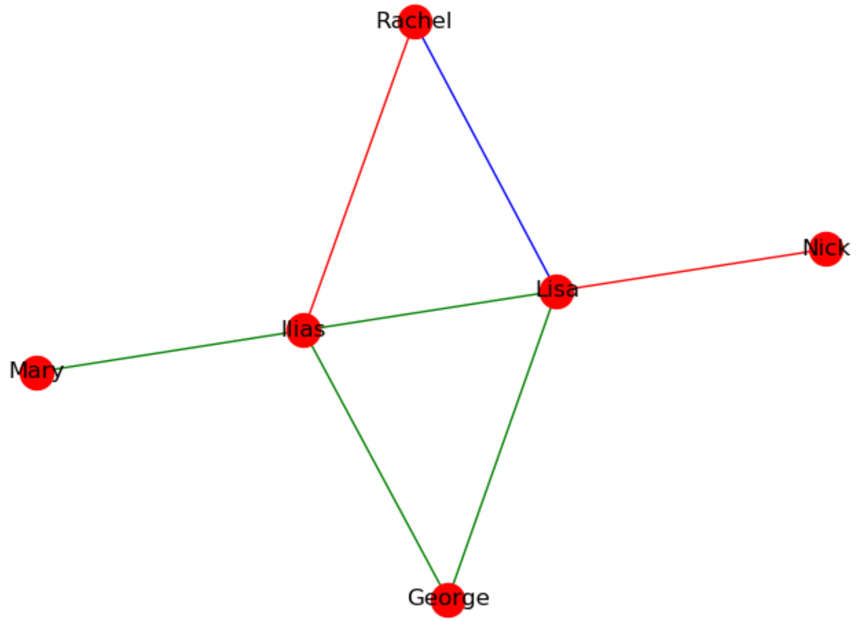
Introduction: a quick example

```
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
```



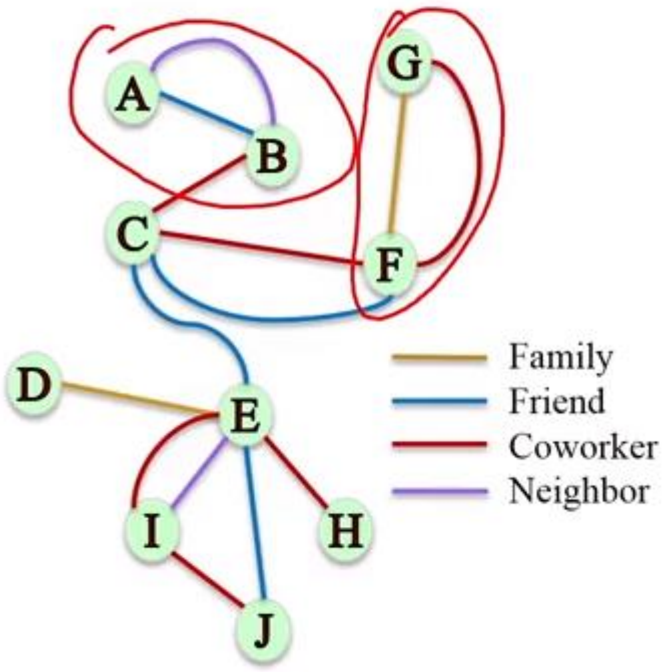
Introduction: a quick example

```
#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
```



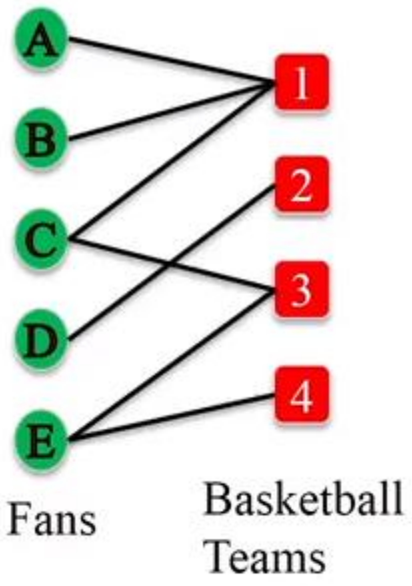
Introduction: a quick example

```python
#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
```

