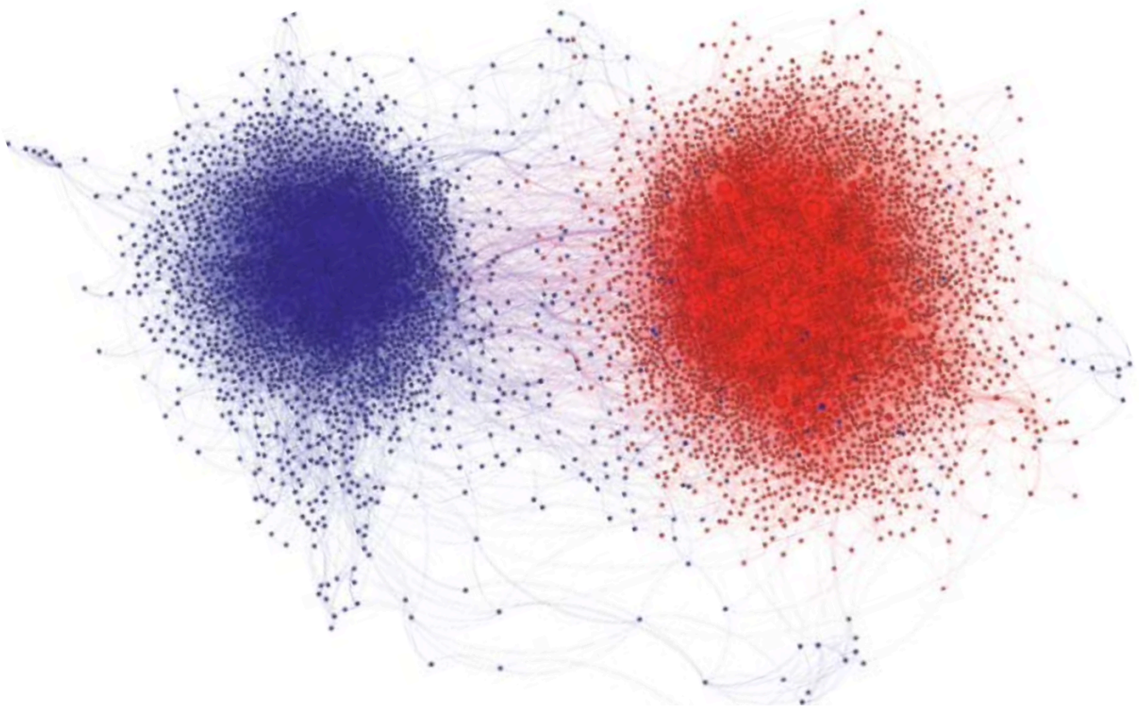
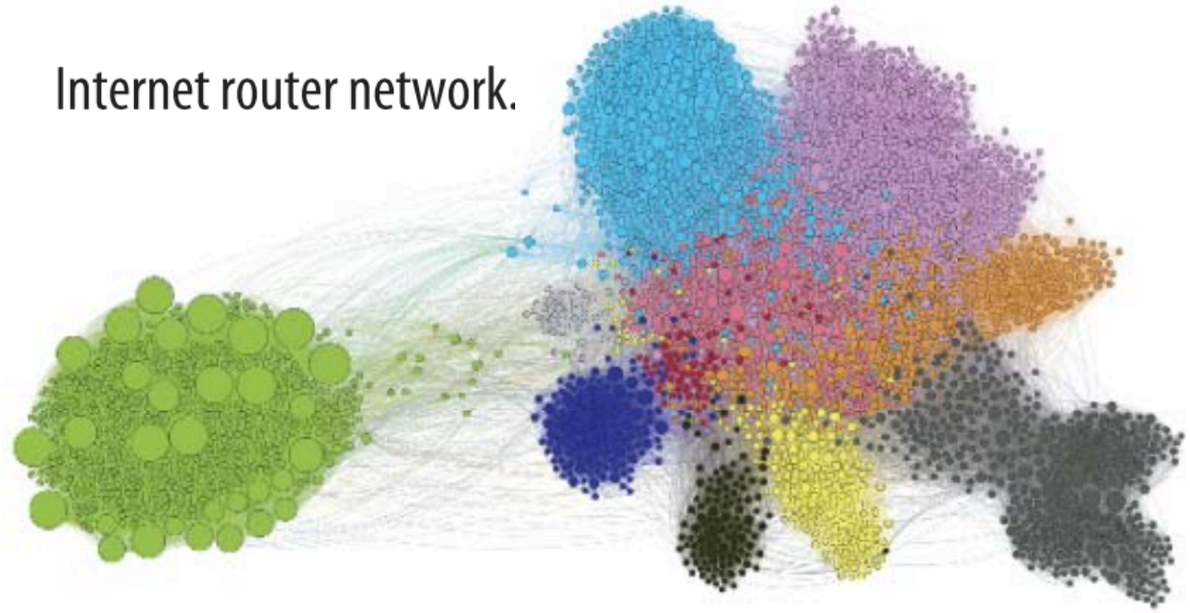
NETWORK FUNDAMENTALS

Motivation



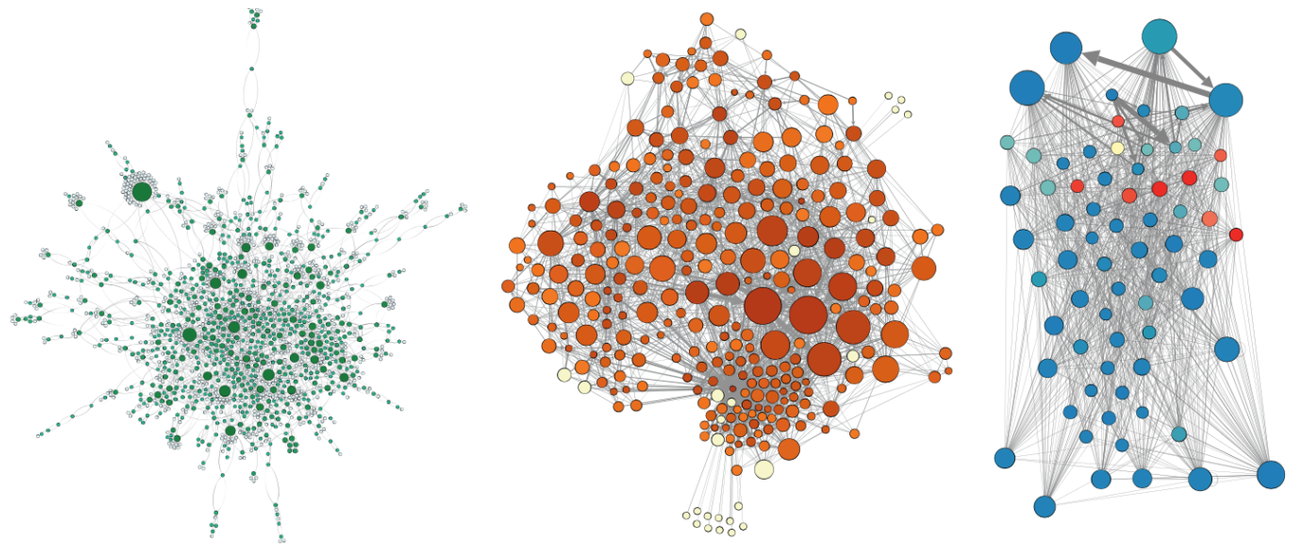
Twitter, among people sharing posts about US politics.

Internet router network.

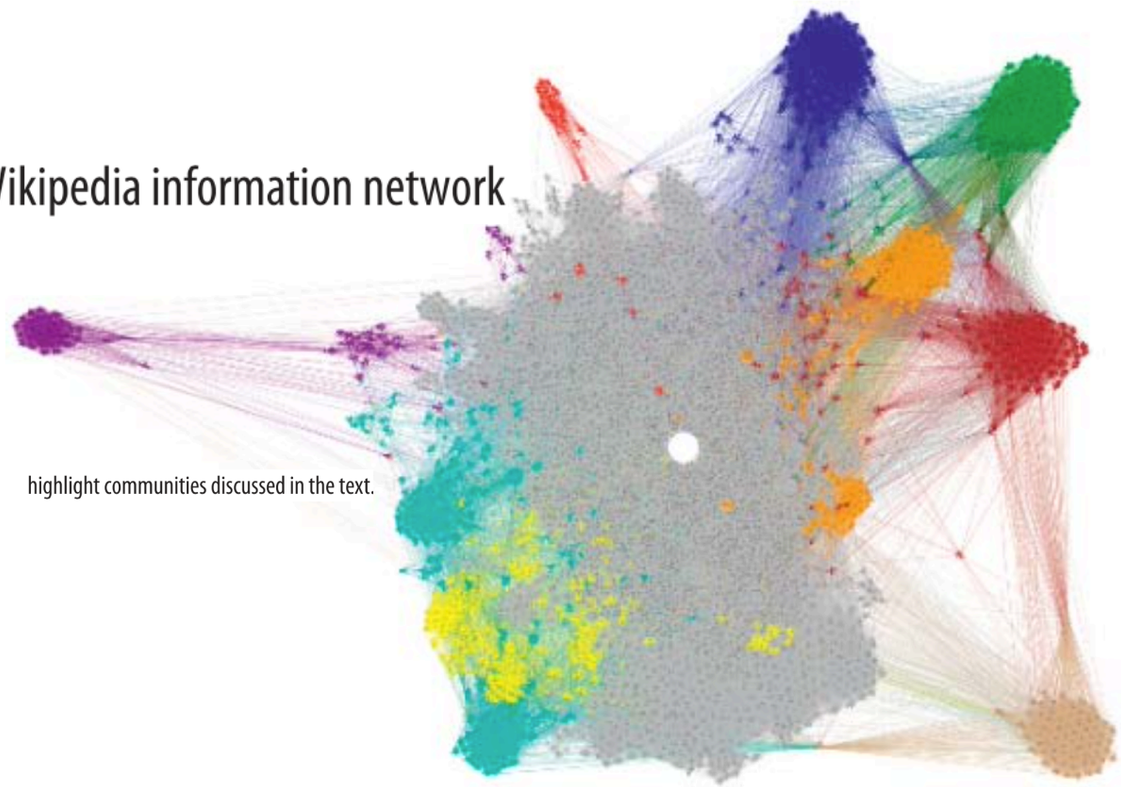


community detection algorithm that identifies dense clusters reflecting the geographic distribution of routers.

Three biological networks. Left: Protein interaction network of yeast.

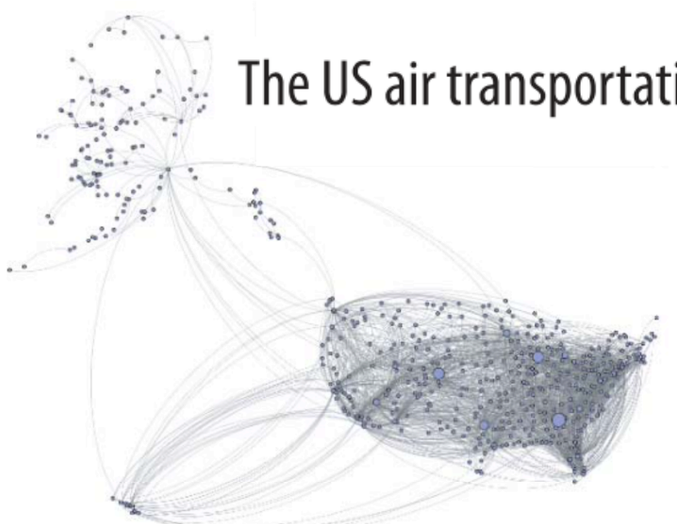


Wikipedia information network



highlight communities discussed in the text.

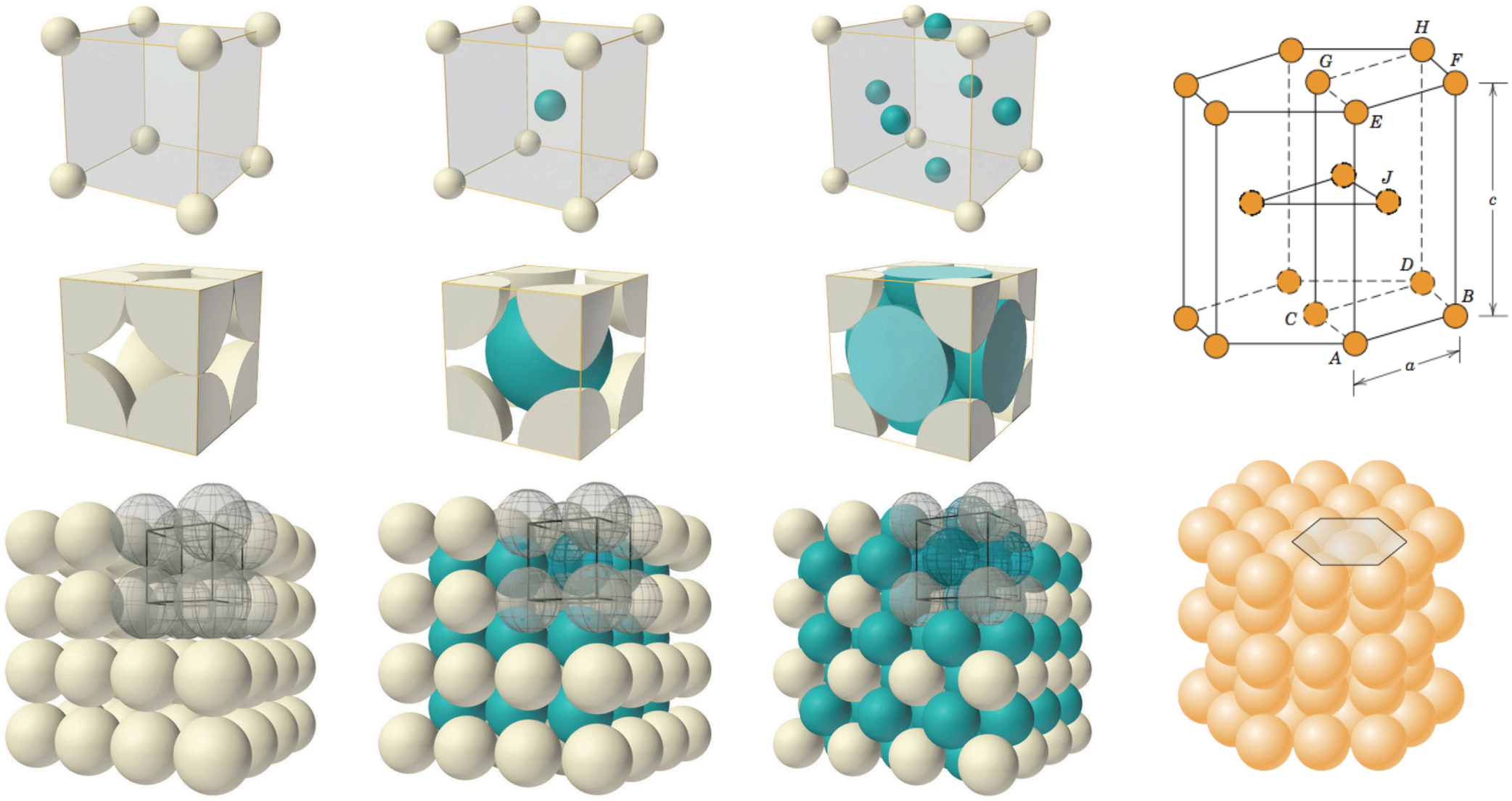
The US air transportation



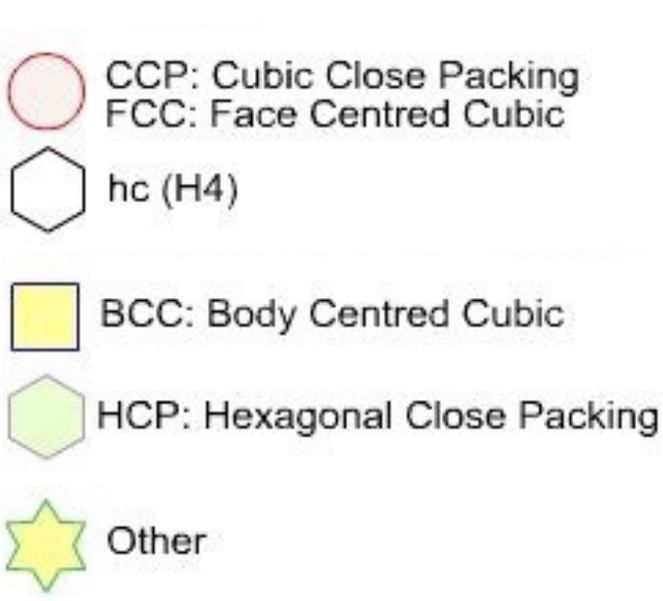
Source: A First Course in NETWORK SCIENCE — Filippo Menczer, Santo Fortunato and Clayton A. Davis

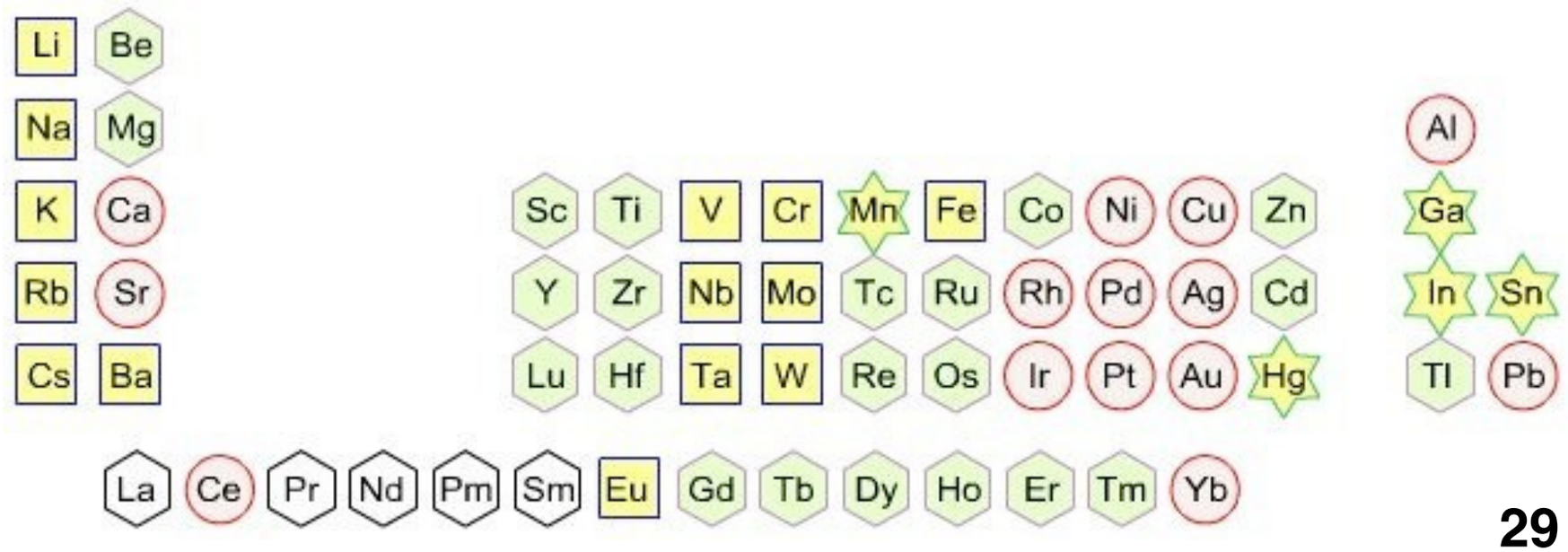
Example-3

SC BCC FCC HCP



(a) Simple cubic (b) Body-centered cubic (c) Face-centered cubic d) Hexagonal





Software tools





igraph is an R connector to the igraph collection of network analysis tools.
An open-source collection of network analysis tools with connectors in R, Python, Mathematica and C/C++.

visNetwork is an R package for interactive network visualization, built on the vis.js Javascript library.

Prepping data, graph/network analysis

- tidygraph
- igraph

Network analysis specific

- sna
- network

Visualizing

- ggraph
- igraph
- networkD3





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

(analysis tool) → NetworkX object → Visualization tools

NetworkX.draw()
PyVis

HvPlot
python-iGraph

Plotly

DyNetx - Dynamic Network library

DyNetx is a Python software package that extends `networkx` with dynamic network models



Gephi is the leading visualization and exploration software for all kinds of graphs and networks. Gephi is open-source and free.

Runs on Windows, Mac OS X and Linux.

Additional software tools

SOURCE: https://guides.library.jhu.edu/datavisualization/network



Gephi

Gephi is the leading visualization and exploration software for all kinds of graphs and networks. Gephi is open-source and free.



Cytoscape

Cytoscape is an open source software platform for visualizing complex networks and integrating these with any type of attribute data.



GraphVis

Graphviz is open source graph visualization software. It has several main graph layout programs. It also has web and interactive graphical interfaces, and auxiliary tools, libraries, and language bindings.



igraph

An open-source collection of network analysis tools with connectors in R, Python, Mathematica and C/C++.



UCINET

UCINET 6 for Windows is a software package for the analysis of social network data. UCINET comes with the NetDraw network visualization tool.



SocNetV

The Social Network Analysis and Visualization Software (SocNetV) is a fee and open-source tool for social network analysis.



Pajek

Analysis and visualization of very large networks on Windows. Pajek is a free alternative to UCINET.

