

# Representations of Graphs & Centrality Measures

Prof. O-Joun Lee

Dept. of Artificial Intelligence,  
The Catholic University of Korea  
*ojlee@catholic.ac.kr*

# Contents

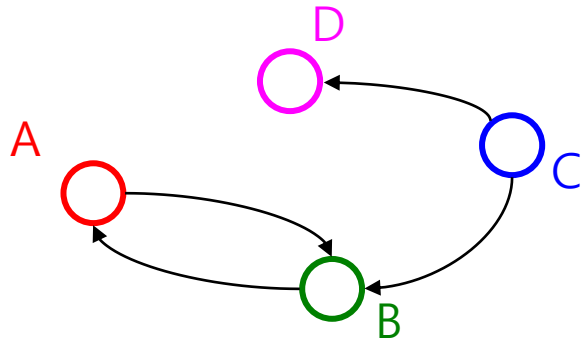


- Graph Representations
  - Adjacency Matrix and Adjacency List
  - Graph Databases
- Graph Centrality Measures
  - Degree Centrality
  - Betweenness Centrality
  - Closeness Centrality
  - Eigenvector Centrality
  - Katz Centrality
  - PageRank

- Graphs efficiently model relationships, perfect for addressing questions like "What's the lowest-cost path from A to B?"
- We need a data structure that represents graphs
- Determining the "Best" Data Structure can depend on:
  - Properties of the graph (dense vs. sparse)
  - Common queries
    - For example: "is  $(u, v)$  an edge?" vs "what are the neighbors of node  $u$ ?"

- There are two standard graph representations:
  - Adjacency Matrix
  - Adjacency List
- Trade-offs: Computational complexity and Space requirement
  - Consider different trade-offs, particularly the balance between time and space efficiency

- Assign each node a number from 0 to  $|V|-1$
- A  $|V| \times |V|$  matrix of Booleans (or 0 vs. 1)
  - Then  $M[u][v] == \text{true}$  means there is an edge from  $u$  to  $v$



	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

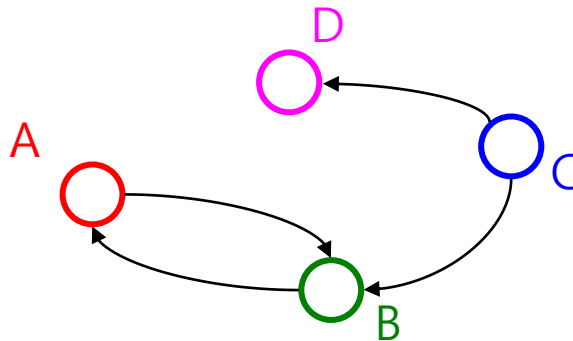
➤ Sample code:

```
import networkx as nx
import numpy as np
import pandas as pd

# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_edges_from([("A", "B"), ("B", "A"), ("C", "B"), ("C", "D")])
# 2D array adjacency matrix
A = nx.adjacency_matrix(G)
A_dense = A.todense()
print(A_dense)
# Pandas format
nx.to_pandas_adjacency(G)
```

```
[[0 1 0 0]
 [1 0 0 0]
 [0 1 0 1]
 [0 0 0 0]]
```

	A	B	C	D
A	0.0	1.0	0.0	0.0
B	1.0	0.0	0.0	0.0
C	0.0	1.0	0.0	1.0
D	0.0	0.0	0.0	0.0



	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

- Running time to:
  - Get a vertex's out-edges:
  - Get a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Best for sparse or dense graphs?

	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0



- Running time to:
  - Get a vertex's out-edges:  $O(|V|)$
  - Get a vertex's in-edges:  $O(|V|)$
  - Decide if some edge exists:  $O(1)$
  - Insert an edge:  $O(1)$
  - Delete an edge:  $O(1)$
- Space requirements:  $O(|V|^2)$
- Best for sparse or dense graphs?
  - Dense graphs

	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0



➤ Sample codes: checking connection between nodes

```
# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_edges_from([("A", "B"), ("B", "A"), ("C", "B"), ("C", "D")])

# Get a vertex's out-edges:
print(f"OUT-edges of node B: {G.out_degree('B')}")
# Get a vertex's in-edges:
print(f"IN-edges of node B: {G.in_degree('B')}")
# Decide if some edge exists:
print(f"Check an edge from A to C: {G.has_edge('A', 'C')}")
# Insert an edge:
G.add_edge("A", "C")
# OR
G.add_edges_from([("A", "D")])
print(f"Check an edge from A to C: {G.has_edge('A', 'C')}")
print(f"Check an edge from A to D: {G.has_edge('A', 'D')}")
# Delete an edge:
G.remove_edge("A", "C")
# OR
G.remove_edges_from([("A", "D")])
print(f"Check an edge from A to C: {G.has_edge('A', 'C')}")
print(f"Check an edge from A to D: {G.has_edge('A', 'D')}")
```

```
OUT-edges of node B: 1
IN-edges of node B: 2
Check an edge from A to C: False
Check an edge from A to C: True
Check an edge from A to D: True
Check an edge from A to C: False
Check an edge from A to D: False
```

	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

- How will the adjacency matrix vary for an undirected graph?
  - Will be symmetric about diagonal axis?
  - Matrix: Could we save space by using only about half the array?

	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	1	0

- But how would you "get all neighbors"?

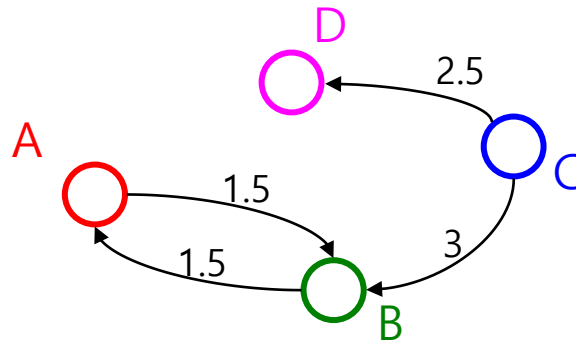
- How can we adapt the representation for weighted graphs?
  - Instead of Boolean, store a number in each cell
  - Need some value to represent 'not an edge'
    - 0, -1, or some other value based on how you are using the graph
    - Might need to be a separate field if no restrictions on weights

➤ Sample code for weighted graph:

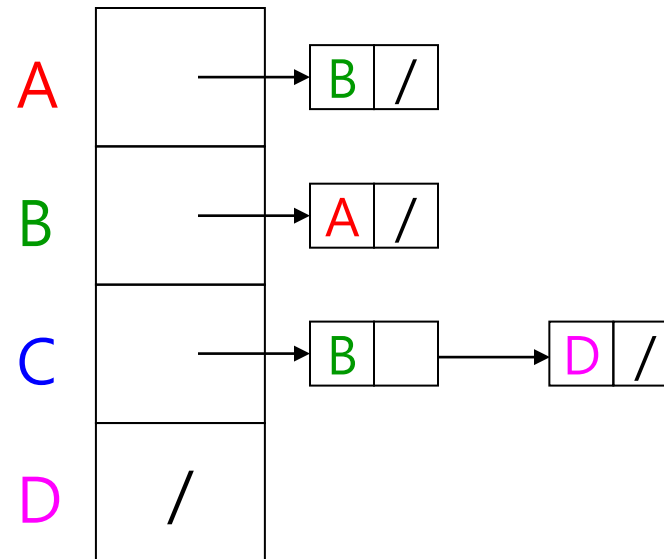
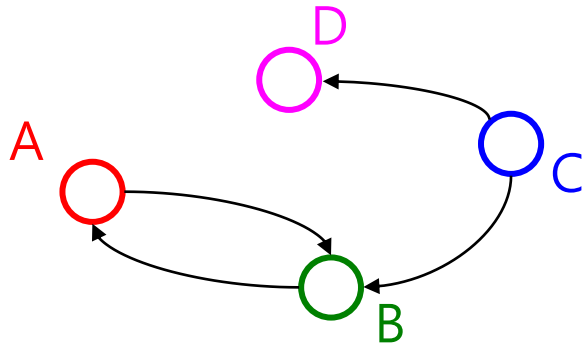
```
# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_weighted_edges_from([("A", "B", 1.5), ("B", "A", 1.5), ("C", "B", 3), ("C", "D", 2.5)])
# 2D array adjacency matrix
A = nx.adjacency_matrix(G)
A_dense = A.todense()
print(A_dense)
# Pandas format of adjacency matrix
nx.to_pandas_adjacency(G)
```

```
[[0.  1.5 0.  0. ]
 [1.5 0.  0.  0. ]
 [0.  3.  0.  2.5]
 [0.  0.  0.  0. ]]
```

