Python for Complex Network Analysis

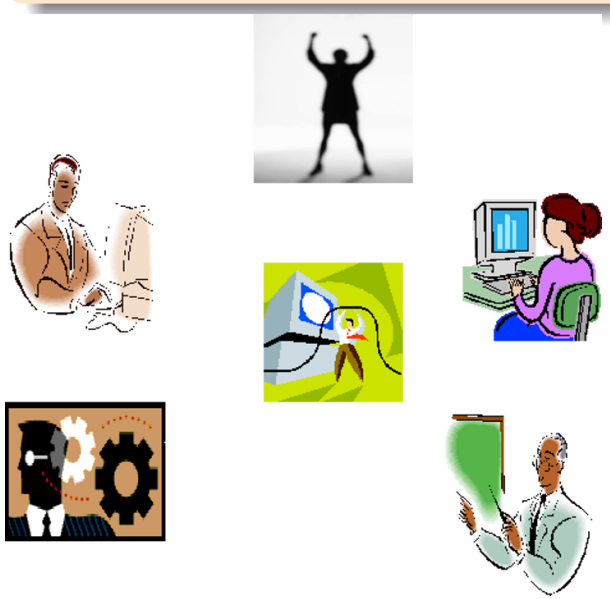
Sajeev G P

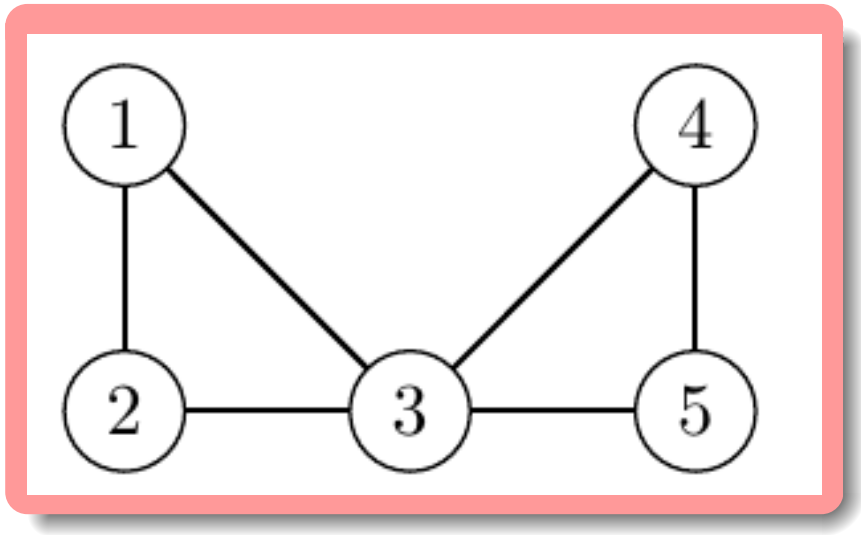
January 6, 2020

Python for Complex Network Analysis

The Course

- Course Code: 18CS387/1
- Course Name: Python for Complex Network Analysis
- Credits: 2
- Class: Monday 2.00-5.00 PM, S301





Course Contents I

Topics

- Complex Networks
- Social Networks
- Python
  - Anaconda
  - NetworkX
  - iGraph
  - Gephi

Outline

- Friendship Networks
- NetworkX
  - Graphs
  - Measures
  - Algorithms
- Case Study

Course Contents II

Network Graph

- Revisiting Graph theory
  - Paths
  - Cycles
  - Connectivity
  - Components
  - Distance
  - Search

Network Graph ..

- More on Graph theory
  - Connectedness
  - Degree Distribution
  - Centrality
  - Cliques
  - Clusters

Books and References

1. Zinoviev, D. (2018). *Complex Network Analysis in Python: Recognize-Construct-Visualize-Analyze-Interpret*. Pragmatic Bookshelf.
2. `https://networkx.github.io/documentation/stable/auto_examples/index.html#algorithms`

Course Contents III

Online

- https://www.coursera.org/learn/python-social-network-analysis
- https://www.datacamp.com/courses/network-analysis-in-python-part-1

Evaluation

Internal Evaluation

- One Class Test (Periodical)- 15 marks
- Written Assignments, Quiz, Viva, Class Performance- 15 Marks
- Programming Assignments- 10 Marks
- Course Project- 40

End Semester Exam

- Computer Based Test- 20 Marks

Basic Example

```
>>> import networkx as nx
>>> G = nx.Graph()
>>> G.add_node("spam")
>>> G.add_edge(1,2)
>>> print(G.nodes())
[1, 2, 'spam']
>>> print(G.edges())
[(1, 2)]
```

Graph Types

- Graph : Undirected simple (allows self loops)
- DiGraph : Directed simple (allows self loops)
- MultiGraph : Undirected with parallel edges
- MultiDiGraph : Directed with parallel edges
- can convert to undirected: `g.to_undirected()`
- can convert to directed: `g.to_directed()`

To construct, use standard python syntax:

```
>>> g = nx.Graph()
>>> d = nx.DiGraph()
>>> m = nx.MultiGraph()
>>> h = nx.MultiDiGraph()
```

Adding Nodes

- `add_nodes_from()` takes any iterable collection and any object

```
>>> g = nx.Graph()
>>> g.add_node('a')
>>> g.add_nodes_from( ['b','c','d'])
>>> g.add_nodes_from('xyz')
>>> h = nx.path_graph(5)
>>> g.add_nodes_from(h)
>>> g.nodes()
[0,1,'c','b',4,'d',2,3,5,'x','y','z'
   ]
```

Adding Edges

- Adding an edge between nodes that don't exist will automatically add those nodes
- `add_nodes_from()` takes any iterable collection and any type (anything that has a `__iter__()` method)

```
>>> g = nx.Graph( [('a','b'),('b','c
    '),('c','a')] )
>>> g.add_edge('a', 'd')
>>> g.add_edges_from([('d', 'c'), ('
    d', 'b')])
```

Adding Node and Edge attributes

- Every node and edge is associated with a dictionary from attribute keys to values
- Type indifferent, just needs to be hashable
  - i.e. can't use `list`, must use `tuple`

```
>>> G = nx.Graph()
>>> G.add_node([1,2])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/pymodules/
```

Node attributes

- Can add node attributes as optional arguments along with most add methods

```
>>> g = nx.Graph()
>>> g.add_node(1,name='Obrian')
>>> g.add_nodes_from([2],name='
    Quintana'])
>>> g[1]['name']
'Obrian'
```

Edge attributes

- Can add edge attributes as optional arguments along with most add methods

```
>>> g.add_edge(1, 2, w=4.7 )
>>> g.add_edges_from([(3,4),(4,5)],
    w =3.0)
>>> g.add_edges_from([(1,2,{'val'
    :2.0})])
# adds third value in tuple as '
    weight' attr
>>> g.add_weighted_edges_from
    ([(6,7,3.0)])
>>> g.get_edge_data(3,4)
{'w' : 3.0}
>>> g.add_edge(5,6)
```

Simple Properties

- Number of nodes :

```
c len(g)
>>> g.number_of_nodes()
>>> g.order()
```

- Number of Edges

```
>>> g.number_of_edges()
```

- Check node membership

```
>>> g.has_node(1)
```

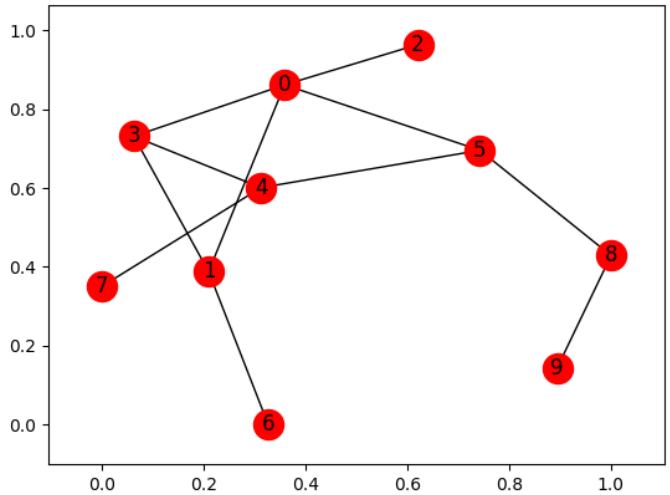
- Check edge presence

```
>>> g.has_edge(1)
```

Loading &Plotting Graphs

- Adjacency List

```
import networkx as nx
G1 = nx.Graph()
# add node/edge pairs
G1.add_edges_from([(0, 1),(0,2)
    ,
                   (0,3),(0,5),
                   (1,3),(1,6),
                   (3,4),(4,5),
                   (4,7),(5,8),
                   (8,9)])
# draw the network G1
nx.draw_networkx(G1)
```



Loading &Plotting Graphs ..

- Create a file for Adjacency List
- `G_adjlist.txt` is the adjaceny list representation of G1.

```
0 1 2 3 5  node 0 is adjacent to nodes 1, 2, 3, 5
1 3 6  node 1 is (also) adjacent to nodes 3, 6
2  node 2 is (also) adjacent to no new nodes
3 4  node 3 is (also) adjacent to node 4
```

```
>>> !cat G_adjlist.txt
>>>G2 = nx.read_adjlist('G1.adjlist')
>>>G3=nx.Graph(G2)
>>>nx.draw_networkx(G2)
```

Loading &Plotting Graphs ..

- Adjacency Matrix

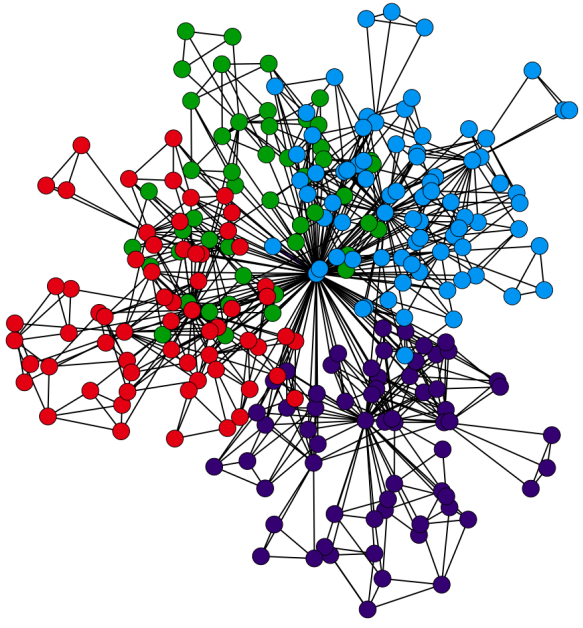
```
import numpy as np
 G_mat =
 np.array([[0, 1, 1, 1, 0, 1, 0, 0, 0, 0],
           [1, 0, 0, 1, 0, 0, 1, 0, 0, 0],
           [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
           [1, 1, 0, 0, 1, 0, 0, 0, 0, 0],
           [0, 0, 0, 1, 0, 1, 0, 1, 0, 0],
           [1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
           [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
           [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
           [0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
           [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]])
G3 = nx.Graph(G_mat)
```

Networks

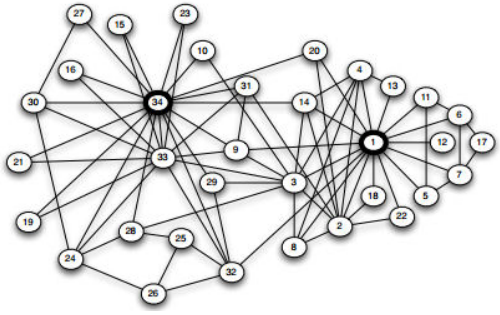
Networks: A set of objects (nodes) with interconnections (edges).

Why study networks?

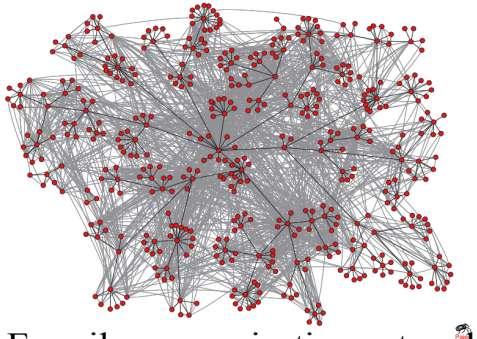
Because they are everywhere!



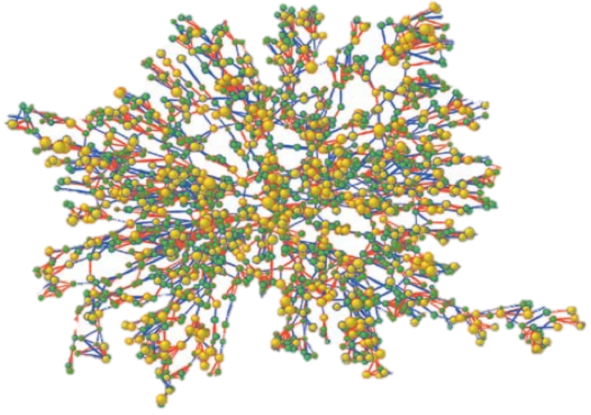
Social Networks



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

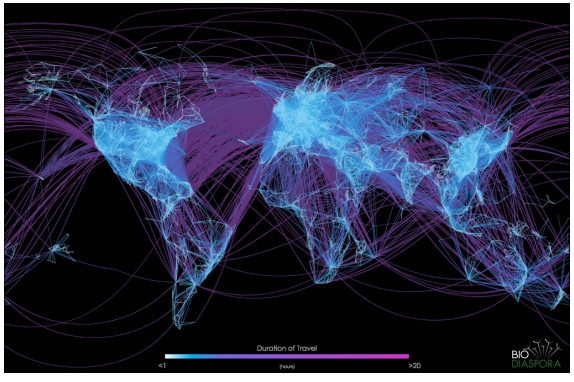


E-mail communication network
among 436 HP employees [Adamic & Adar 2005]



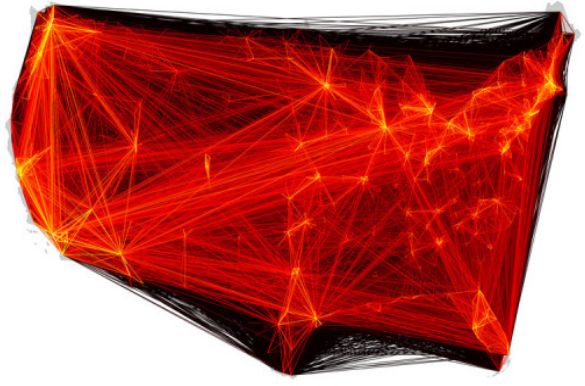
Network of friendship, marital tie, and
family tie among 2200 people
[Christakis & Fowler 2007]