Introduction

A network (graph), is a set of elements, called nodes, along with a set of connections between pairs, called links

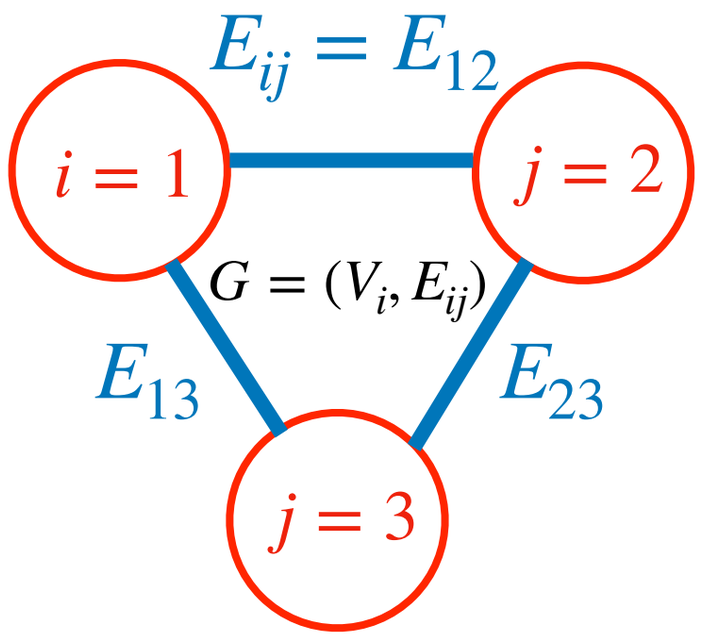
Networks provide a theoretical framework allowing for conceptual representation of interrelations in a wide array of systems;

Vertices/nodes
(objects/entities)

$V_i \quad i \in [1,N]$

$N = 3 \quad vertices$

N=**Number of nodes**



Edges/links
(connections)

$E_{ij} \quad i,j \in [1,N]$

Eij is an object, commonly a scalar or a dictionary, describing the link

L=N**umber of links**

Nomenclature

Network Science	Graph Teory
Network	Graph
Node	Vertex
Link	Edge

Types of links

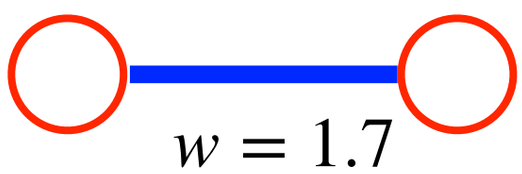
There are different types of links, which define different classes of networks.



un-directed

(symmetric exchange)

"two way street"



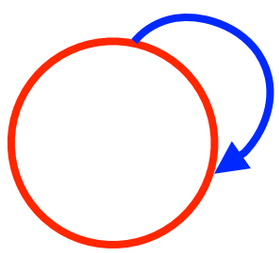
weighted



Directed

(asymmetric exchange)

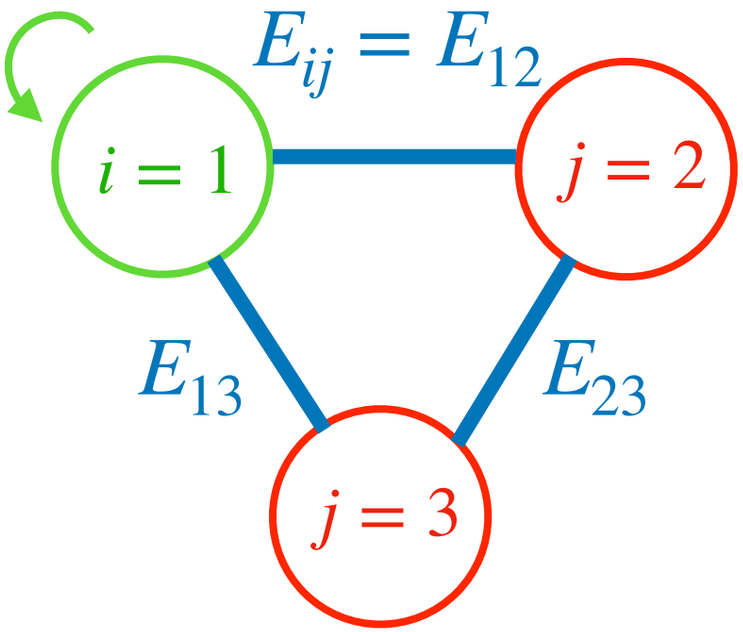
"one way street"



loop

(self connection)

The links represent the presence of a relation among the elements represented by the nodes. As we have seen earlier, links can correspond to social, physical, communication, geographic, conceptual, chemical, biological, or other interactions.



$$\mathbf{E}_{ij} = \begin{pmatrix} E_{11} & E_{12} & E_{13} \\ E_{21} & E_{22} & E_{23} \\ E_{31} & E_{32} & E_{33} \end{pmatrix}$$

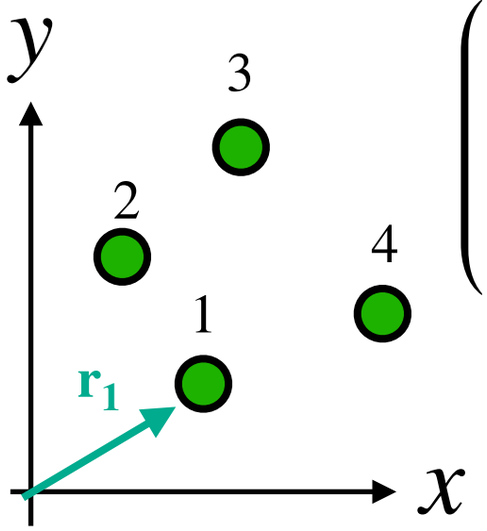
Pair-wise relationships

N^2 *possible edges*

"self relationships"

common to call connected nodes neighbors.

Example-2

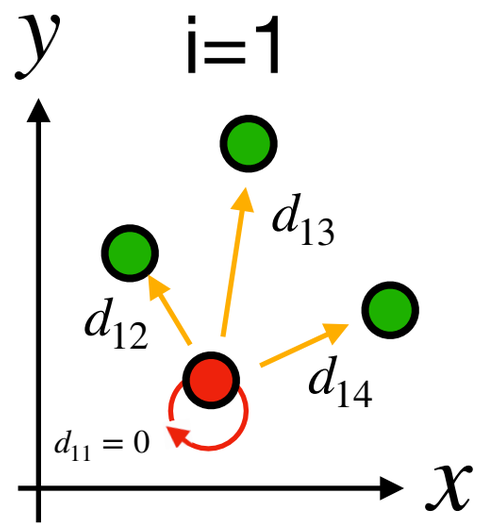


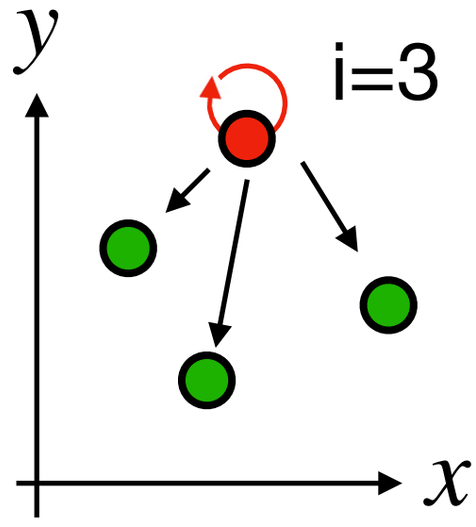
$$\begin{pmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \mathbf{r}_3 \\ \mathbf{r}_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{pmatrix}$$

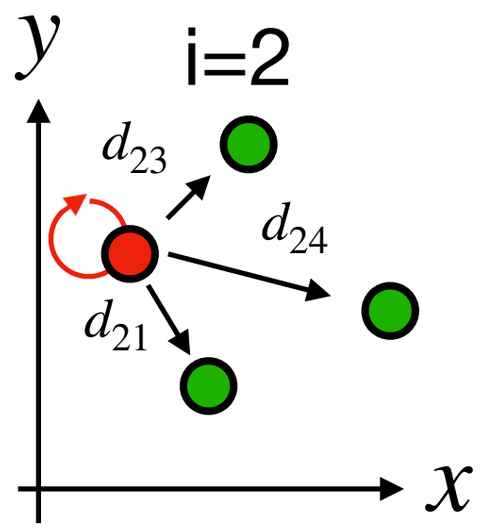
$N = 4$ **data point**

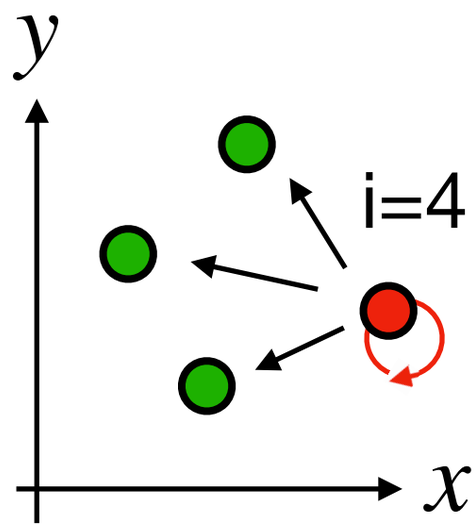
L_2 **norm: Euclidean distance**

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$









$N^2 = 16$ **pairwise distances**

$$\mathbf{d}_{ij} = \begin{pmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \\ d_{41} & d_{42} & d_{43} & d_{44} \end{pmatrix}$$

zero main diagonal $\mathbf{d}_{ii} = 0$

symetric $\mathbf{d}_{ij} = \mathbf{d}_{ji}$

Any collection of N points forms, at most, N^2 inter-connections (i.e. a fully connected network)

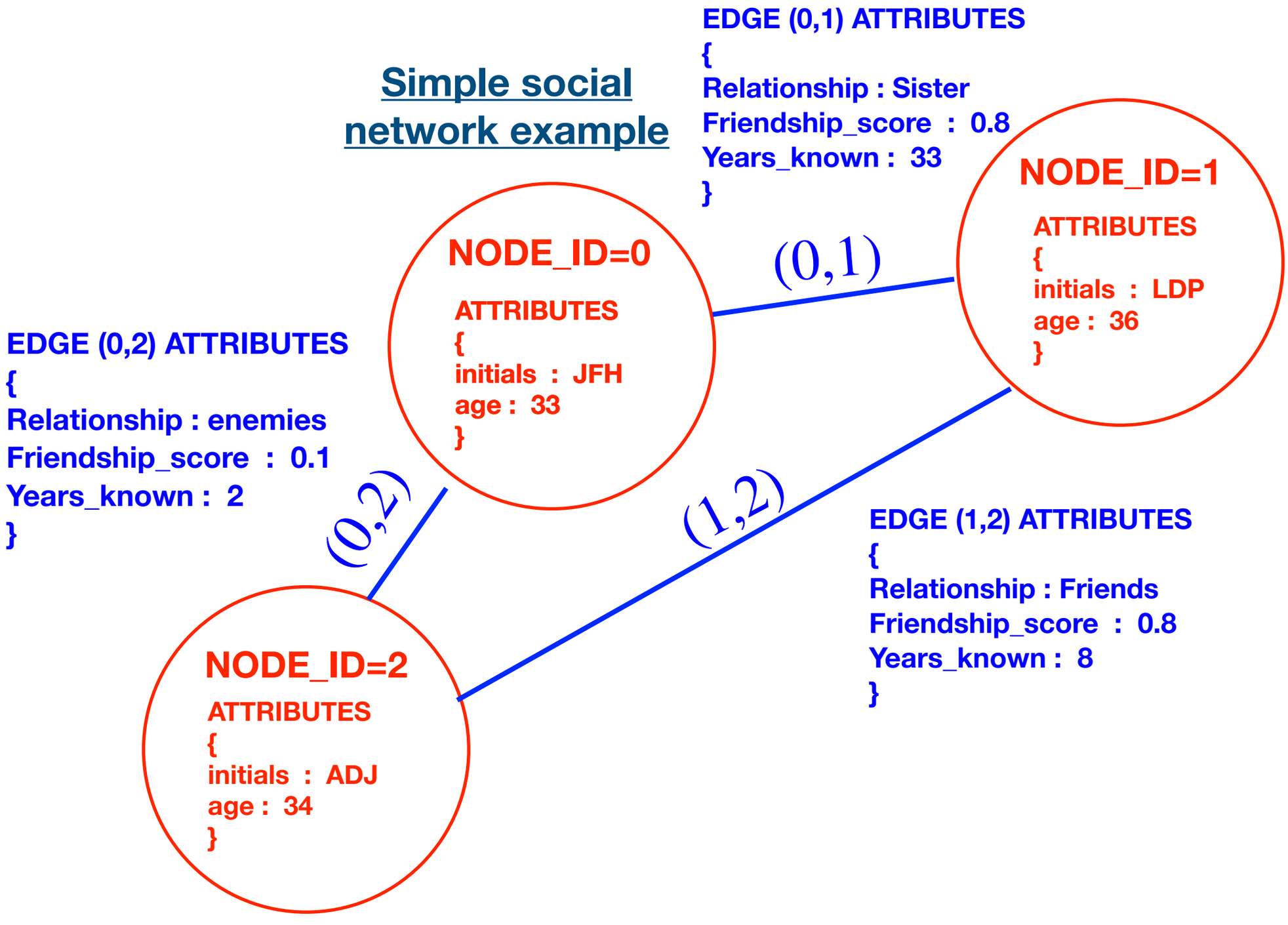
$$\frac{N^2}{2} - N = 6$$ **unique pairwise distances**

Main idea: Often in data science, we are interested in a collection of N data-points, in network analysis we are interested in the N^2 (or less) connections between those points

Node and edge Attributes

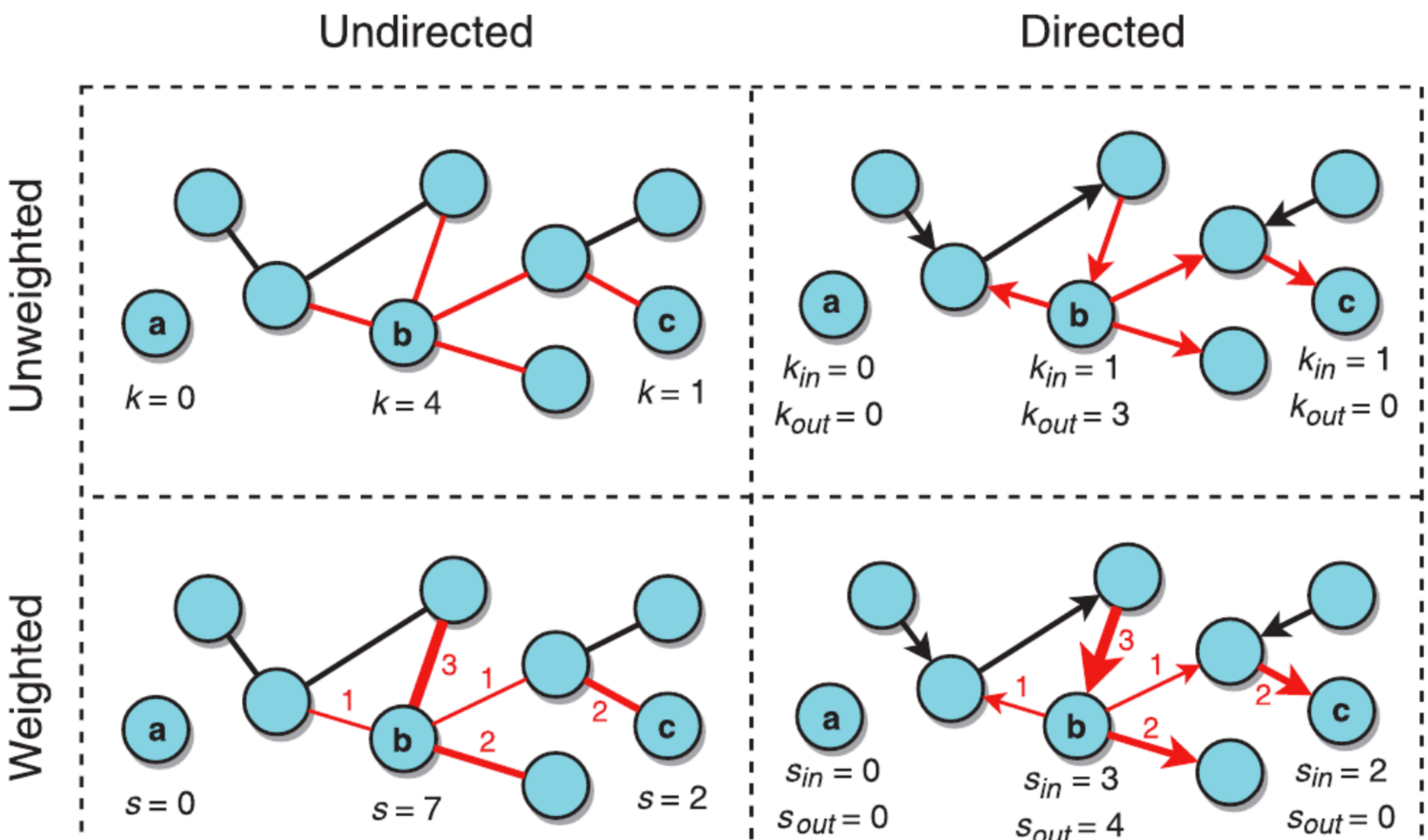
- In NetworkX each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hash-able).

- Default values are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.



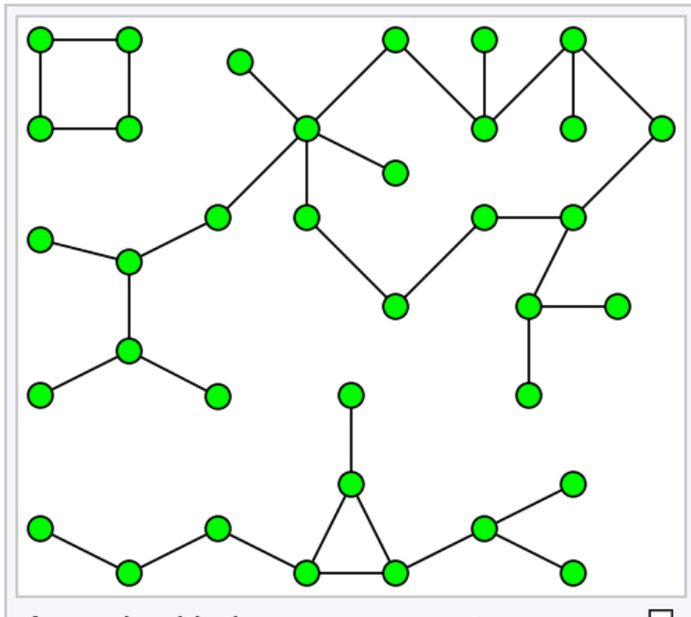
Sub-networks

Often, we are interested in a subset of a network, which is itself a network and is called a subnetwork (or subgraph).

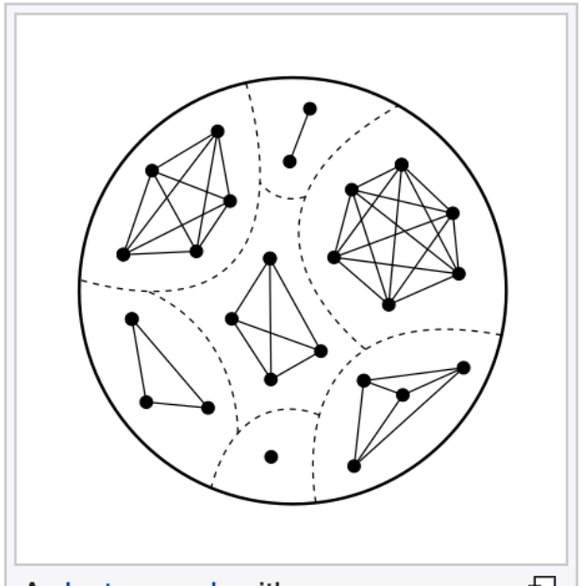


Components (islands)

A component of a graph is a connected subgraph that is **not** part of any larger connected subgraph (i.e. an island). Components are sometimes called “connected components”.