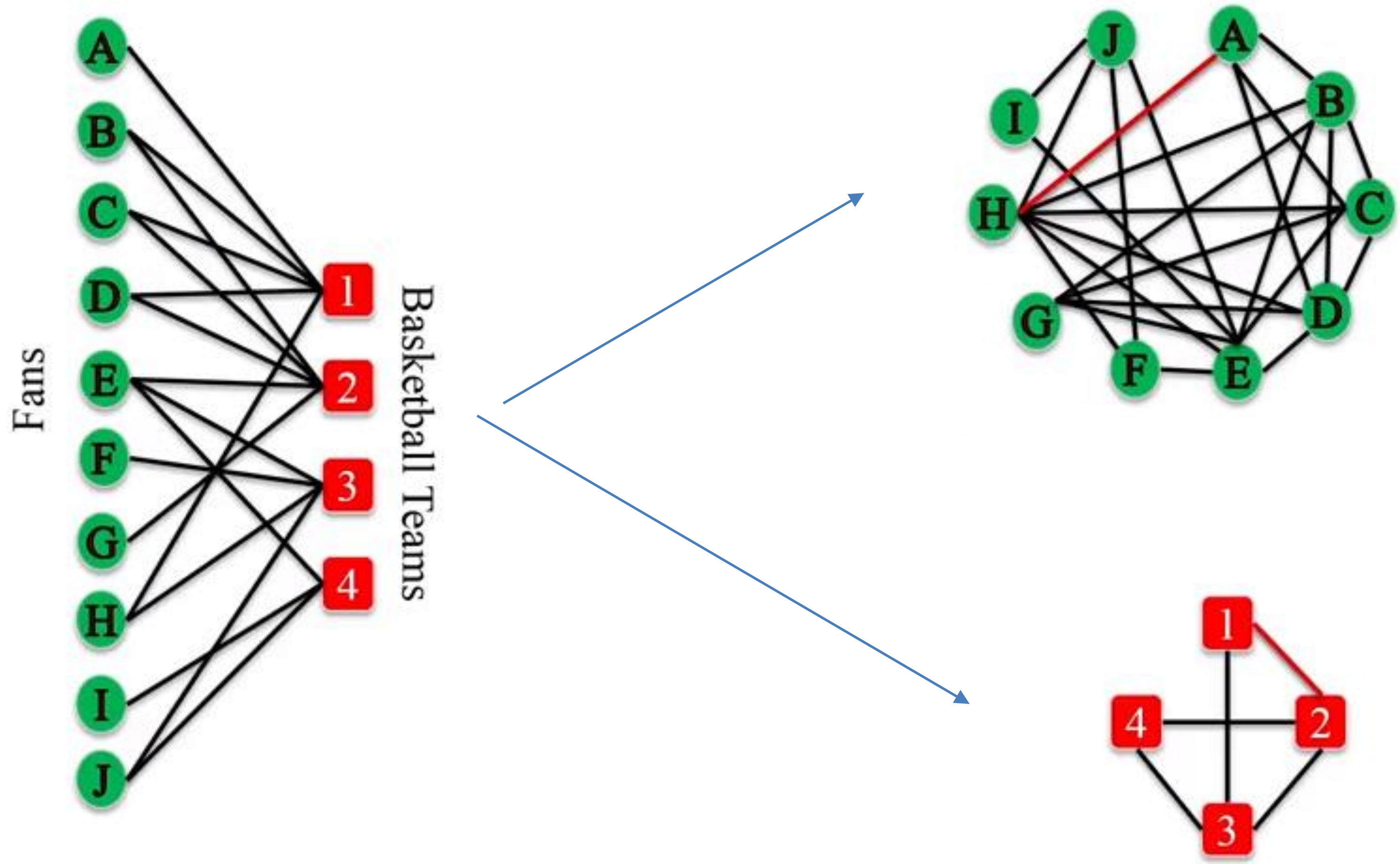


Introduction: a quick example

```python
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
```



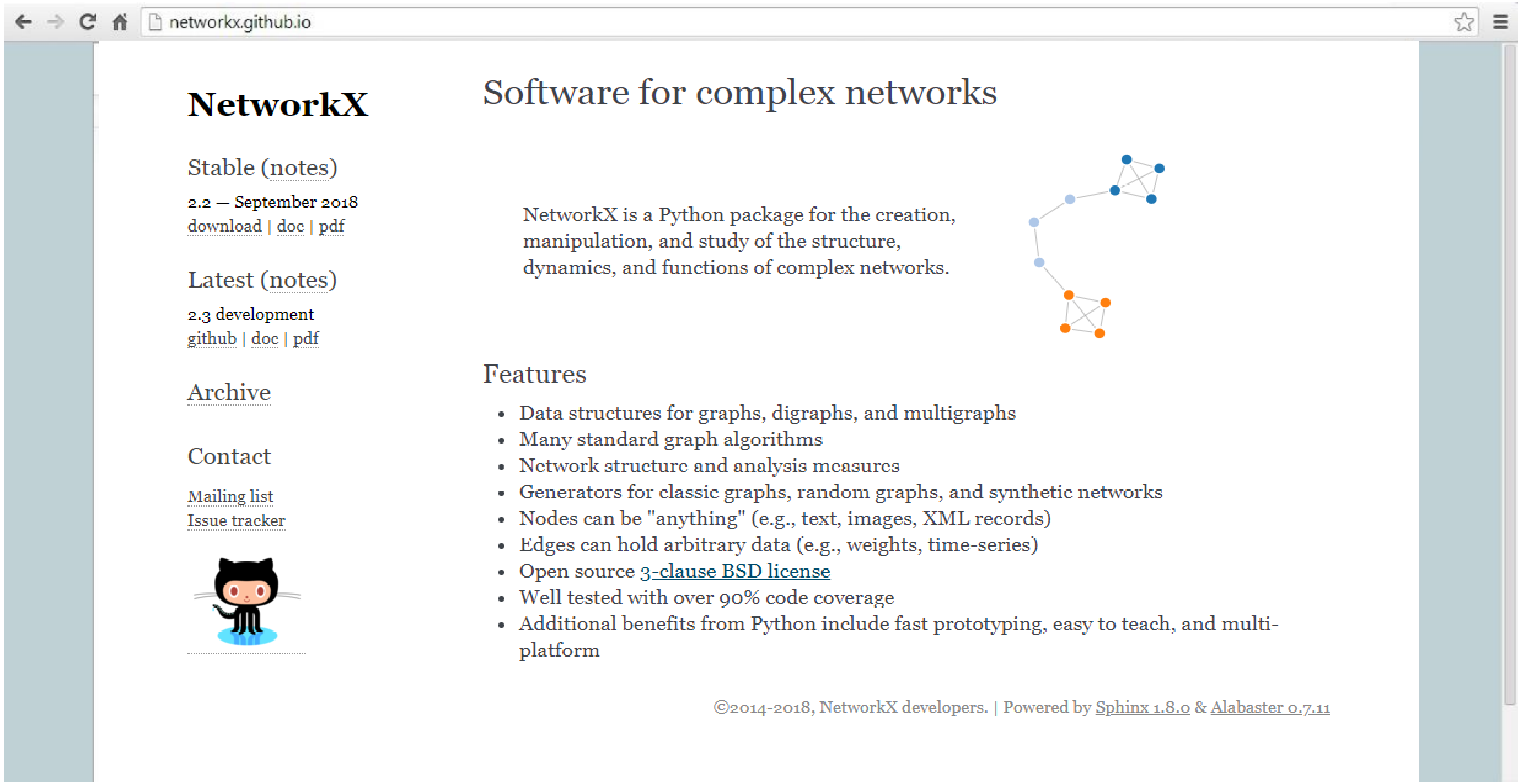
Introduction: a quick example




```python
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/



networkx.github.io

NetworkX

Stable (notes)

2.2 — September 2018

download | doc | pdf

Latest (notes)

2.3 development

github | doc | pdf

Archive

Contact

Mailing list

Issue tracker

Software for complex networks

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

Features

- 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 3-clause BSD license
- Well tested with over 90% code coverage
- Additional benefits from Python include fast prototyping, easy to teach, and multi-platform

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:

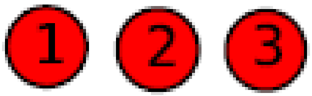
```
>>> g = nx.Graph() # empty graph
```

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

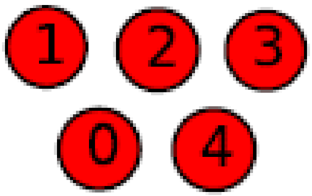
Getting started: adding nodes

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

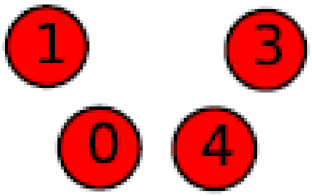