Transportation and Mobility Networks



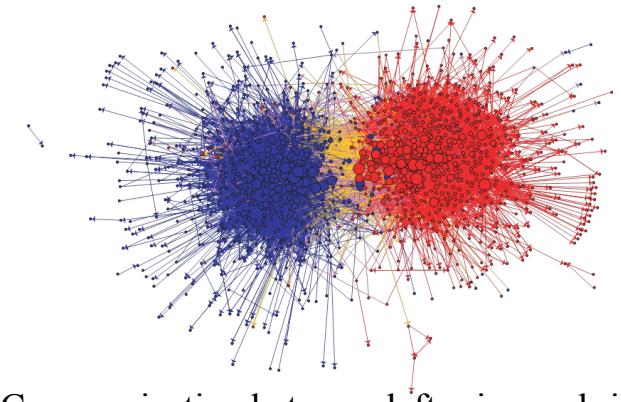
Network of direct flights around the world
[Bio.Diaspora]

Ann Arbor bus transportation network

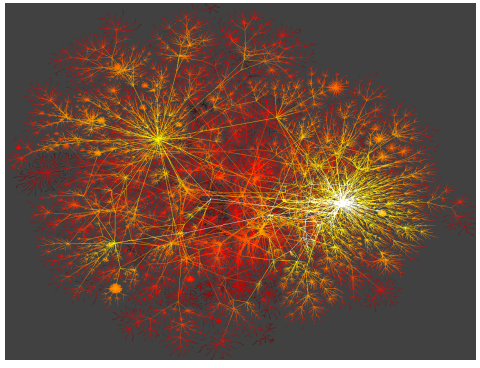


Human mobility network based
on location of dollar bills (Where's George)
[Thiemann et al. 2010]

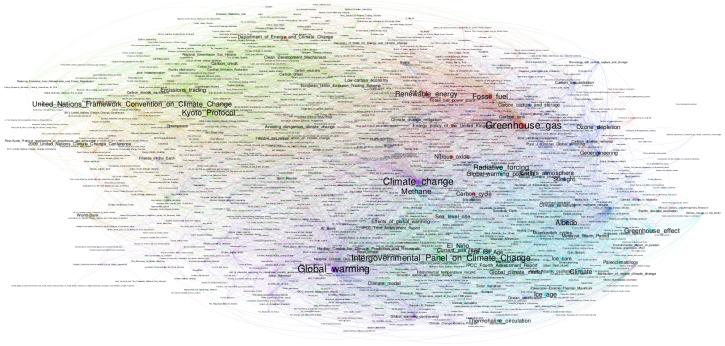
Information Networks



Communication between left-wing and right-wing political blogs [Adamic & Glance 2005]

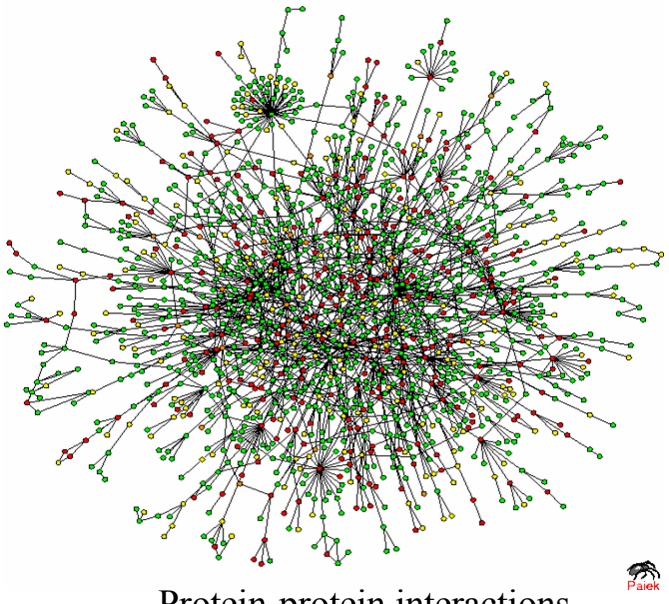


Internet Connectivity [K. C. Claffy]

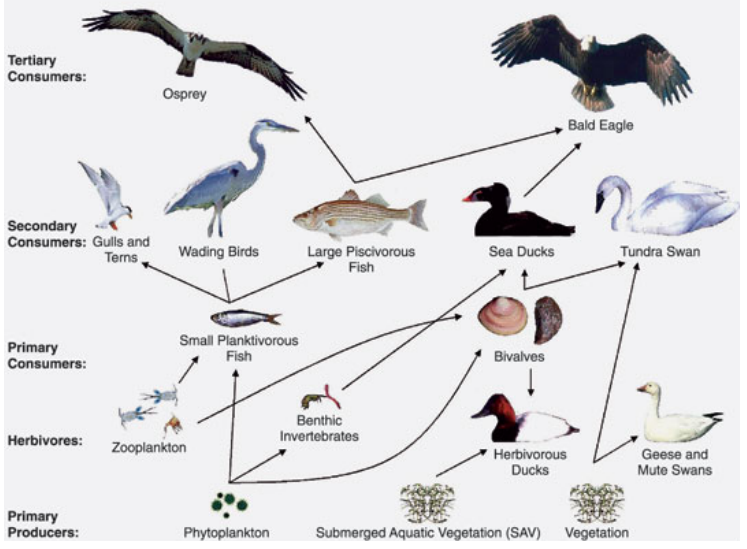


Network of Wikipedia articles about climate change [EMAPS]

Biological Networks



Protein-protein interactions
[Jeong et al. 2001]



Chesapeake Bay Waterbird Food Web
[Perry et al. 2005]

And More…

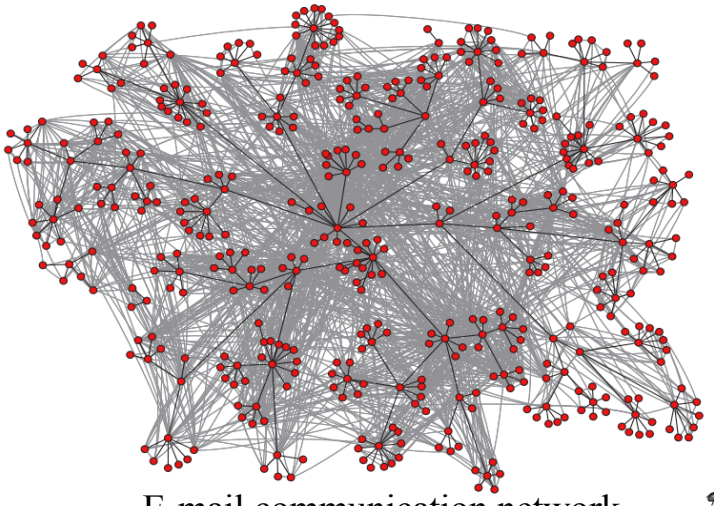
- Financial networks
- Co-authorship networks
- Trade networks
- Citation networks

Networks Applications

Networks are everywhere, but what can we do with them?

Is a rumor likely to spread in this network?

Who are the most influential people in this organization?



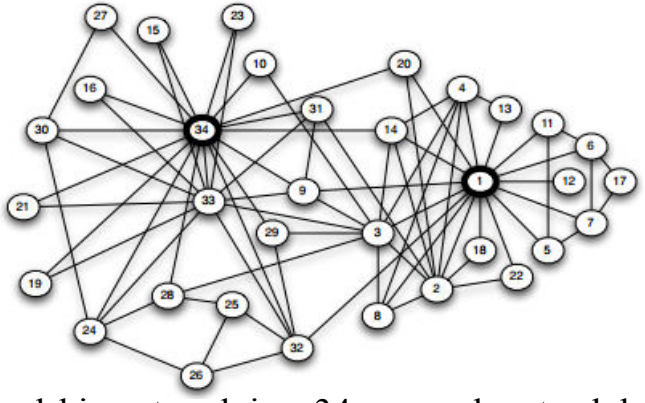
E-mail communication network among 436 HP employees

Networks Applications

Networks are everywhere, but what can we do with them?

Is this club, likely to split into two groups?

If so, which nodes will go to which group?



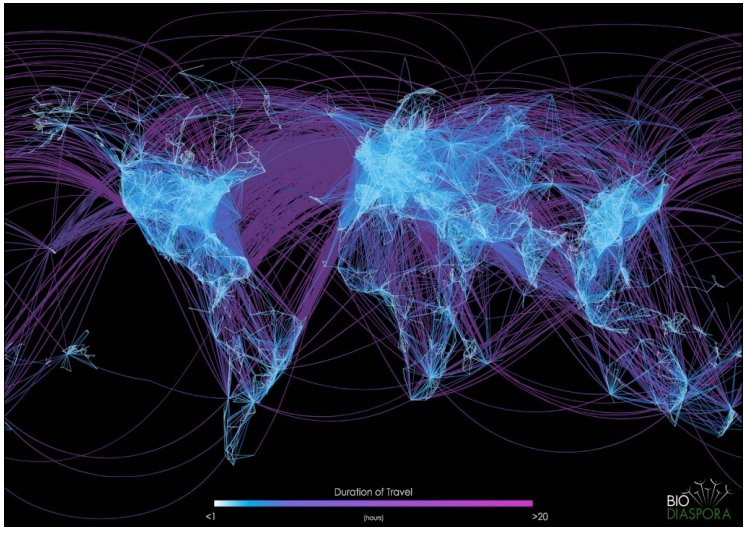
Friendship network in a 34-person karate club

Networks Applications

Networks are everywhere, but what can we do with them?

Which airports are at highest risk for virus spreading?

Are some parts of the world more difficult to reach?

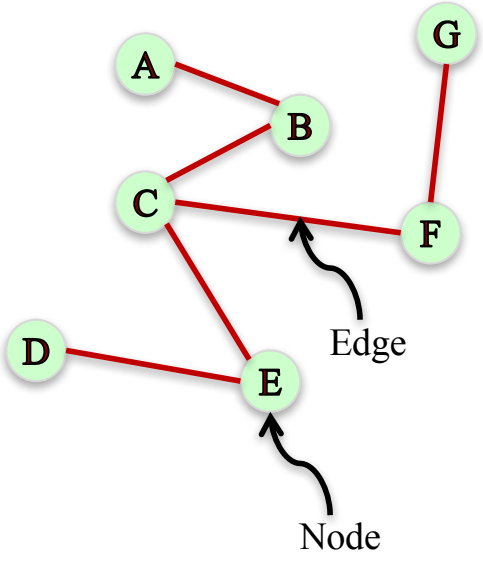


Network of direct flights around the world

Network Definition and Vocabulary

Network (or Graph): A representation of connections among a set of items.

- Items are called nodes (or vertices)
- Connections are called edges (or link or ties)

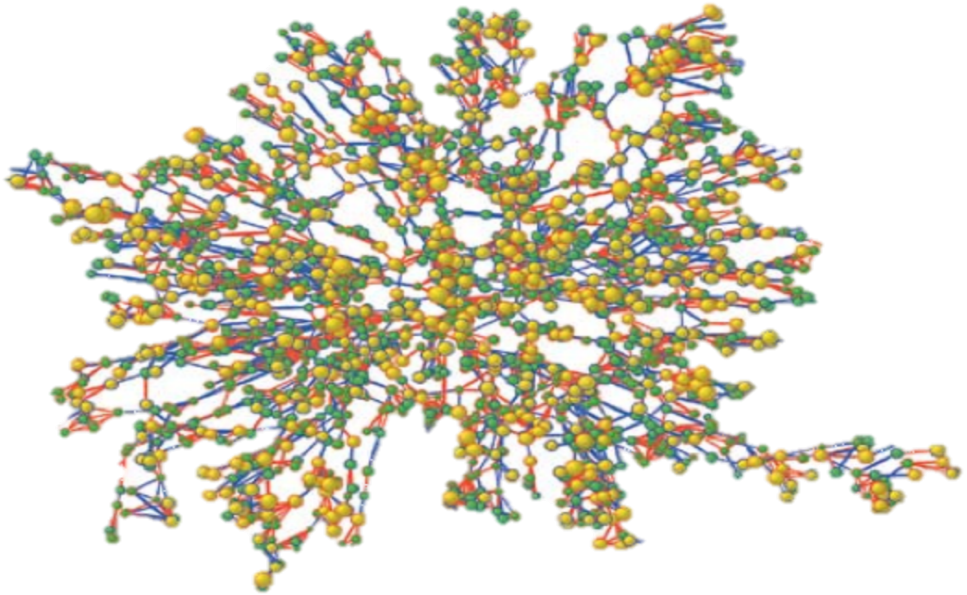



```
import networkx as nx
G=nx.Graph()
G.add_edge('A','B')
G.add_edge('B','C')
```