	A	B	C	D
A	0.0	1.5	0.0	0.0
B	1.5	0.0	0.0	0.0
C	0.0	3.0	0.0	2.5
D	0.0	0.0	0.0	0.0



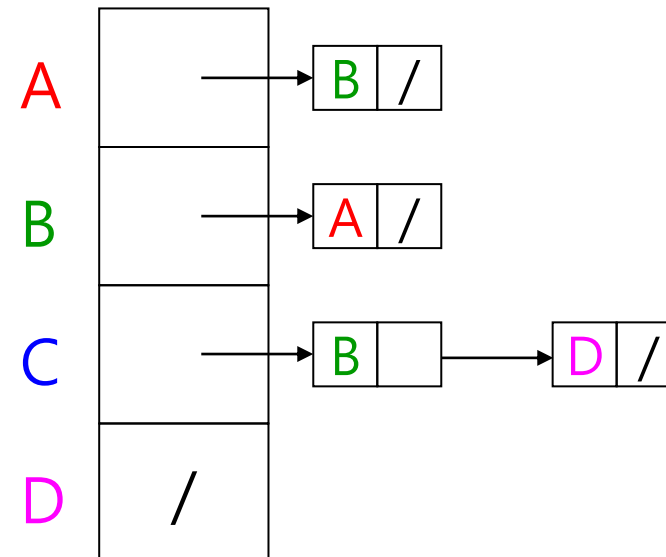
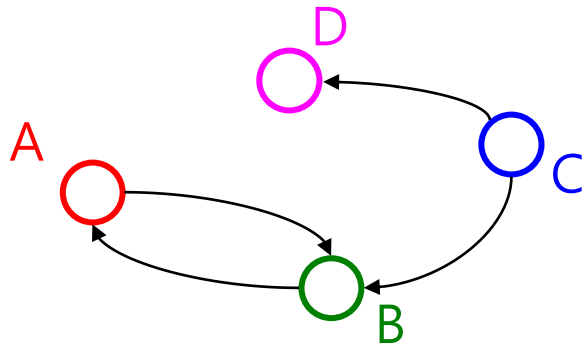
- Assign each node a number from 0 to  $|V|-1$ 
  - An array of length  $|V|$  in which each entry stores a list of all adjacent vertices (e.g., linked list)



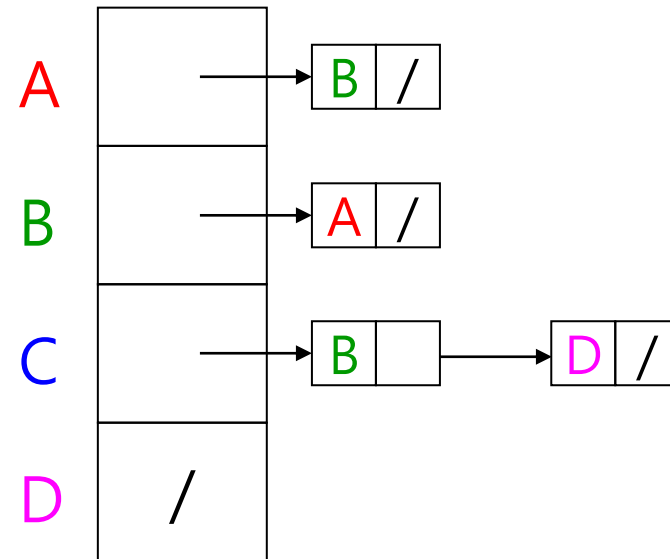
## ➤ Sample Code

```
# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_edges_from([("A", "B"), ("B", "A"), ("C", "B"), ("C", "D")])
adjacency_list = nx.generate_adjlist(G)
for line in adjacency_list:
    print(line)
```

```
A B
B A
C B D
D
```



- Running time to:
  - Get a vertex's out-edges:
  - Get a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Best for sparse or dense graphs?





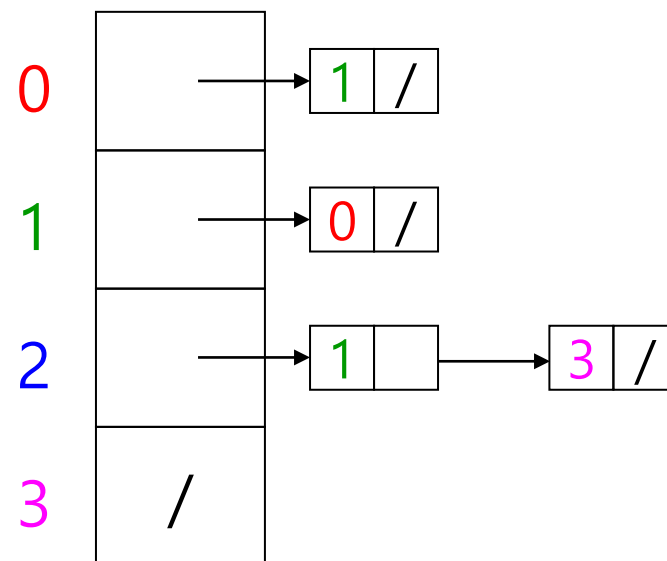
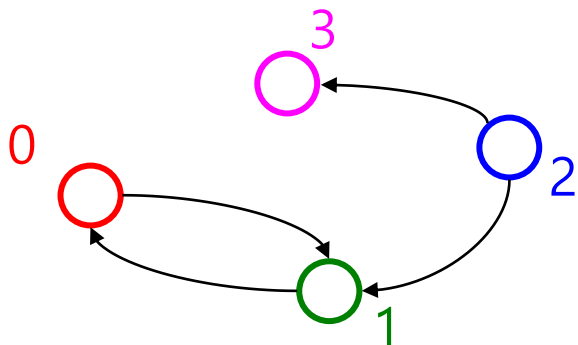
- Running time to:
  - Get a vertex's out-edges:  $O(d)$  where  $d$  is out-degree of vertex
  - Get a vertex's in-edges:  $O(|E|)$  (could keep a second adjacency list for this!)
  - Decide if some edge exists:  $O(d)$  where  $d$  is out-degree of source
  - Insert an edge:  $O(1)$  (unless you need to check if it's already there)
  - Delete an edge:  $O(d)$  where  $d$  is out-degree of source
- Space requirements:  $O(|V|+|E|)$
- Best for sparse or dense graphs? **Sparse graphs**

## ➤ Sample code:

```
# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_edges_from([(0, 1), (1, 0), (2, 1), (2, 3)])

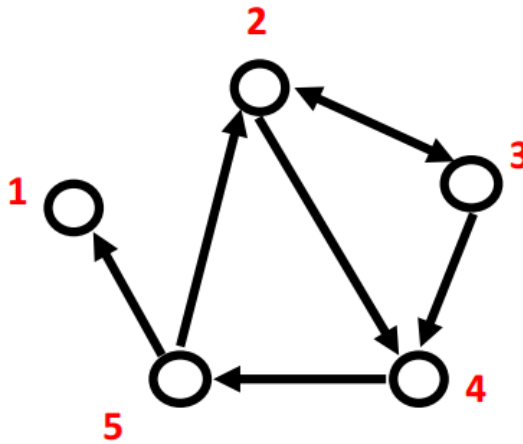
A = nx.adjacency_matrix(G)
print(A)
```

```
(0, 1) 1
(1, 0) 1
(2, 1) 1
(2, 3) 1
```

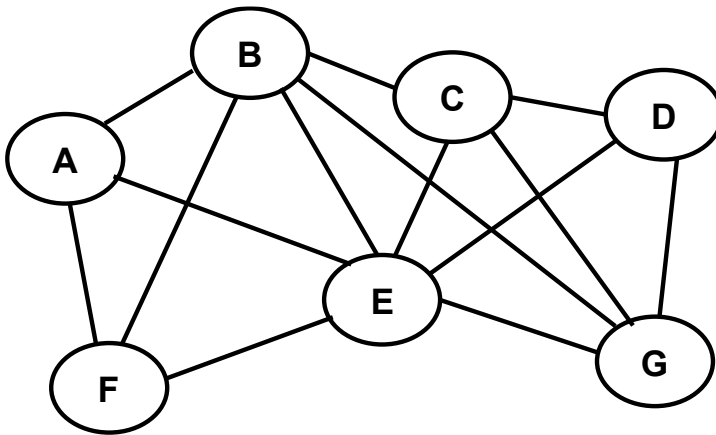


- Graphs are often sparse
  - Streets form grids
  - Airlines rarely fly to all cities
- Adjacency lists should generally be your default choice
  - Slower performance compensated by greater space savings

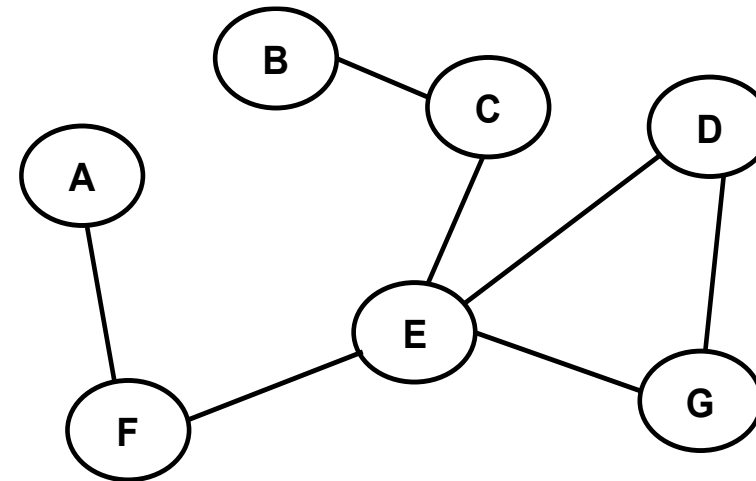
- Adjacency list: Easier to work with if network is large and sparse
- Allows us to quickly retrieve all neighbours of a given node:
  - 1:
  - 2: 3, 4
  - 3: 2, 4
  - 4: 5
  - 5: 1, 2