```
# A list of nodes
>>> g.add_nodes_from([2, 3])
```




```
# 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)
```

```
# 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

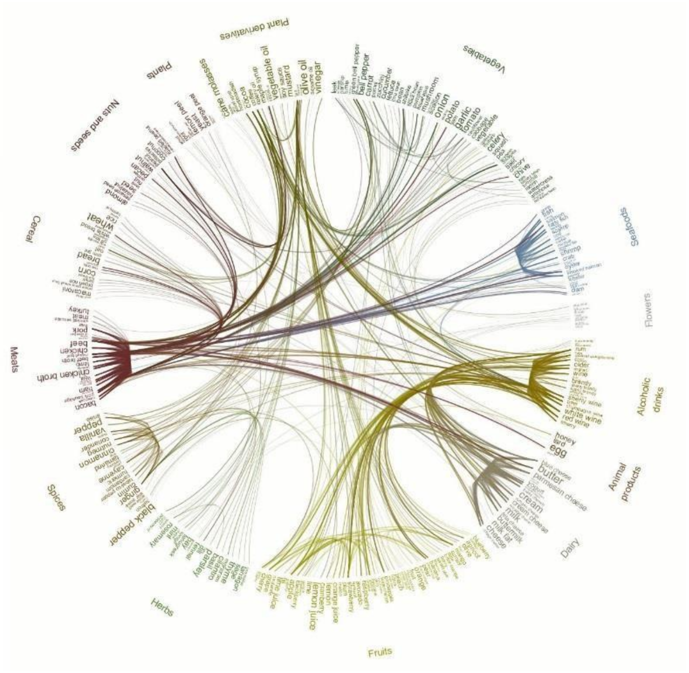
```
#using an edgelist with weights
def load_from_edgelist(file):
    G2 = nx.read_edgelist('graphs/'+file, data=[('Weight', int)])
    print (nx.info(G2))
    print (G2.edges(data=True))
    return G2
```

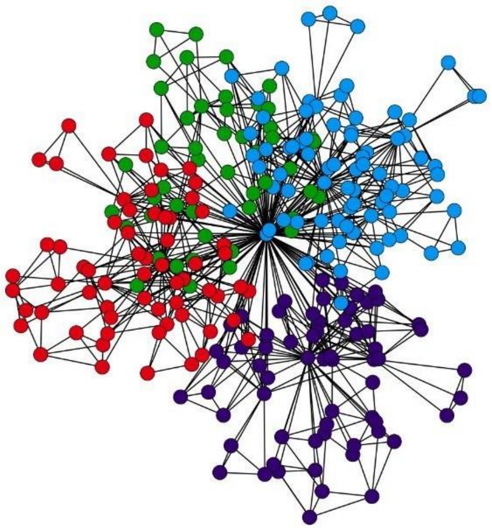
 # from pandas <<powerful Python data analysis toolkit>>

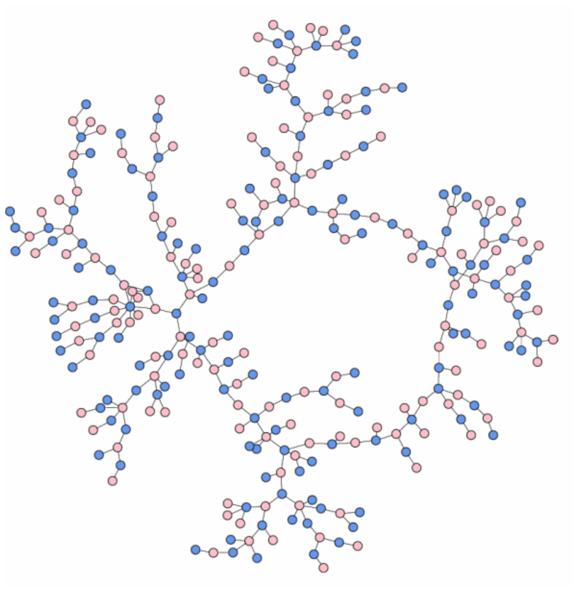
```
#using a pandas dataframe
def load_from_pandas(file):
    df=pd.read_csv('graphs/'+file,delim_whitespace=True,header=None,names=['n1','n2','weight'])
    print (df.head(3))
    G3 = nx.from_pandas_edgelist(df,'n1','n2',edge_attr='weight')
    print (nx.info(G3))
    return G3
```