Nodes: People

Edges: Friendship, marital, or family ties

(Mostly) **Symmetric relationships**

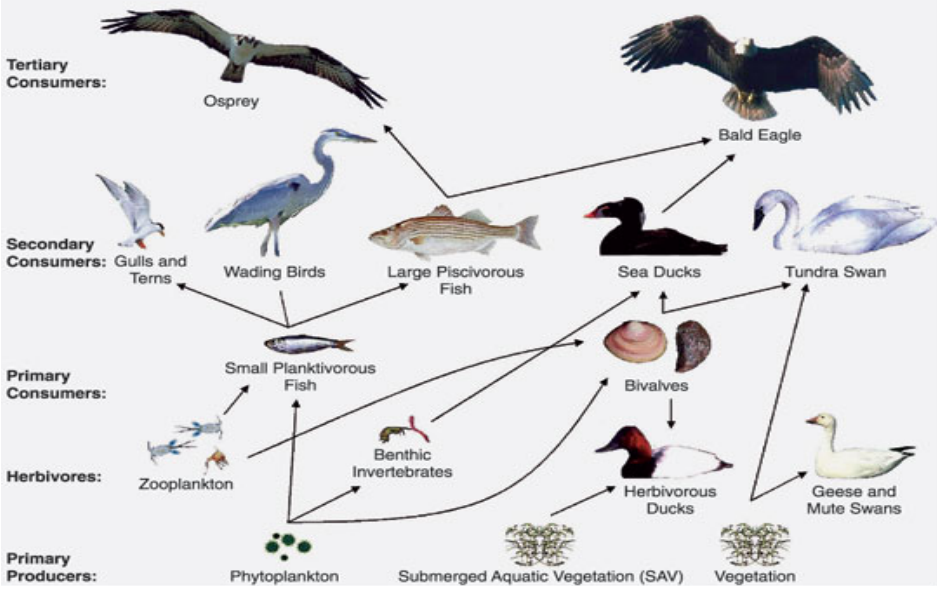


Network of friendship, marital tie, and family tie among 2200 people

Nodes: Birds

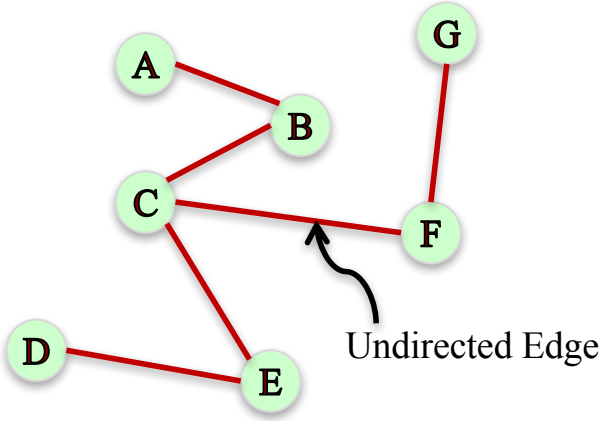
Edges: What eats what

Asymmetric relationships



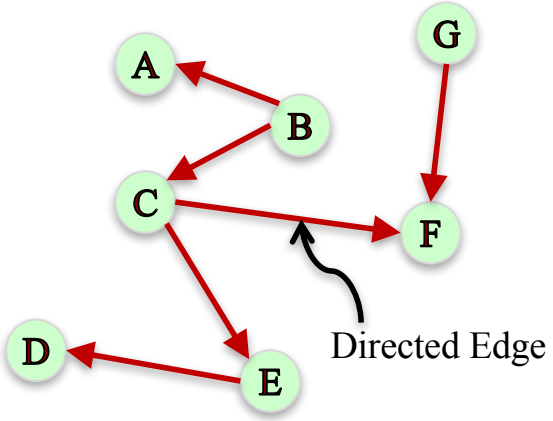
Chesapeake Bay Water bird Food Web

Edge Direction



Undirected network:
edges have no direction

```
G=nx.Graph()
G.add_edge('A','B')
G.add_edge('B','C')
```



Directed network:
edges have direction

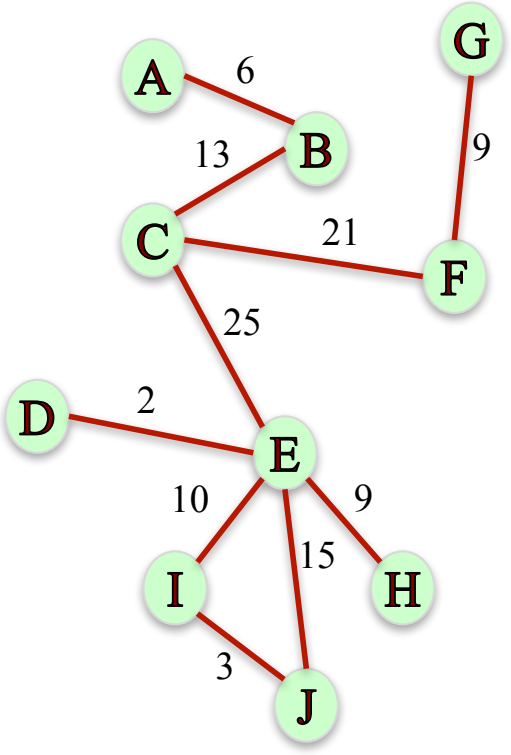
```
G=nx.DiGraph()
G.add_edge('B', 'A')
G.add_edge('B','C')
```

Weighted Networks

Not all relationships are equal.
Some edges carry higher weight than others.

Weighted network: a network where edges are assigned a (typically numerical) weight.

```
G=nx.Graph()
G.add_edge('A','B', weight = 6)
G.add_edge('B','C', weight = 13)
```



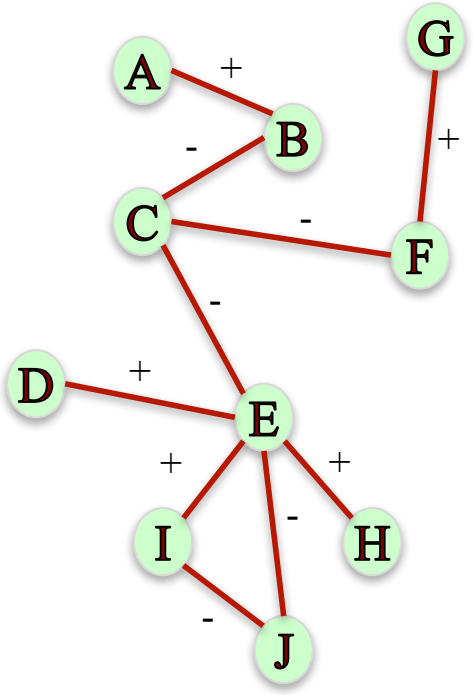
Number of times coworkers had lunch together in one year

Signed Networks

Some networks can carry information about friendship and antagonism based on conflict or disagreement.
Ex: In Epinions and Slashdot people can declare friends and foes.

Signed network: a network where edges are assigned positive or negative sign.

```
G=nx.Graph()
G.add_edge('A','B', sign= '+')
G.add_edge('B','C', sign= '-')
```

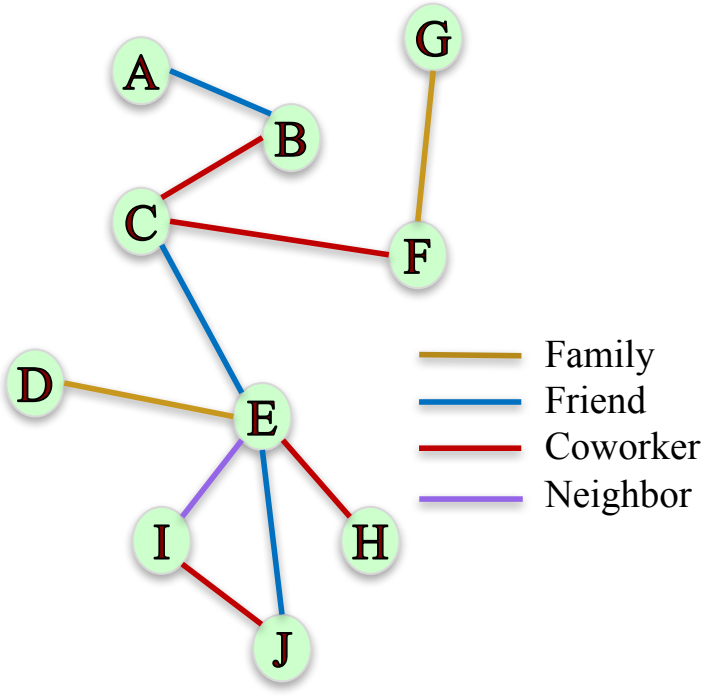


Friends and enemies

Other Edge Attributes

Edges can carry many other labels or attributes

```
G=nx.Graph()
G.add_edge('A','B', relation= 'friend')
G.add_edge('B','C', relation= 'coworker')
G.add_edge('D','E', relation= 'family')
G.add_edge('E','I', relation= 'neighbor')
```

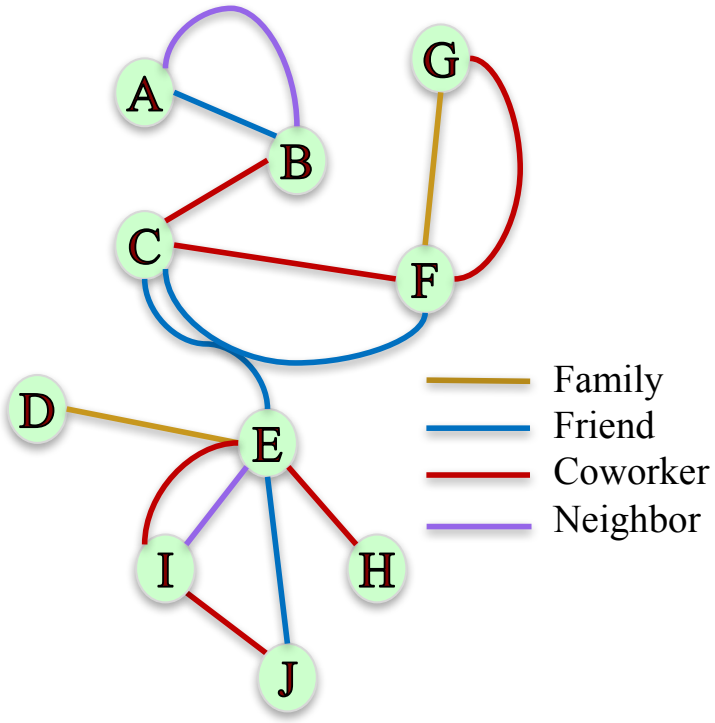


Mutigraphs

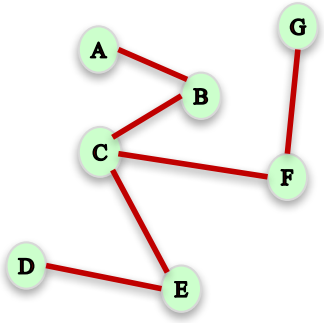
A pair of nodes can have different types of relationships simultaneously

Multigraph: A network where multiple edges can connect the same nodes (parallel edges).

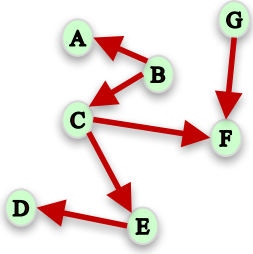
```
G=nx.MultiGraph()
G.add_edge('A','B', relation= 'friend')
G.add_edge('A','B', relation= neighbor')
G.add_edge('G','F', relation= 'family')
G.add_edge('G','F', relation= 'coworker')
```



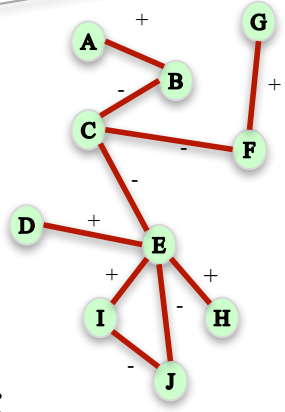
Summary



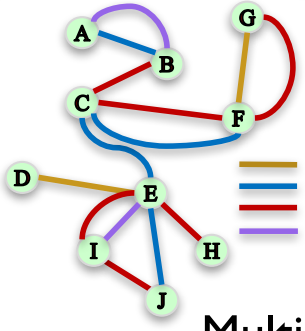
Undirected:
G=nx.Graph()



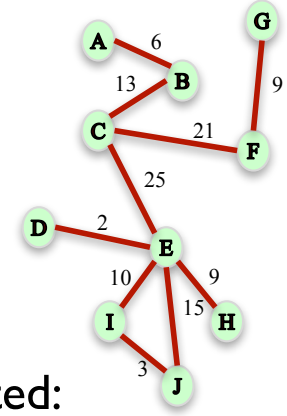
Directed:
G=nx.DiGraph()



Signed:
G.add_edge('A','B', sign= '+')



Multigraph:
G=nx.MultiGraph()



Weighted:
G.add_edge('A','B', weight = 6)

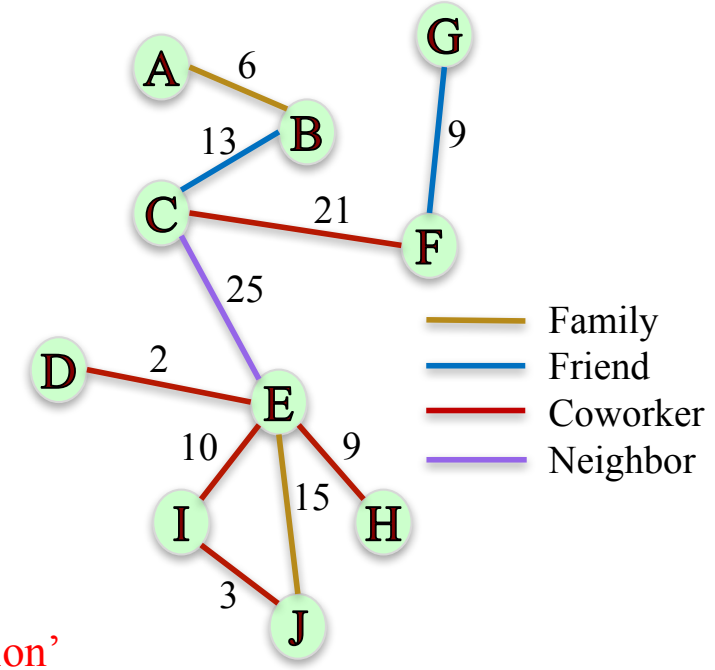
Edge Attributes in NetworkX

```
G=nx.Graph()
G.add_edge('A','B', weight= 6, relation = 'family')
G.add_edge('B','C', weight= 13, relation = 'friend')
```

```
In: G.edges() #list of all edges
Out: [('A', 'B'), ('C', 'B')]
```

```
In: G.edges(data= True)  #list of all edges with attributes
Out: [('A', 'B', {'relation': 'family', 'weight': 6}),
('C', 'B', {'relation': 'friend', 'weight': 13})]
```

```
In: G.edges(data= 'relation') #list of all edges with attribute 'relation'
Out: [('A', 'B', 'family'), ('C', 'B', 'friend')]
```