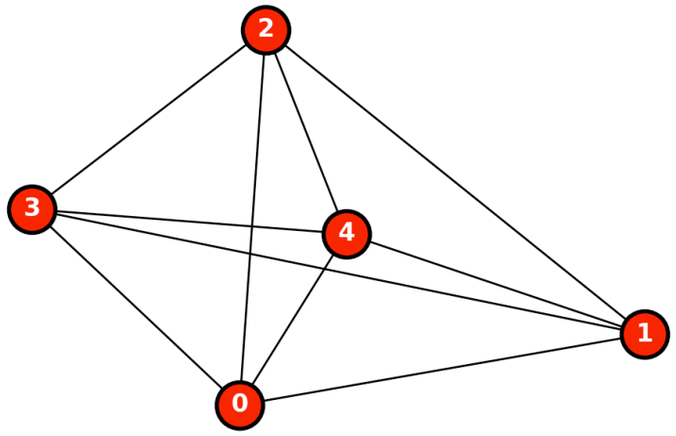


A graph with three components



A cluster graph with seven components

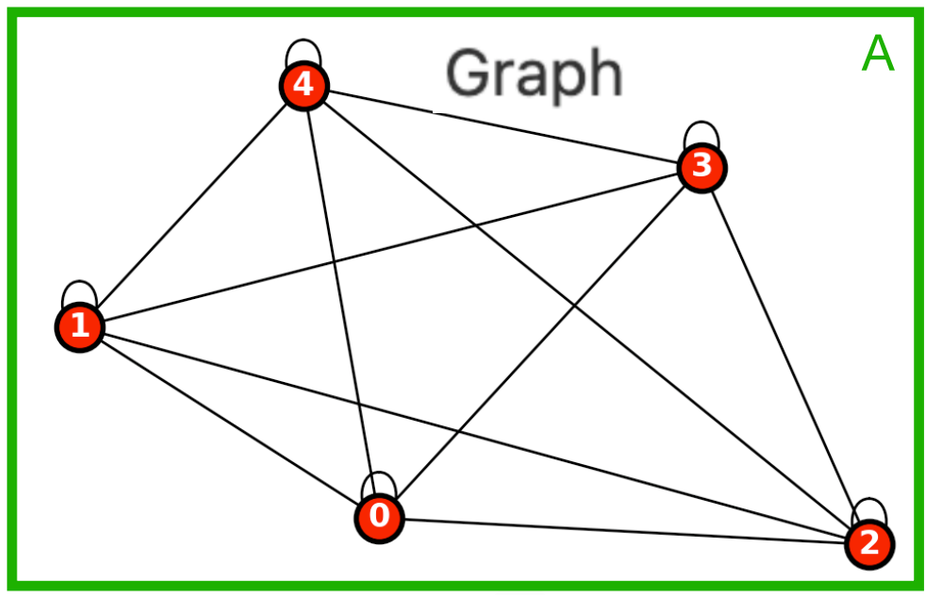
Types in NetworkX

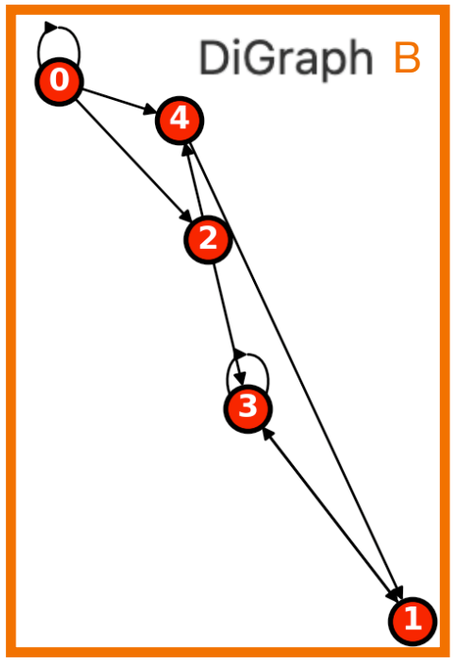


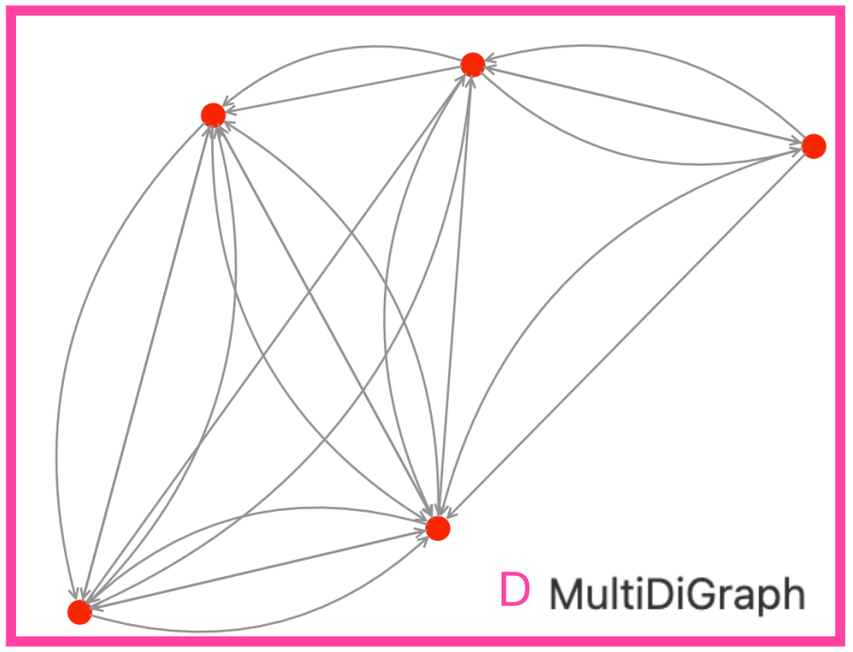
SIMPLE: undirected+no loops

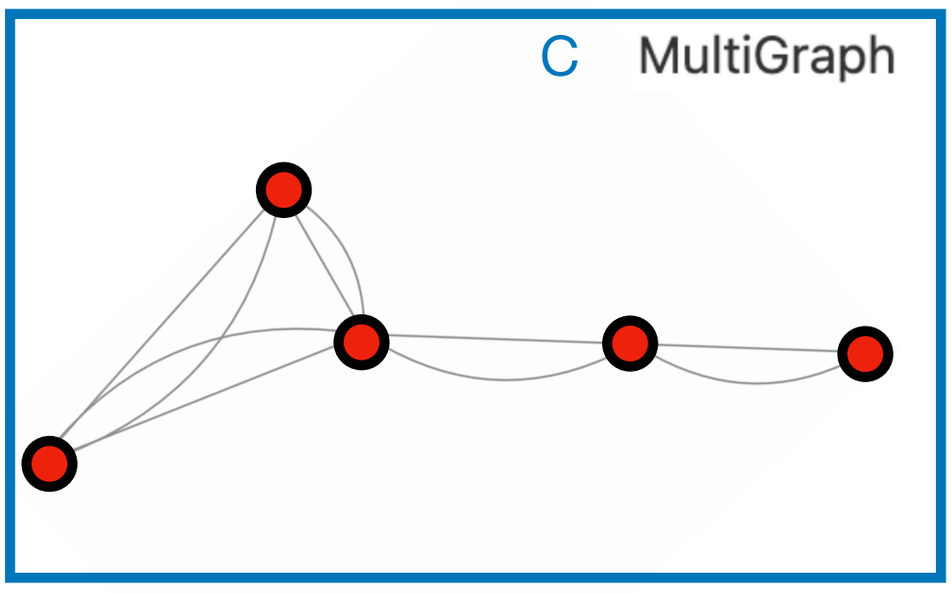
NetworkX
Network Analysis in Python

Type	Self-loops allowed	Parallel edges allowed	
Graph	undirected	Yes	No	A
DiGraph	directed	Yes	No	B
MultiGraph	undirected	Yes	Yes	C
MultiDiGraph	directed	Yes	Yes	D









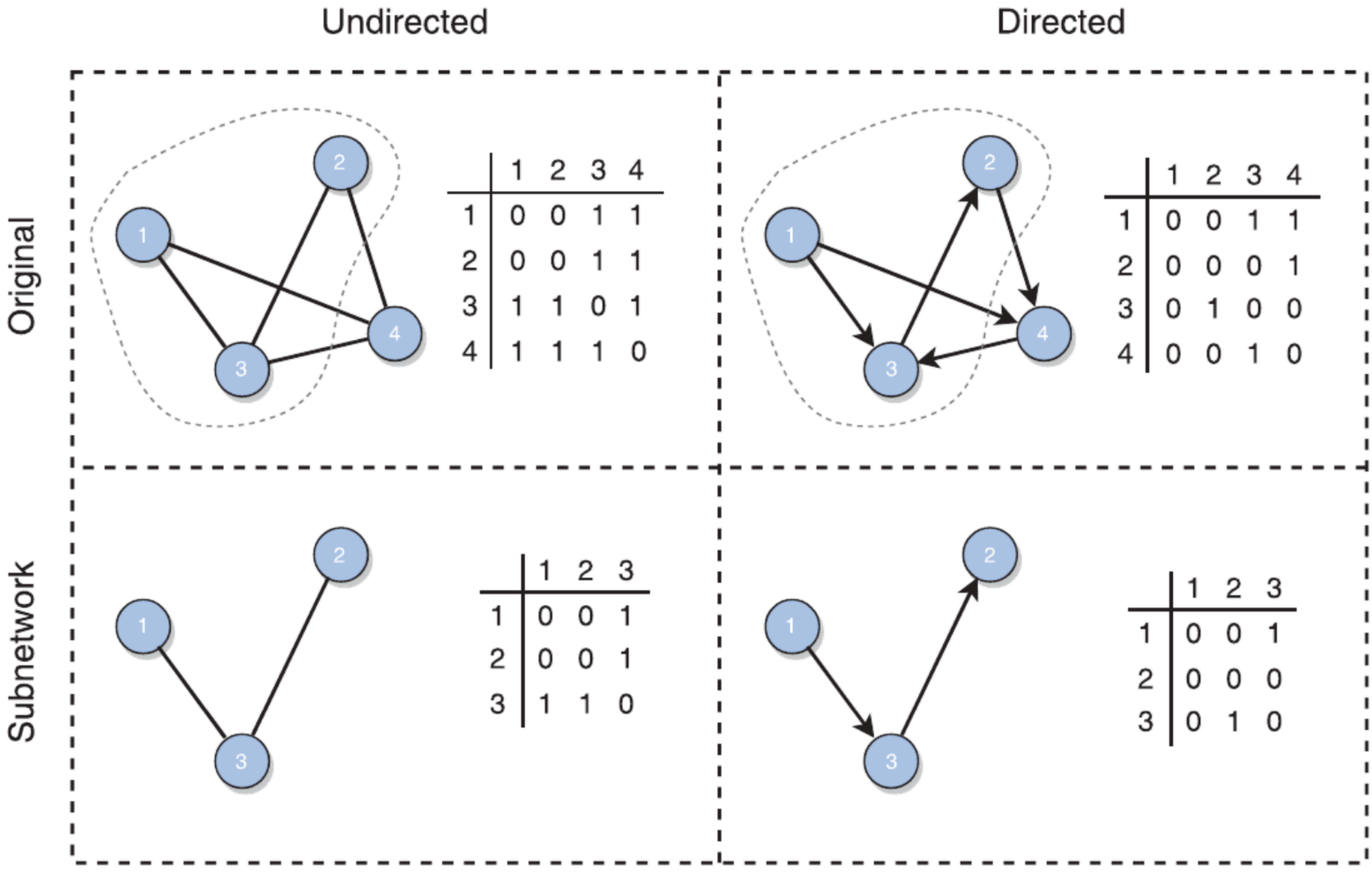
Definition of a Network

A network G has two parts, a set of N elements, called *nodes* or *vertices*, and a set of L pairs of nodes, called *links* or *edges*. The link (i, j) joins the nodes i and j. A network can be *undirected* or *directed*. A directed network is also called a *digraph*. In directed networks, links are called *directed links* and the order of the nodes in a link reflects the direction: the link (i, j) goes from the source node i to the target node j. In undirected networks, all links are bi-directional and the order of the two nodes in a link does not matter. A network can be *unweighted* or *weighted*. In a weighted network, links have associated *weights*: the *weighted link* (i, j, w) between nodes i and j has weight w. A network can be both directed and weighted, in which case it has directed weighted links.

Network Representations

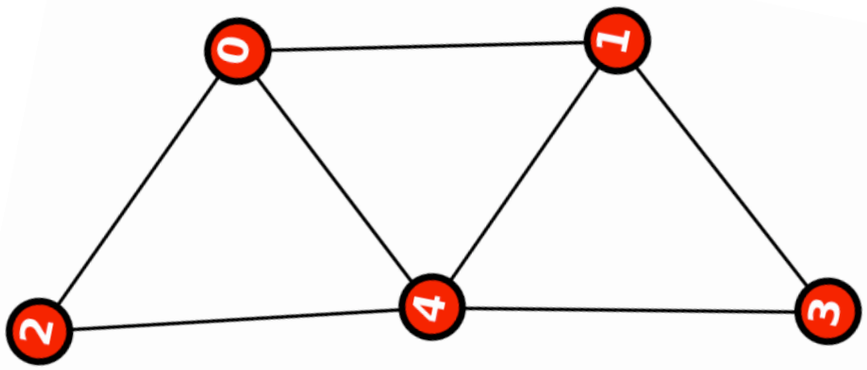
Adjacency matrix

Element a_{ij} of the adjacency matrix represents the link between nodes i and j. $a_{ij} = 1$ if i and j are adjacent, $a_{ij} = 0$ otherwise.



Network and subnetwork examples. We also show the adjacency matrix representation of each network

Example-1



EDGELIST:

```
0 1 {}
0 2 {}
0 4 {}
1 3 {}
1 4 {}
2 4 {}
3 4 {}
```

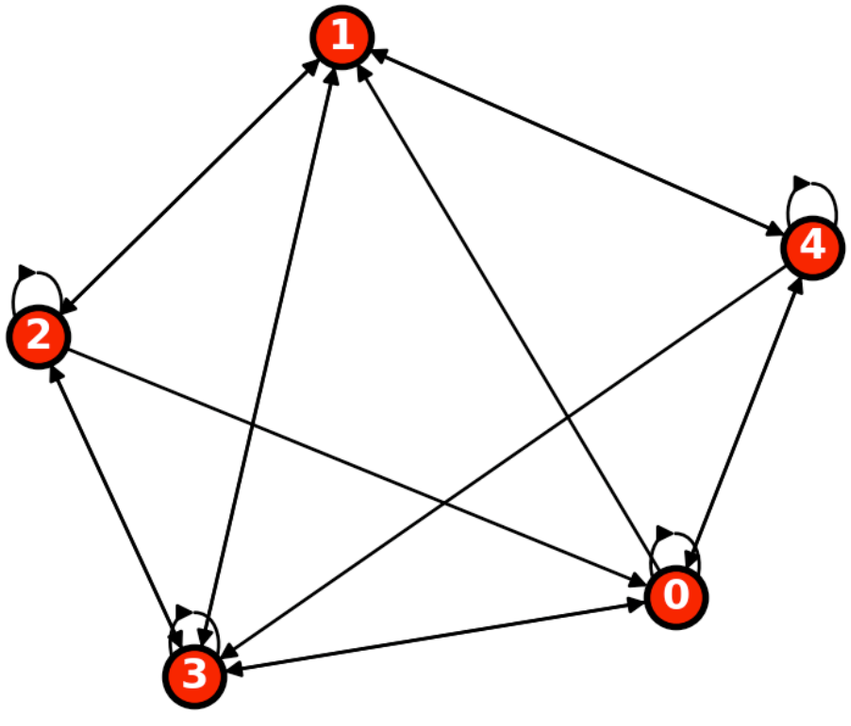
ADJACENCY MATRIX:

	0	1	2	3	4
0	0	1	1	0	1
1	1	0	0	1	1
2	1	0	0	0	1
3	0	1	0	0	1
4	1	1	1	1	0

ADJACENCY LIST:

```
0 1 2 4
1 3 4
2 4
3 4
4
```

Example-2



EDGELIST:

```
0 0 {}
0 1 {}
0 3 {}
0 4 {}
1 2 {}
1 3 {}
1 4 {}
2 0 {}
2 1 {}
2 2 {}
2 3 {}
3 0 {}
3 1 {}
3 2 {}
3 3 {}
4 0 {}
4 1 {}
4 3 {}
4 4 {}
```

ADJACENCY MATRIX:

	0	1	2	3	4
0	1	1	0	1	1
1	0	0	1	1	1
2	1	1	1	1	0
3	1	1	1	1	0
4	1	1	0	1	1

ADJACENCY LIST:

```
0 0 1 3 4
1 2 3 4
2 0 1 2 3
3 0 1 2 3
4 0 1 3 4
```

Network-X : Building Graphs https://networkx.org/documentation/stable/reference/classes/graph.html

From “scratch”:

1) Initialize an empty graph object

```
import networkx as nx
G = nx.Graph()      OR G = nx.DiGraph()
G = nx.MultiGraph() OR G = nx.MultiDiGraph()
```

2) Add nodes

```
#ONE AT A TIME
G.add_node(1)

#MANY AT A TIME
G.add_nodes_from([2, 3])
G.add_nodes_from(range(100, 110))
```

2) Add Edges

```
#ONE AT A TIME
G.add_edge(1, 2)

#MANY AT A TIME
G.add_edges_from([(1, 2), (1, 3)])
```

3) Add Node/Edge Attributes

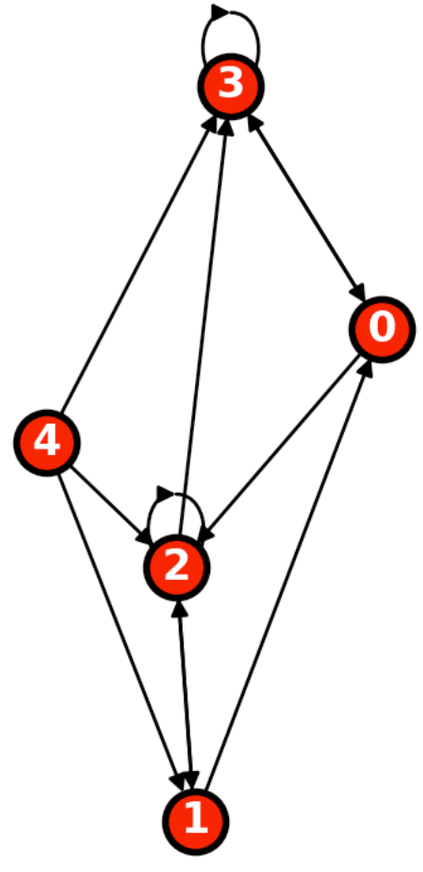
Add node attributes using add_node(), add_nodes_from() or G.nodes

```
G.add_node(1, time="5pm")
G.add_nodes_from([3], time="2pm")
G.nodes[1]

G.nodes[1]["room"] = 714  # node must exist already to use G.nodes
del G.nodes[1]["room"]  # remove attribute
list(G.nodes(data=True))
```

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edges.

```
G.add_edge(1, 2, weight=4.7)
G.add_edges_from([(3, 4), (4, 5)], color="red")
G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
G[1][2]["weight"] = 4.7
G.edges[1, 2]["weight"] = 4
```



Network-X: Generators:

https://networkx.org/documentation/stable/reference/generators.html

```
import networkx as nx
#GENERATE A FULLY CONNECTED GRAPH WITH 100 NODES
G = nx.complete_graph(100)
```

SOME EXAMPLES:

Classic

Generators for some classic graphs.

balanced_tree(r, h[, create_using])	Returns the perfectly balanced **r**-ary tree of height **h**.
barbell_graph(m1, m2[, create_using])	Returns the Barbell Graph: two complete graphs connected by a path.
binomial_tree(n[, create_using])	Returns the Binomial Tree of order n.
complete_graph(n[, create_using])	Return the complete graph **K_n** with n nodes.
complete_multipartite_graph(*subset_sizes)	Returns the complete multipartite graph with the specified subset sizes.
circular_ladder_graph(n[, create_using])	Returns the circular ladder graph CL_n of length n.
circulant_graph(n, offsets[, create_using])	Returns the circulant graph $Ci_n(x_1, x_2, \ldots, x_m)$ with n nodes.
cycle_graph(n[, create_using])	Returns the cycle graph C_n of cyclically connected nodes.
dorogovtsev_goltsev_mendes_graph(n[, ...])	Returns the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.
empty_graph([n, create_using, default])	Returns the empty graph with n nodes and zero edges.
full_rary_tree(r, n[, create_using])	Creates a full r-ary tree of **n** nodes.
ladder_graph(n[, create_using])	Returns the Ladder graph of length n.

Directed

Generators for some directed graphs, including growing network (GN) graphs and scale-free graphs.

gn_graph(n[, kernel, create_using, seed])	Returns the growing network (GN) digraph with **n** nodes.
gnr_graph(n, p[, create_using, seed])	Returns the growing network with redirection (GNR) digraph with **n** nodes and redirection probability **p**.
gnc_graph(n[, create_using, seed])	Returns the growing network with copying (GNC) digraph with **n** nodes.
random_k_out_graph(n, k, alpha[, ...])	Returns a random **k**-out graph with preferential attachment.
scale_free_graph(n[, alpha, beta, gamma, ...])	Returns a scale-free directed graph.

Social Networks

Famous social networks.

karate_club_graph()	Returns Zachary's Karate Club graph.
davis_southern_women_graph()	Returns Davis Southern women social network.
florentine_families_graph()	Returns Florentine families graph.
les_miserables_graph()	Returns coappearance network of characters in the novel Les Miserables.

... AND MANY MORE!!!!

Network Properties

Number of nodes (N):

Number of links (L):

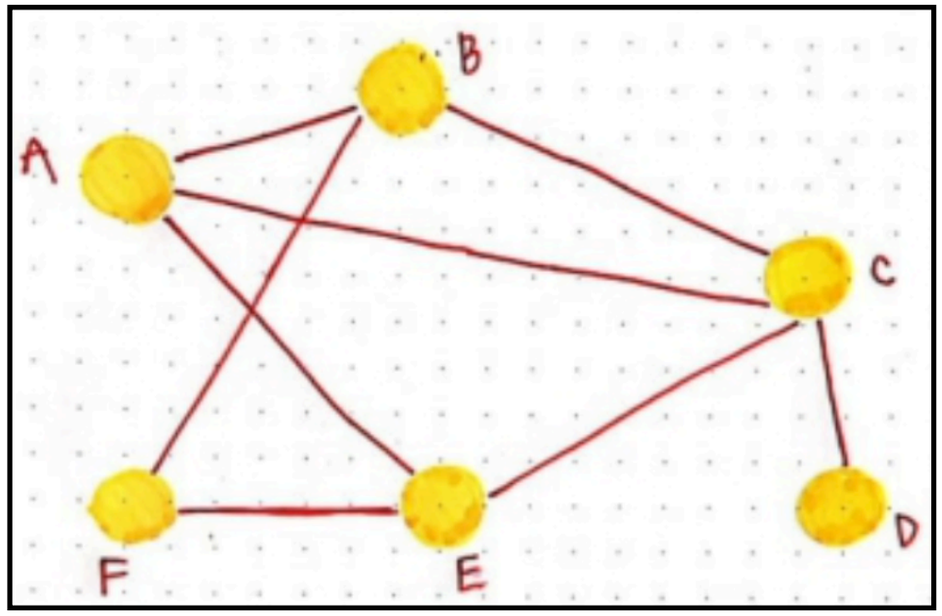
Density (d): Graph density represents the ratio between the edges present in a graph and the maximum number of edges that the graph can contain

Average degree (k): Average number of links per node

Diameter: The diameter of a graph is the length of the "longest shortest path" between any two graph vertices

Degree

The number of links incident upon a node (i.e., the number of ties that a node has).