- If the graph has  $n$  vertices (nodes)
  - Maximum # of edges is  $n^2$
- In dense graphs number of edges is close to  $n^2$
- In sparse graphs number of edges is close to  $n$



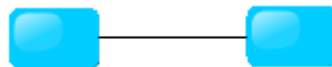
Dense graphs  
(many edges between nodes)



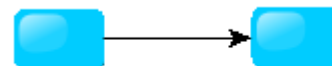
Sparse graphs  
(few edges between nodes)

- A database should store and present all types of graph

- Undirected Graph



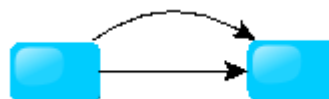
- Directed Graph



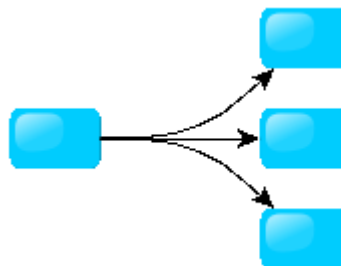
- Pseudo Graph



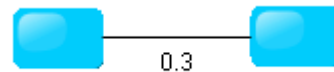
- Multi Graph



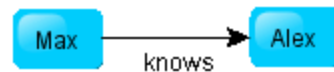
- Hyper Graph



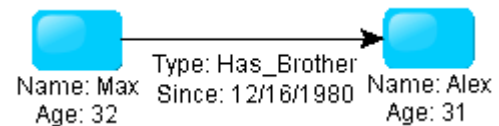
➤ Weighted Graph



➤ Labelled Graph



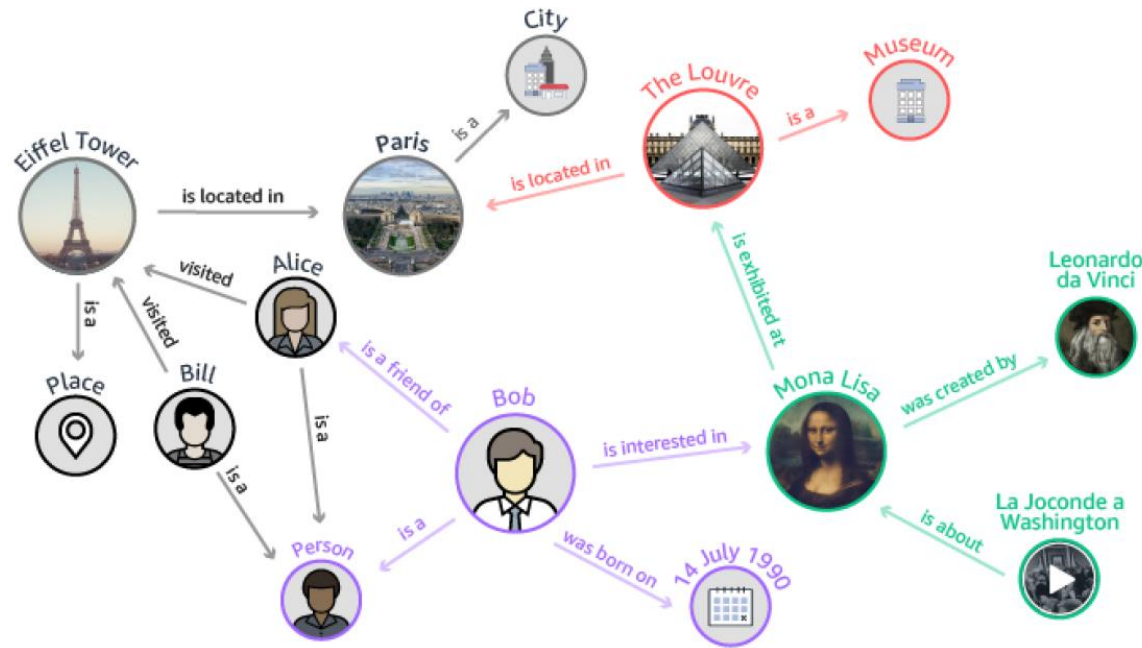
➤ Property Graph



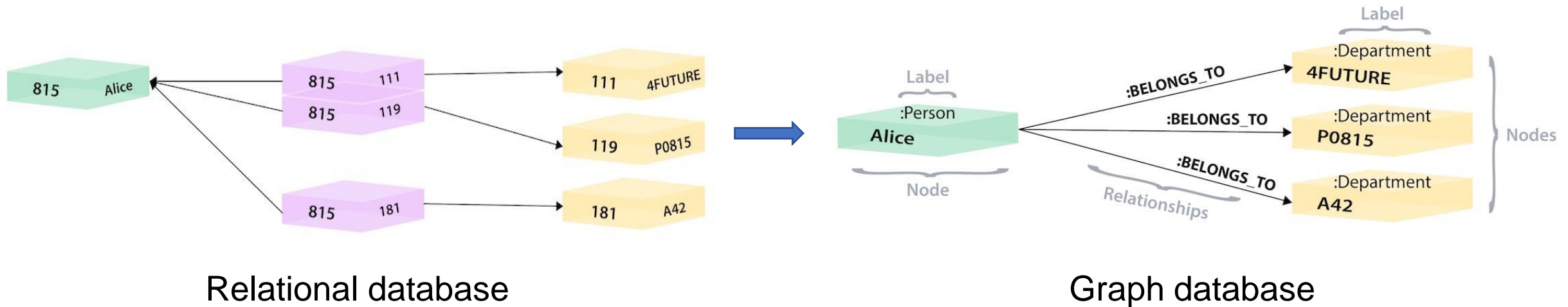


- A database with an explicit graph structure
- Each node knows its adjacent nodes
- As the number of nodes increases, the cost of a local step (or hop) remains the same
- Plus, an Index for lookups

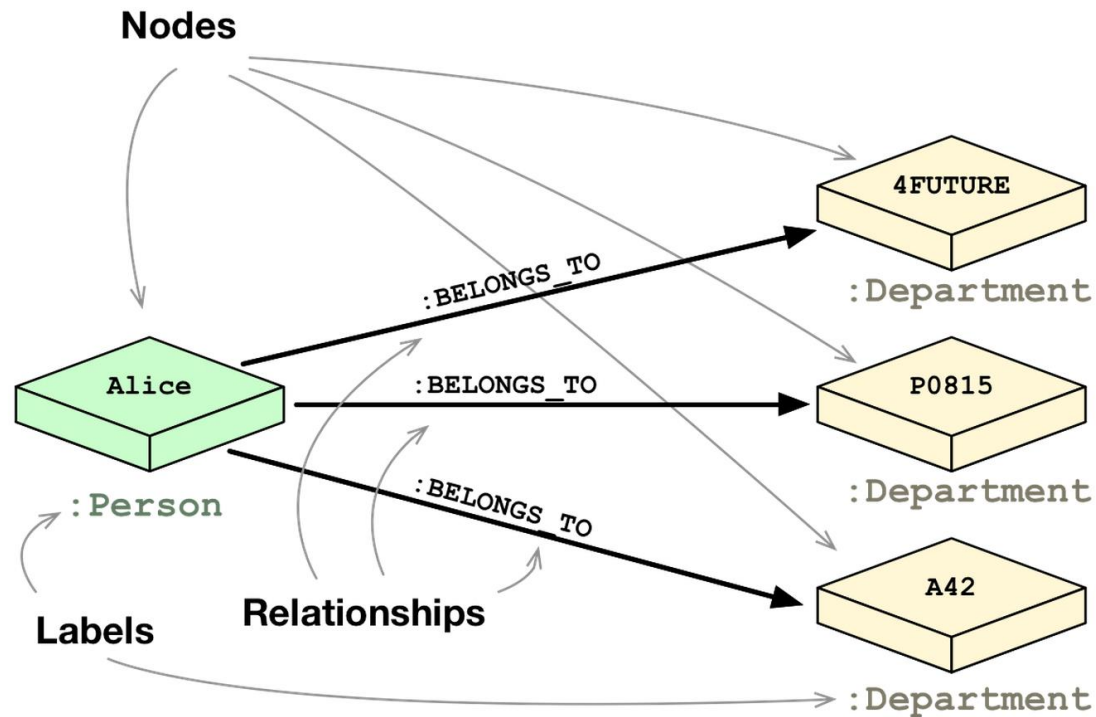
- A graph database stores nodes and relationships instead of tables, or documents. Data is stored just like you might sketch ideas on a whiteboard.
- The data is stored without restricting it to a pre-defined model, allowing a very flexible way of thinking about and using it.



- To find the user Alice and her person ID of 815.
  - We search the Person-Department table (orange middle table) to locate all the rows that reference Alice's person ID (815).
- Once we retrieve the 3 relevant rows, we go to the Department table on the right to search for the actual values of the department IDs (111, 119, 181).
- Now we know that Alice is part of the 4Future, P0815, and A42 departments.