Number of times coworkers had lunch together in one year

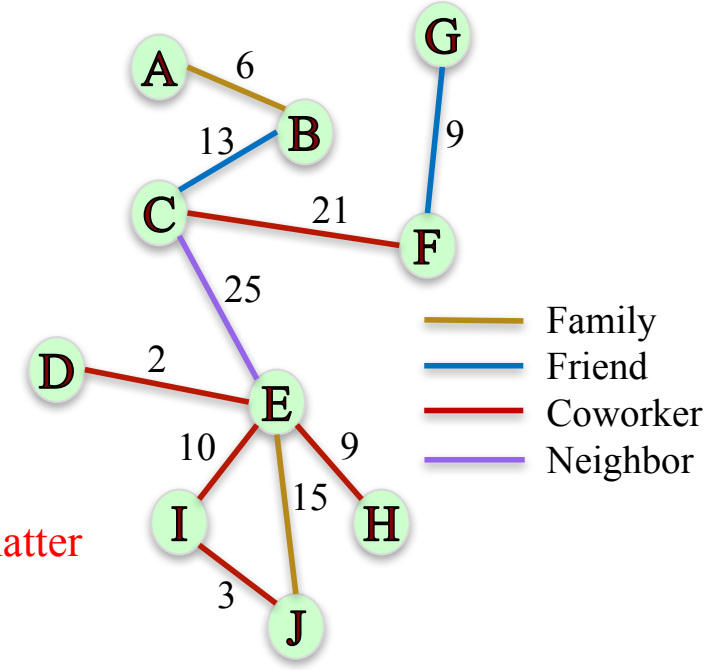
Edge Attributes in NetworkX

Accessing attributes of a specific edge:

In: G.edge['A']['B'] # dictionary of attributes of edge (A, B)
Out: {'relation': 'family', 'weight': 6}

In: G.edge['B']['C']['weight']
Out: 13

In: G.edge['C']['B']['weight'] # undirected graph, order does not matter
Out: 13



Number of times coworkers had lunch together in one year

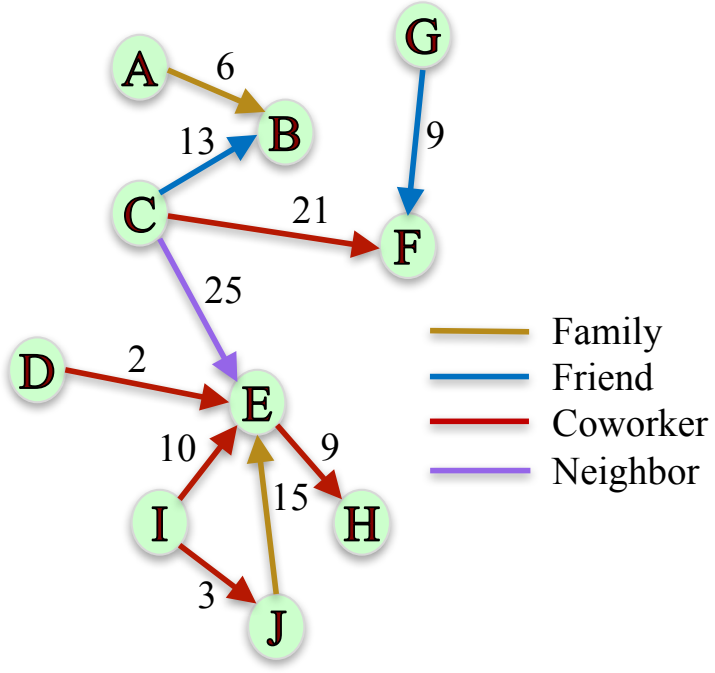
Edge Attributes in NetworkX

Directed, weighted network:

```
G=nx.DiGraph()
G.add_edge('A','B', weight= 6, relation = 'family')
G.add_edge('C', 'B', weight= 13, relation = 'friend')
```

Accessing edge attributes:

```
In: G.edge['C']['B']['weight']
Out: 13

In: G.edge['B']['C']['weight'] # directed graph, order matters
Out: KeyError: 'C'
```



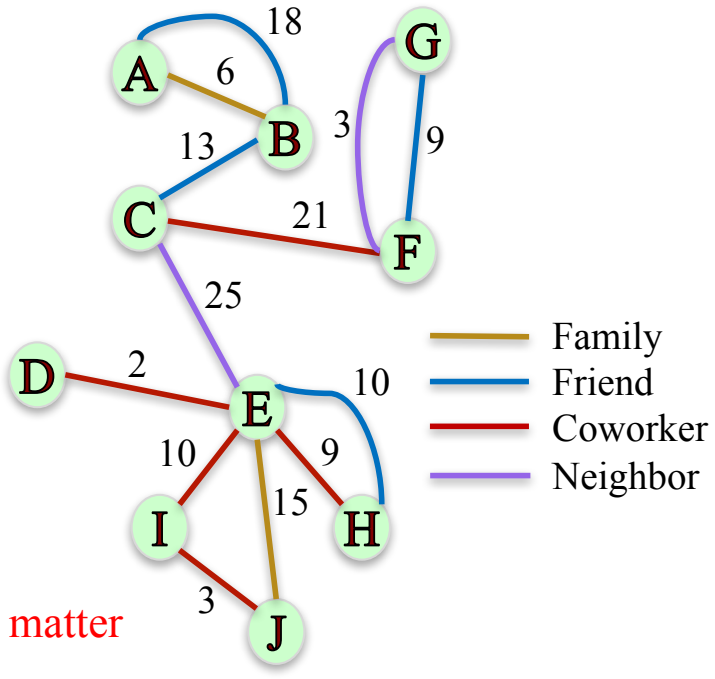
Edge Attributes in NetworkX

MultiGraph:

```
G=nx.MultiGraph()
G.add_edge('A','B', weight= 6, relation = 'family')
G.add_edge('A','B', weight= 18, relation = 'friend')
G.add_edge('C','B', weight= 13, relation = 'friend')
```

Accessing edge attributes:

```
In: G.edge['A']['B'] # One dictionary of attributes per (A,B) edge
Out: {0: {'relation': 'family', 'weight': 6},
1: {'relation': 'friend', 'weight': 18}}

In: G.edge['A']['B'][0]['weight'] # undirected graph, order does not matter
Out: 6
```



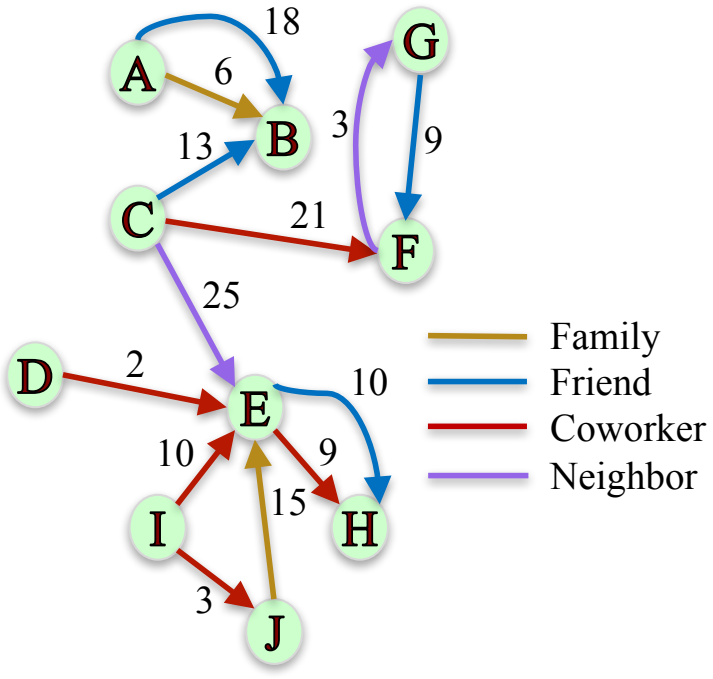
Edge Attributes in NetworkX

Directed MultiGraph:

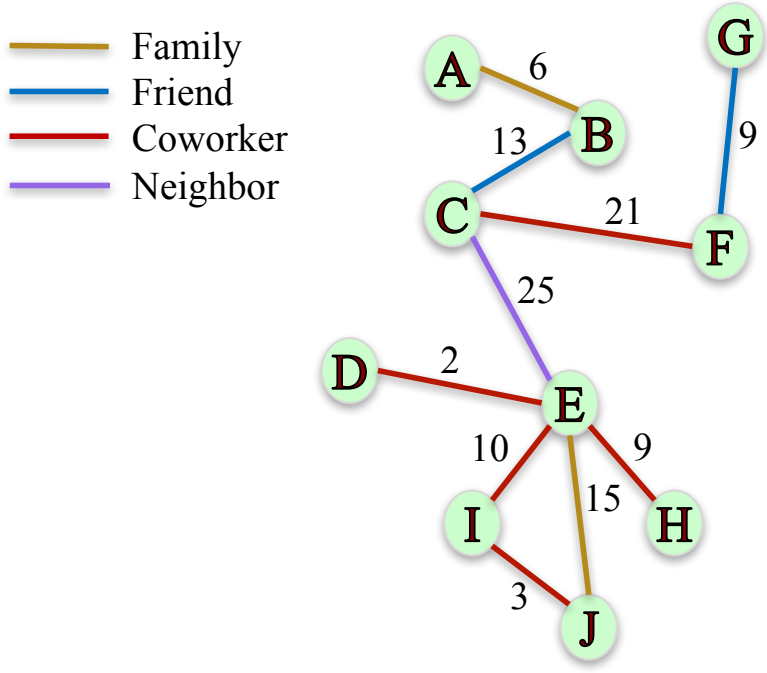
```
G=nx.MultiDiGraph()
G.add_edge('A','B', weight= 6, relation = 'family')
G.add_edge('A','B', weight= 18, relation = 'friend')
G.add_edge('C','B', weight= 13, relation = 'friend')
```

Accessing edge attributes:

```
In: G.edge['A']['B'][0]['weight']
Out: 6

In: G.edge['B']['A'][0]['weight'] # directed graph, order matters
Out: KeyError: 'A'
```



Node Attributes in NetworkX



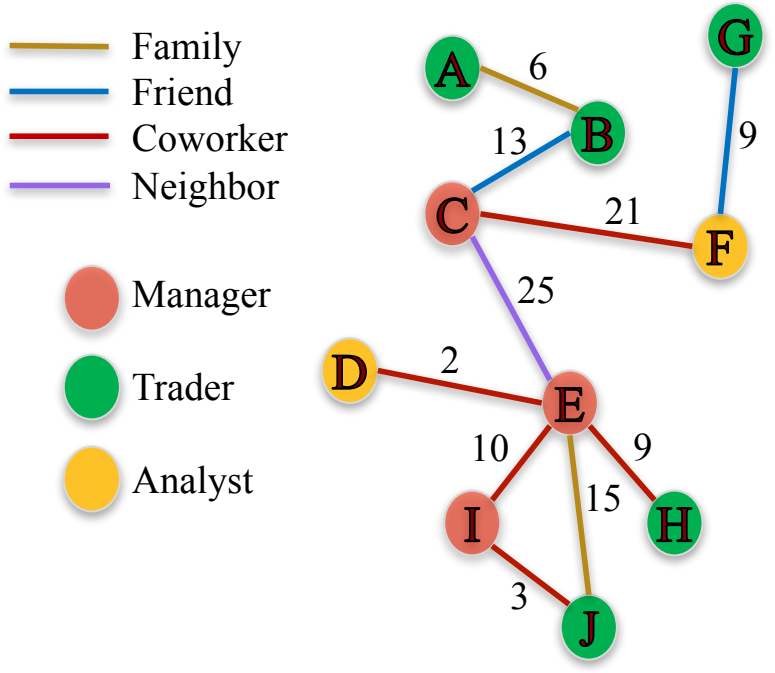
Number of times coworkers had lunch together in one year

Node Attributes in NetworkX

```
G=nx.Graph()
G.add_edge('A','B', weight= 6, relation = 'family')
G.add_edge('B','C', weight= 13, relation = 'friend')
```

Adding node attributes:

```
G.add_node('A', role = 'trader')
G.add_node('B', role = 'trader')
G.add_node('C', role = 'manager')
```



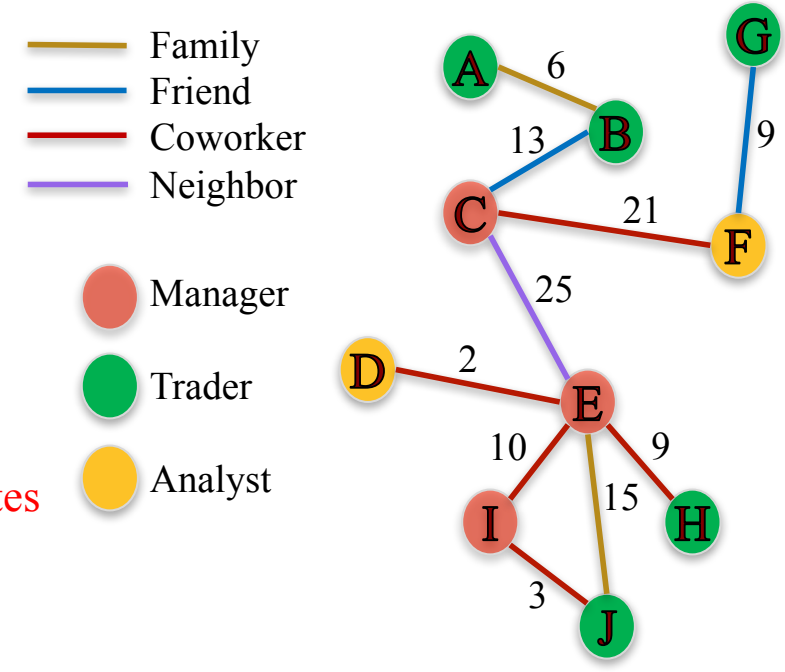
Number of times coworkers had lunch together in one year

Node Attributes in NetworkX

```
G=nx.Graph()
G.add_edge('A','B', weight= 6, relation = 'family')
G.add_edge('B','C', weight= 13, relation = 'friend')
```

Accessing node attributes:

```
In: G.nodes() # list of all nodes
Out: ['A', 'C', 'B']
In: G.nodes(data= True) #list of all nodes with attributes
Out: [('A', {'role': 'trader'}), ('C', {'role': 'manager'})
, ('B', {'role': 'trader'})]
In: G.node['A']['role']
Out: 'manager'
```



Number of times coworkers had lunch together in one year

Summary

Adding node and edge attributes:

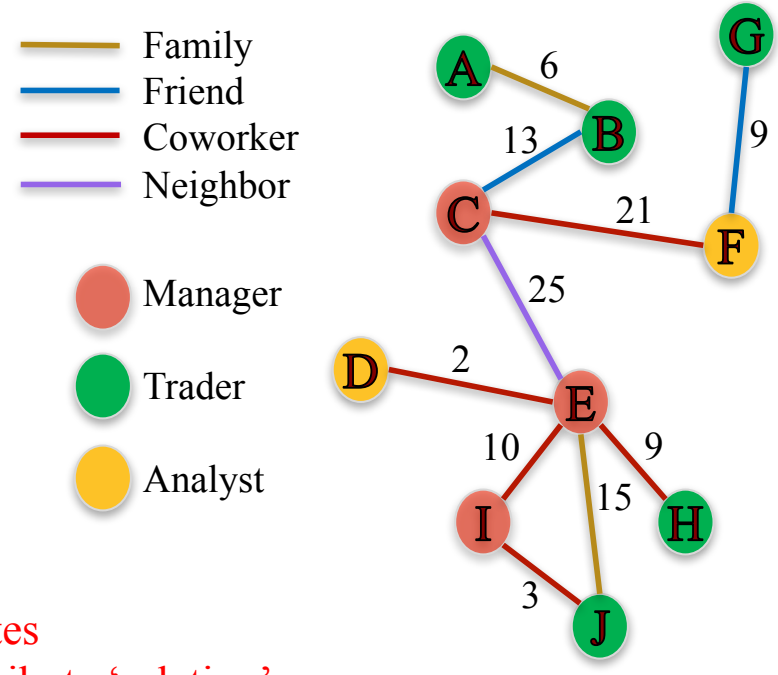
```
G=nx.Graph()
G.add_edge('A','B', weight= 6, relation = 'family')
G.add_node('A', role = 'trader')
```

Accessing node attributes:

```
G.nodes(data= True) #list of all nodes with attributes
G.node['A']['role'] #role of node A
```

Accessing Edge attributes:

```
In: G.edges(data= True)  #list of all edges with attributes
In: G.edges(data= 'relation') #list of all edges with attribute 'relation'
G.edge['A']['B']['weight'] # weight of edge (A,B)
```

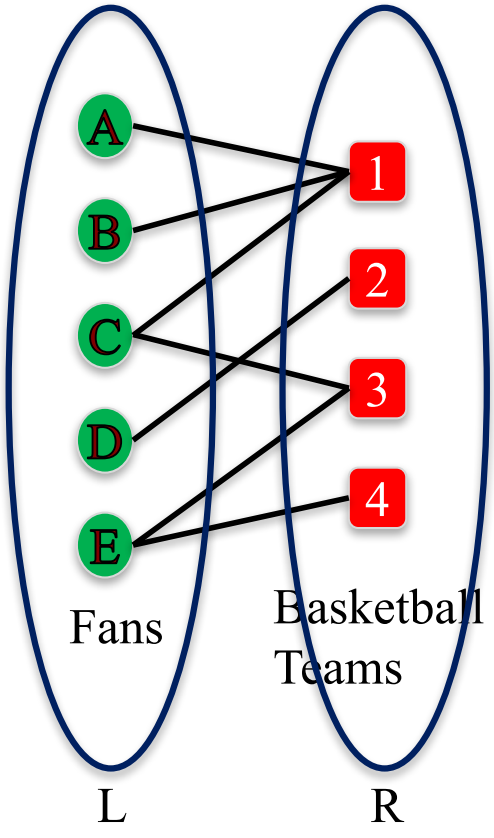


Number of times coworkers had lunch together in one year

Bipartite Graphs

Bipartite Graph: a graph whose nodes can be split into two sets *L* and *R* and every edge connects an node in *L* with a node in *R*.

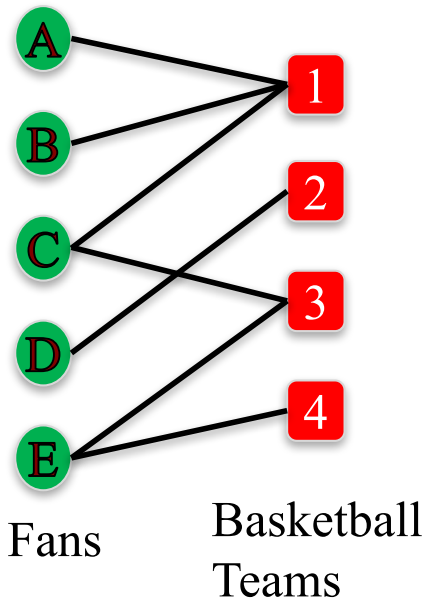
```
from networkx.algorithms import bipartite
B = nx.Graph() #No separate class for bipartite graphs
B.add_nodes_from(['A','B','C','D', 'E'], bipartite=0) #label one set of nodes 0
B.add_nodes_from([1,2,3,4], bipartite=1) #label other set of nodes 1
B.add_edges_from([('A',1), ('B',1), ('C',1), ('C',3), ('D',2), ('E',3), ('E', 4)])
```



Bipartite Graphs

Checking if a graph is bipartite:

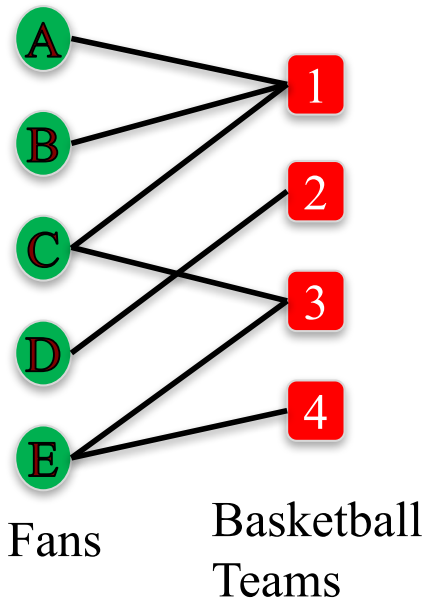
```
In: bipartite.is_bipartite(B) # Check if B is bipartite
Out: True

In: B.add_edge('A', 'B')
In: bipartite.is_bipartite(B)
Out: False

B.remove_edge('A', 'B')
```



Bipartite Graphs

Checking if a set of nodes is a bipartition of a graph:

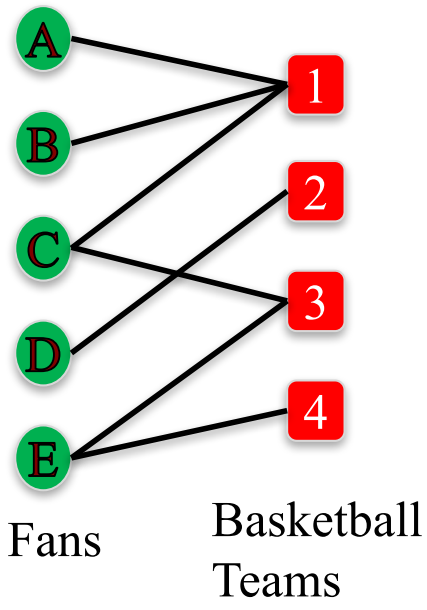
```
In: X = set([1,2,3,4])
In: bipartite.is_bipartite_node_set(B,X)
Out: True

X = set(['A', 'B', 'C', 'D', 'E'])
In: bipartite.is_bipartite_node_set(B,X)
Out: True

X = set([1,2,3,4, 'A'])
In: bipartite.is_bipartite_node_set(B,X)
Out: False
```



Bipartite Graphs

Getting each set of nodes of a bipartite graph:

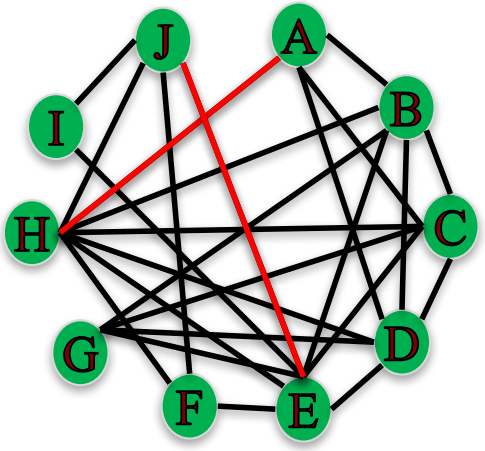
```
In: bipartite.sets(B)
Out: ({'A', 'B', 'C', 'D', 'E'}, {1, 2, 3, 4})

In: B.add_edge('A', 'B')
In: bipartite.sets(B)
Out: NetworkXError: Graph is not bipartite.

B.remove_edge('A', 'B')
```



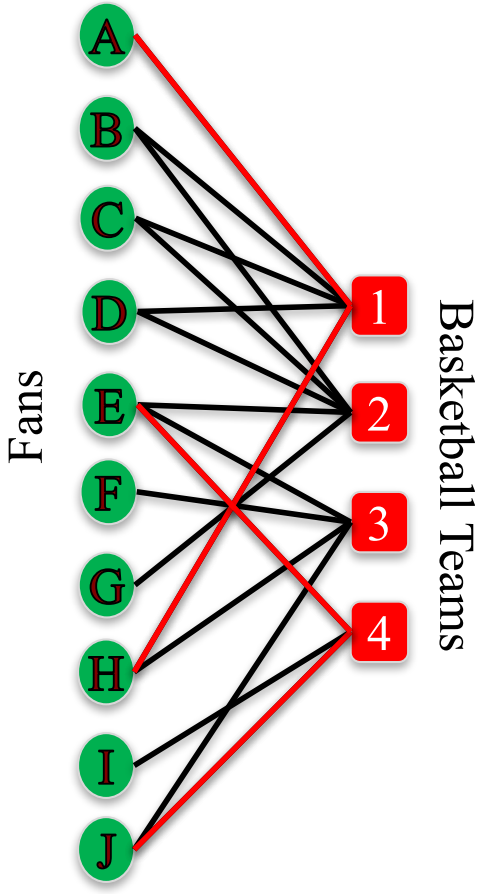
Projected Graphs

***L*-Bipartite graph projection:** Network of nodes in group *L*, where a pair of nodes is connected if they have a common neighbor in *R* in the bipartite graph.

Similar definition for *R*-Bipartite graph projection

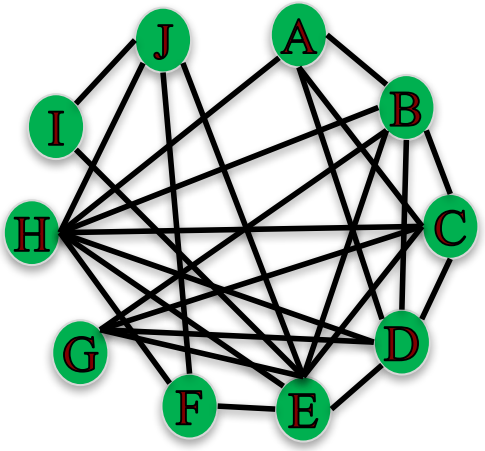


Network of fans who have a team in common

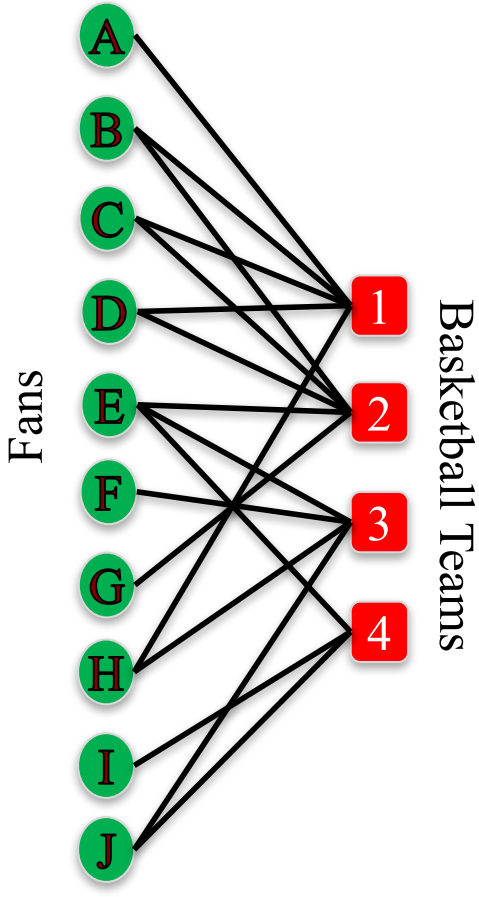


Projected Graphs

```
B = nx.Graph()
B.add_edges_from([('A',1), ('B',1),
('C',1),('D',1),('H',1), ('B', 2), ('C', 2), ('D',
2),('E', 2), ('G', 2), ('E', 3), ('F', 3), ('H', 3),
('J', 3), ('E', 4), ('I', 4), ('J', 4) ])

X = set(['A','B','C','D', 'E', 'F','G', 'H', 'I','J'])
P = bipartite.projected_graph(B, X)
```

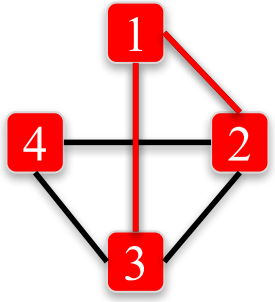


Network of fans who have a team in common

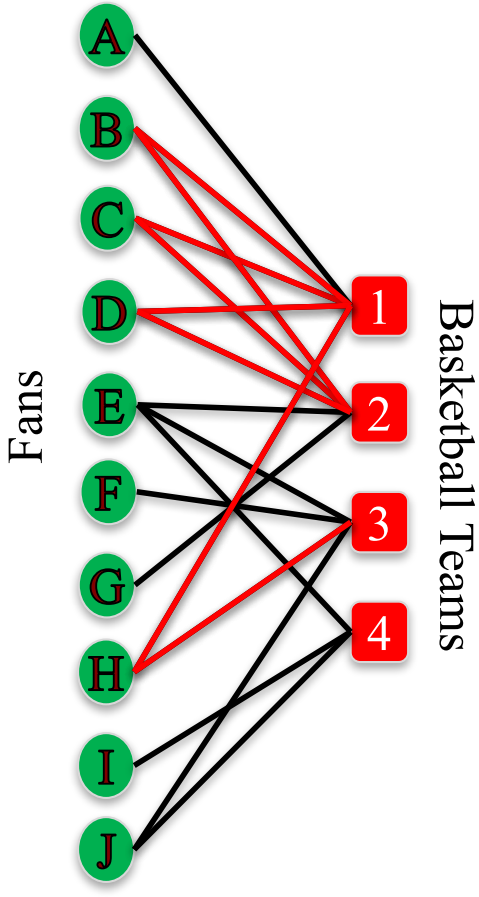


Projected Graphs

```
B = nx.Graph()
B.add_edges_from([('A',1), ('B',1),
('C',1),('D',1),('H',1), ('B', 2), ('C', 2), ('D',
2),('E', 2), ('G', 2), ('E', 3), ('F', 3), ('H', 3),
('J', 3), ('E', 4), ('I', 4), ('J', 4) ])

X = set([1,2,3,4])
P = bipartite.projected_graph(B, X)
```



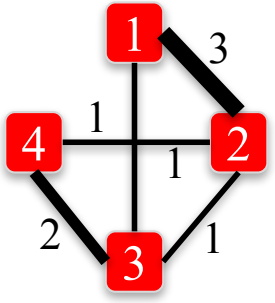
Network of teams who have a fan common



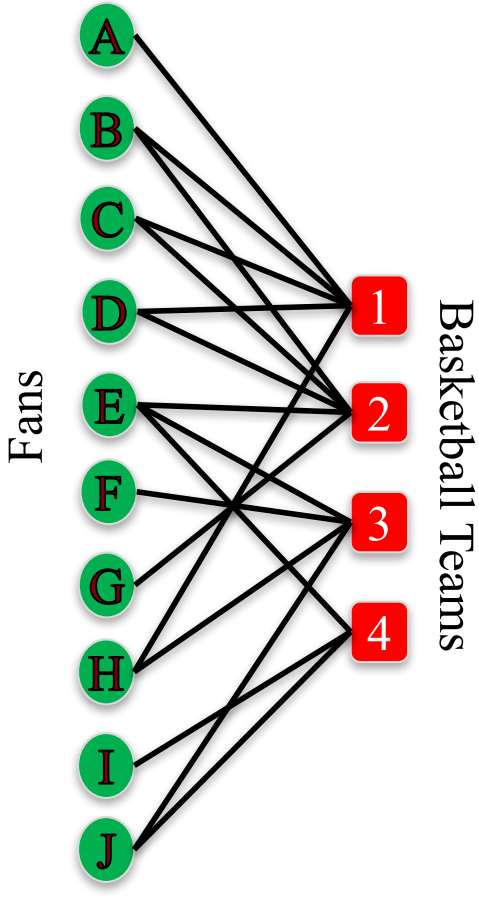
We need weights on the edges!