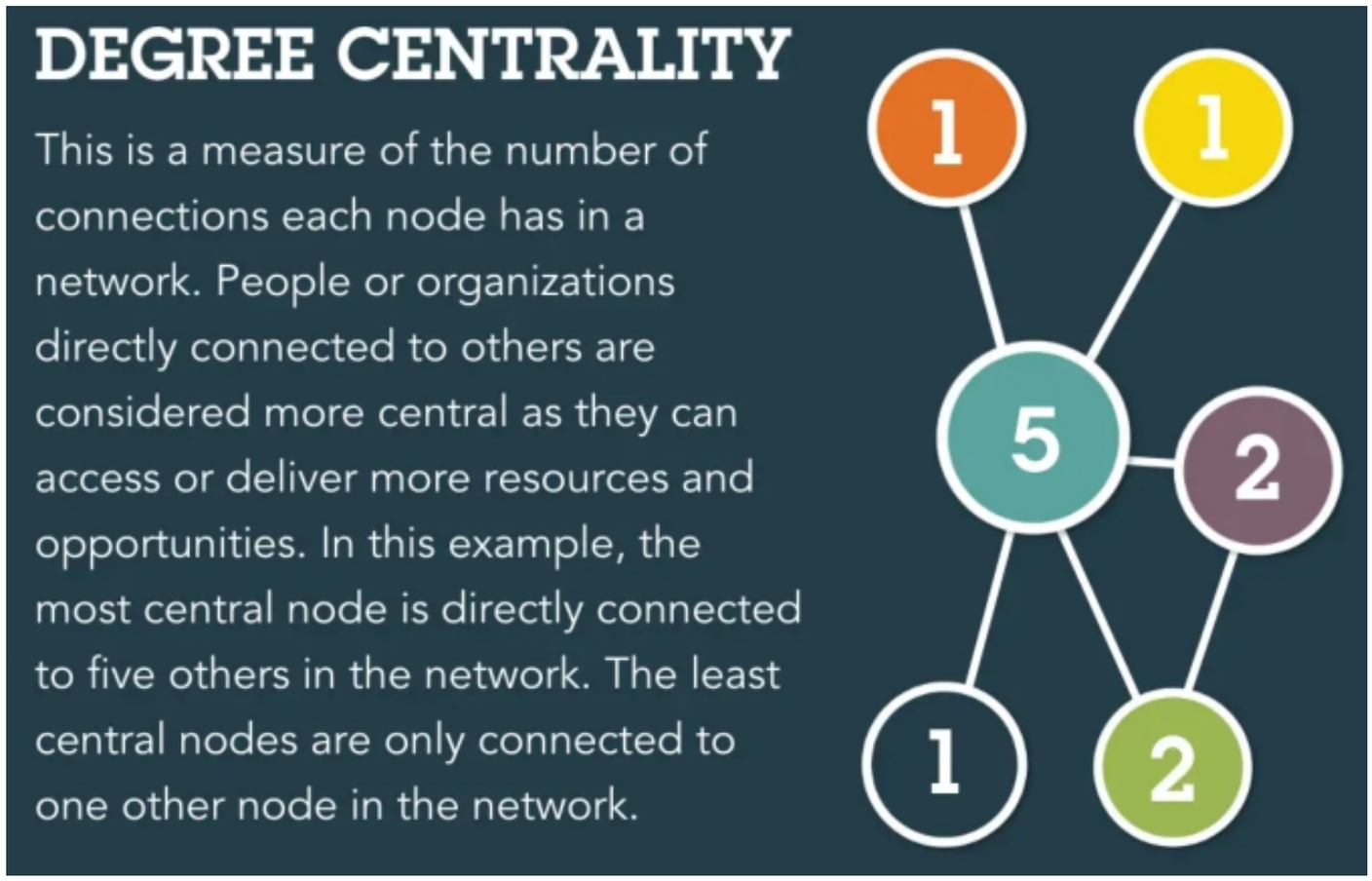
* Two vertices are adjacent if they share a common edge (an edge joins the vertices).

* The degree of a vertex is the total number of vertices that are adjacent to the vertex.

VERTEX	DEGREE
A	3
B	3
C	4
D	1
E	3 → odd vertex
F	2 → even vertex

- Degree Quantifies risk of a node catching something is flowing through the network (such as a virus, or meme

- directed network (where ties have direction), we usually define two separate measures of degree centrality,
namely indegree and outdegree



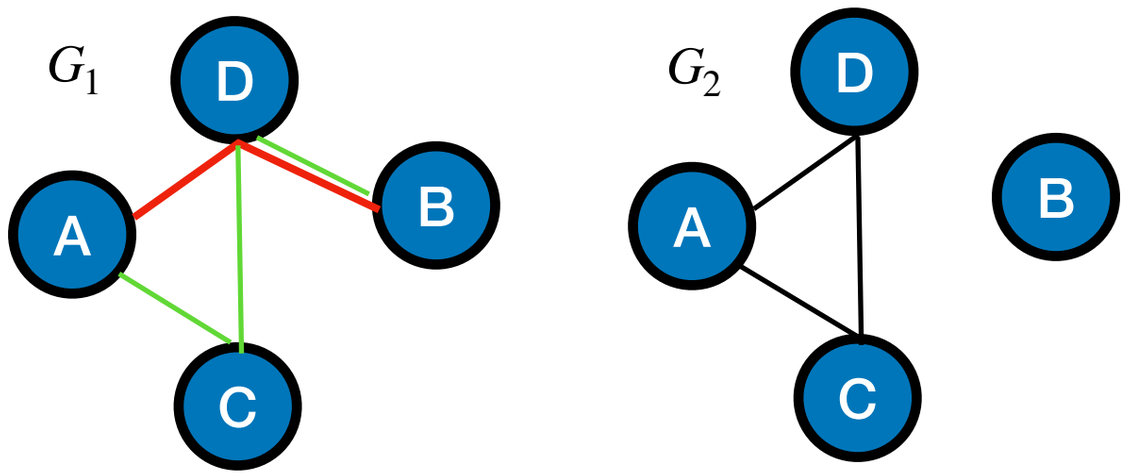
NetworkX:

```
import networkx as nx
print("DEGREE:")
print(nx.degree(G))

print("DEGREE HISTOGRAM:")
print(nx.degree_histogram(G))
```

Paths

- **Existence**: A path between nodes A and B exists if it is possible to go from node-A to node-B by traversing the links in the network



- **singleton**: a node, with no neighbors (i.e. degree=zero)

- The path is the **sequence** of links traversed. $P_1 = (e_{AD}, e_{DB})$ $P_2 = (e_{AC}, e_{CD}, e_{DB})$

  ** Typically there are multiple paths between the same two nodes.

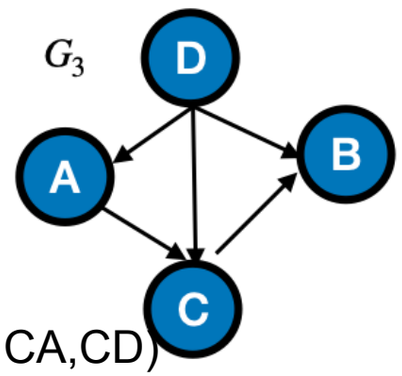
- **Path length:** The number of links in a path. $l_{AB}^1 = 2$ $l_{AB}^2 = 3$

- **Simple path:** Path that never goes through the same link more than once;

- **Distance** l_{AB} between two nodes is defined as the minimum number of links that must be traversed in a connecting path between them. Such a path is called the **shortest path**, and its length is called the **shortest-path length.**

  ** You can have multiple shortest paths between two nodes, but they all must have the same (shortest) length.

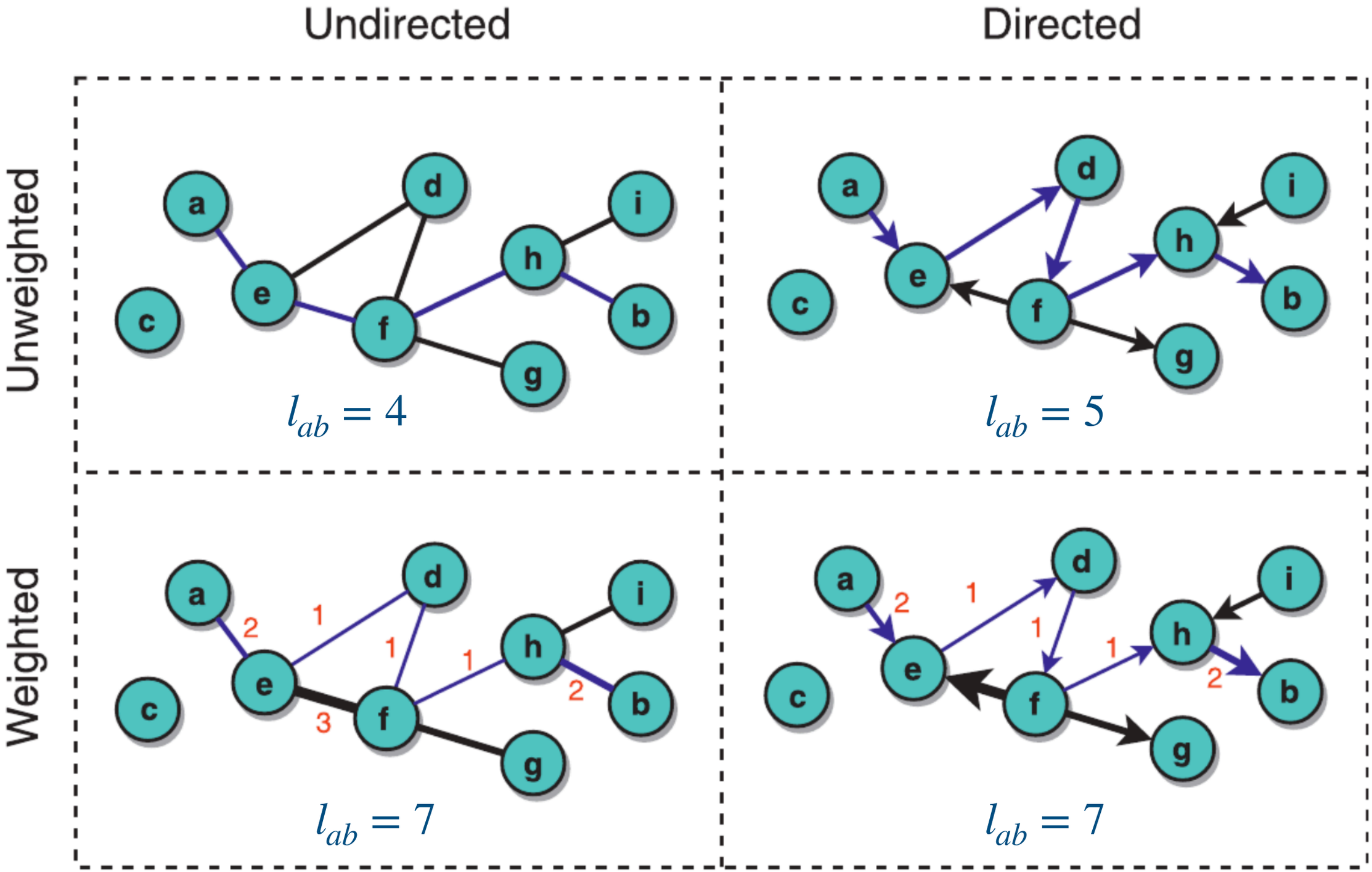
- **Directed networks**, paths must OBEY directed links
  - i.e. (paths can't go down one-way streets)



  ** Beware that in a directed network there may be no paths between some pairs of nodes. (e.g. CA,CD)

Examples

Sometimes, we can **associate a numeric distance with links** between adjacent nodes (e.g. geographic distance). In such cases we can redefine the path length as the sum of these distances
You can think of an unweighted network as a special case in which all links have distance one.

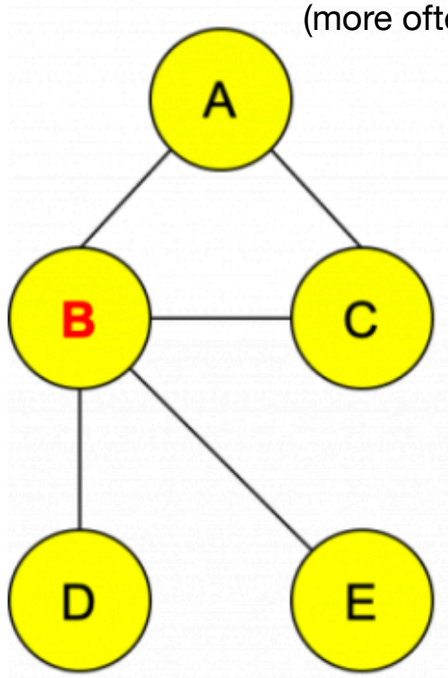


Shortest paths (blue) in undirected, directed, unweighted, and weighted networks. Link weights represent distances and are shown in red.

Distance matrix

Matrix containing the **distances (i.e. length of shortest paths)** between all Node in the Network

Un-directed Graph:



(more often distances are represented with a dictionary (only consider paths that exist)

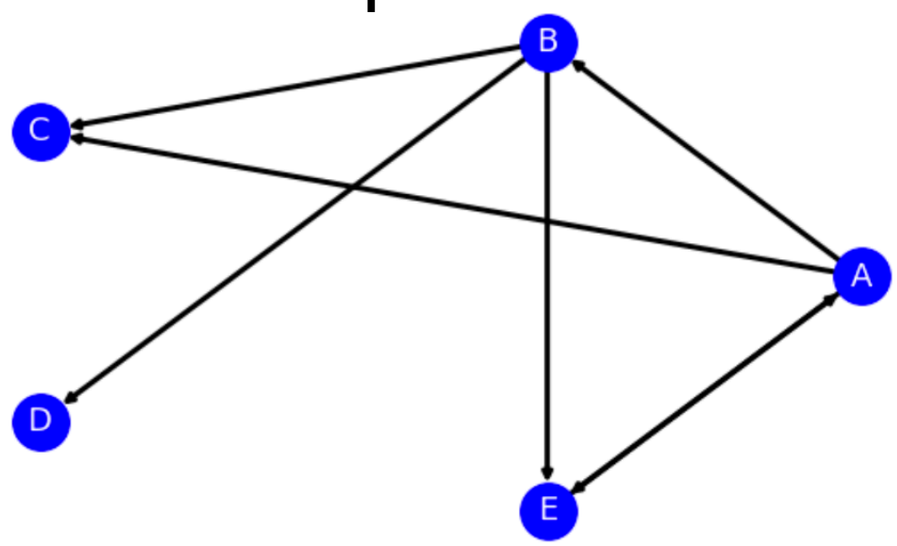
	A	B	C	D	E
A	0	1	1	2	2
B	1	0	1	1	1
C	1	1	0	2	2
D	2	1	2	0	2
E	2	1	2	2	0

(symetric)

$= l_{ij}$

zero represents the absence of a path, in this case technically $l_{ij} = \infty$

Directed Graph:




```
[[0. 1. 1. 2. 1.]
 [2. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [1. 2. 2. 3. 0.]]
```

$= l_{ij}$

(asymetric)

Average path length (Network property)**:** The average shortest-path length across all pairs of nodes.

$$\langle l \rangle = \underset{l_{ij} \neq 0}{\text{mean}}(l_{ij})$$

```
np.mean(lij[lij!=0]    print(nx.average_shortest_path_length(G))
```

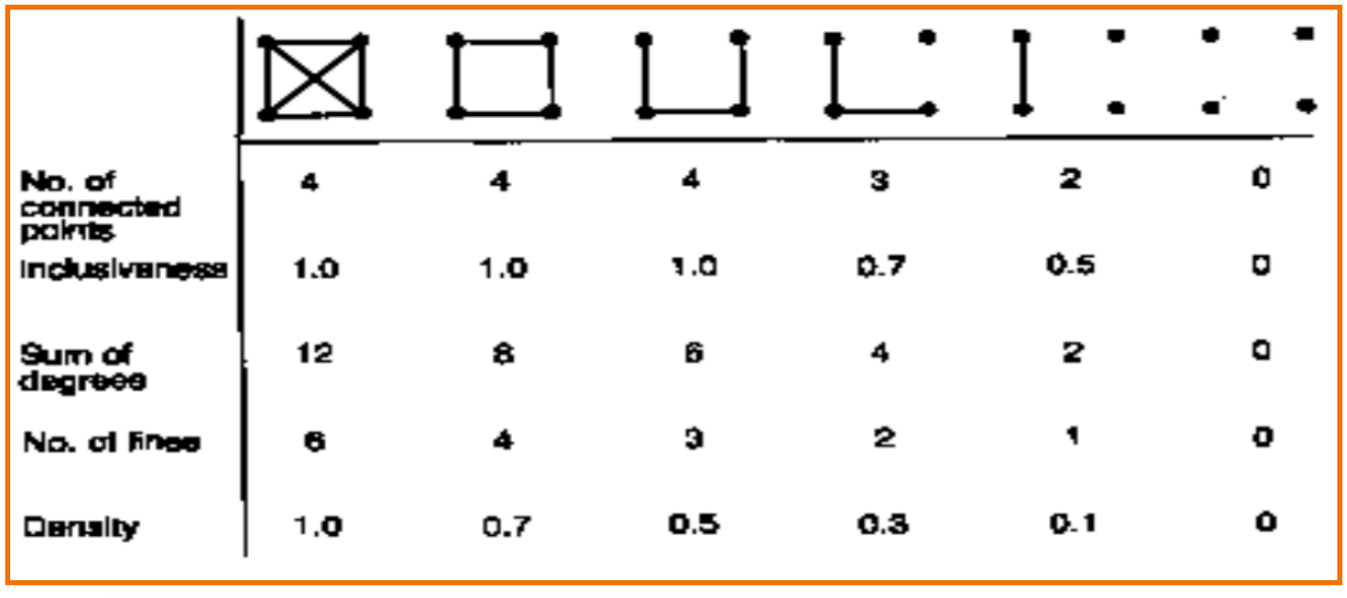
Density

- Measure of graph's connectivity
  - property that quantifies the entire graph (not just a single node or edge)
- Fraction of "occupied edges" out of "total edges"
- also known as "wiring probablity" or "connection probablity"

Un-directed graphs: $$d = \frac{\text{"filled" edges}}{\text{possible edges}} = \frac{m}{\frac{n^2 - n}{2}} = \frac{2m}{n(n-1)}$$

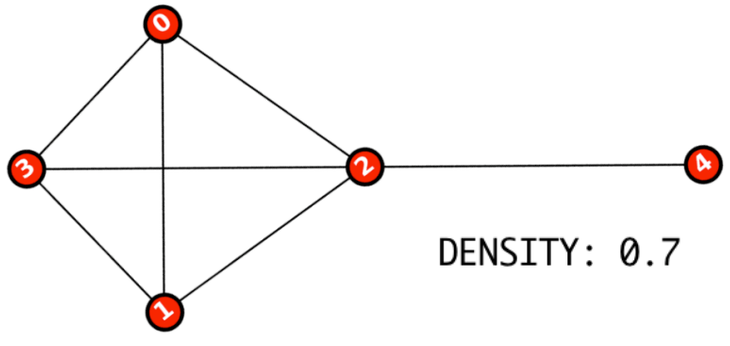
`n` is the number of nodes
`m` is the number of edges in `G`.

http://www.analytictech.com/mb119/chap4c.htm



No. of connected points	4	4	4	3	2	0
Inclusiveness	1.0	1.0	1.0	0.7	0.5	0
Sum of degrees	12	8	6	4	2	0
No. of lines	6	4	3	2	1	0
Density	1.0	0.7	0.5	0.3	0.1	0

d>1 : fully connected with loops
d=1 : "complete" (fully connected)
d=0 : No edges



DENSITY: 0.7

ADJACENCY MATRIX:

	0	1	2	3	4
0	0	1	1	1	0
1	1	0	1	1	0
2	1	1	0	1	1
3	1	1	1	0	0
4	0	0	1	0	0

- lower diagonal is redundant and not considered

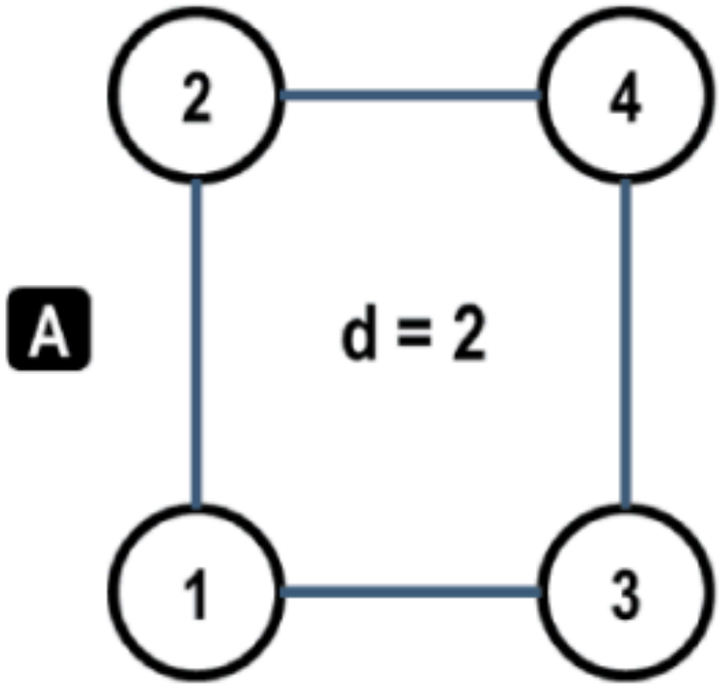
```
import networkx as nx
print("DENSITY:", nx.density(G))
```