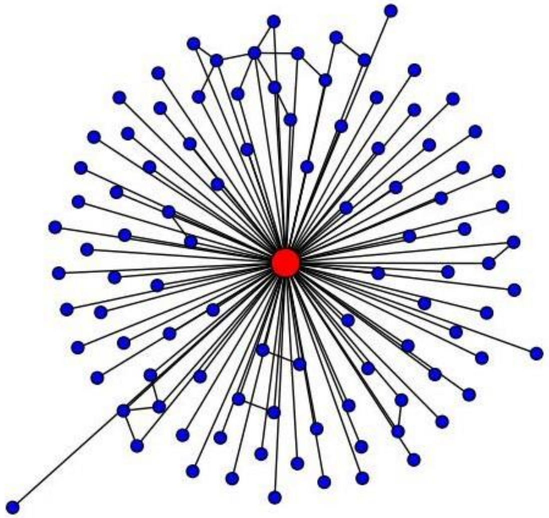
Introduction: drawing and plotting

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

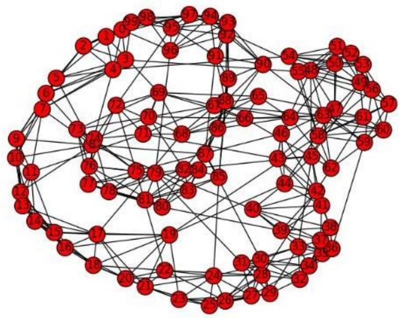


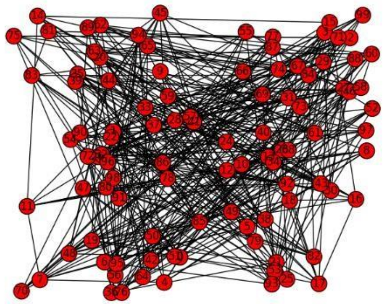


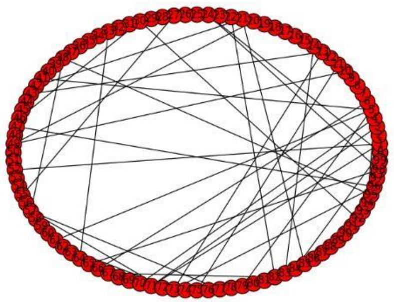




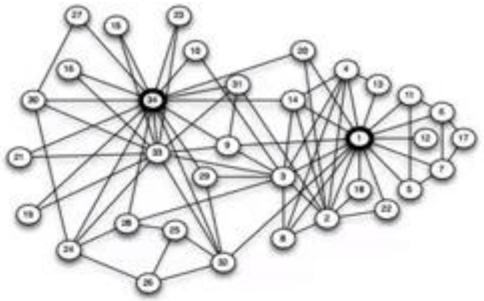
Getting started: drawing graphs



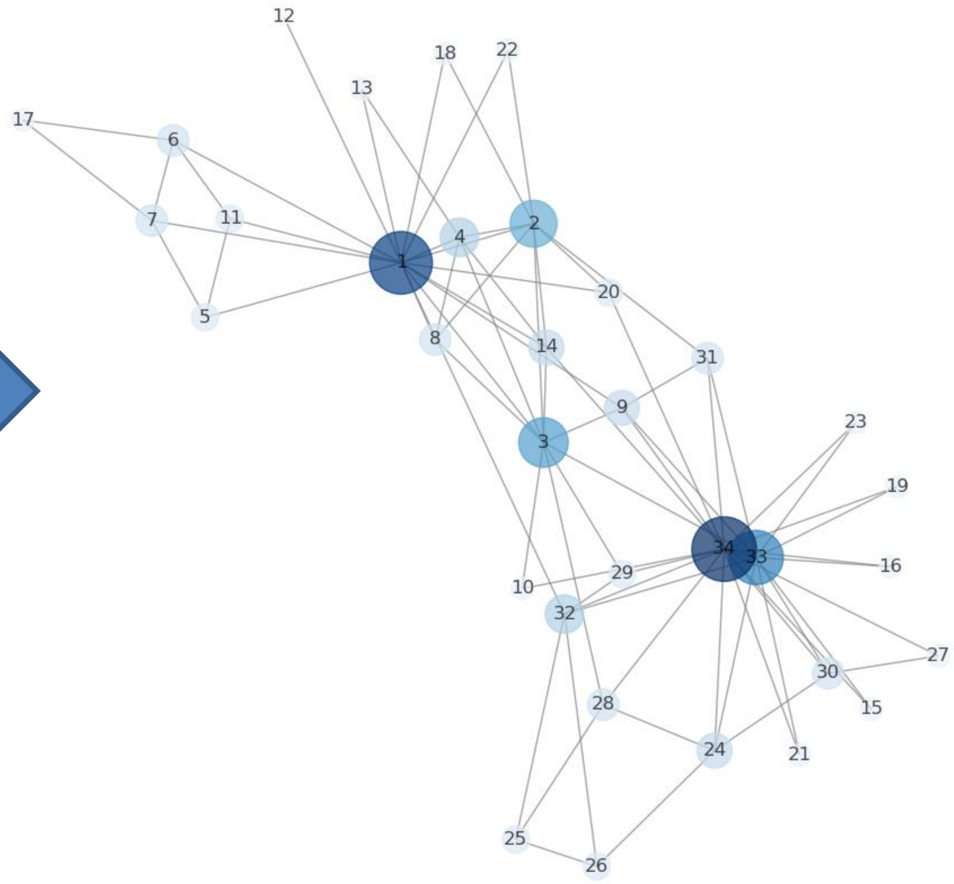




Getting started: drawing graphs



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:

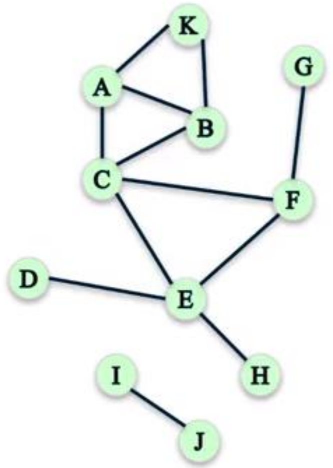
$$\frac{\#\text{ of pairs of } C'\text{s friends who are friends}}{\#\text{ of pairs of } C'\text{s friends}}$$

# 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 = $\frac{2}{6} = \frac{1}{3}$

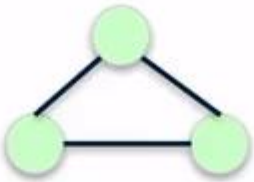



```
def clus_coef_v1(G,each_node): #Fraction of pairs of the 
    if each_node==True:
        for node in G.nodes():
            print (nx.clustering(G,node))
    print ('Average with v1:',nx.average_clustering(G))
```

Transitivity

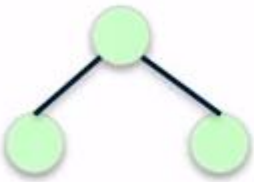
Percentage of "open triads" that are triangles in a network.