Projected Graphs

***L*-Bipartite weighted graph projection:** An *L*-Bipartite graph projection with weights on the edges that are proportional to the number of common neighbors between the nodes.



Weighted Network of teams who have a fan common

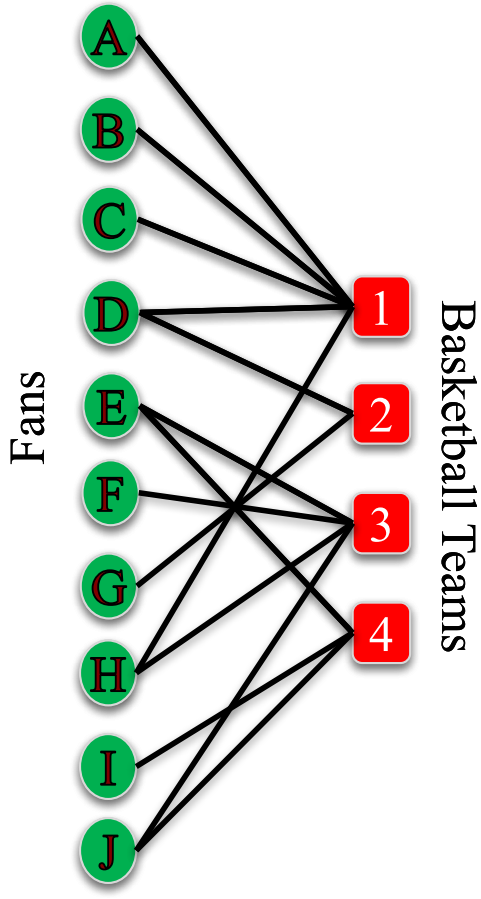



```
X = set([1,2,3,4])
P = bipartite.weighted_projected_graph(B, X)
```

Summary

No separate class for bipartite graphs in NetworkX
Use Graph(), DiGraph(), MultiGraph(), etc.
Use from networkx.algorithms import bipartite for bipartite related algorithms (Many algorithms only work on Graph()).

- nx.bipartite.is_bipartite(B) # Check if B is bipartite
- bipartite.is_bipartite_node_set(B,X) # Check if node set X is a bipartition
- bipartite.sets(B) # Get each set of nodes of bipartite graph B
- bipartite.projected_graph(B, X) # Get the bipartite projection of node set X
- bipartite.weighted_projected_graph(B, X) # Get the weighted bipartite projection of node set X



Class Test 1 I

Network of Cities

- A file with a pair of city names and distance between them is given (`edge_list.txt`)
- A file with city name and corresponding population is given.
- Form network with node and edge and edge attributes

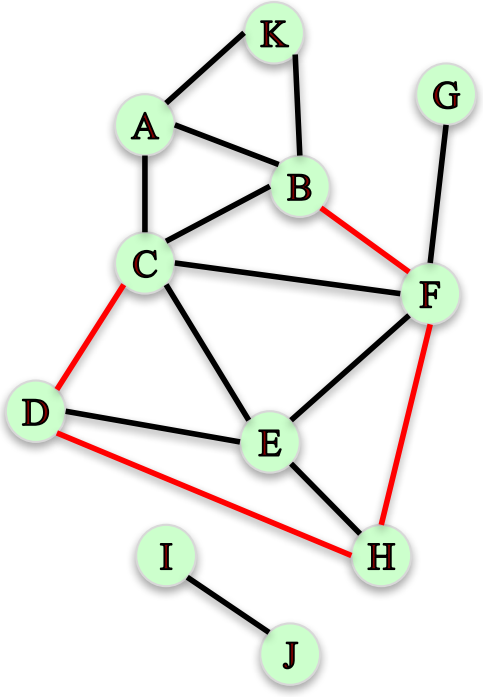
Class Test 1 II

```
import networkx as nx
G = nx.read_weighted_edgelist
('edge_list.txt', delimiter =" ")
population = {
        'Kolkata' : 4486679,
        'Delhi' : 11007835,
        ...
        }
for i in list(G.nodes()):
    G.nodes[i]['population'] = population[i]
nx.draw_networkx(G, with_label = True)
```

Triadic Closure

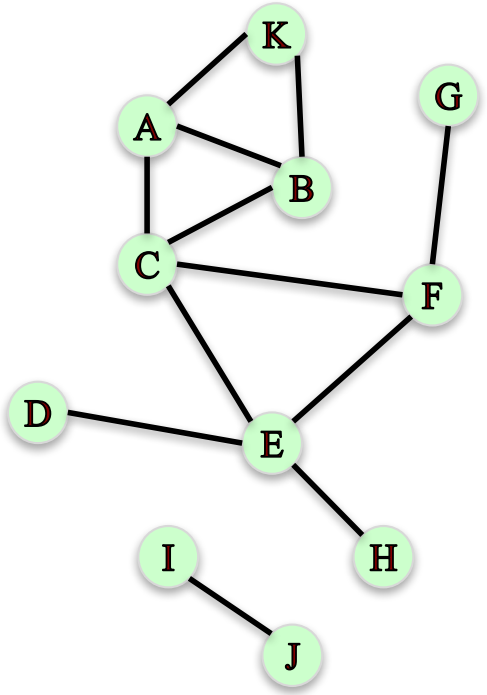
Triadic closure: The tendency for people who share connections in a social network to become connected.

How can we measure the prevalence of triadic closure in a network?



Local Clustering Coefficient

Local clustering coefficient of a node:
Fraction of pairs of the node's friends that are friends with each other.



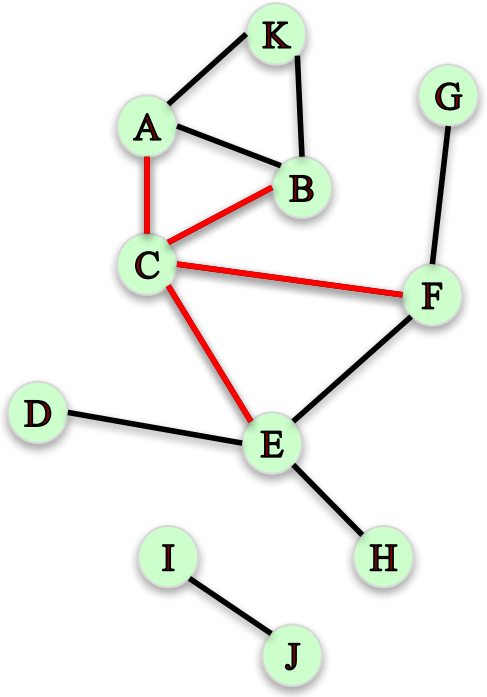
Local Clustering Coefficient

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 =



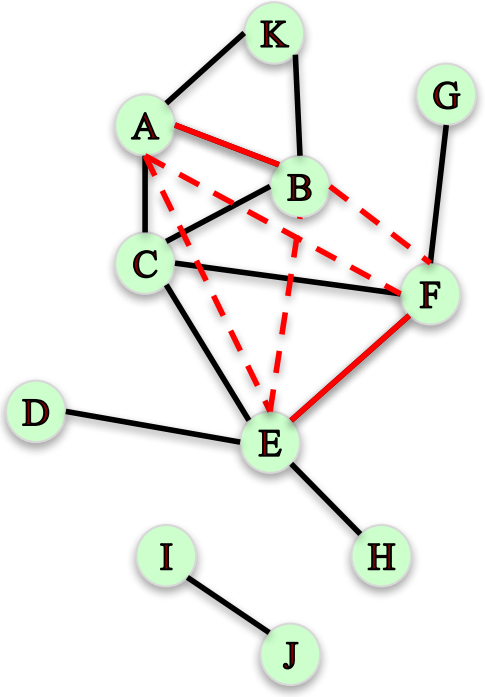
Local Clustering Coefficient

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)

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



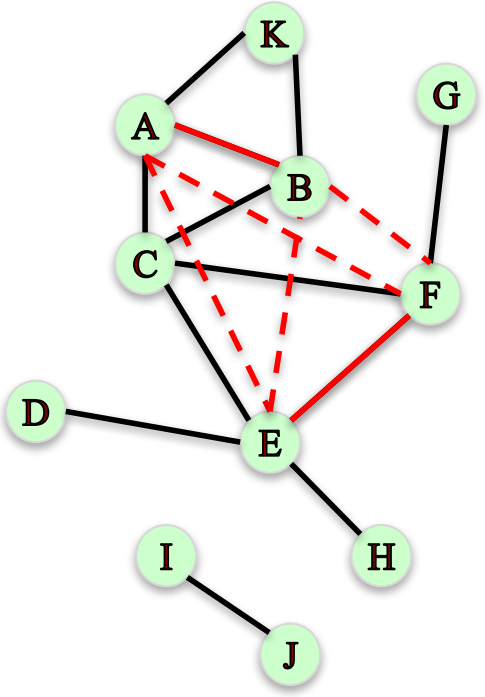
Local Clustering Coefficient

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)

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



Local Clustering Coefficient

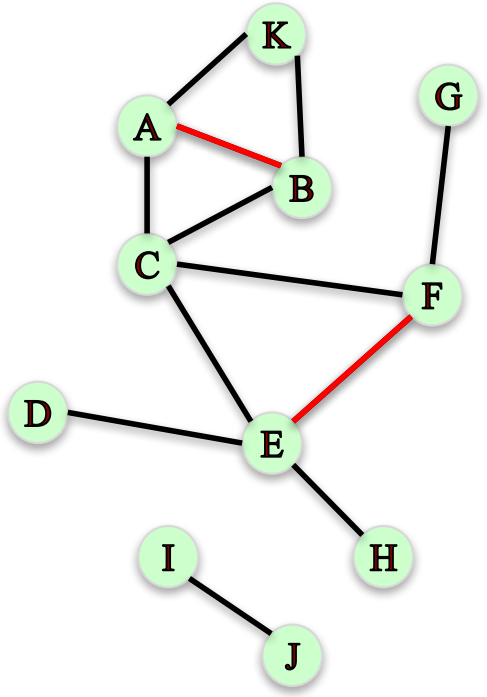
Compute the local clustering coefficient of node C:

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

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

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

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



Local Clustering Coefficient

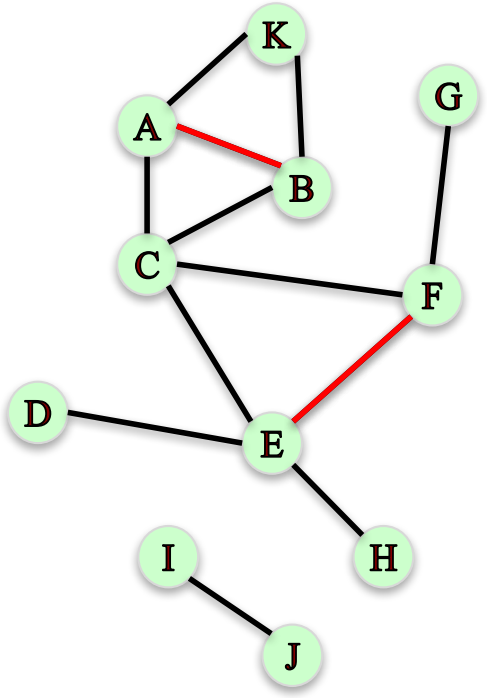
Compute the local clustering coefficient of node C:

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

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

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

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



Local Clustering Coefficient

Compute the local clustering coefficient of node C:

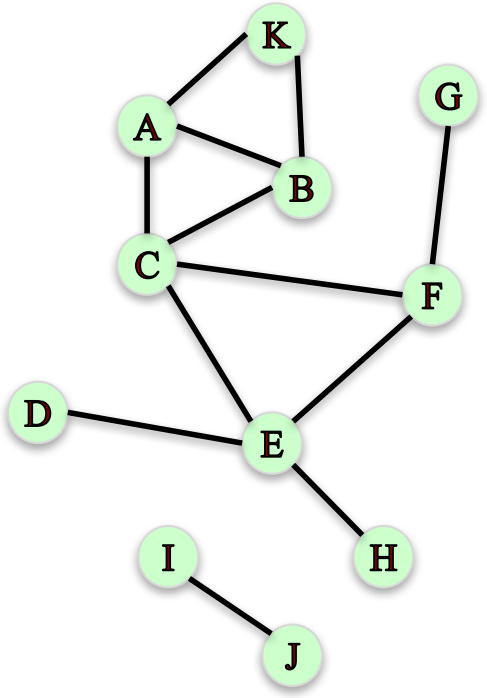
$$\frac{\#\text{ of pairs of C's friends who are friends}}{\#\text{ of pairs of C's friends}}$$