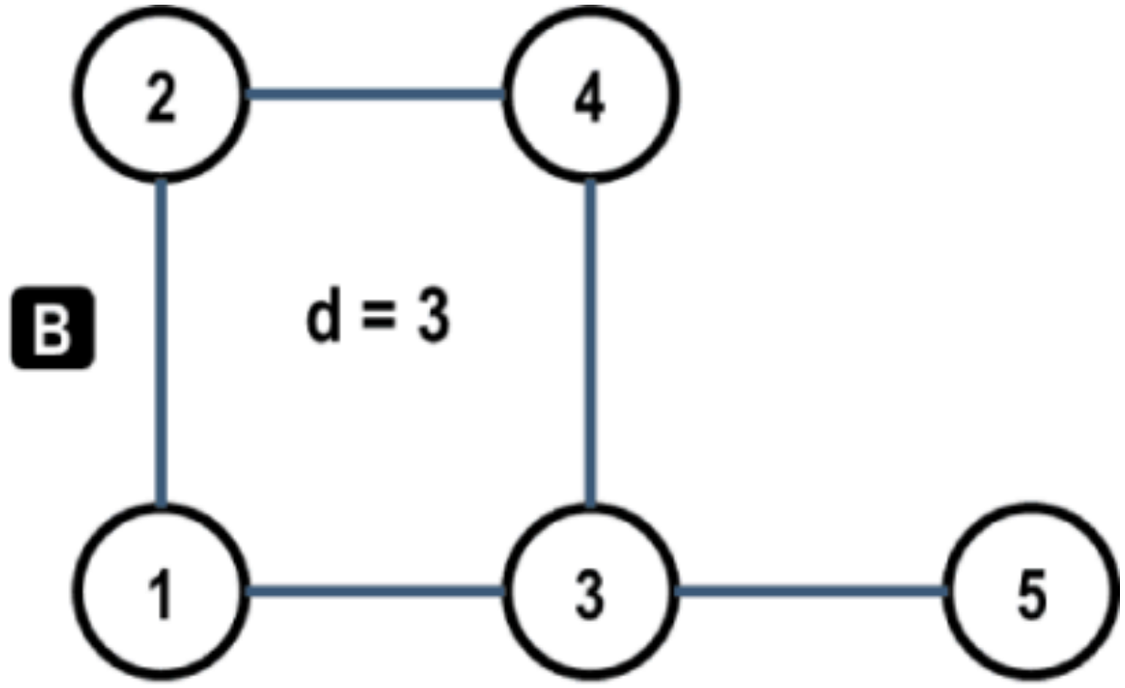
Directed graphs: $$d = \frac{\text{"filled" edges}}{\text{possible edges}} = \frac{m}{n^2 - n} = \frac{m}{n(n-1)}$$

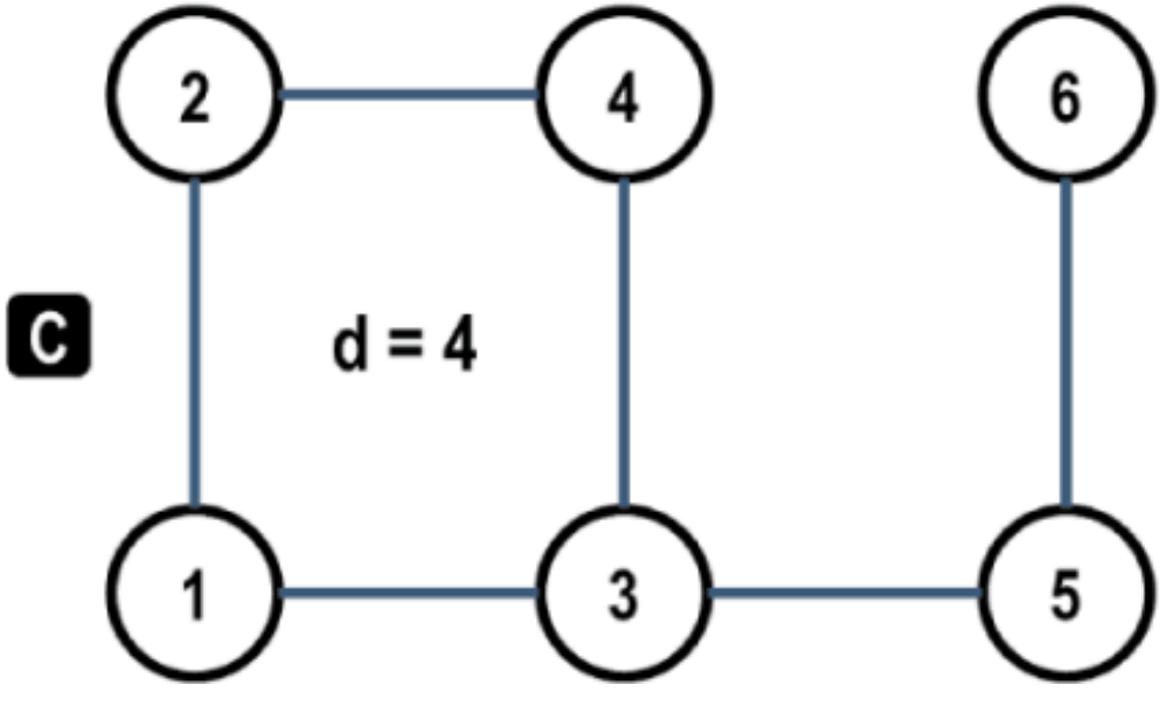
- lower diagonal is NOT redundant and is considered

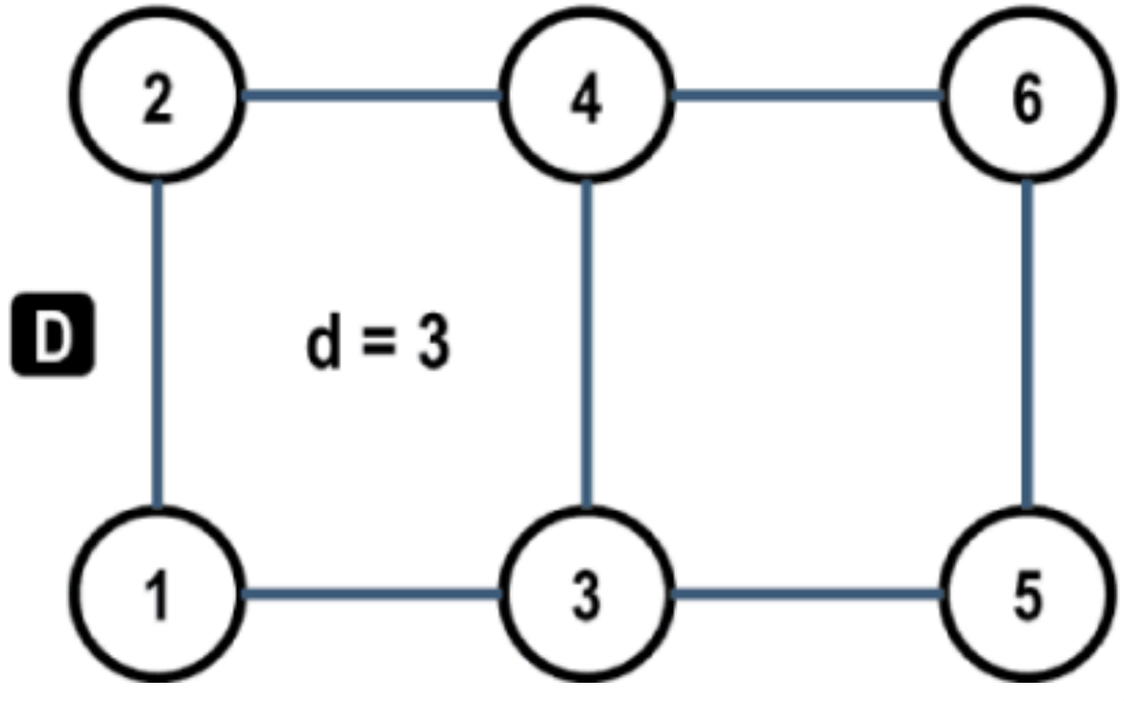
Diameter

Diameter: The diameter of a graph is the length of the "longest shortest path" between any two graph vertices









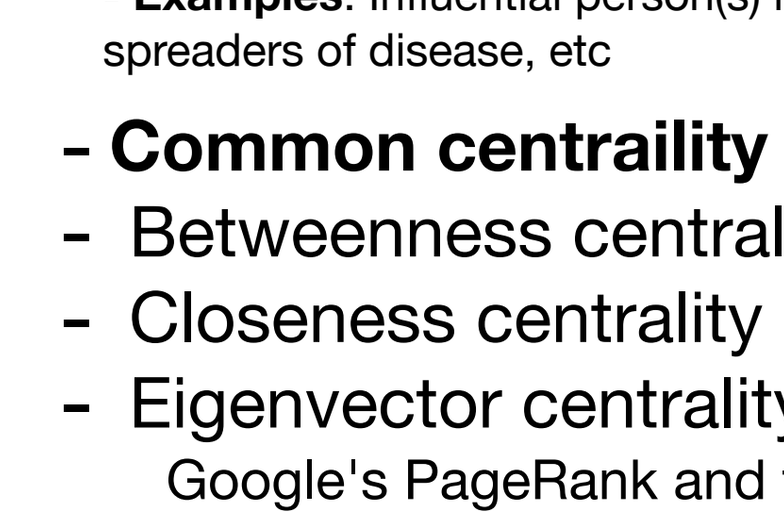
Centrality https://en.wikipedia.org/wiki/Centrality

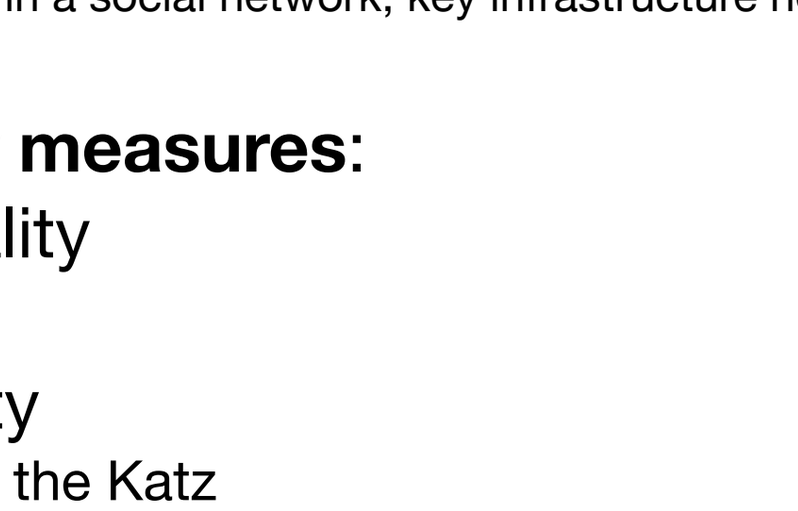
- In graph theory and network analysis, indicators of centrality assign numbers or **rankings to nodes by their network position.**

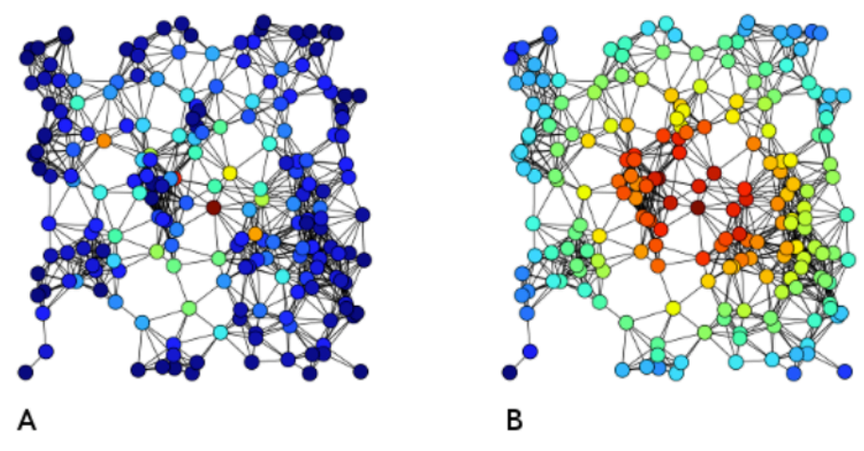
  - **Examples**: Influential person(s) in a social network, key infrastructure nodes in the Internet or urban networks, super-spreaders of disease, etc

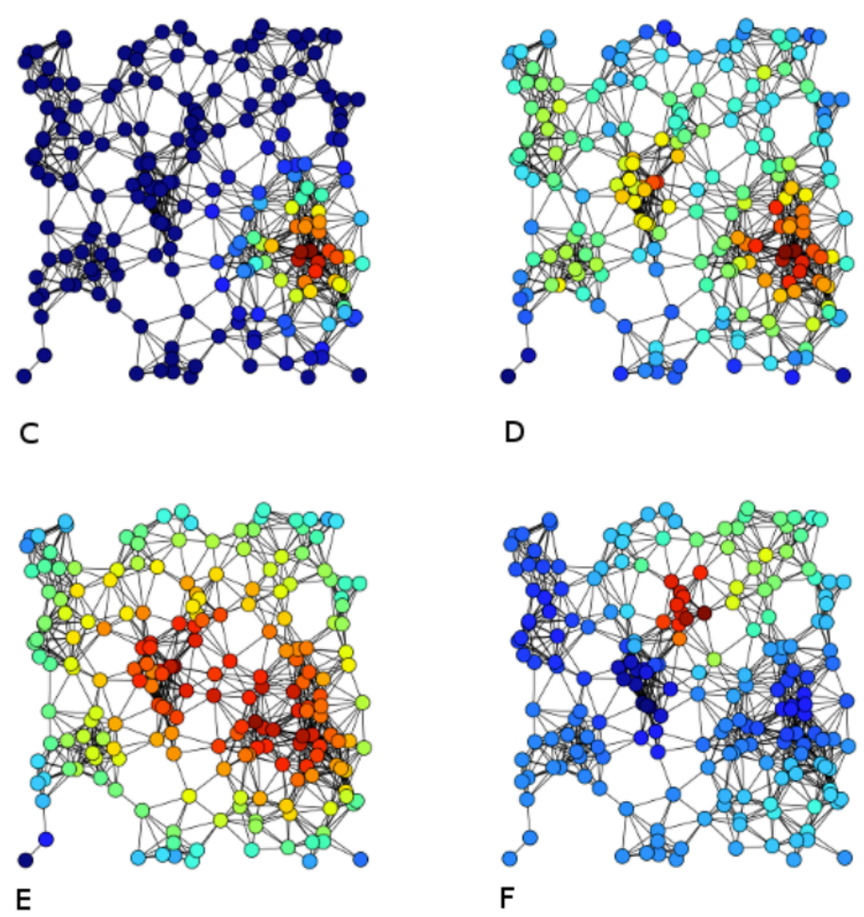
- **Common centraility measures**:
- Betweenness centrality
- Closeness centrality
- Eigenvector centrality

  Google's PageRank and the Katz centrality are variants of the eigenvector centrality
- Degree centrality,
- Harmonic centralityKatz ce
- ntrality of the same graph.







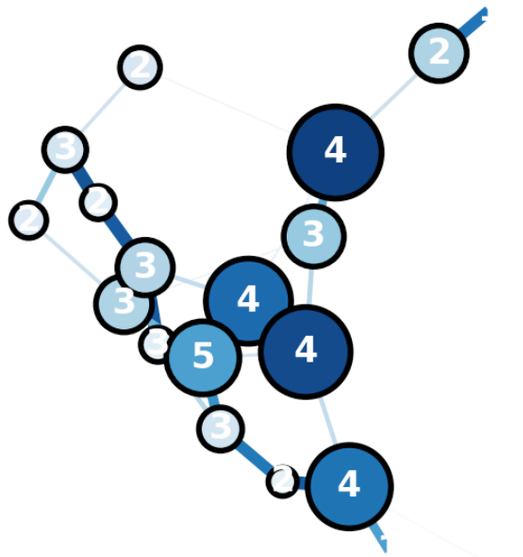


A) Betweenness centrality, B) Closeness centrality, C) Eigenvector centrality, D) Degree centrality, E) Harmonic centrality and F) Katz centrality

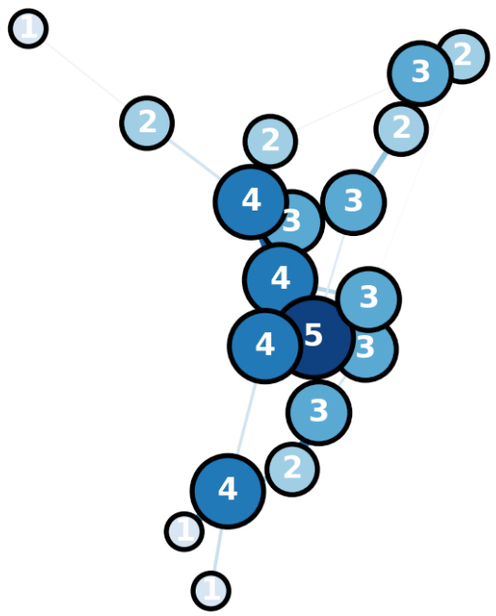
NetworkX: Centrality

- centality encoded as node color and size

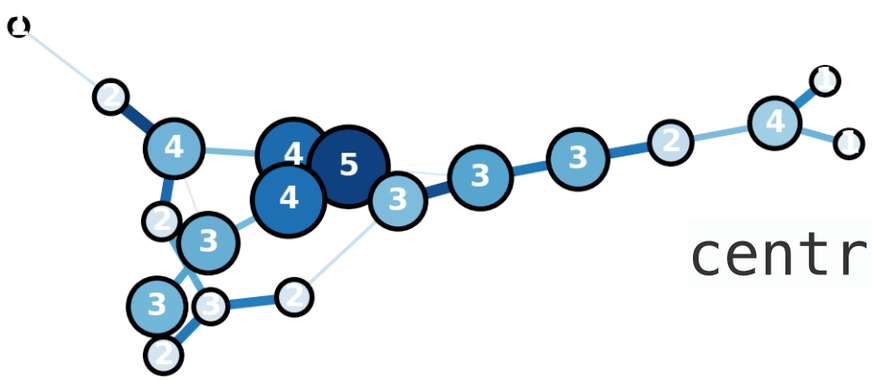
```
centrality = np.array([centrality[n] for n in centrality]);
node_size=1000*centrality/max(centrality)
colors_n = [cmap(u/max(centrality)) for u in centrality]
```



```
centrality = nx.betweenness_centrality(G)
```



```
centrality=nx.degree(G)
```



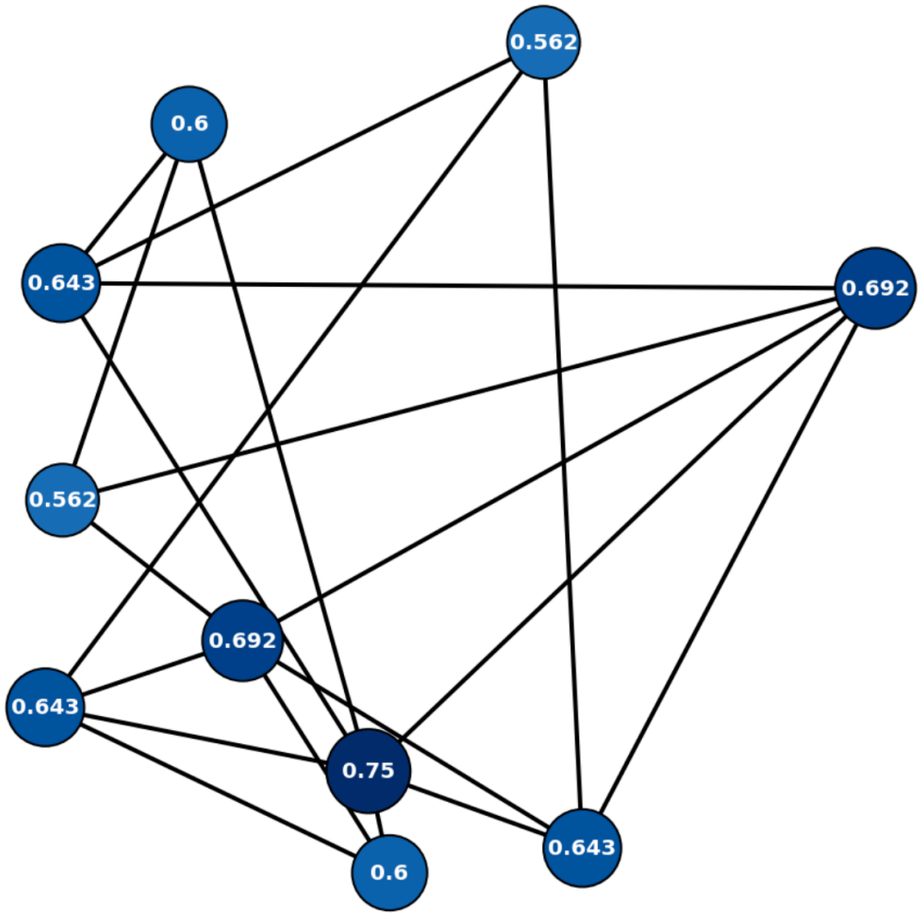
```
centrality = nx.eigenvector_centrality(G)
```

Centrality measures

https://en.wikipedia.org/wiki/Centrality#Degree_centrality

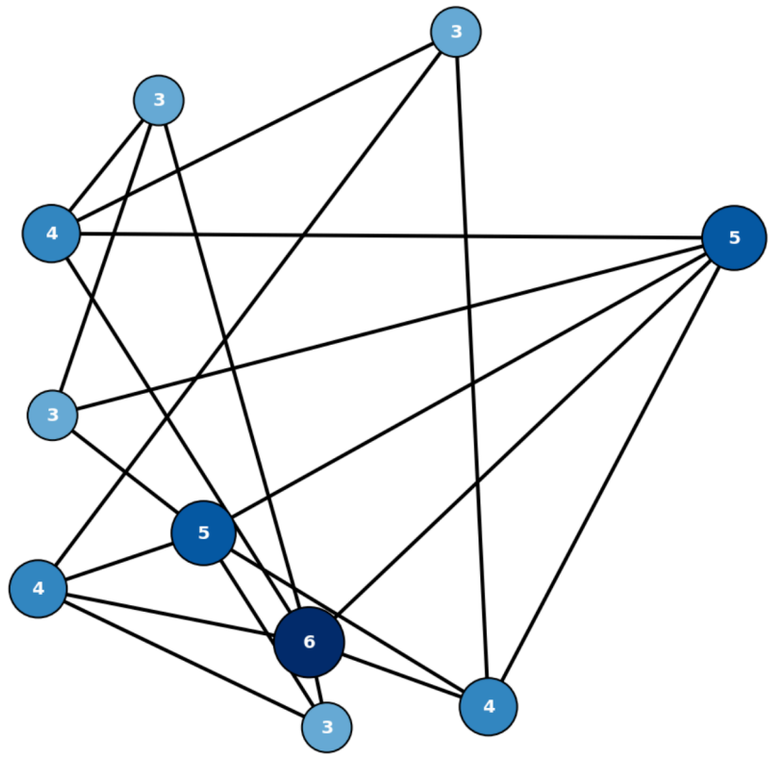
DEGREE CENTRAILITY:

Degree is a simple centrality measure that counts how many neighbors a node has. **A node is important if it has many neighbors, or, in the directed case, if there are many other nodes that link to it, or if it links to many other nodes.**



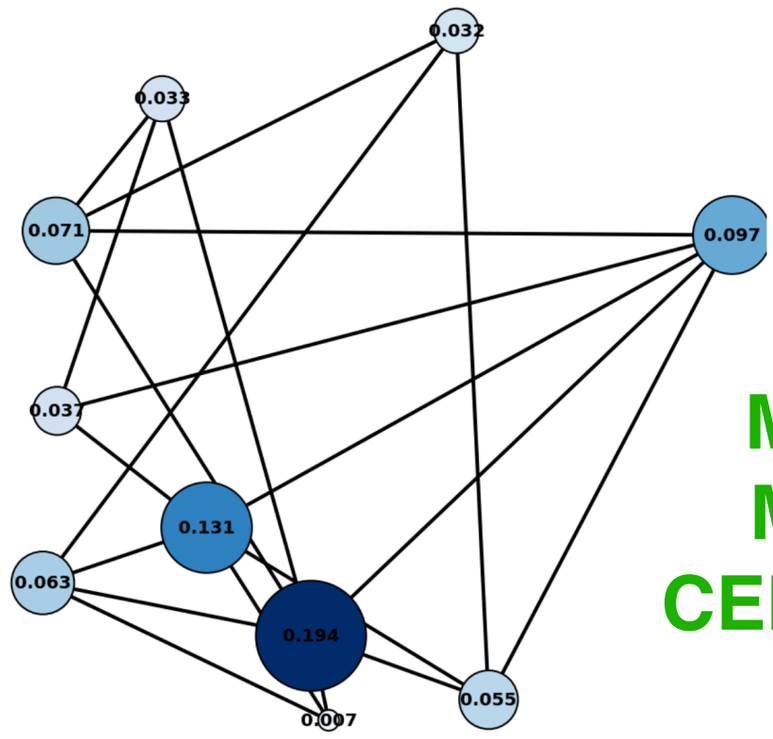
BETWEENESS CENTRAILITY:

Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between two other nodes.