- Nodes: Alice, 4FUTURE, P0815, A42
- Labels: Person, Department
- Relationships: BELONGS\_TO





The #1 Database for Connected Data

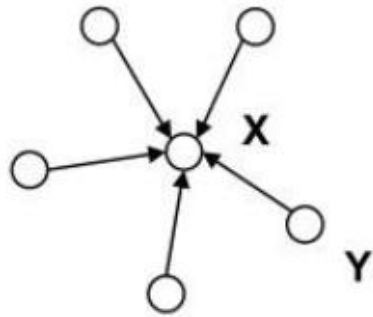


Azure Cosmos DB

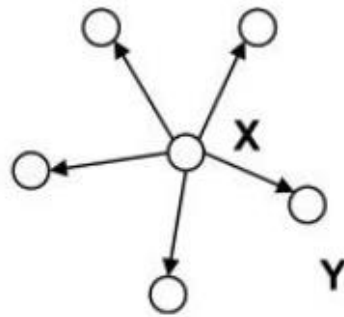


- Knowing the network structure, we can calculate various useful quantities or measures that capture particular features of network topology
- Centrality measures represent the most important nodes in graphs:
  - The most influential person in a social network.
  - The most critical nodes in an infrastructure.
  - The highest spreaders of disease.

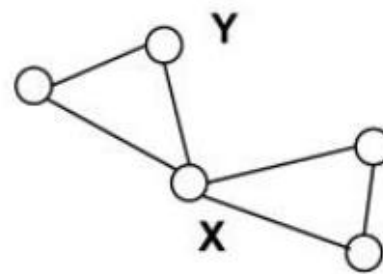
- Who is important based on the network position?
- For example, in each network, 'X' has higher centrality than 'Y' according to a particular measure.



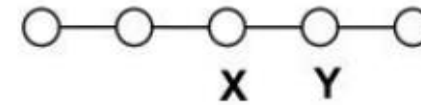
indegree



outdegree



betweenness

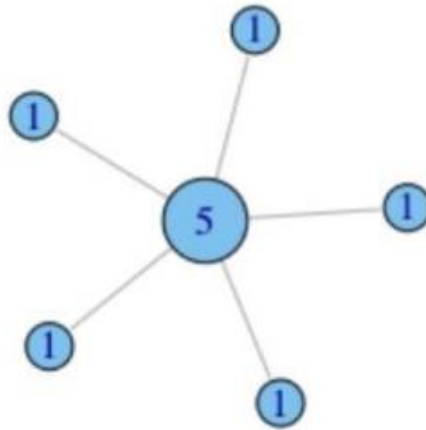


closeness

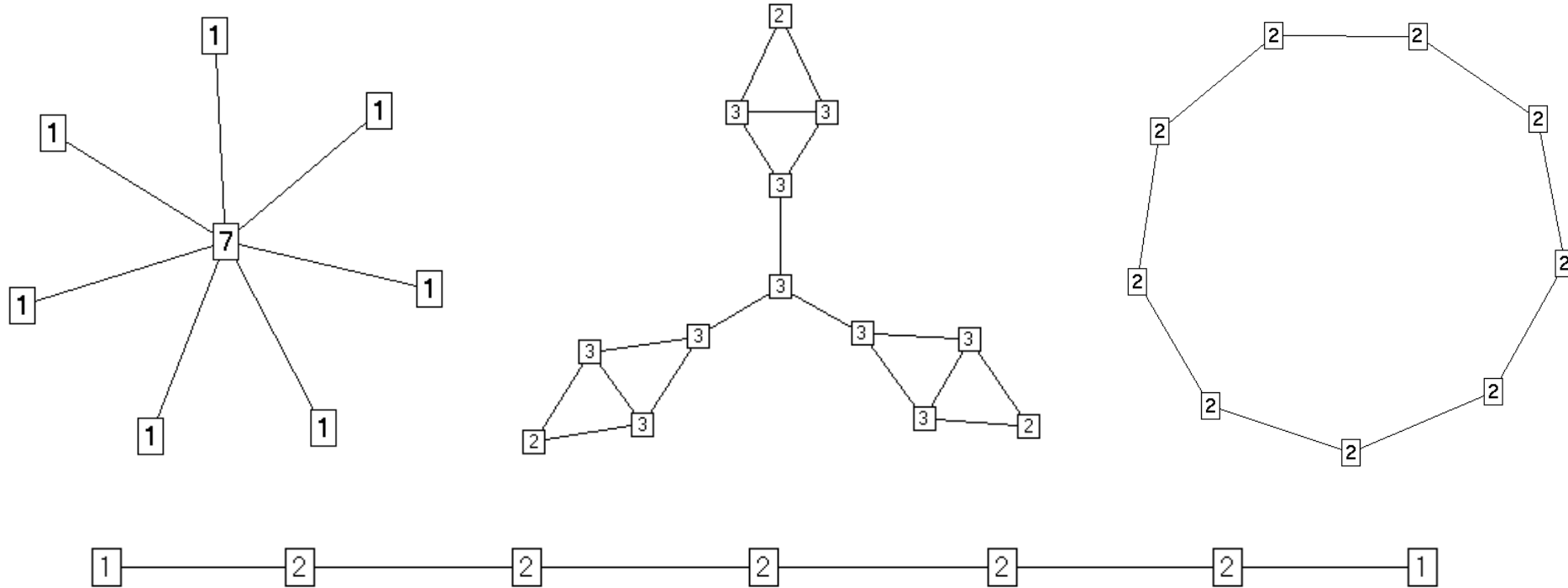


- Degree centrality
- Betweenness centrality
- Closeness centrality
- Eigenvector centrality
- Katz centrality
- PageRank

- People who have many friends is most important.
- When is the number of connections the best centrality measure?
  - People who will do favors for you
  - People you can talk to (influence set, information access, ...)
  - Influence of several actions.

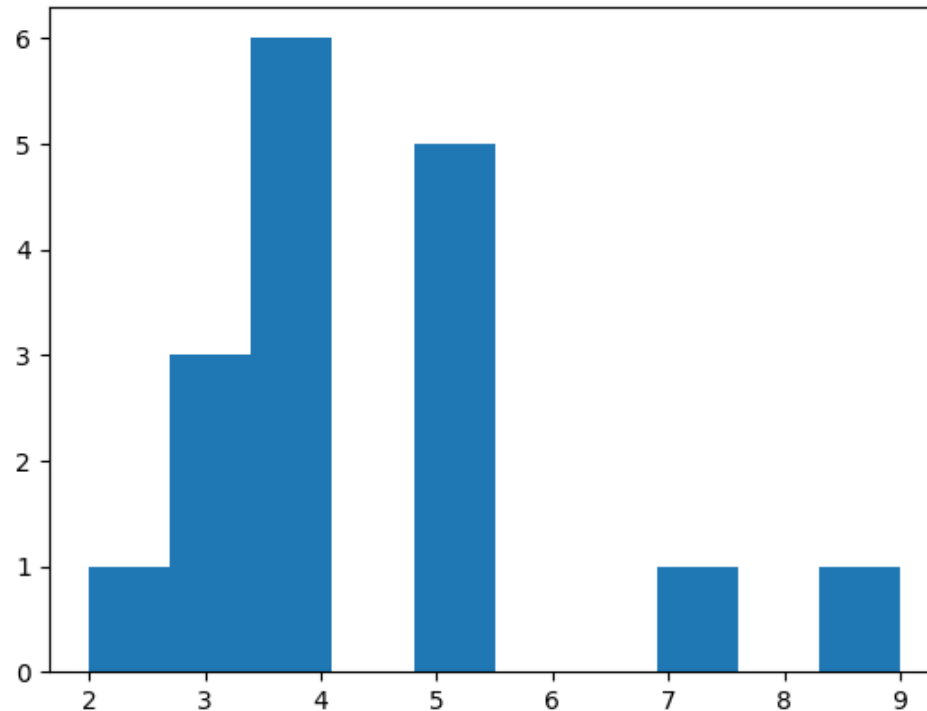


- The most intuitive notion of centrality focuses on degree:
  - The actor with the most ties is the most important:



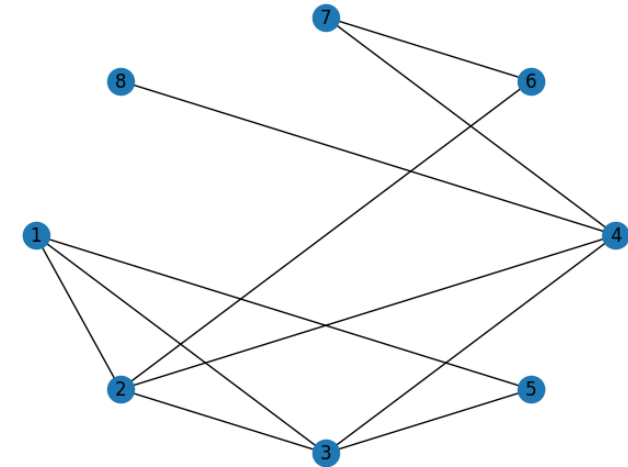
$$C_D(v_i) = d(v_i) = \sum_{j=1}^n A_{ij}$$

- In a simple random graph, degree will have a Poisson distribution, and the nodes with high degree are likely to be at the intuitive center.
- Deviations from a Poisson distribution suggest non-random processes, which is at the heart of current “scale-free” work on networks.