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

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

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

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



Local Clustering Coefficient

Compute the local clustering coefficient of node F:

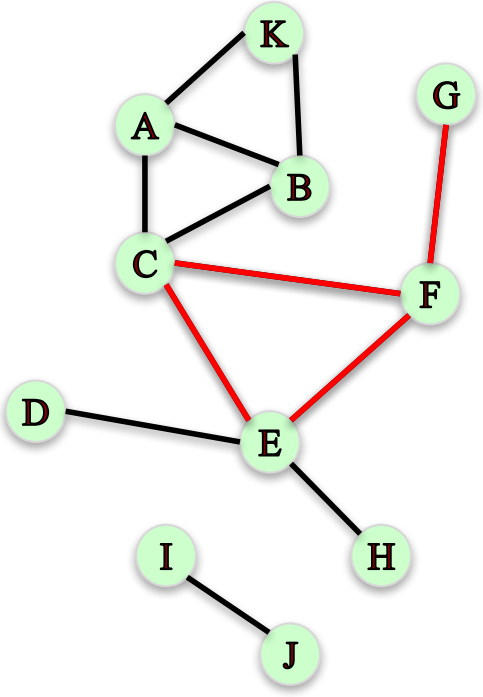
$$\frac{\text{\# of pairs of F's friends who are friends}}{\text{\# of pairs of F's friends}}$$

$d_F = 3$

$$\text{\# of pairs of F's friends} = \frac{d_F(d_F - 1)}{2} = \frac{6}{2} = 3$$

\# of pairs of F's friends who are friends = 1

Local clustering coefficent of F = $\frac{1}{3}$



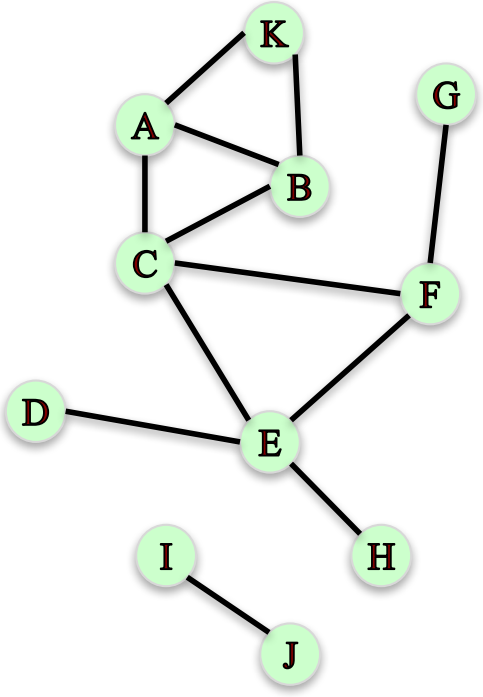
Local Clustering Coefficient

Compute the local clustering coefficient of node J:

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

# of pairs of J's friends = 0 (Can not divide by 0)

We will assume that the local clustering coefficient of a node of degree less than 2 to be 0.



Local Clustering Coefficient

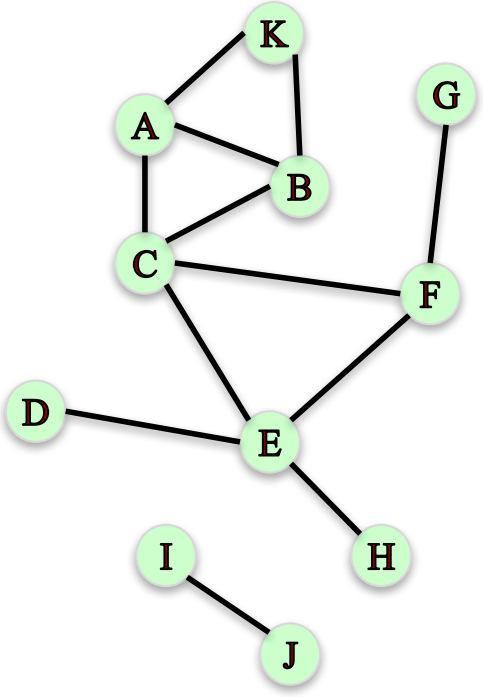
Local clustering coefficient in NetworkX:

```
G = nx.Graph()
G.add_edges_from([('A', 'K'), ('A', 'B'), ('A', 'C'), ('B', 'C'), ('B', 'K'),
('C', 'E'), ('C', 'F'), ('D', 'E'), ('E', 'F'), ('E', 'H'), ('F', 'G'), ('I', 'J')])
```

```
In: nx.clustering(G, 'F')
Out: 0.3333333333333333
```

```
In: nx.clustering(G, 'A')
Out: 0.6666666666666666
```

```
In: nx.clustering(G, 'J')
Out: 0.0
```

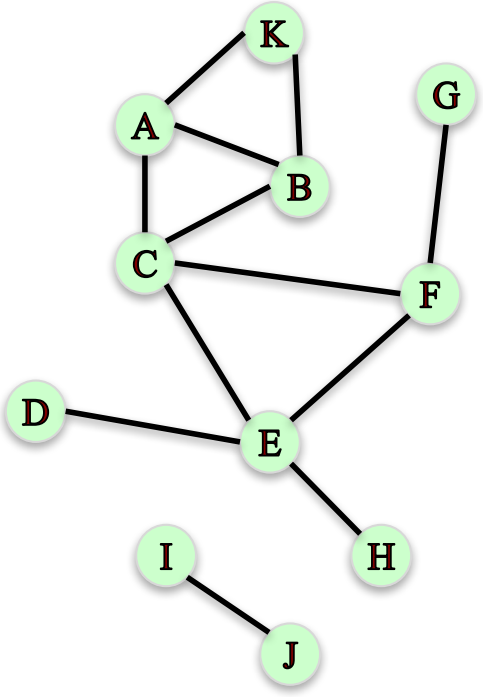


Global Clustering Coefficient

Measuring clustering on the whole network:

Approach 1: Average local clustering coefficient over all nodes in the graph.

In: nx.average_clustering(G)
Out: 0.2878787878787878785

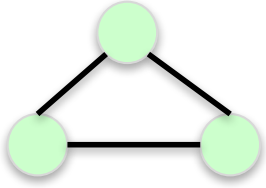


Global Clustering Coefficient

Measuring clustering on the whole network (Approach 2):

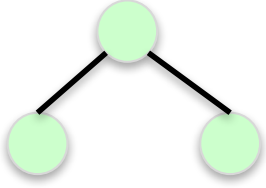
Percentage of "open triads" that are triangles in a network.

Triangles:



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

Open triads:

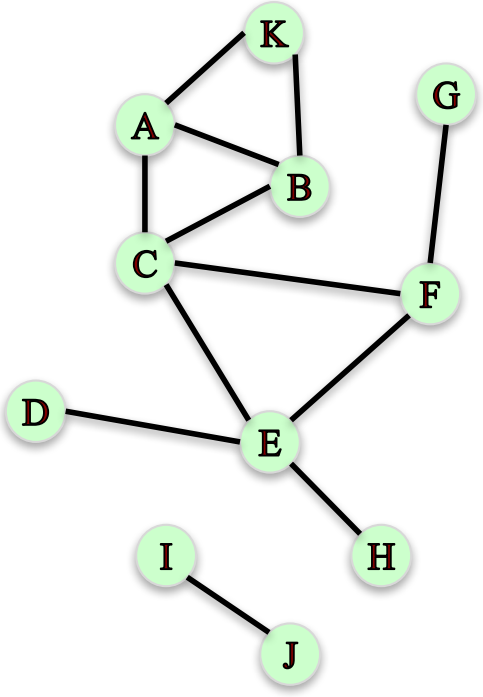


Global Clustering Coefficient

Measuring clustering on the whole network:

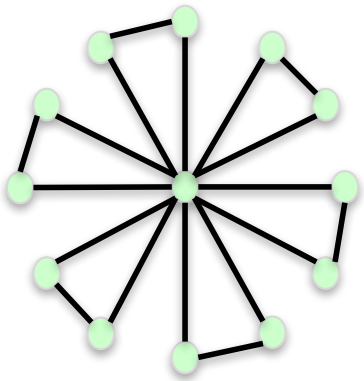
Transitivity: Ratio of number of triangles and number of "open triads" in a network.

In: nx.transitivity(G)
Out: 0.409090909091



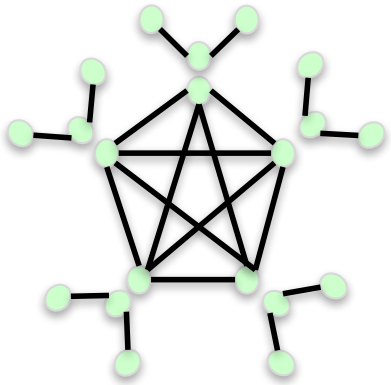
Transitivity vs. Average Clustering Coefficient

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

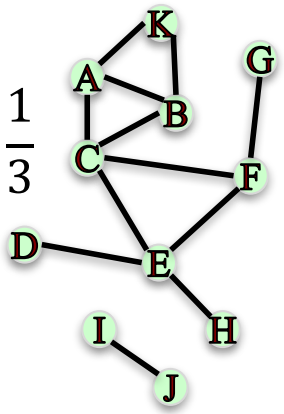
Summary

Clustering coefficient measures the degree to which nodes in a network tend to "cluster" or form triangles.

Local Clustering Coefficient

Fraction of pairs of the node's friends that are friends with each other.

LCC of C = $\frac{2}{6} = \frac{1}{3}$



Global Clustering Coefficient

Average Local Clustering Coefficient

nx.average_clustering(G)

Transitivity

Ratio of number of triangles and number of "open triads".

Puts larger weight on high degree nodes.

nx.transitivity(G)

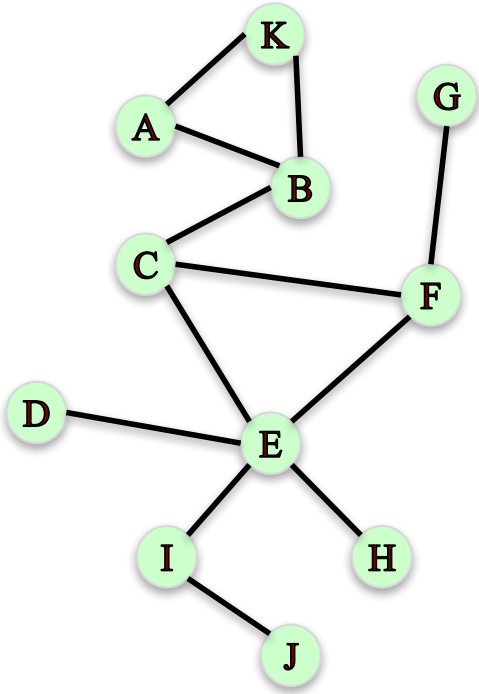
Distance

How "far" is node A from node H?

Are nodes far away or close to each other in this network?

Which nodes are "closest" and "farthest" to other nodes?

We need a sense of distance between nodes to answer these questions



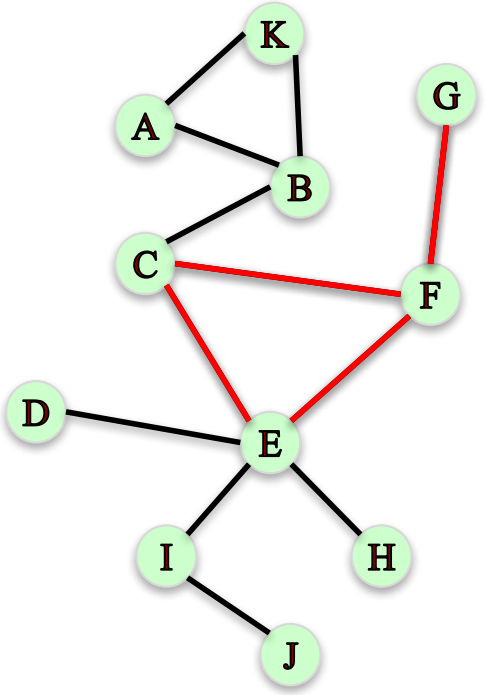
Paths

Path: A sequence of nodes connected by an edge.

Find two paths from node G to node C:

G – F – C

G – F – E – C



Distance

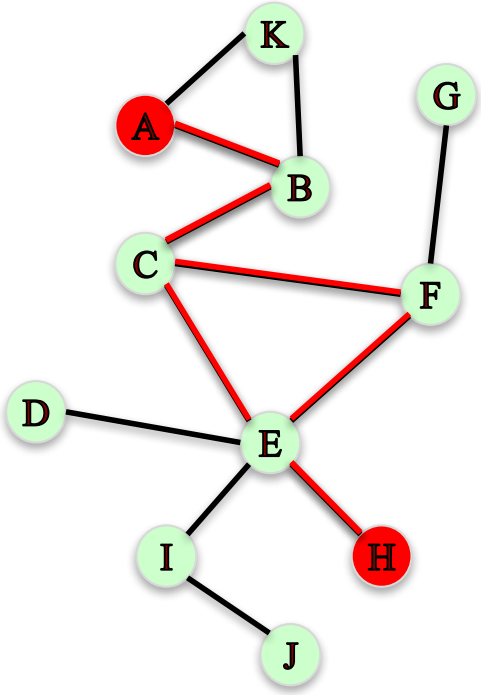
How far is node A from node H?

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.

Path 1 has length 4, Path 2 has length 5



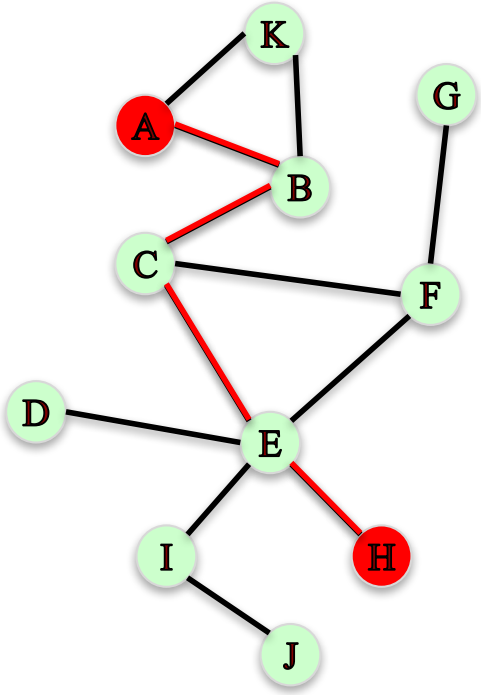
Distance

Distance between two nodes: the length of the shortest path between them.