CLOSENESS CENTRAILITY:

In a connected graph, the normalized closeness centrality (or closeness) of a node is the **average length of the shortest path between the node and all other nodes in the graph.** Thus the more central a node is, the closer it is to all other nodes.

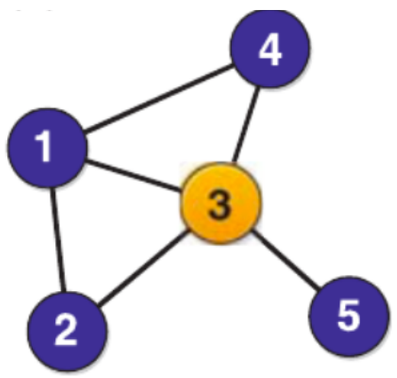


MORE ON THE MATH BEHIND CENTRALITY NEXT WEEK

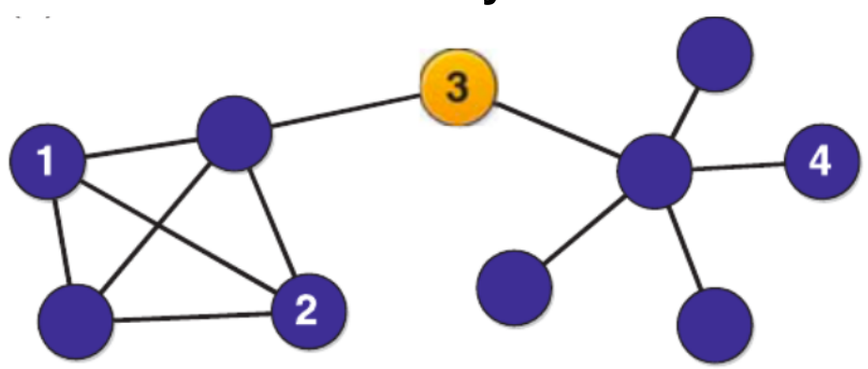
Node betweenness

- Betweenness centrality is a way of detecting the amount of influence a node has over the flow of information in a graph.

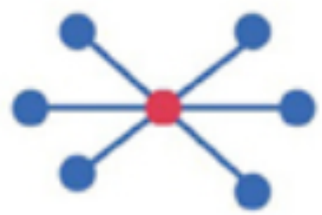
- Each node receives a **score, based on the number of shortest paths that pass through the node**. Nodes that more frequently lie on shortest paths between other nodes will have higher betweenness centrality scores.



The orange node has high degree (k3= 4) as well as high betweenness (b3 = 3.5).



The orange node has low degree ($k_3 = 2$) betweenness ($b_3 = 3.5$)



Mored likely to be involved in diffusion (spread) processes

Formal defintion:

Given two nodes, there may be more than one shortest path between them in the network, all having the same length. For instance, if nodes X and Y are not connected to each other but have two common neighbors S and T, there are two distinct shortest paths of length two running from X to Y: $X-S-Y$ and $X-T-Y$. Let σ_{hj} be the total number of shortest paths from h to j and $\sigma_{hj}(i)$ the number of these shortest paths that pass through node i. The betweenness of i is defined as

$$b_i = \sum_{h \neq j \neq i} \frac{\sigma_{hj}(i)}{\sigma_{hj}}. \qquad (3.3)$$

In Eq. (3.3) the sum runs over all pairs of vertices h and j, distinct from i and from each other. If no shortest path between h and j crosses i [$\sigma_{hj}(i) = 0$], the contribution of the pair (h, j) to the betweenness of i is 0. If all shortest paths between h and j cross i [$\sigma_{hj}(i) = \sigma_{hj}$], the contribution is 1. If a node is a *leaf* (i.e. it has only one neighbor), it cannot be crossed by any path. Therefore its betweenness is zero. Since the potential contributions come from all pairs of nodes, the betweenness grows with the network size.

NetworkX

```
nx.betweenness_centrality(G)
```

```
# dict nodes ->
# betweenness centrality
```

Code

Also see —>
CODES/week-2

```python
import networkx as nx
import matplotlib.pyplot as plt
import random
import numpy as np
# from matplotlib import rcParams
# rcParams.update({'figure.autolayout': True})

#GENERATE GRAPH
N=10
G = nx.complete_graph(N)
G = nx.gnp_random_graph(N, 0.4, seed=10374196)

#SPECIFIY X-Y POSITIONS FOR PLOTTING
# pos=nx.spring_layout(G,k=1./np.sqrt(N),iterations=500)
pos=nx.random_layout(G)

#PLOT NETWORK
def plot_network(font_color="black"):
    #INITIALIZE PLOT
    fig, ax = plt.subplots()
    fig.set_size_inches(12, 12)

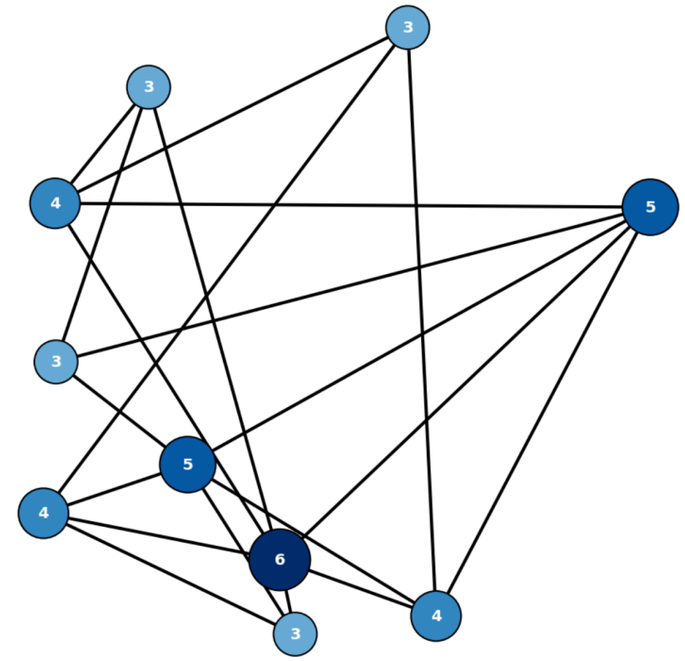
    #PLOT
    nx.draw(
    G,
    node_color=colors_n,
    edgecolors="black",
    edge_color="black",
    linewidths=2,
    labels=labels,
    font_color=font_color,
    font_weight="bold",
    font_size=20,
    width=4,
    node_size=node_size,
    with_labels=True,
    pos=pos,
    ax=ax,
    )
    #ax.set_aspect('auto', 'box')
    # plt.tight_layout()
    plt.show()

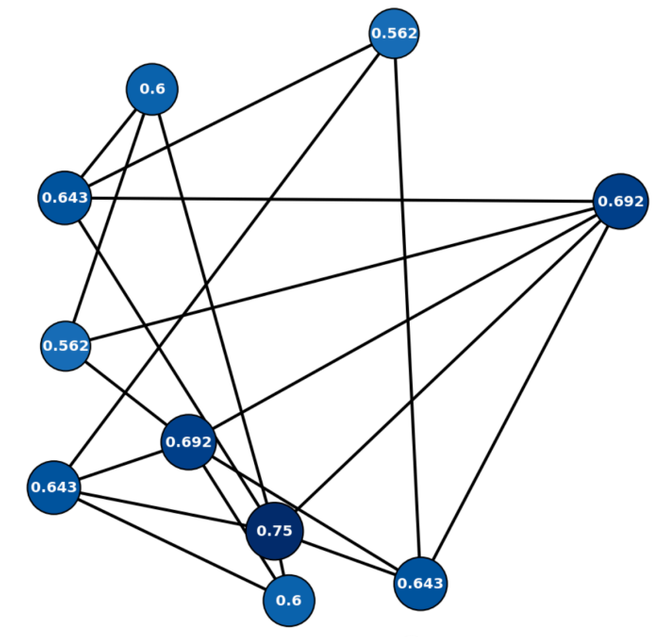
#SAMPLE CMAP FOR COLORS
cmap=plt.cm.get_cmap('Blues')

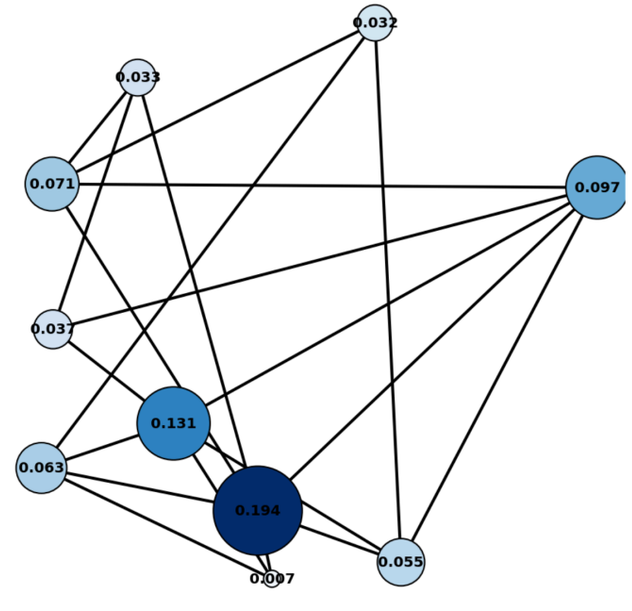
#DEGREE CENTRALITY
centrality=nx.degree(G)
labels={}
for n,d in centrality: labels[n]=d #str(n)+"-"+str(d)
centrality=np.array([d for n,d in centrality]); #print(G_DEGREE)
node_size=6000*centrality/max(centrality)
colors_n = [cmap(u/max(centrality)) for u in centrality]
plot_network("white")


# CLOSENESS_CENTRALITY
centrality =nx.closeness_centrality(G)
for key in centrality.keys(): centrality[key]=round(centrality[key],3)
labels=centrality
centrality = np.array([centrality[n] for n in centrality]); #print(G_DEGREE)
node_size=6000*centrality/max(centrality)
colors_n = [cmap(u/max(centrality)) for u in centrality]
plot_network("white")

#BETWEENESS CENTRALITY
centrality = nx.betweenness_centrality(G)
for key in centrality.keys(): centrality[key]=round(centrality[key],3)
labels=centrality
centrality = np.array([centrality[n] for n in centrality]); #print(G_DEGREE)
node_size=15000*centrality/max(centrality)
colors_n = [cmap(u/max(centrality)) for u in centrality]
plot_network()
```

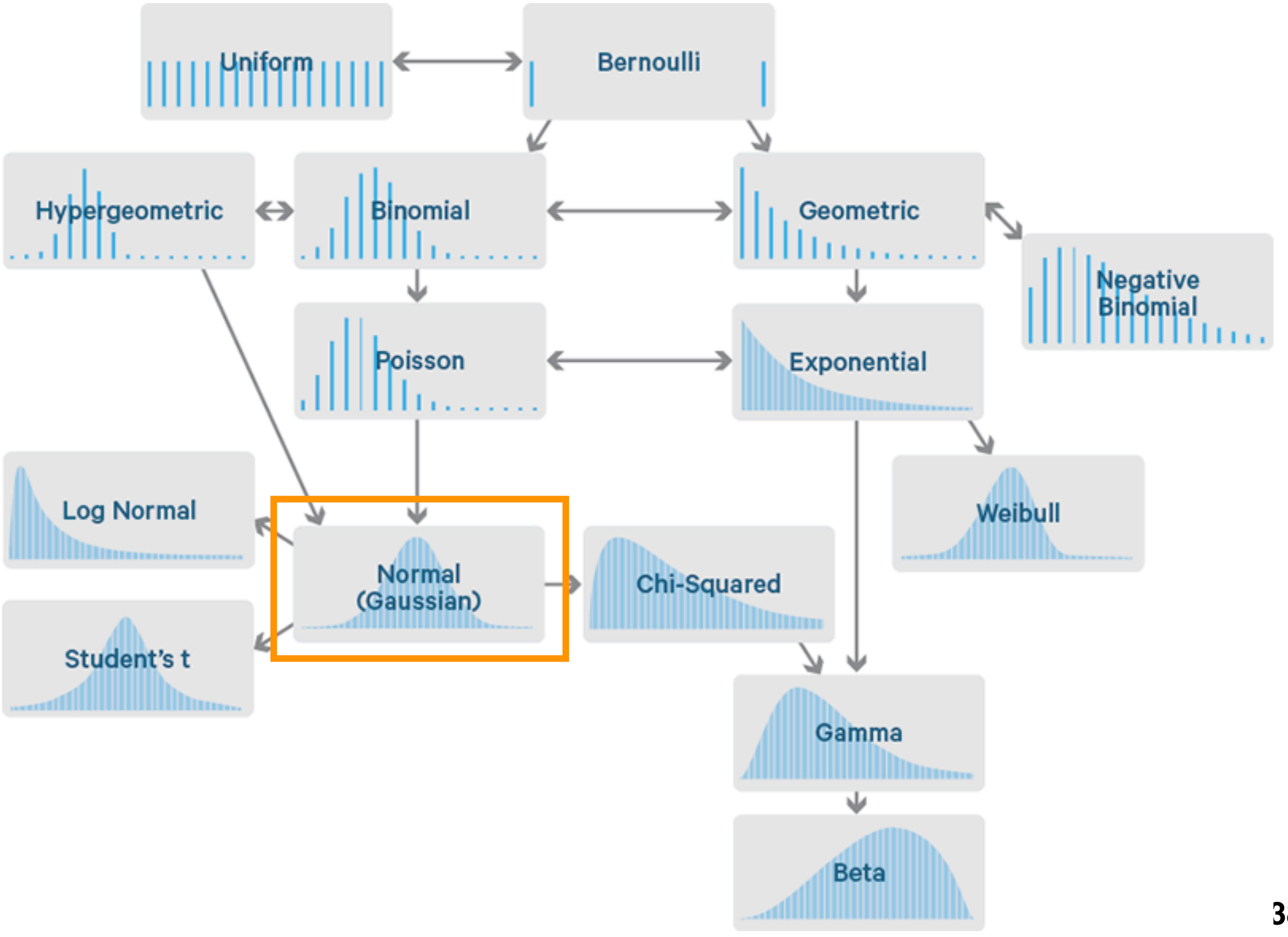






Node degree distributions

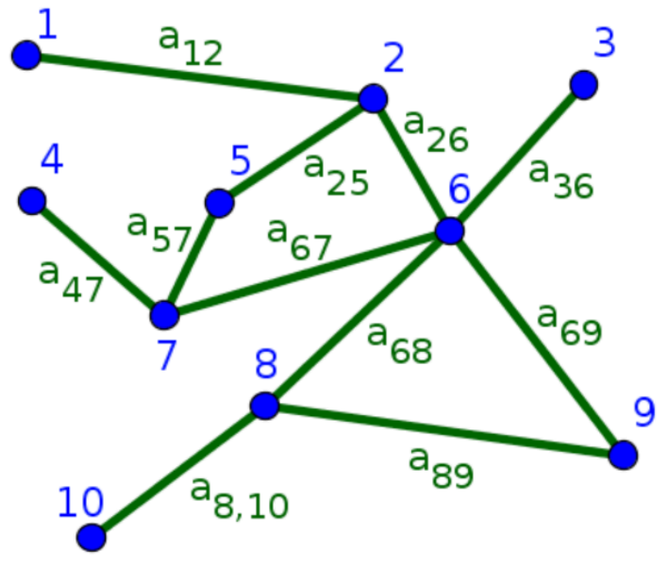
Common probability distributions

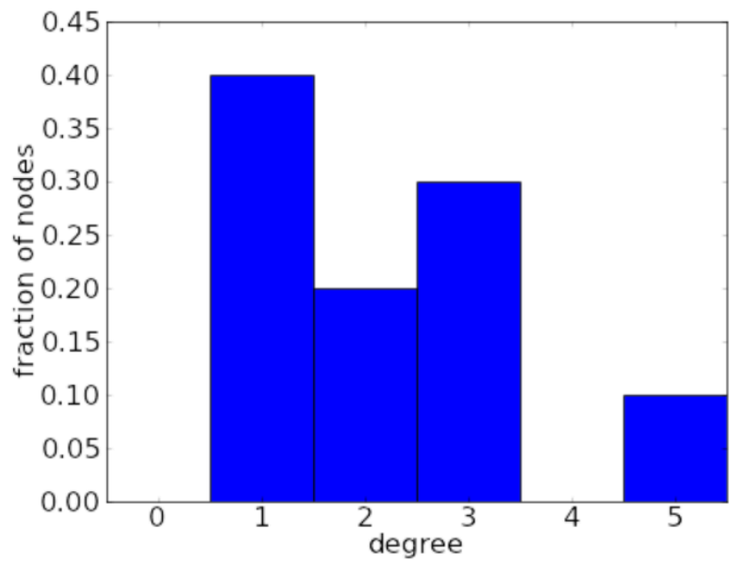


Node degree distribution

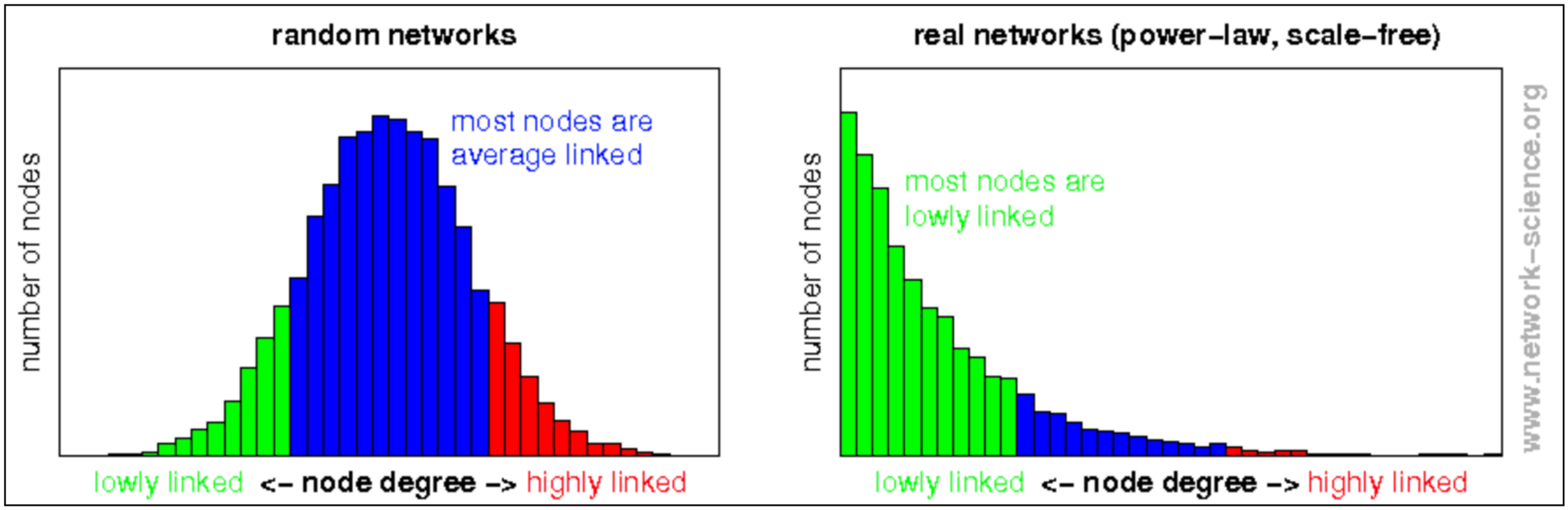
https://mathinsight.org/degree_distribution

In the study of graphs and networks, the degree of a node in a network is the number of connections it has to other nodes and the degree distribution is the probability distribution of these degrees over the whole network.