Triangles:



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

Open triads:



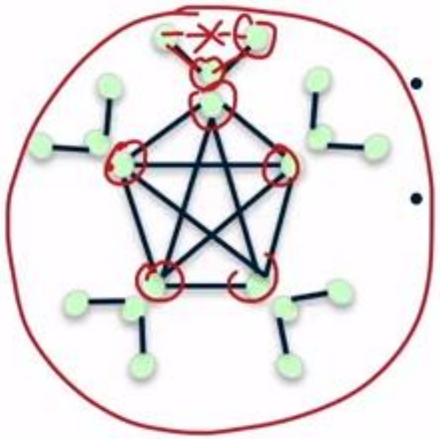
```
def clus_coef_v2(G): #Ratio of numbers of triangles and number of "open triads"
    print ('Average with v2: ',nx.transitivity(G))
```

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

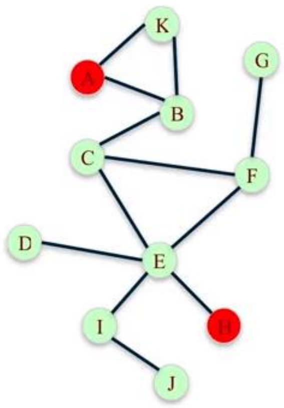
Distance

How far is node A from node H?

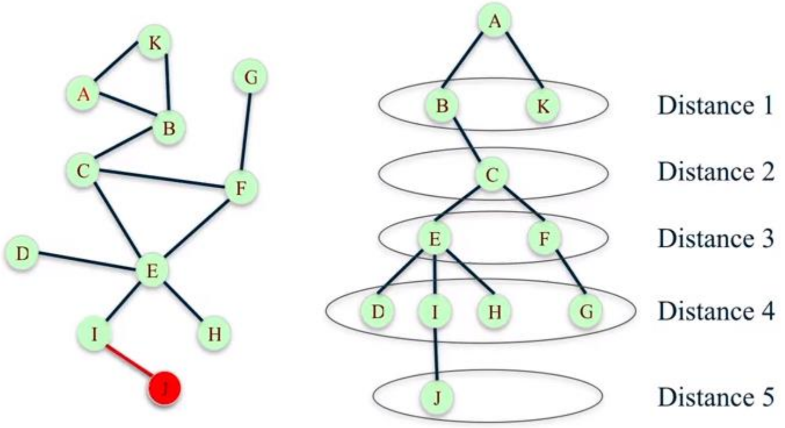
Path 1: A – B – C – E – H (4 "hops")

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

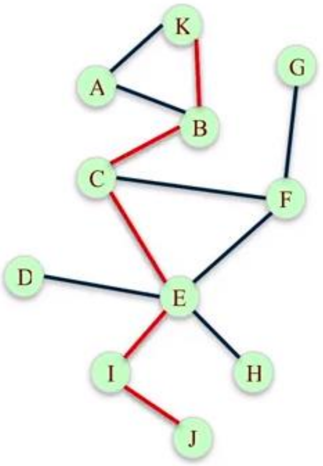
How to characterize the distance between all pairs of nodes in a graph?

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




```python
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
```

```python
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.

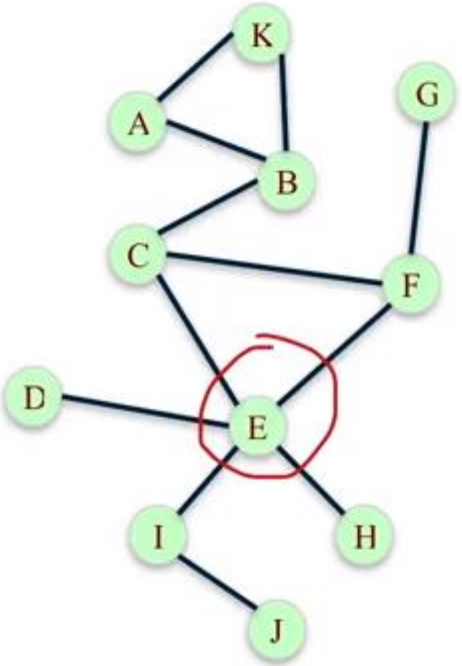
In: nx.eccentricity(G)

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