The distance between node A and H is 4

In: nx.shortest_path(G,'A', 'H')
Out: ['A', 'B', 'C', 'E', 'H']

In: nx.shortest_path_length(G,'A', 'H')
Out: 4

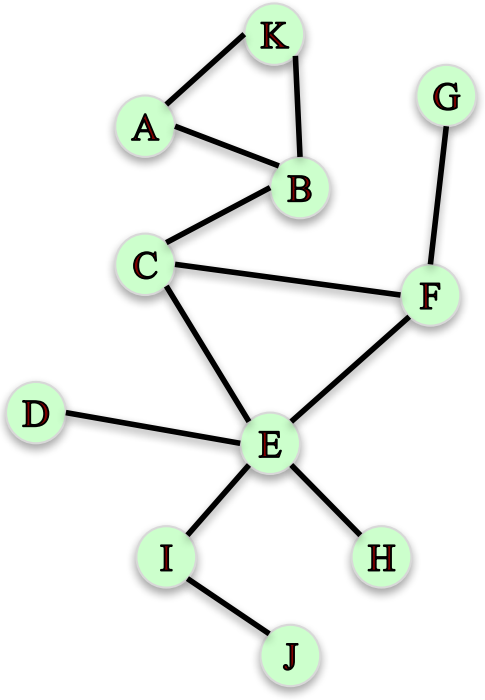


Distance

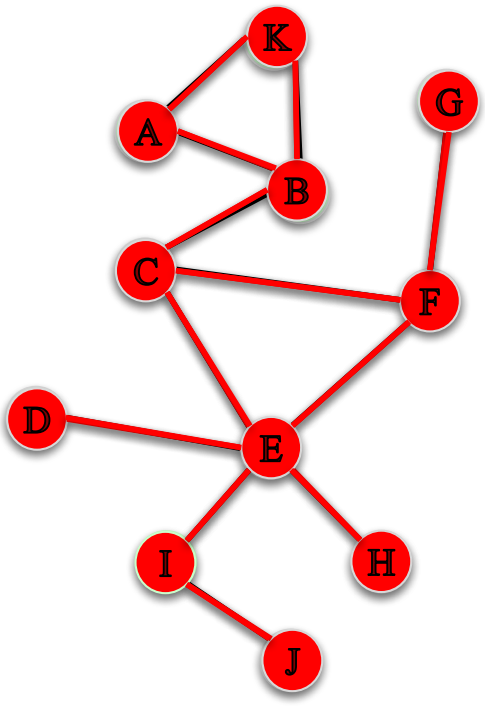
Finding the distance from node A to every other node.

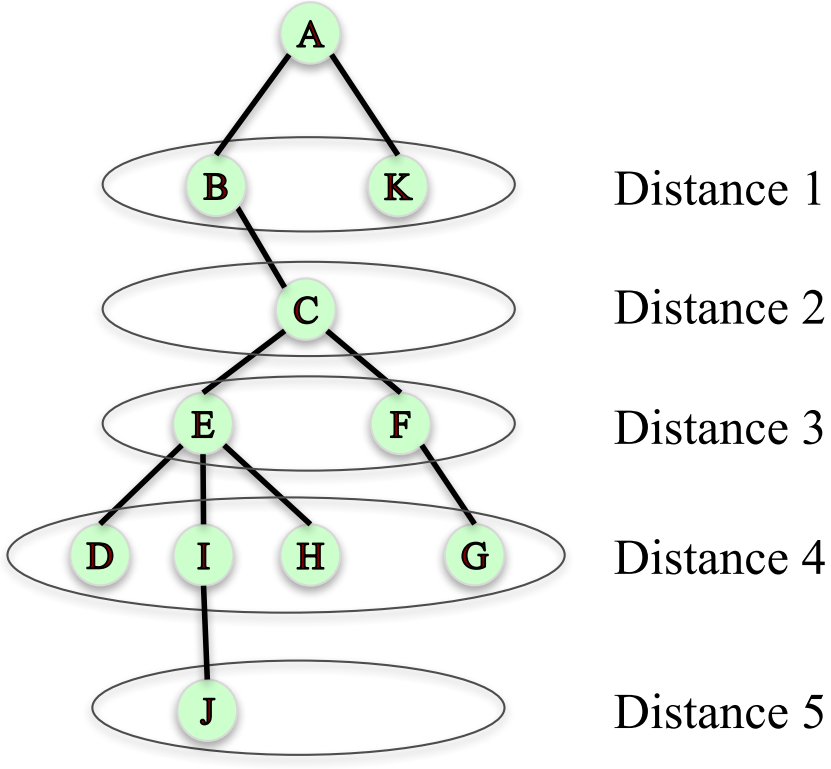
Easy to do manually in small networks but tedious in large (real) networks.

Breadth-first search: a systematic and efficient procedure for computing distances from a node to all other nodes in a large network by "discovering" nodes in layers.



Breadth-First Search





Breadth-First Search

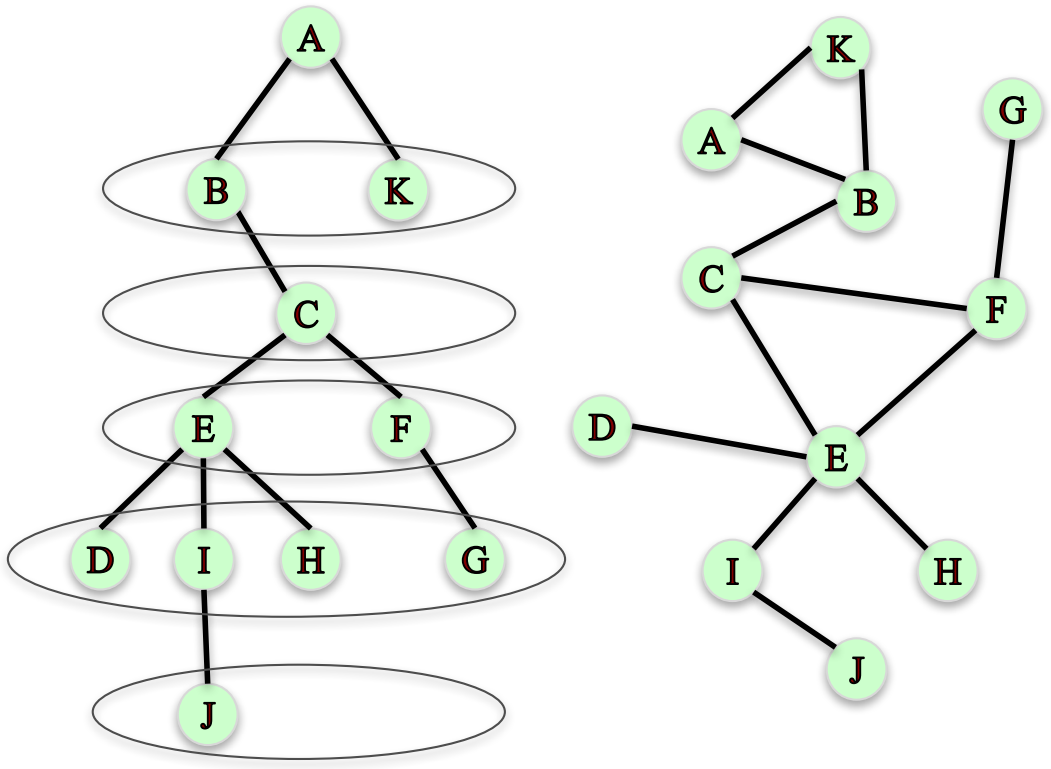
In: T = nx.bfs_tree(G, 'A')

In: T.edges()

Out: [('A', 'K'), ('A', 'B'), ('B', 'C'), ('C', 'E'), ('C', 'F'), ('E', 'I'), ('E', 'H'), ('E', 'D'), ('F', 'G'), ('I', 'J')]

In: nx.shortest_path_length(G,'A')

Out: {'A': 0, 'B': 1, 'C': 2, 'D': 4, 'E': 3, 'F': 3, 'G': 4, 'H': 4, 'I': 4, 'J': 5, 'K': 1}



Distance Measures

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

Average distance between every pair of nodes.

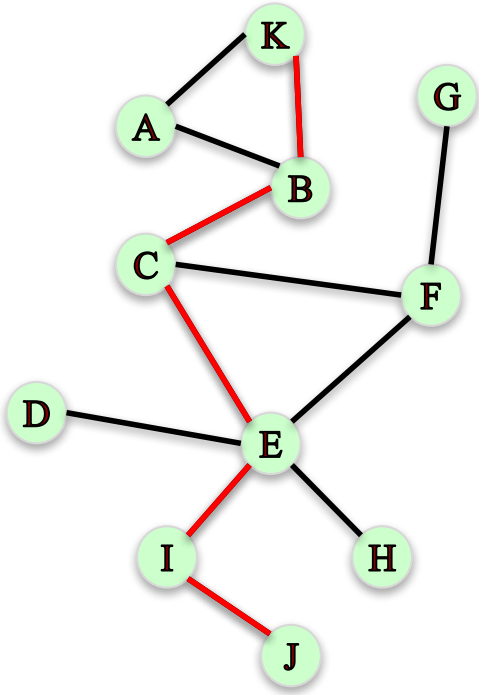
In: nx.average_shortest_path_length(G)

Out: 2.52727272727

Diameter: maximum distance between any pair of nodes.

In: nx.diameter(G)

Out: 5



Distance Measures

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.

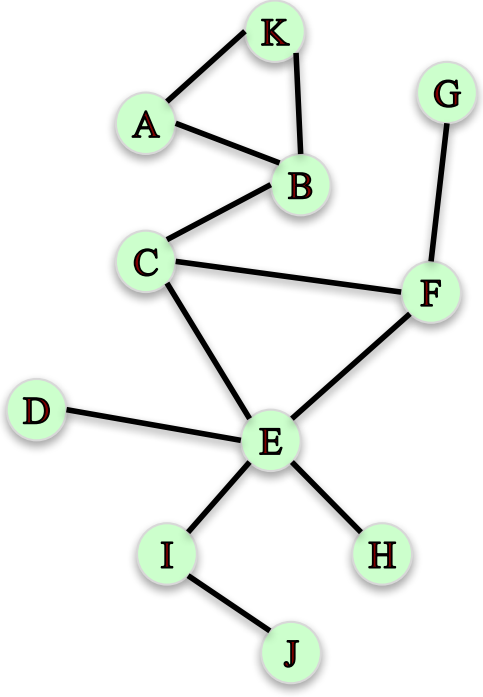
In: nx.eccentricity(G)

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



Distance Measures

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

The **Periphery** of a graph is the set of nodes that have eccentricity equal to the diameter.

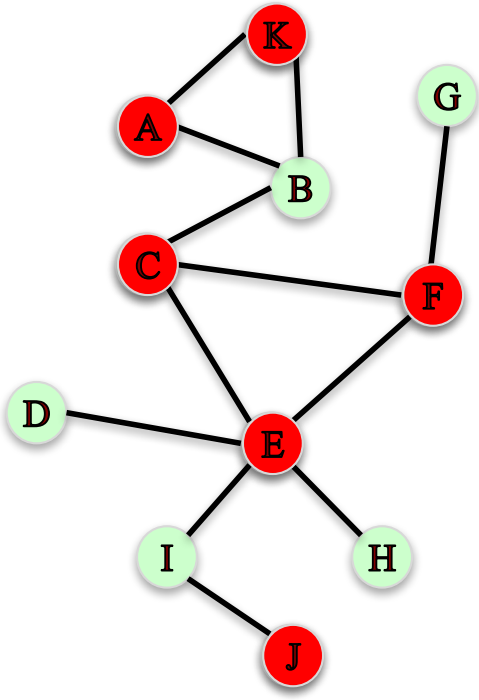
In: nx.periphery(G)

Out: ['A', 'K', 'J']

The **center** of a graph is the set of nodes that have eccentricity equal to the radius.

In: nx.center(G)

Out: ['C', 'E', 'F']



Karate Club Network

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

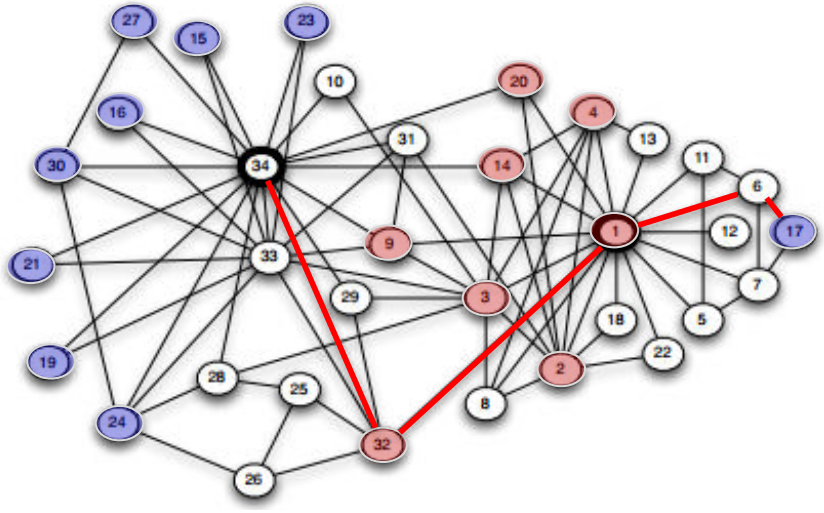
Average shortest path = 2.41

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

Summary

Distance between two nodes: length of the shortest path between them.

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

Characterizing distances in a network:

Average distance between every pair of nodes.

Diameter: maximum distance between any pair of nodes.

Radius: the minimum eccentricity in the graph.

Identifying central and peripheral nodes:

The **Periphery** is the set of nodes with eccentricity = diameter.

The **center** is the set of nodes with eccentricity = radius.

Resources

- NetworkX Docs
  `http://networkx.lanl.gov/tutorial/index.html`
- NetworkX Tutorial
  `http://networkx.lanl.gov/contents.html`
- Matplotlib Docs
  `http://matplotlib.sourceforge.net/contents.html`
- Matplotlib Tutorial
  `http://matplotlib.sourceforge.net/users/pyplot_tutorial.html`
- Numpy Docs
  `http://numpy.scipy.org/`
- MacPorts
  `http://macports.org`