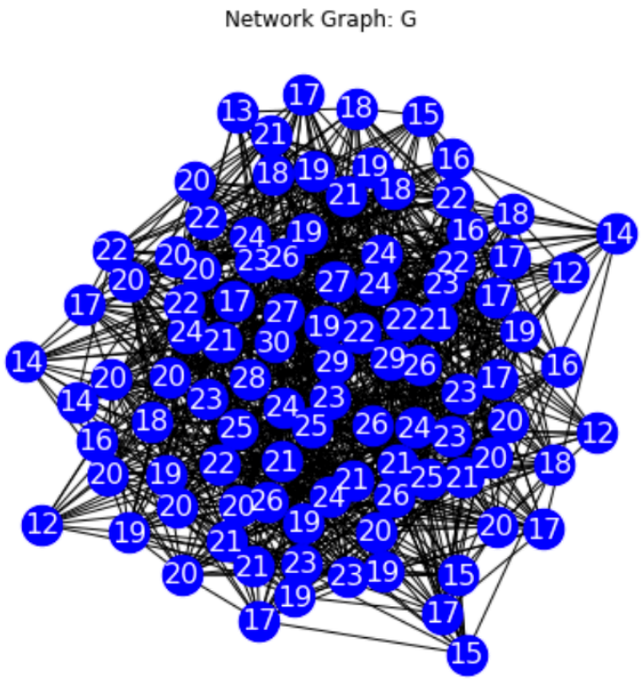
For this undirected network, the degrees are $k_1 = 1, k_2 = 3, k_3 = 1, k_4 = 1, k_5 = 2, k_6 = 5, k_7 = 3, k_8 = 3, k_9 = 2$, and $k_{10} = 1$. Its degree distribution is $P_{\text{deg}}(1) = 2/5, P_{\text{deg}}(2) = 1/5, P_{\text{deg}}(3) = 3/10, P_{\text{deg}}(5) = 1/10$, and all other $P_{\text{deg}}(k) = 0$.

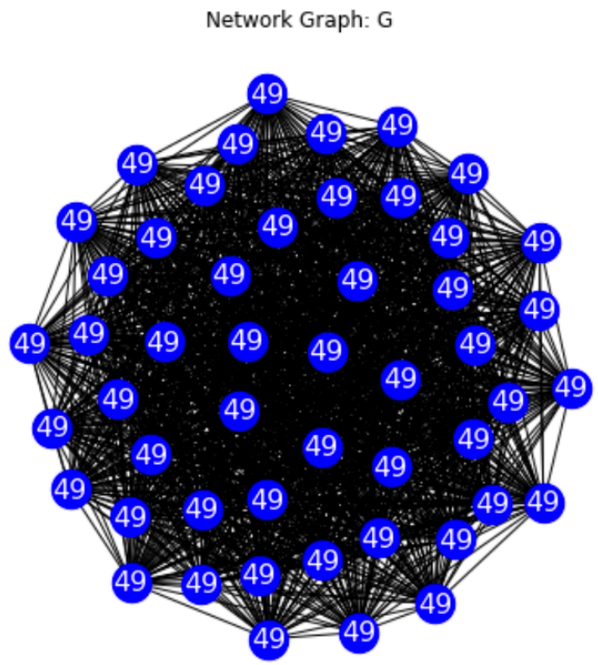


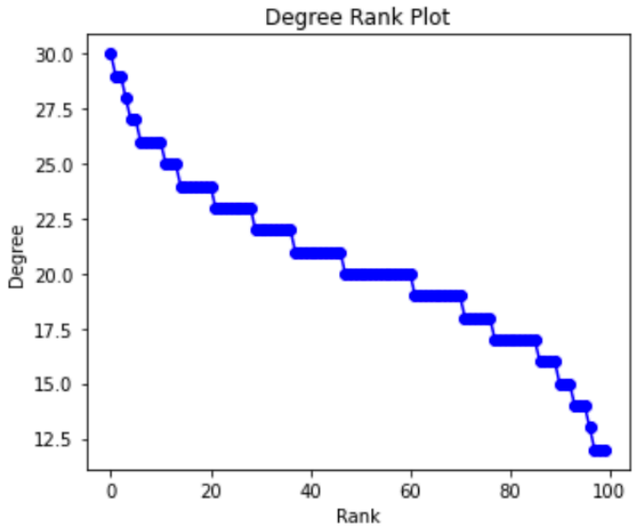
http://www.network-science.org/powerlaw_scalefree_node_degree_distribution.html

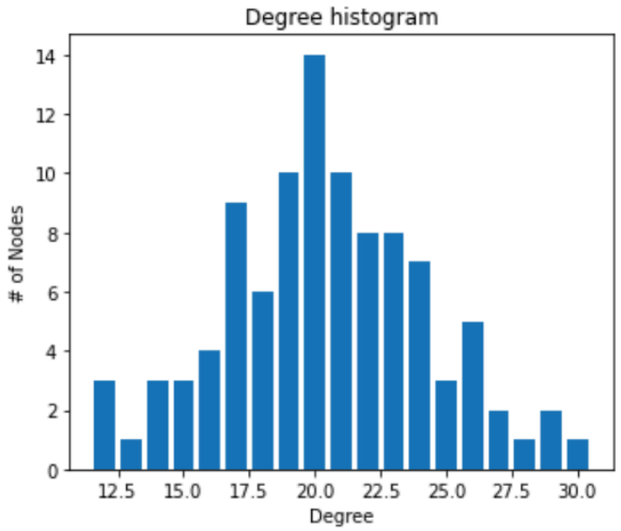
Example

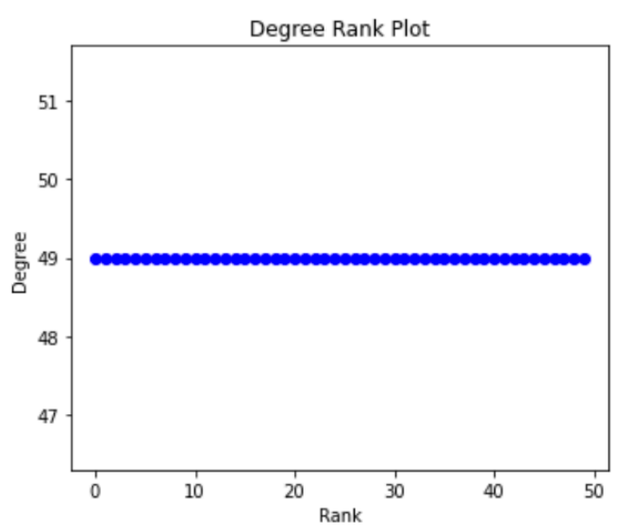
```
RANDOM GRAPH
```

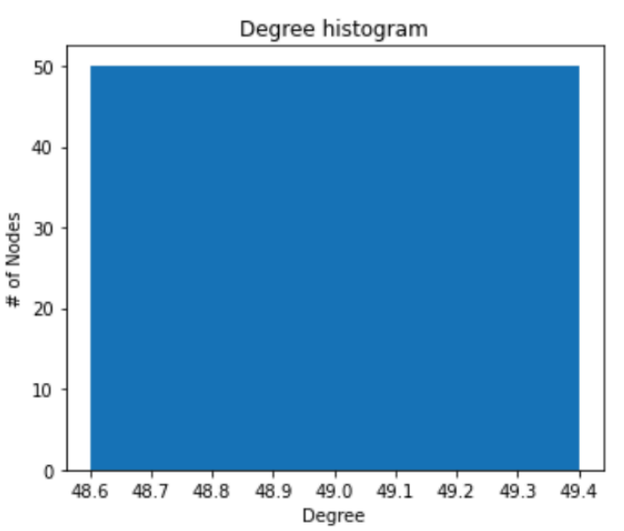






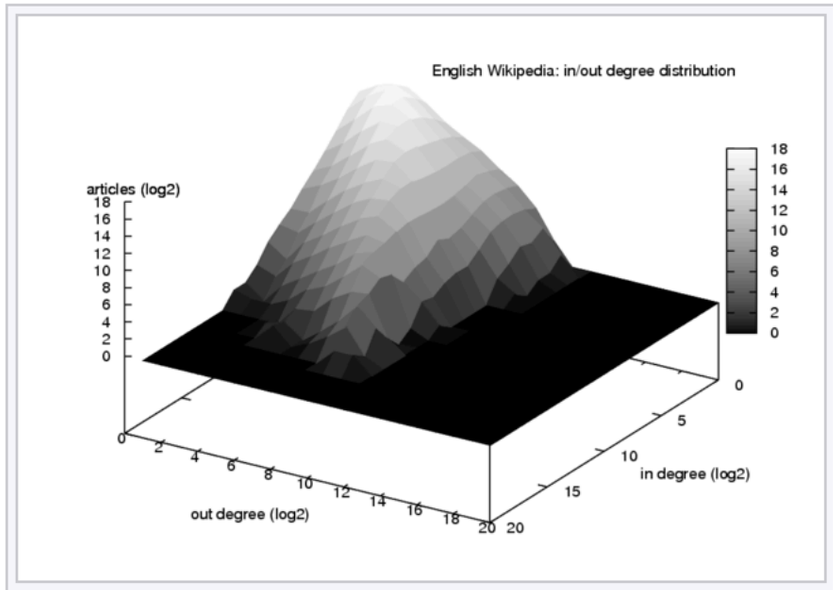






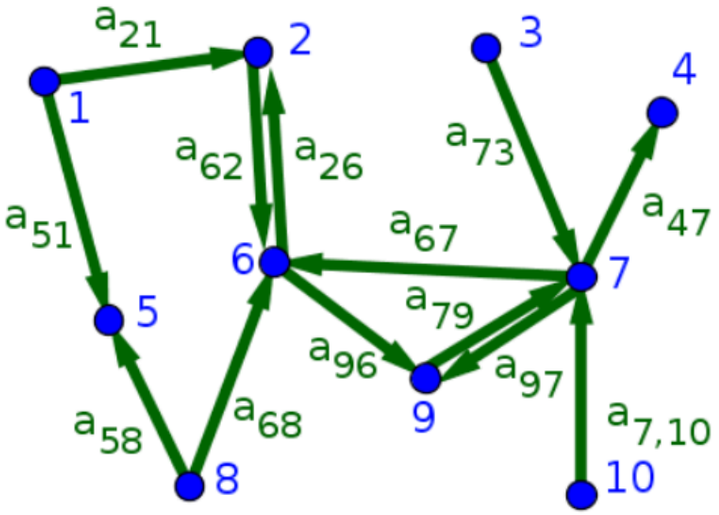
Node degree distribution

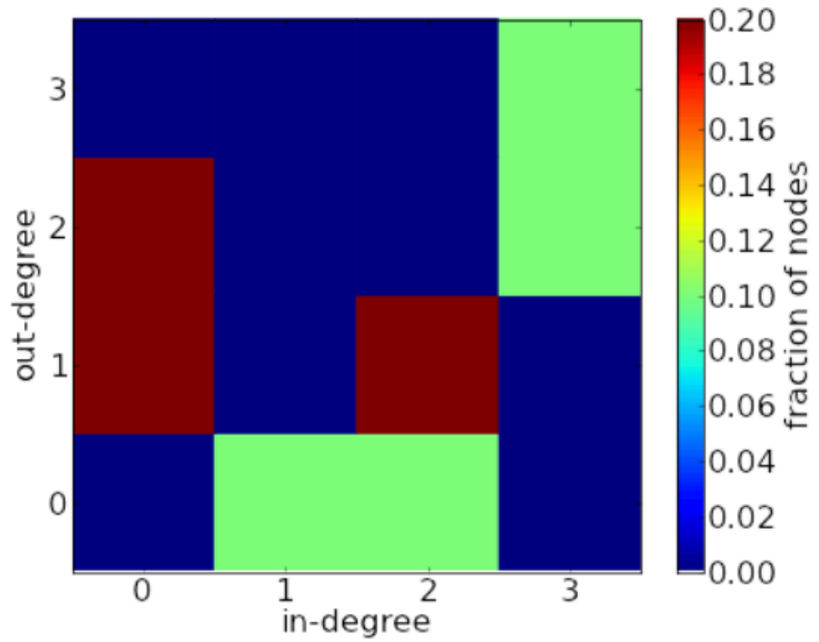
In a directed network, each node has some in-degree k_{in} and some out-degree k_{out} which are the number of links which have run into and out of that node

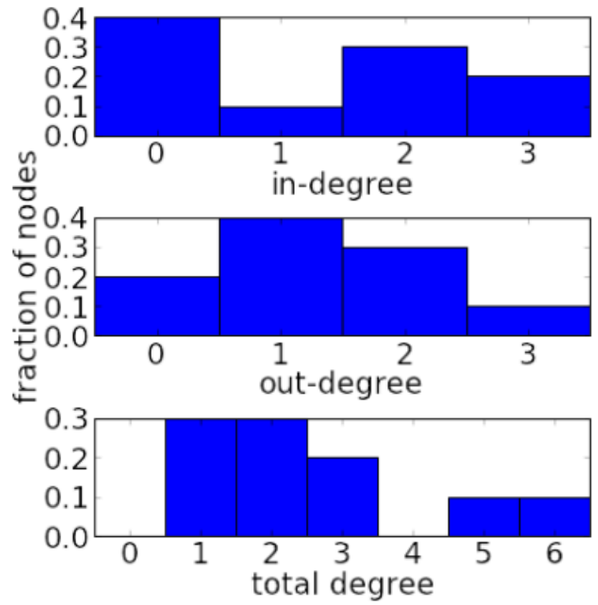


In/out degree distribution for Wikipedia's hyperlink graph (logarithmic scales)

https://mathinsight.org/degree_distribution







For the above network, the in- and out-degrees are $(k_1^{\text{in}}, k_1^{\text{out}}) = (0, 2)$, $(k_2^{\text{in}}, k_2^{\text{out}}) = (2, 1)$, $(k_3^{\text{in}}, k_3^{\text{out}}) = (0, 1)$, $(k_4^{\text{in}}, k_4^{\text{out}}) = (1, 0)$, $(k_5^{\text{in}}, k_5^{\text{out}}) = (2, 0)$, $(k_6^{\text{in}}, k_6^{\text{out}}) = (3, 2)$, $(k_7^{\text{in}}, k_7^{\text{out}}) = (3, 3)$, $(k_8^{\text{in}}, k_8^{\text{out}}) = (0, 2)$, $(k_9^{\text{in}}, k_9^{\text{out}}) = (2, 1)$, and $(k_{10}^{\text{in}}, k_{10}^{\text{out}}) = (0, 1)$. A two-dimensional histogram of these values is plotted with the color plot. The marginal degree distributions, involving just the in-degrees or just the out-degrees, are shown in the first two bar plots. The third bar plot shows the total degree distribution, based on the total degrees $k_1^{\text{tot}} = 2$, $k_2^{\text{tot}} = 3$, $k_3^{\text{tot}} = 1$, $k_4^{\text{tot}} = 1$, $k_5^{\text{tot}} = 2$, $k_6^{\text{tot}} = 5$, $k_7^{\text{tot}} = 6$, $k_8^{\text{tot}} = 2$, $k_9^{\text{tot}} = 3$, and $k_{10}^{\text{tot}} = 1$.

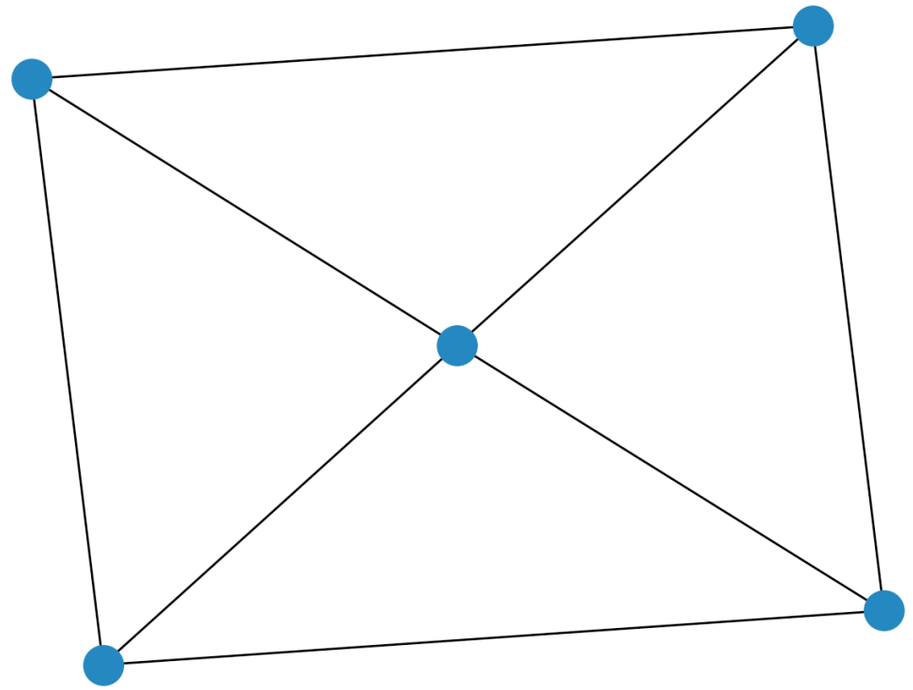
NETWORK VIZ

- **Network visualization** involves the visualization of the relationships (edges or links) between data elements (nodes).

NetworkX: Basic plotting

Example-1:

```
#BASIC PLOT
nx.draw(G)
plt.show()
```

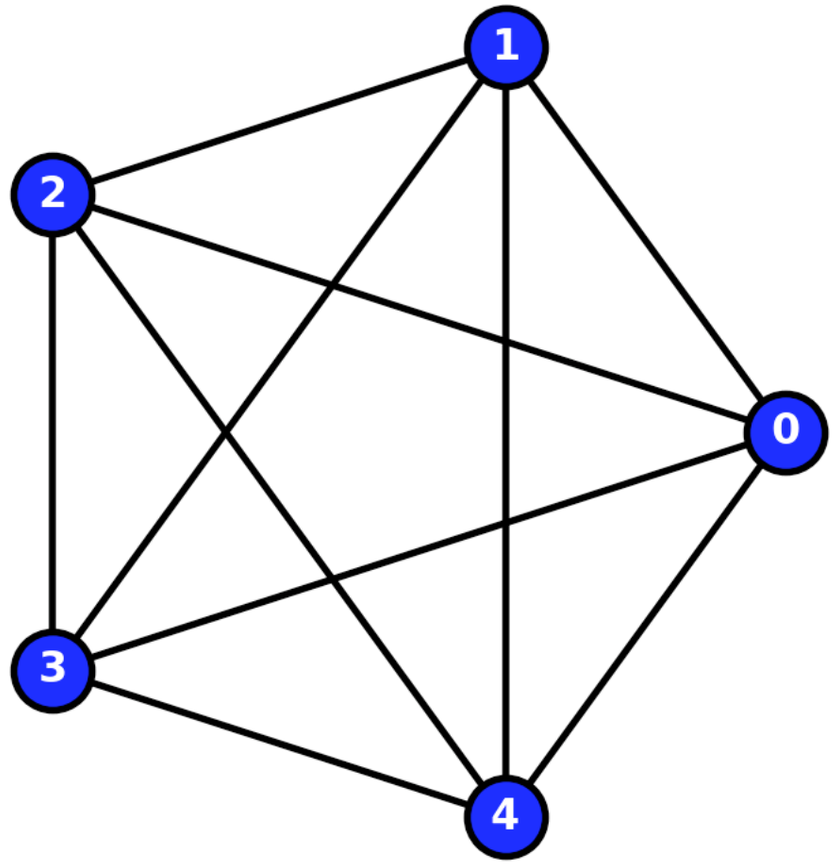


Basic NetworkX plot

Example-2: Customization

```
#CUSTOMIZE
#specifiy x-y positions for plotting
pos=nx.circular_layout(G)

#initialize plot
fig, ax = plt.subplots()
nx.draw(
G,
node_color='blue',
edgecolors="black",
edge_color="black",
linewidths=2,
font_color="white",
font_weight="bold",
node_size=500,
width=2,
with_labels=True,
pos=pos,
ax=ax
)
ax.set(title='Basic NetworkX plot')
ax.set_aspect('equal', 'box')
#fig.savefig("test.png")
plt.show()
```



Plotting: Layouts-1

- POSITIONS IN X,Y SPACE ARE ARBITRARY

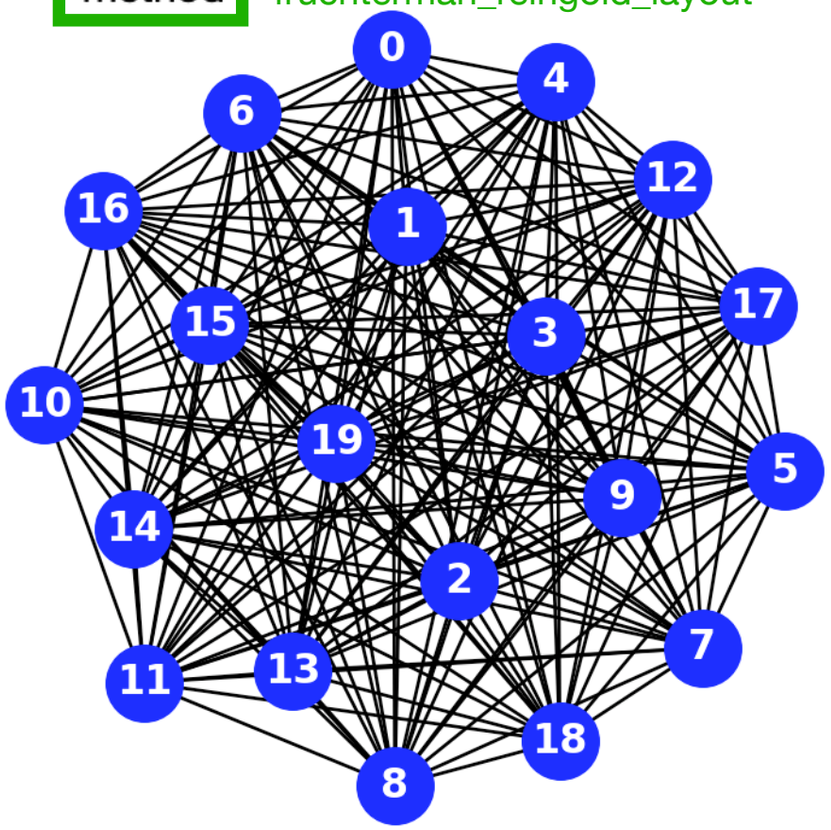
- ONLY CONNECTIVTY MATTERS SO THE SAME GRAPH MAY LOOK DIFFERENT DEPENDING ON POSITIONS