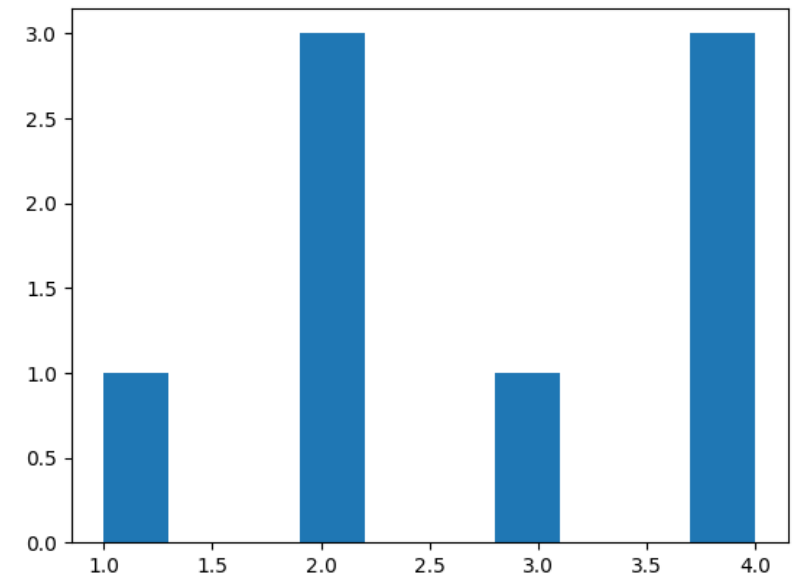
Create graph using edgelist from edge\_list.txt file

```
# Undirected graph
G = nx.read_edgelist('./data/edge_list.txt')
nx.draw(G, pos=nx.shell_layout(G), with_labels = True)
#nx.draw(G, pos=nx.spiral_layout(G), with_labels = True)
# Directed graph
# G = nx.read_edgelist('edge_list.txt', create_using=nx.DiGraph())
```



## Histogram degree plot

```
degrees = [G.degree(n) for n in G.nodes()]
plt.hist(degrees)
```



- If we want to measure the degree to which the graph as a whole is centralized, we look at the *dispersion* of centrality:
  - Simple: variance of the individual centrality scores.

$$S_D^2 = \left[ \sum_{i=1}^n (C_D(v_i) - \bar{C}_d)^2 \right] / n$$

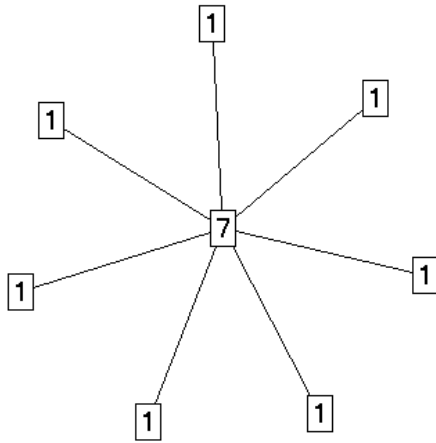
- Or, using Freeman's general formula for centralization (which ranges from 0 to 1): How much variation is there in the centrality scores among nodes?

$$C_D(G) = \frac{\sum_{i=1}^n [C_D(v^*) - C_D(v_i)]}{(n-1)(n-2)},$$

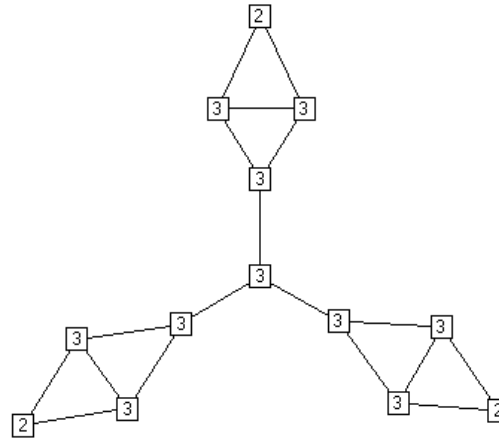
where :

$v^*$  : the node with the highest degree in G

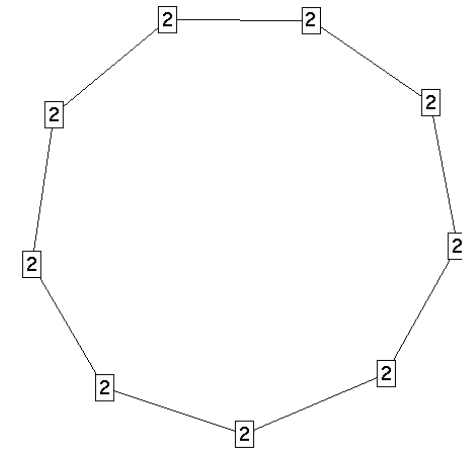
## ➤ Degree centralization scores



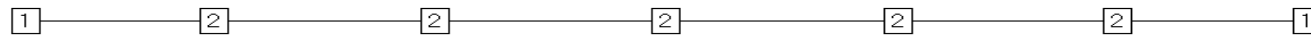
Freeman: 1.0  
Variance: 3.9



Freeman: .02  
Variance: .17



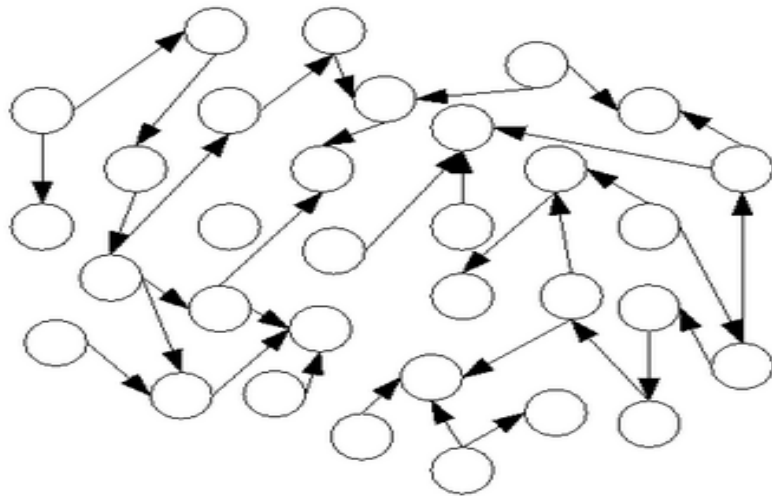
Freeman: 0.0  
Variance: 0.0



Freeman: .07  
Variance: .20

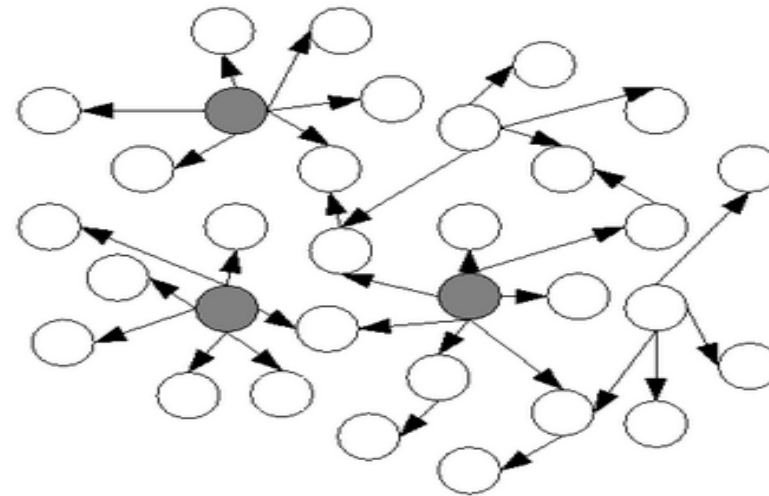


- Random network models introduce an edge between any pair of vertices with a probability  $p$ .
- The problem here is NOT randomness, but rather the distribution used (which, in this case, is uniform)



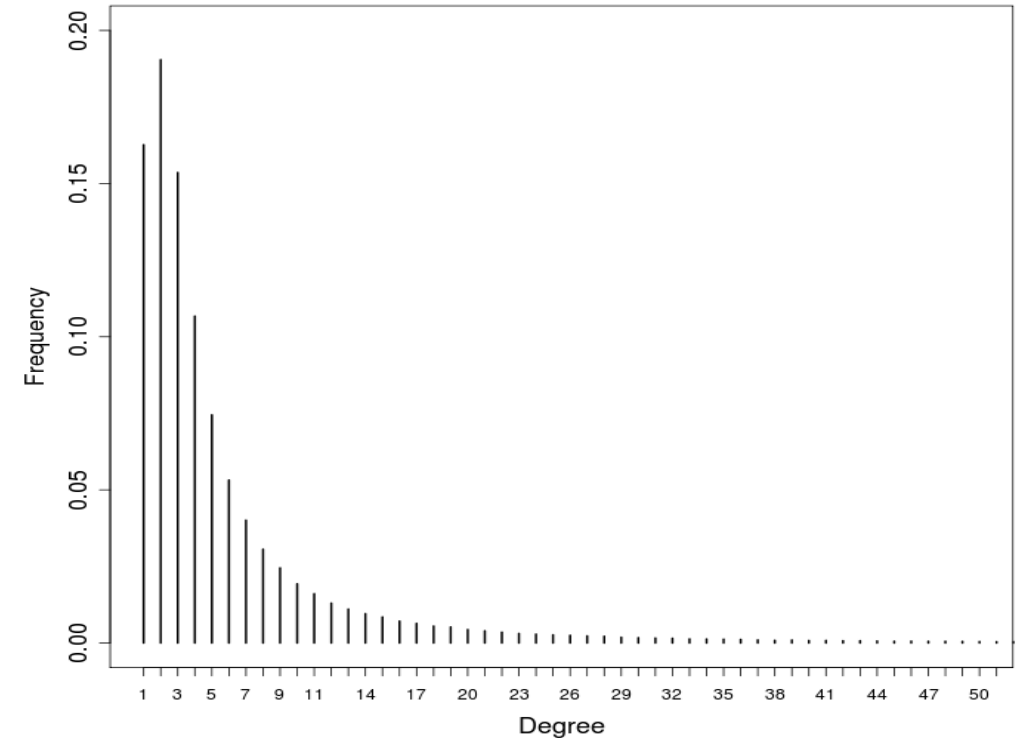
(a) Random network

- Real networks are not exactly like these:
  - Tend to have a relatively few nodes of high connectivity (the “Hub” nodes)
  - These networks are called “Scale-free” networks