```python
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
```

```python
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)
```

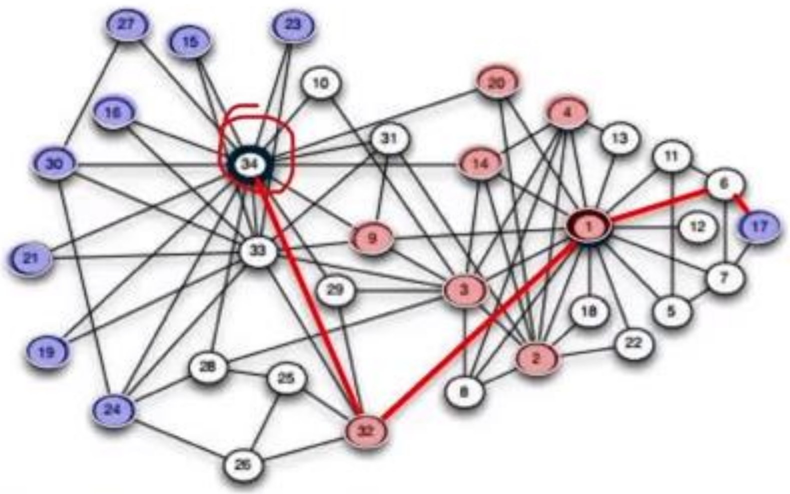
Average shortest path = 2.41

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

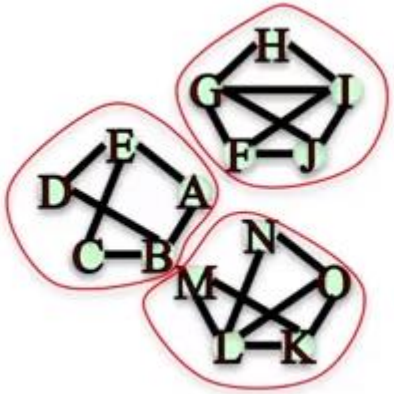
Connected Components

Undirected Graphs

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

Connected components

nx.connected_components(G)

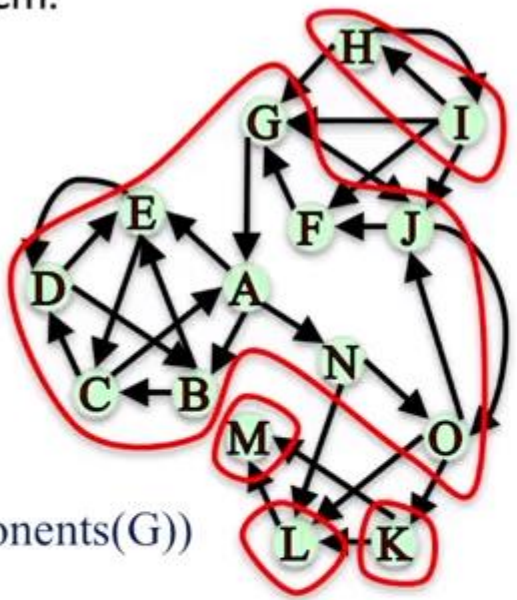


Directed Graphs

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

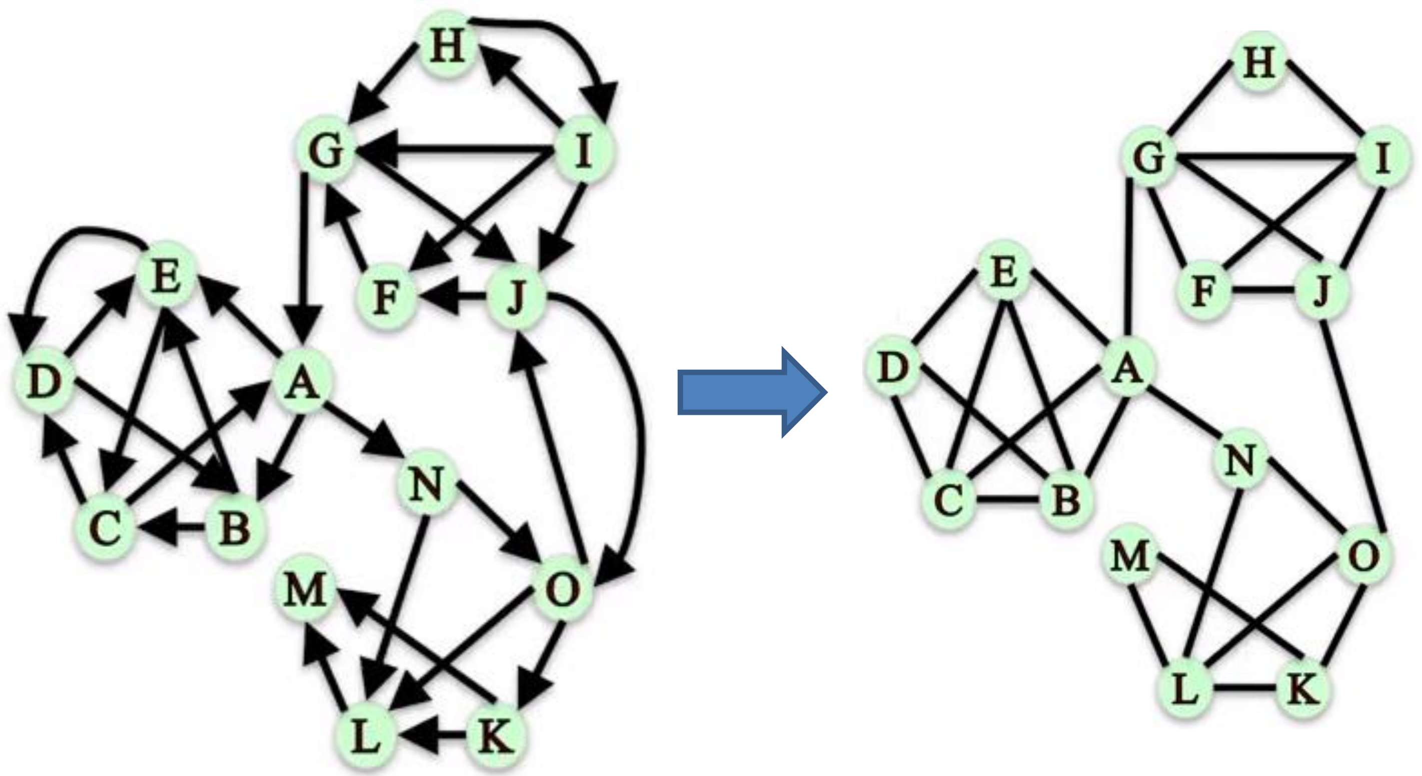
Strongly connected components

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.

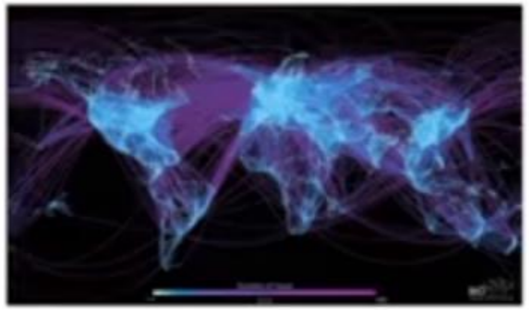