- THERE ARE MANY POSITION ALGORITHMS

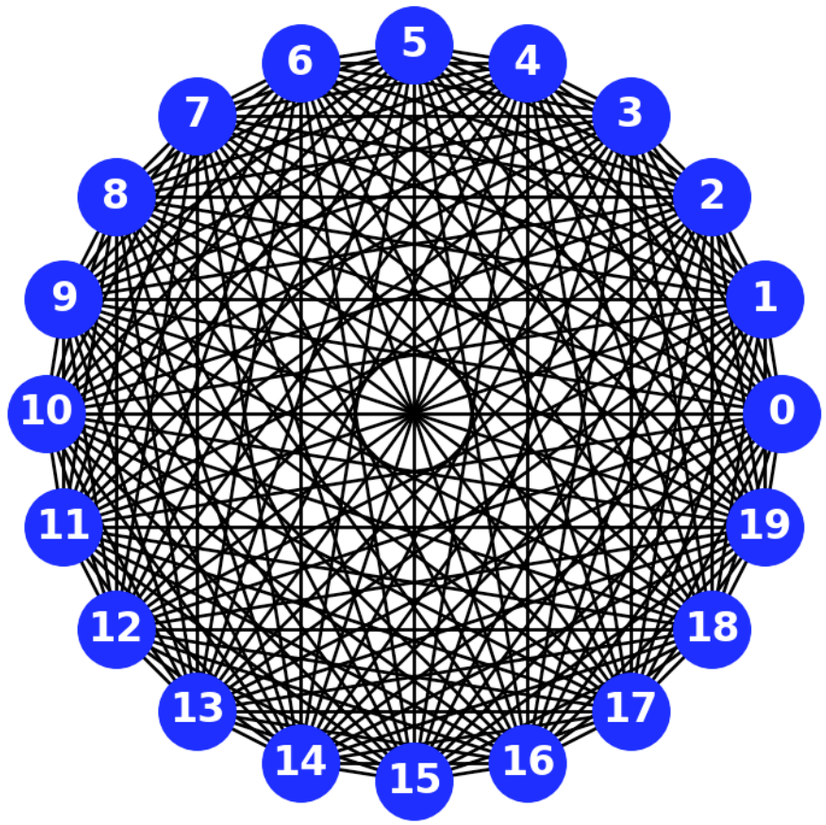
common method

Layout: Spring

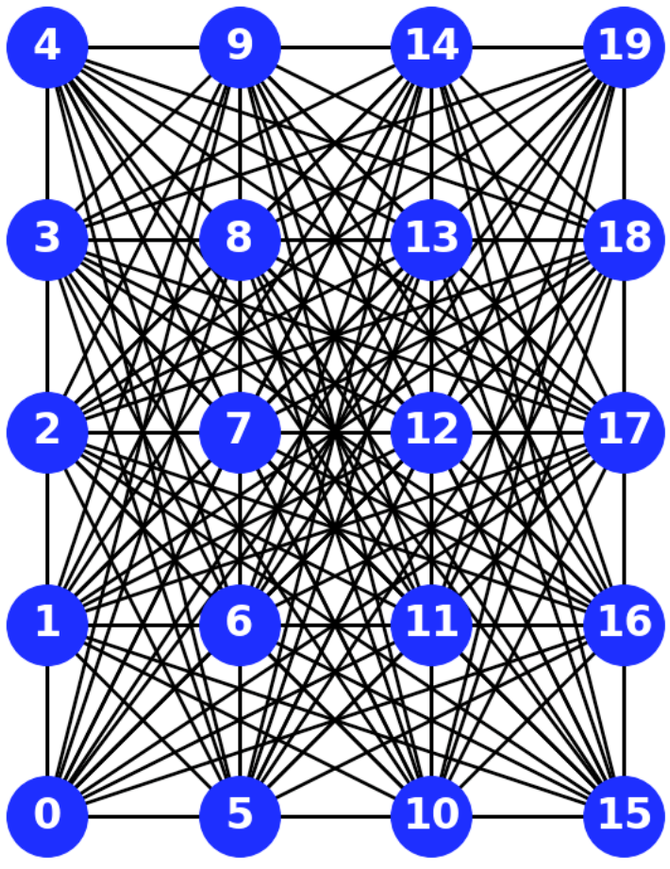
fruchterman_reingold_layout



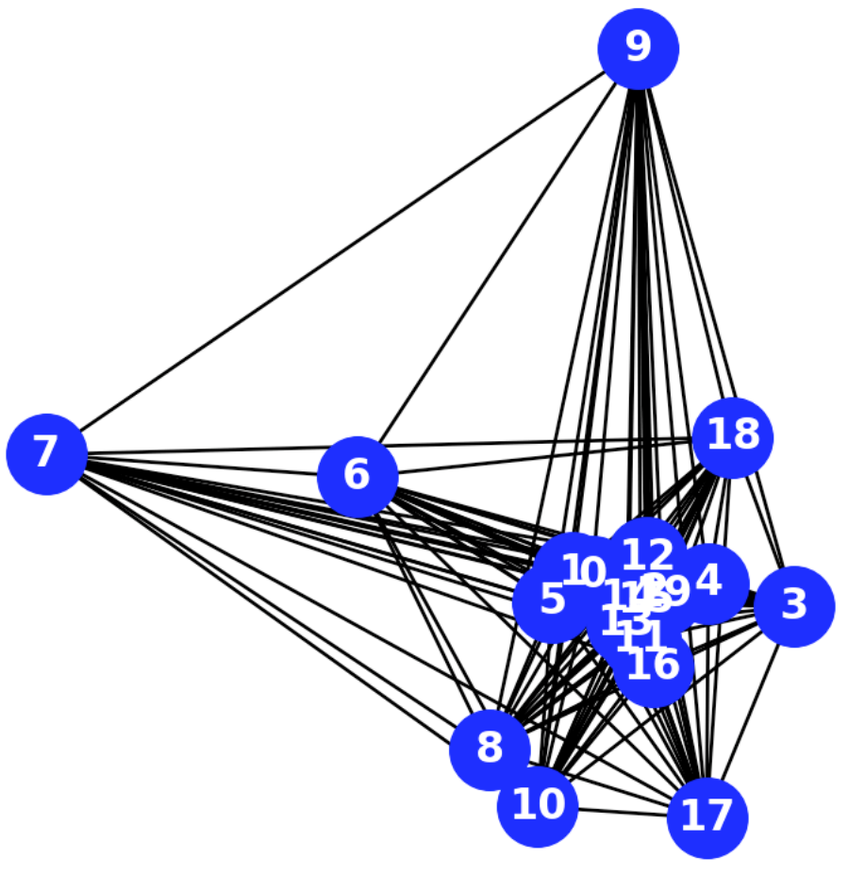
Layout: Circular



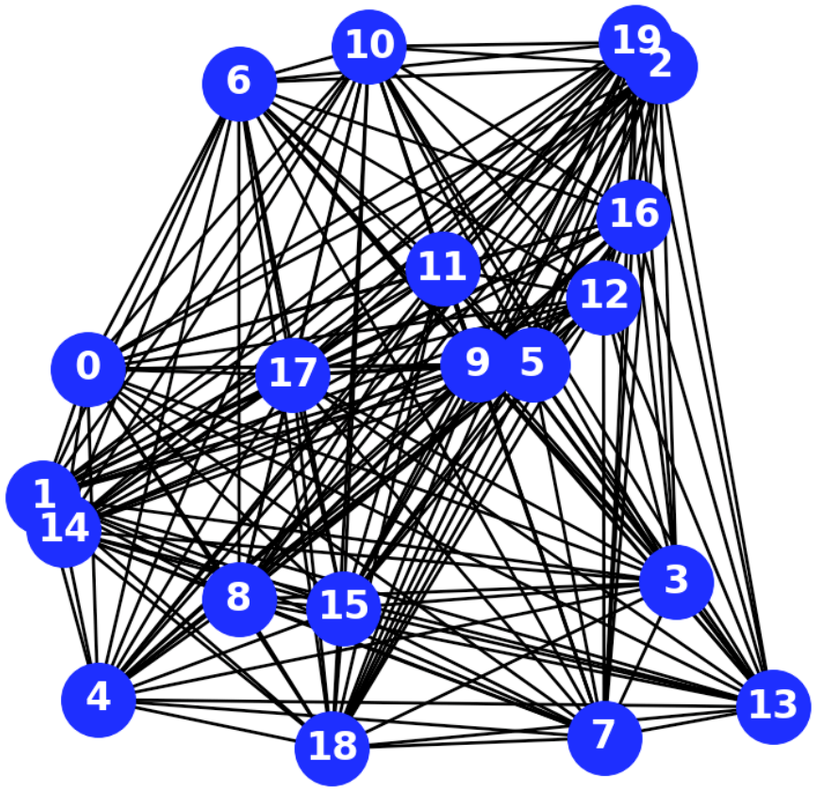
Layout: Manual



Layout: Spectral



Layout: Random



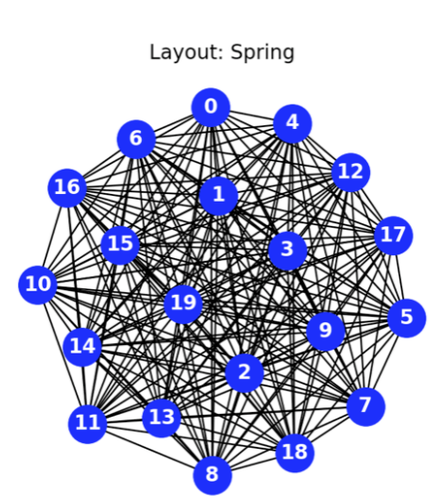
Code:

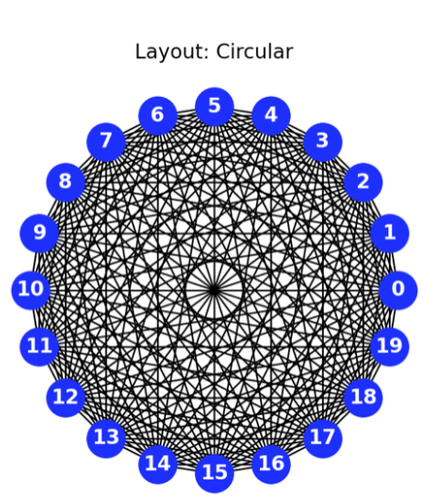
```
import networkx as nx
import matplotlib.pyplot as plt

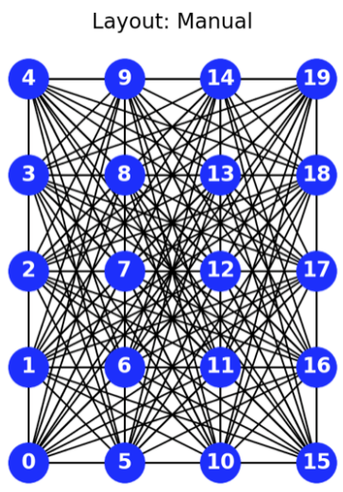
#GENERATE GRAPH
G = nx.complete_graph(20)

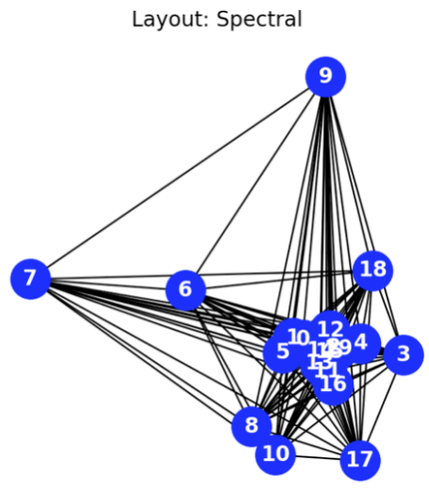
for i in range(1,7+1):
    #if(i==1): title="Layout: Spring";       pos = nx.fruchterman_reingold_layout(G,iterations=200)
    if(i==1): title="Layout: Spring";       pos = nx.spring_layout(G,iterations=200)
    if(i==2): title="Layout: Spectral";     pos = nx.spectral_layout(G)
    if(i==3): title="Layout: Circular";     pos = nx.circular_layout(G)
    if(i==4): title="Layout: Kamada-Kawai"; pos = nx.kamada_kawai_layout(G)
    if(i==5): title="Layout: Random";       pos = nx.random_layout(G)
    if(i==6): title="Layout: Shell";        pos = nx.shell_layout(G)
    #CAN MANUALLY SPECIFY GRAPH LOCATIONS WITH A DICTIONARY OF (X,Y) ARRAYS
    if(i==7):
        title="Layout: Manual"; pos={}; j=0; i=0
        for k in range(0,len(list(G.edges))):
            if(k%5==0): j=j+1
            if(k%5==0): i=0
            i=i+1
            pos[k]=[j,i]

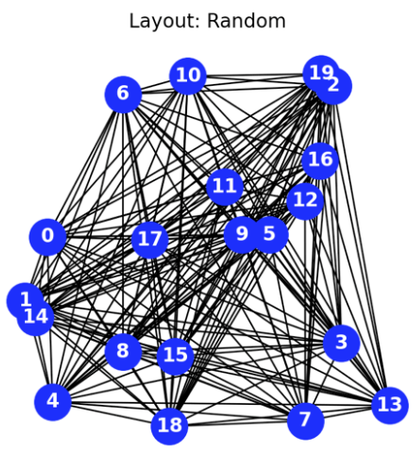
    fig, ax = plt.subplots()
    nx.draw(
        G,
        node_color="blue",
        node_size=500,
        font_color="white",
        font_weight="bold",
        with_labels=True,
        pos=pos,
    )
    ax.set(title=title)
    ax.set_aspect('equal', 'box')
    plt.title(title)
    plt.show()
```





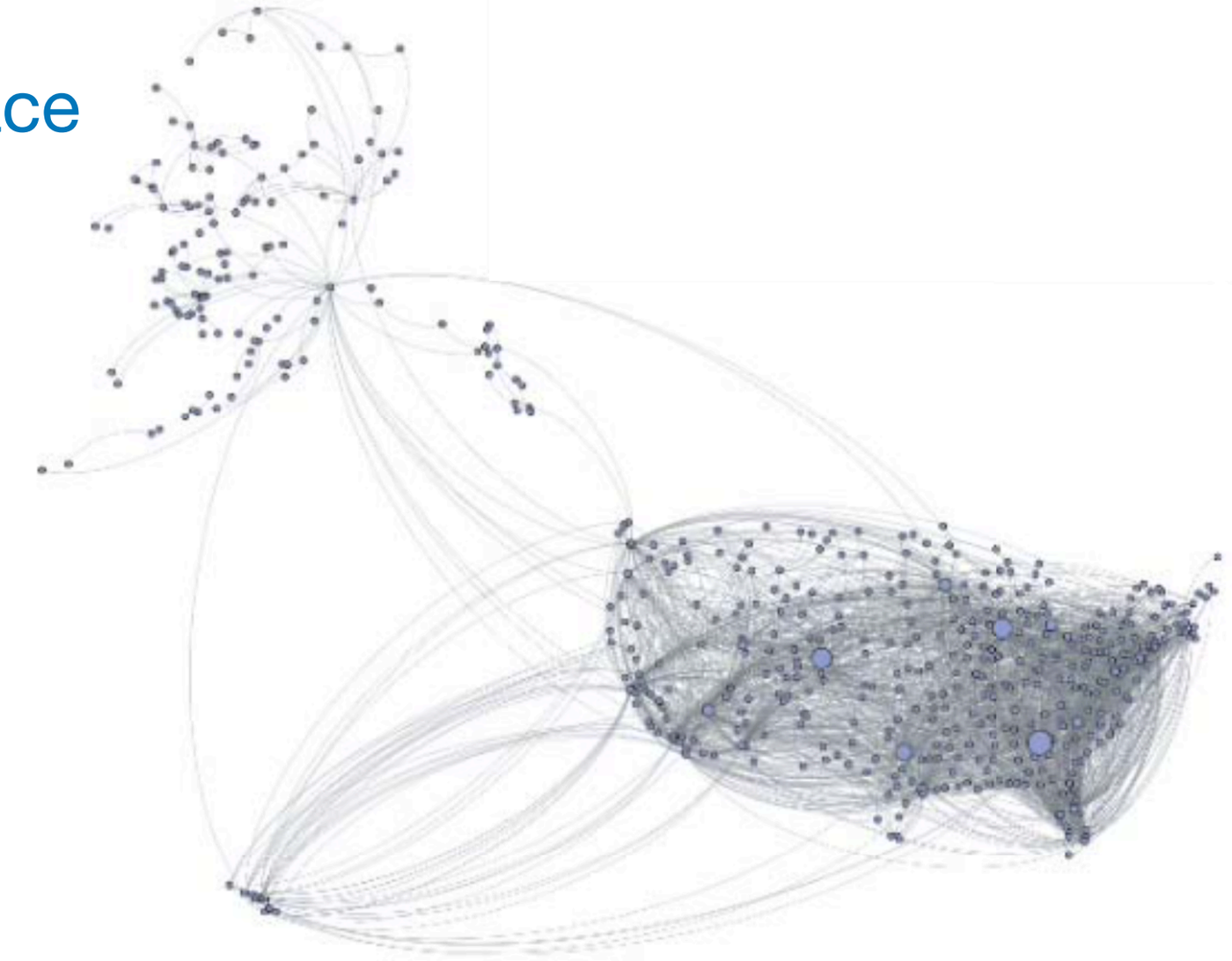




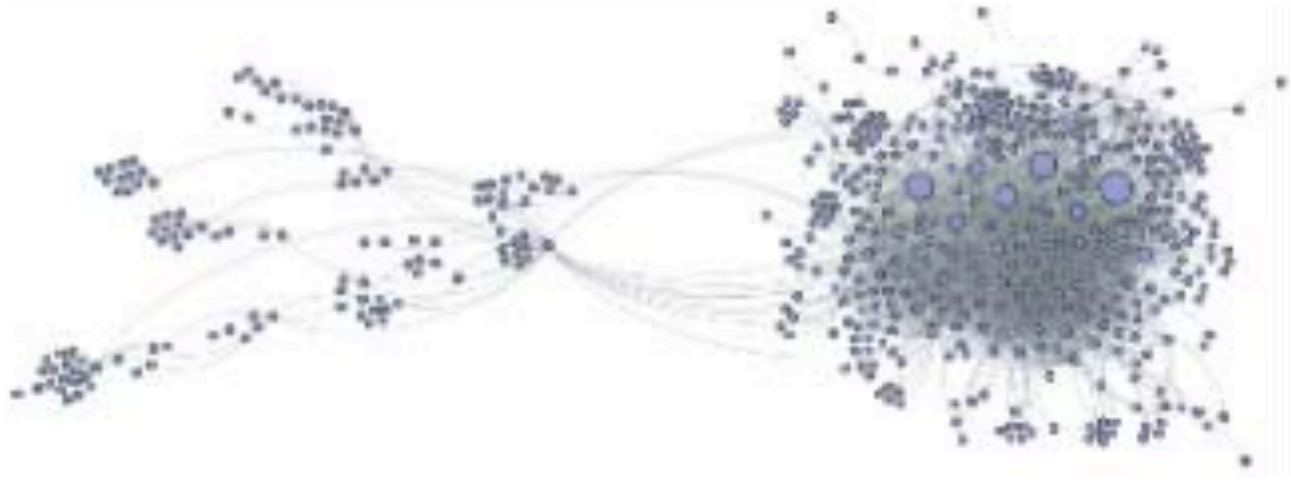


Example The US air transportation network

Layout-1: Real-space



Layout-2: Force-directed



Plotting node+edge attributes

```
import networkx as nx
import matplotlib.pyplot as plt
import random

#GENERATE GRAPH
G = nx.complete_graph(10)

#ASSIGN RANDOM ATTRIBUTES TO NODES+EDGES
for u in G.nodes():
     G.nodes[u]['at1']=0.2+1*random.uniform(0, 1)
#for node in G.nodes(data=True): print(node)

for u,v in G.edges():
     G[u][v]['weight']=5*random.uniform(0, 1)
#for line in nx.generate_edgelist(G, data=True): print(line)
#GET MAX WEIGHT

#SPECIFIY X-Y POSITIONS FOR PLOTTING
pos=nx.circular_layout(G)

#GENERATE PLOT
fig, ax = plt.subplots()

#assign colors based on attributes
weights_n      = [1000*G.nodes[u]['at1'] for u in G.nodes()]
weights_e      = [G[u][v]['weight'] for u,v in G.edges()]

#SAMPLE CMAP FOR COLORS
cmap=plt.cm.get_cmap('Blues')
colors_e  = [cmap(G[u][v]['weight']/5.0) for u,v in G.edges()]
colors_n  = [cmap(G.nodes[u]['at1']/1.2) for u in G.nodes()]

nx.draw(
G,
node_color=colors_n,
edgecolors="black",
edge_color=colors_e,
linewidths=2,
font_color="white",
font_weight="bold",
node_size=weights_n,
width=weights_e,
with_labels=True,
pos=pos,
ax=ax
)

ax.set(title='Color and size plotted by attribute')
ax.set_aspect('equal', 'box')
plt.show()
```