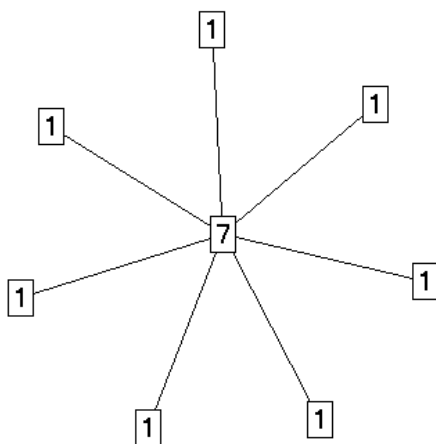


(b) Scale-free network

- In most real networks, the degree distribution is highly asymmetric:
  - Most of the nodes (the trivial many) have low degrees
  - A small but significant fraction of nodes (the vital few) have an extraordinarily high degree. A highly connected node, a node with remarkably high degree, is called hub showing a long tail.



## ➤ Degree centralization scores



Freeman: 1.0

Variance: 3.9

```
: import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

# Instantiate the graph
G = nx.Graph()
# add node/edge pairs
G.add_edges_from([(1, 0), (1, 2), (1, 3), (1, 5), (1, 4), (1, 6), (1, 7)])
# Degree plot for undirected and weighted graph
degrees = [G.degree(n) for n in G.nodes()]

#degrees = [G.degree(n, weight='weight') for n in G.nodes()]
plt.hist(degrees)
mean = sum(degrees) / len(degrees)

v_start = max(degrees)
n = len(degrees)
var = sum((i - mean) ** 2 for i in degrees) / n
freeman = sum((v_start - i) for i in degrees) / ((n-1)*(n-2))
print(f'variance: {var}')
print(f'Freeman: {freeman}')
```

```
variance: 3.9375
Freeman: 1.0
```

- Using Freeman's general formula for centralization (which ranges from 0 to 1):

$$C_D(G) = \frac{\sum_{i=1}^n [C_D(v^*) - C_D(v_i)]}{(n-1)(n-2)},$$

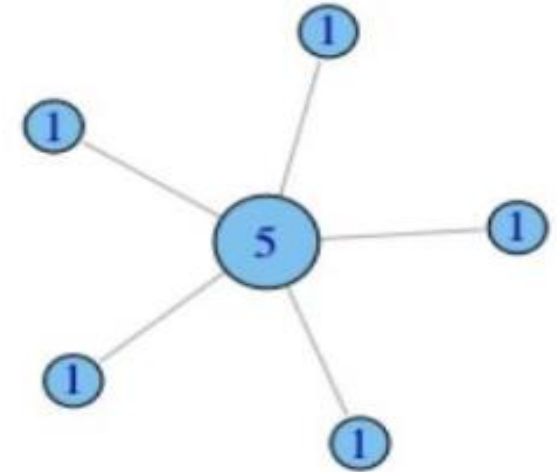
where:

$v^*$ : the node with the highest degree in  $G$



$$C_D = 0.167$$

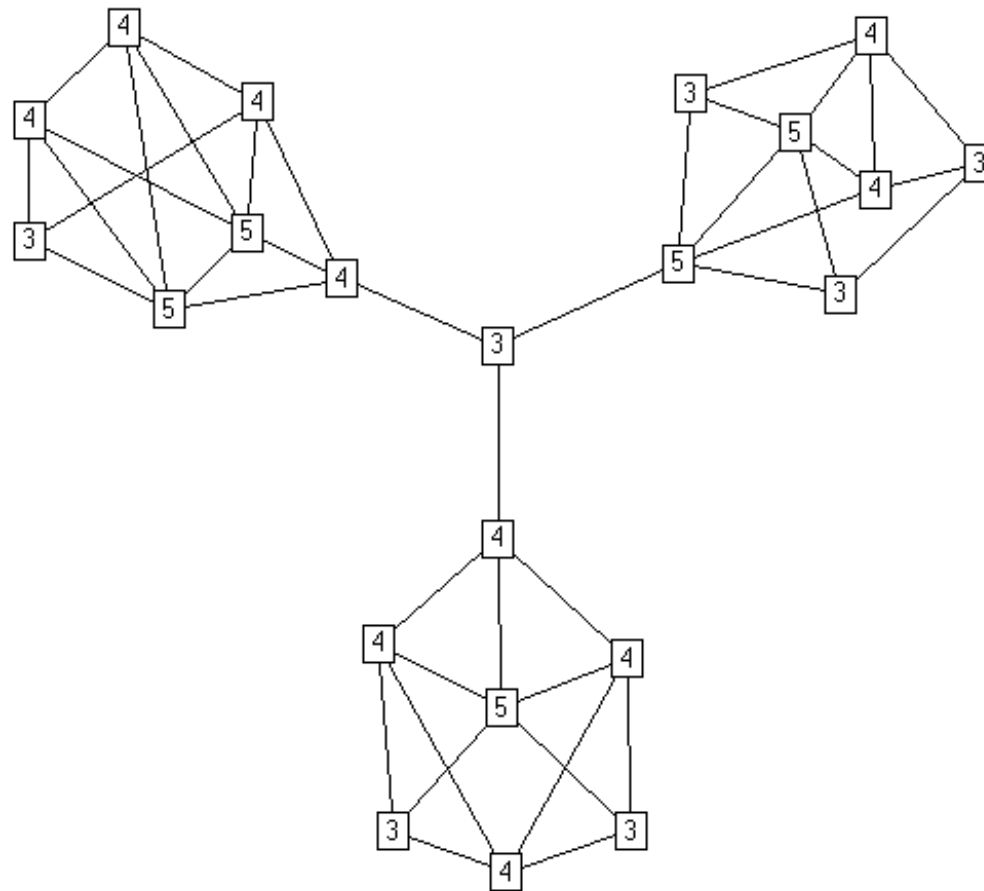
$$C_D(G) = \frac{(2-1) + (2-0) + \dots + (2-1)}{(5-1)(5-2)} = 0.167$$