Network Robustness

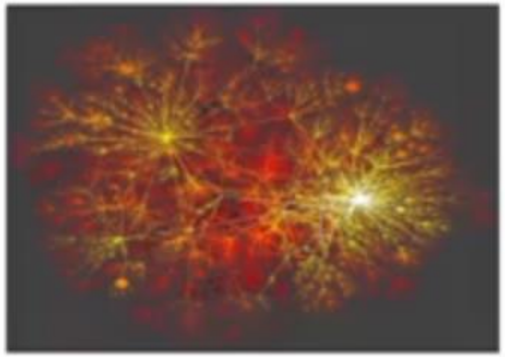
Network robustness: the ability of a network to maintain its general structural properties when it faces failures or attacks.

Attacks?

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



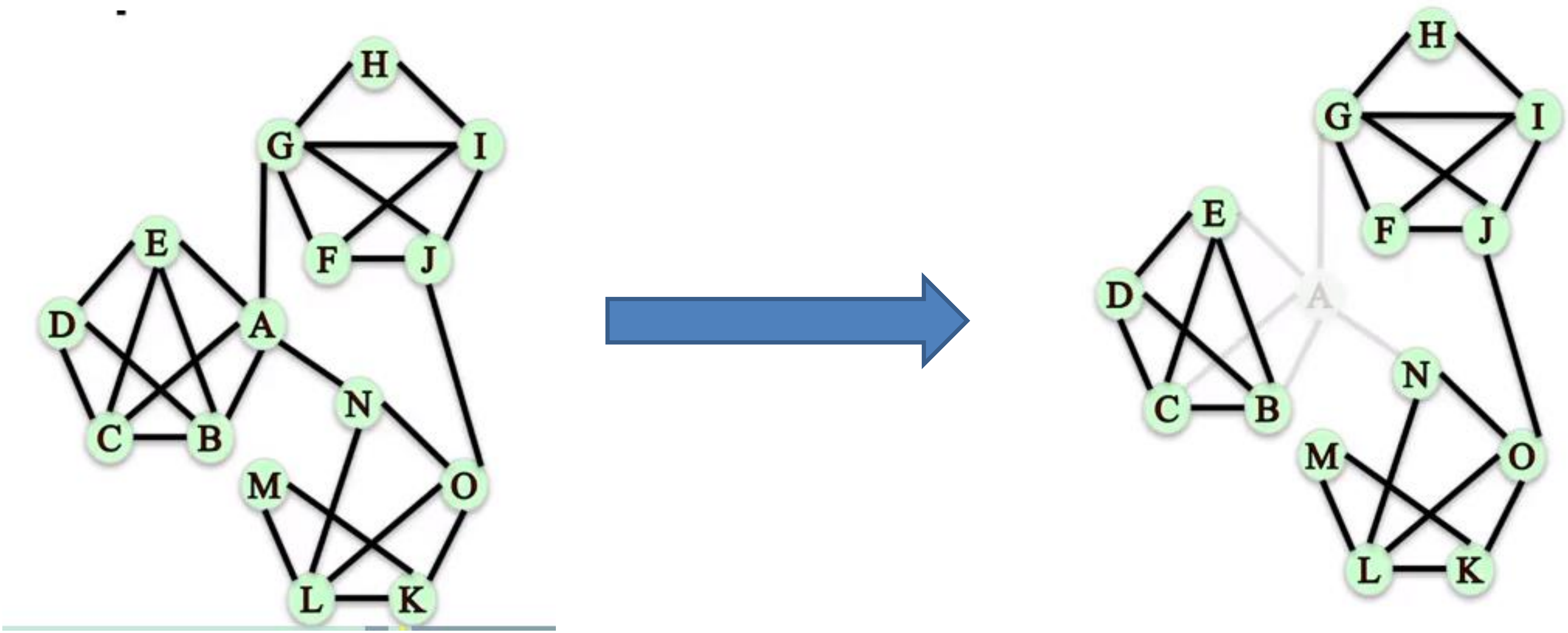
Network of direct flights around the world [Bio.Diaspora]



Internet Connectivity [K. C. Claffy]

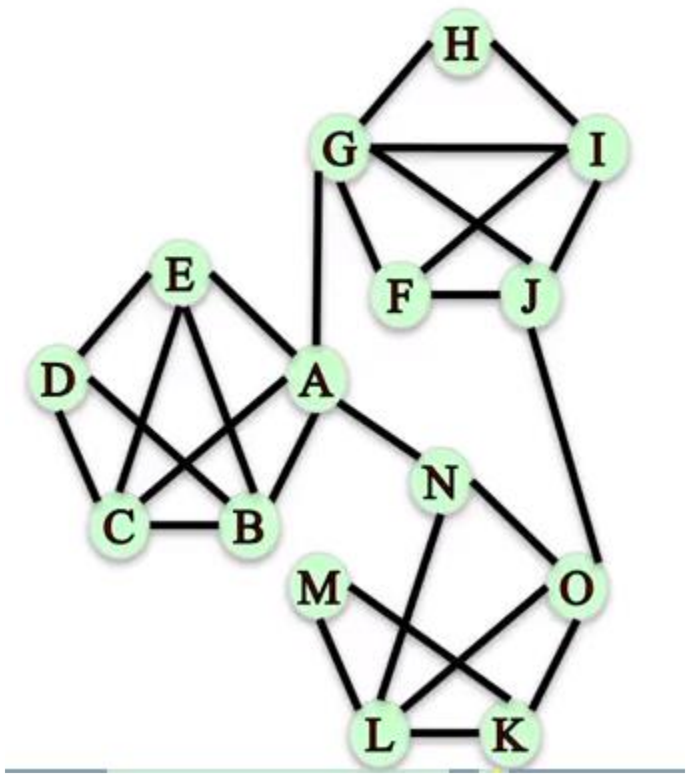
Network Robustness

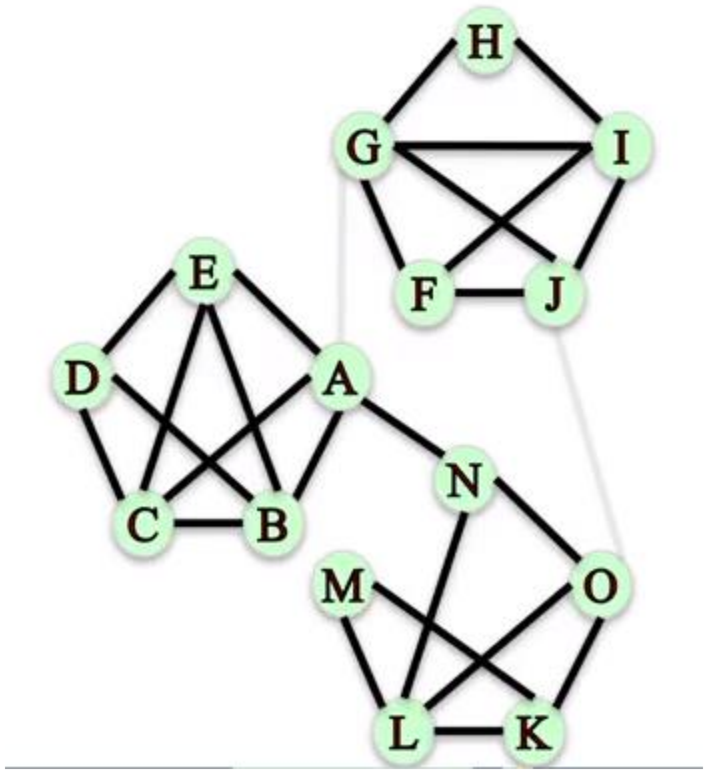
```
def get_smallest_number_of_nodes_to_disconnect(G_un):
    print (nx.node_connectivity(G_un),nx.minimum_node_cut(G_un))
    return nx.node_connectivity(G_un), nx.minimum_node_cut(G_un)
```



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)
```