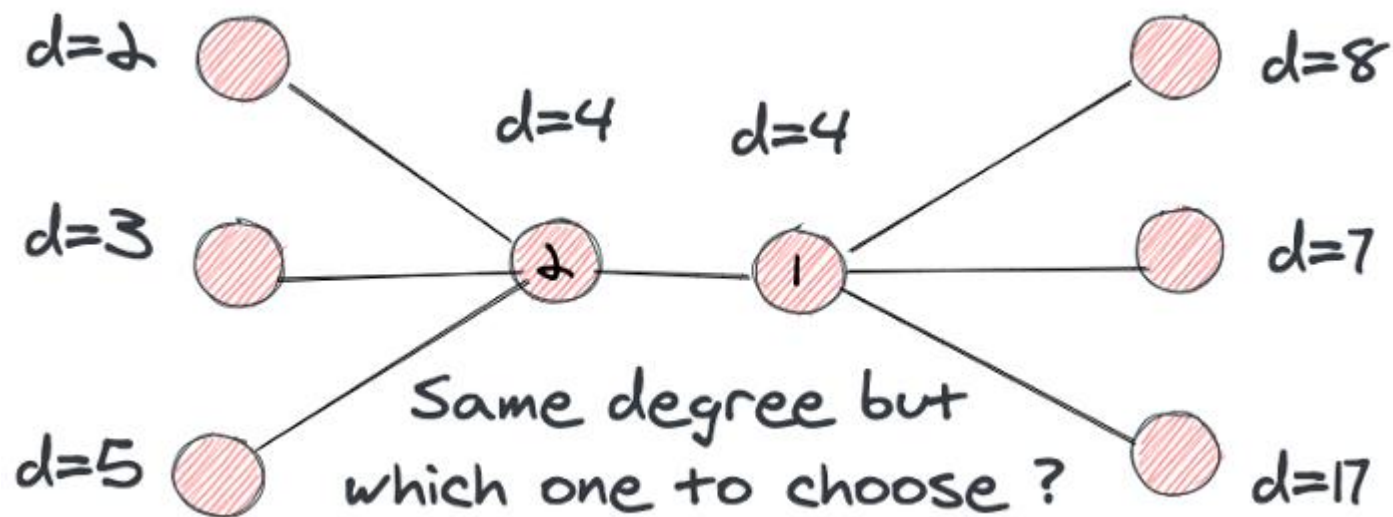
$$C_D = 1.0$$

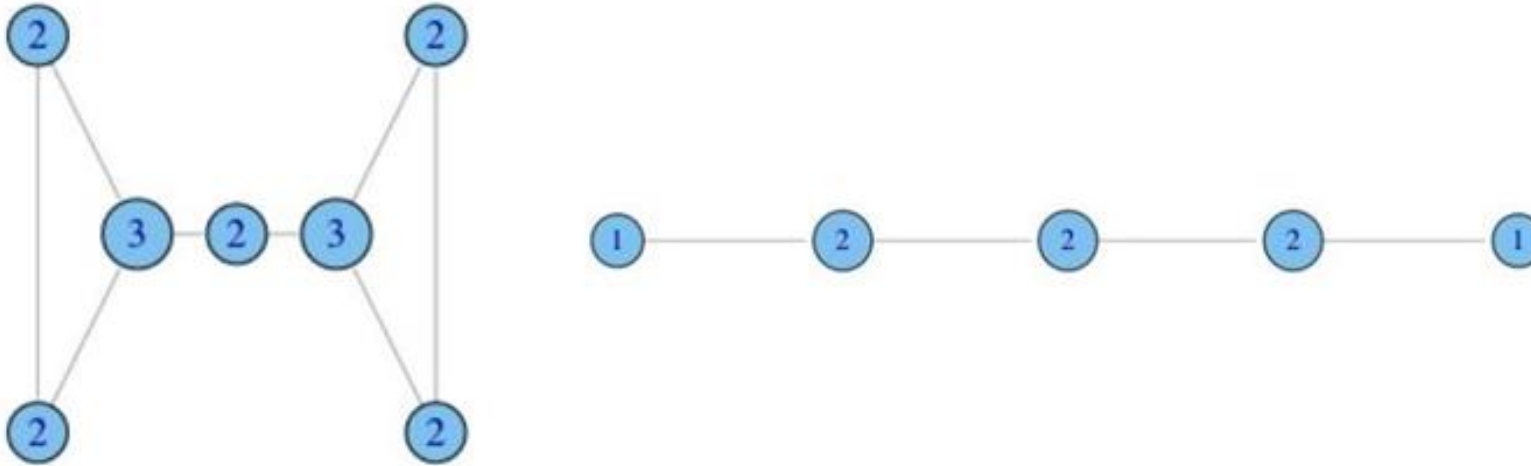
$$C_D(G) = \frac{(5-1) + \dots + (5-1)}{(6-1)(6-2)} = \frac{20}{20} = 1$$

- Degree centrality, however, can be deceiving, because it is a purely local measure



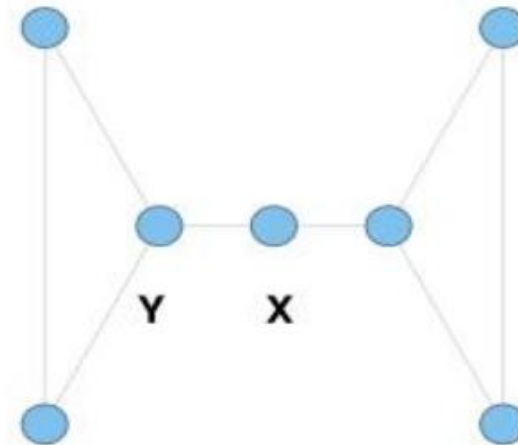
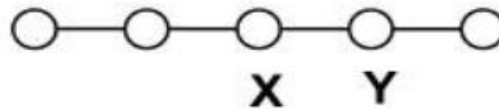
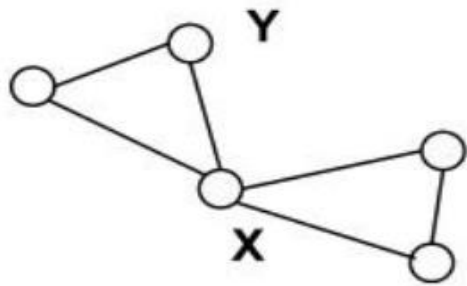
- Node degree is local, not global





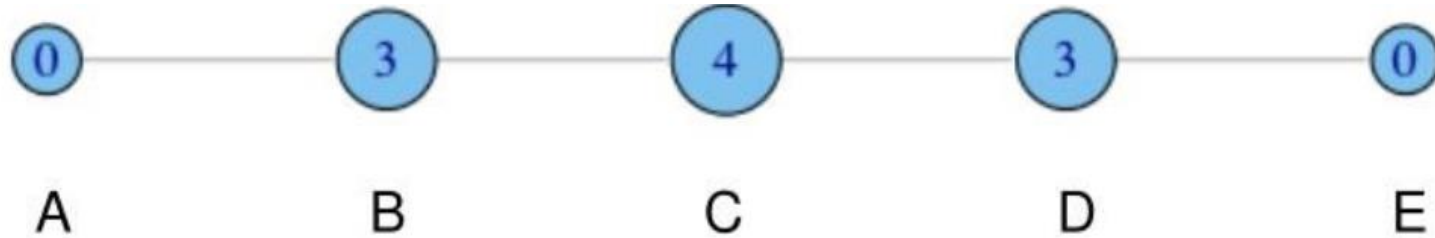
- Ability to broker between groups.
- The likelihood that information originating anywhere in the network reaches you.

- Intuition:
  - How many pairs of individuals would have to go through you in order to reach one another in the minimum number of hops?
- Who has higher betweenness, X or Y?





- Non-normalized version:
  - A lies between no two other nodes
  - B lies between A and 3 other nodes: C,D,E
  - C lies between 4 pairs: (A,D), (A,E), (B,D), (B,E)



- Betweenness centrality of node  $v_i$ :

$$B(v_i) = \sum_{v_j, v_k \in G} \frac{SPD_{v_j \rightarrow v_k}(v_i)}{SPD_{v_j \rightarrow v_k}}$$

The number of shortest paths between  $v_j$  and  $v_k$  that pass through the vertex  $v_i$

The number of shortest paths from  $v_j$  to  $v_k$

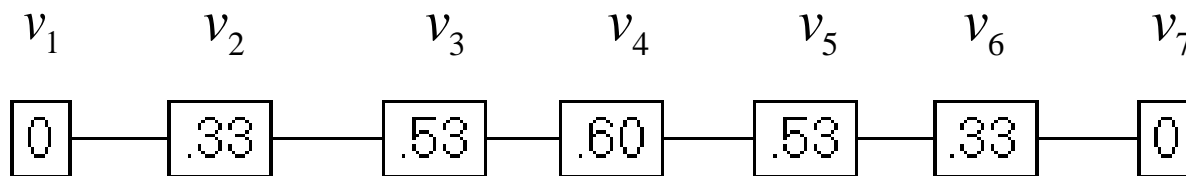
- Usually normalized by:

No. of nodes in the graph

$$\bar{B}(v_i) = B(v_i) / [(n-1)(n-2) / 2]$$

- Vertices with high betweenness centrality have influence in the network by virtue of their control over information passing between others.
  - They get to see the messages as they pass through
  - They could get paid for passing the message along
- Thus, they get a lot of power: their removal would disrupt communication

➤ Betweenness centrality of node  $v_2$  :



$$SPD_{v_1 \rightarrow v_3}(v_2) = 1, SPD_{v_1 \rightarrow v_4}(v_2) = 1, SPD_{v_1 \rightarrow v_5}(v_2) = 1, SPD_{v_1 \rightarrow v_6}(v_2) = 1, SPD_{v_1 \rightarrow v_7}(v_2) = 1$$

$$SPD_{v_3 \rightarrow v_4}(v_2) = 1, SPD_{v_3 \rightarrow v_5}(v_2) = 1, SPD_{v_3 \rightarrow v_6}(v_2) = 1, SPD_{v_3 \rightarrow v_7}(v_2) = 1$$

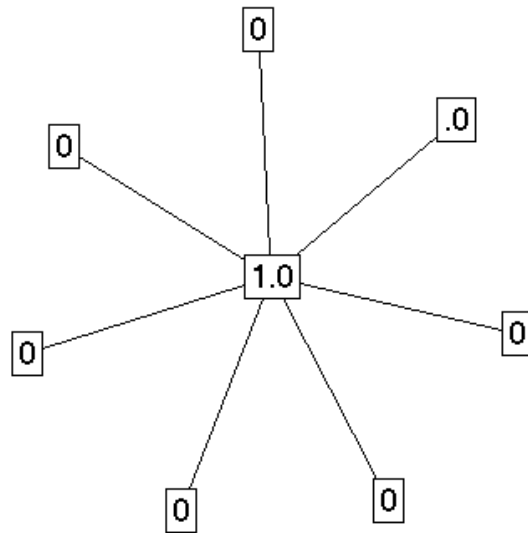
$$B(v_2) = \sum_{v_j, v_k \in G} \frac{SPD_{v_j \rightarrow v_k}(v_2)}{SPD_{v_j \rightarrow v_k}} = \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} = 5$$