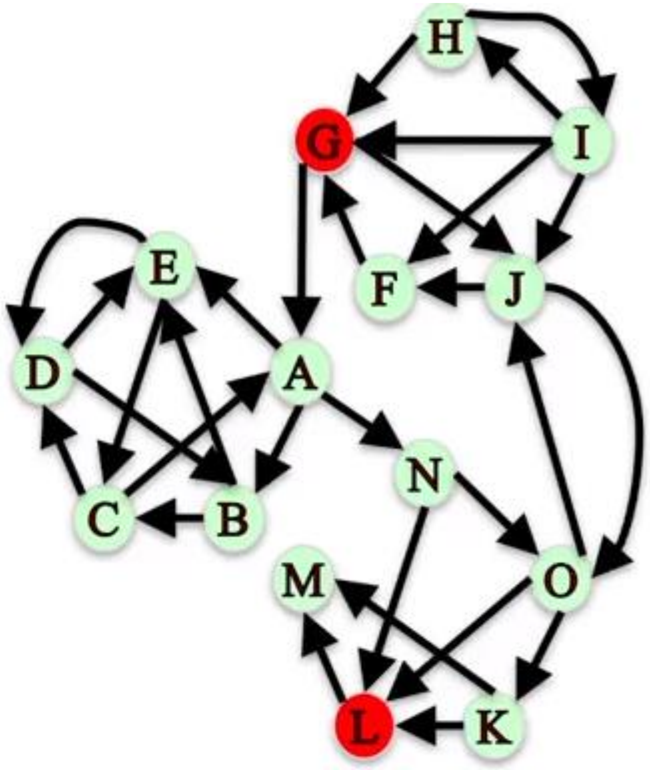


Network Robustness

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?

```python
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.

```
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.

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

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.

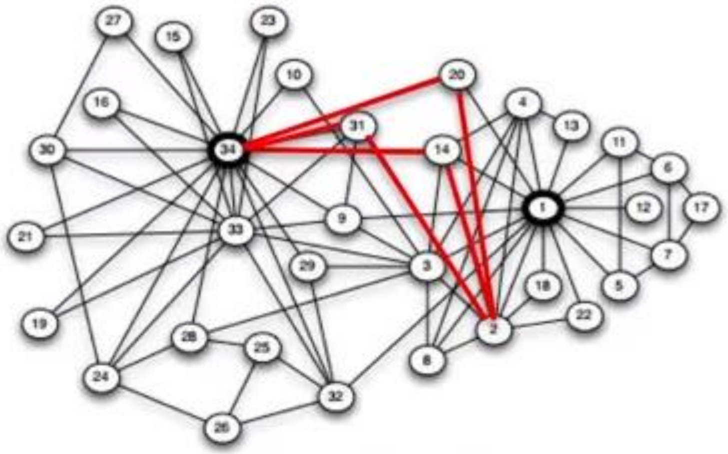
Recall: the distance between two nodes is the length of the shortest path between them.

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.

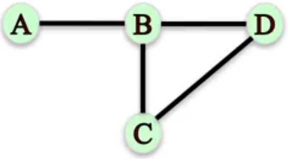
$\sigma_{s,t}(v)$ = the number shortest paths between nodes s and t *that pass through node* v.

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:

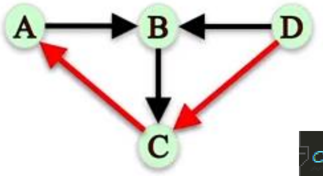
$$C_{btw}(B) = \frac{\sigma_{A,B}(B)}{\sigma_{A,B}} + \frac{\sigma_{A,C}(B)}{\sigma_{A,C}} + \frac{\sigma_{A,D}(B)}{\sigma_{A,D}} + \frac{\sigma_{B,C}(B)}{\sigma_{B,C}} + \frac{\sigma_{B,D}(B)}{\sigma_{B,D}} + \frac{\sigma_{C,D}(B)}{\sigma_{C,D}} = \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1}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

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?

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

```python
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]
```