$$\bar{B}(v_2) = \frac{B(v_2)}{(n-1)(n-2)/2} = \frac{5}{(7-1)(7-2)/2} = 0.33$$

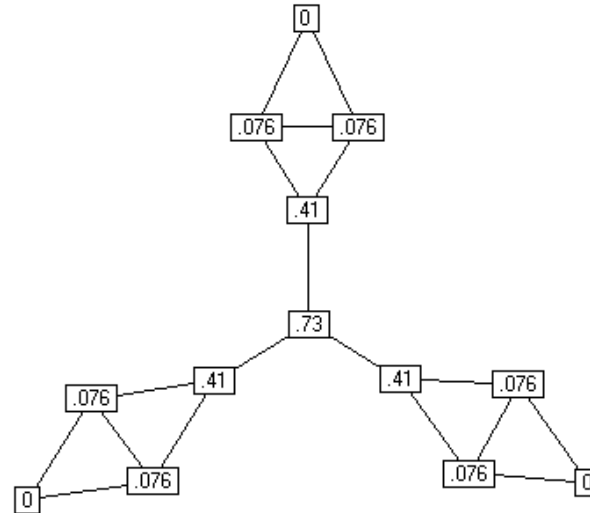
➤ Centralization:

$$C_D(G) = \frac{\sum_{i=1}^n [B(v^*) - B(v_i)]}{(n-1)},$$

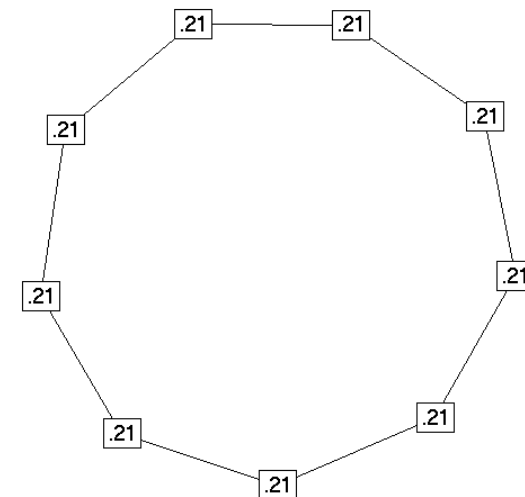
where :  $v^*$  is the node with the highest betweenness in G  
B: Betweenness centrality



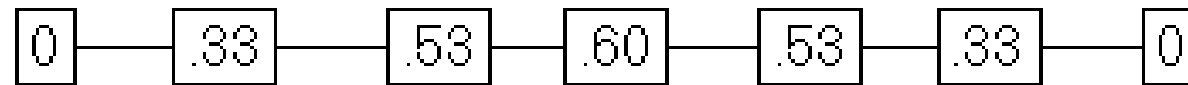
Centralization: 1.0



Centralization: .59



Centralization: 0



Centralization: .31

```
: import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
```

```
# Instantiate the graph
```

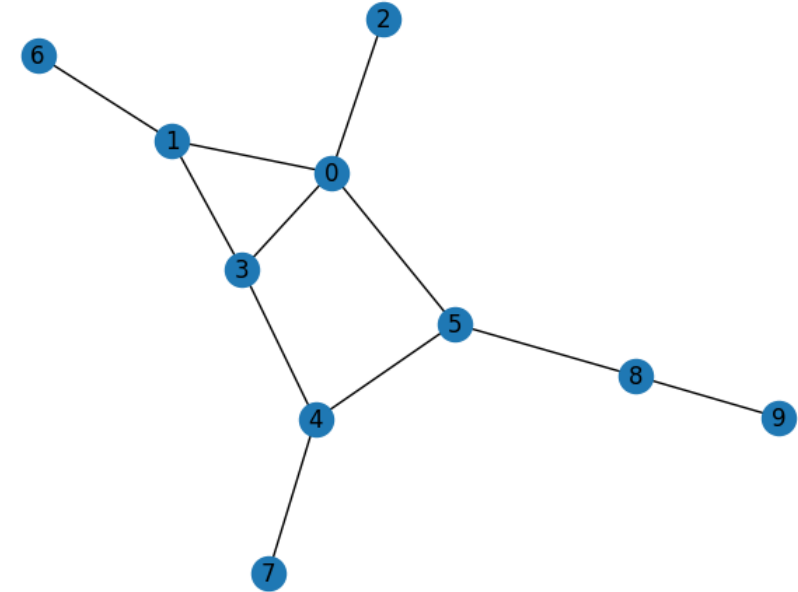
```
G = nx.Graph()
```

```
# add node/edge pairs
```

```
G.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)])
```

```
: #Sort for identifying most influential nodes using betweenness centrality
for node in sorted(betweenness_centrality, key=betweenness_centrality.get, reverse=True):
    print(node, betweenness_centrality[node])
```

```
5 0.4444444444444444
0 0.4305555555555555
4 0.2638888888888889
1 0.2222222222222222
8 0.2222222222222222
3 0.16666666666666666
2 0.0
6 0.0
7 0.0
9 0.0
```



- Betweenness centrality is a measure of a node's influence over the flow of information in the network, often used to find nodes that serve as bridge between different network partitions
- Pros:
  - Entire network: The position of a node is related to the whole network
  - Flow perspective: It represents a different kind of importance compared to other centralities
- Cons:
  - Limit access: Access limited through only a few key persons might slow down the flow of information if some of resources or people become unavailable

- Problem:
  - What if: It is not so important to have many direct friends? Or be “between” others.
- But one still wants to be in the “middle” of things
  - Node  $v_i$  not too far from the center



- The closeness is defined so that if a vertex is close to every other vertex, then the value is larger than if the vertex is not close to everything else.

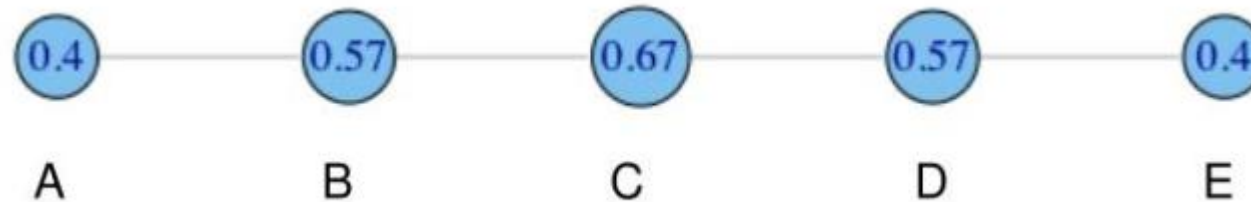
d: The number of nodes in the shortest path between 2 nodes

$$C(v_i) = \left[ \sum_{j=1}^n |d(v_i, v_j)| \right]^{-1}$$

- Normalized Closeness Centrality:

$$\bar{C}(v_i) = \frac{n-1}{\sum_{j=1}^n |d(v_i, v_j)|}$$

- The Closeness centrality of node A = Sum of the number of nodes between A and other nodes:
  - SPD(A,B)=1
  - SPD(A,C)=2
  - SPD(A,D)=3
  - SPD(A,E)=4



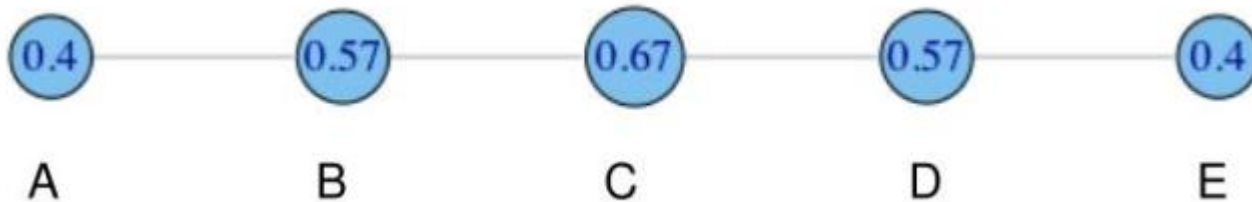
$$\bar{C}(A) = \frac{N - 1}{\sum_{j=1}^n |d(v_i, v_j)|} = \frac{4}{1 + 2 + 3 + 4} = 0.4$$

```
G = nx.Graph()
# add node/edge pairs
G.add_edges_from([(0, 1),
                  (1, 2),
                  (2, 3),
                  (3, 4)])

nx.draw(G, with_labels = True)

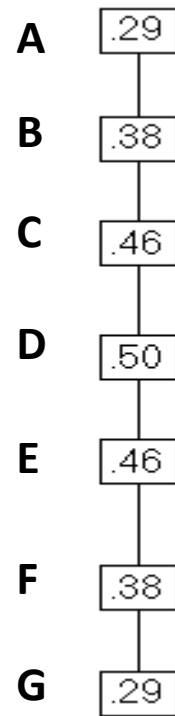
closeness centrality = nx.closeness centrality(G)
print(closeness centrality)
#Sort for identifying most influential nodes using closeness centrality
#for node in sorted(closeness centrality, key=closeness centrality.get, reverse=True):
# print(node, closeness centrality[node])
```

```
{0: 0.4, 1: 0.5714285714285714, 2: 0.6666666666666666, 3: 0.5714285714285714, 4: 0.4}
```



$$\bar{C}(A) = \frac{N - 1}{\sum_{j=1}^n |d(v_i, v_j)|} = \frac{4}{1 + 2 + 3 + 4} = 0.4$$

- Given a graph with 7 nodes: A,B,C,D,E,F,G
- Distance matrix denotes the shortest path distance between two nodes



Distance							Closeness normalized	
A	B	C	D	E	F	G	.048	.286
B	0	1	2	3	4	5	.063	.375
C	1	0	1	2	3	4	.077	.462
D	2	1	0	1	2	3	.083	.500
E	3	2	1	0	1	2	.077	.462
F	4	3	2	1	0	1	.063	.375
G	5	4	3	2	1	0	.048	.286

- Comparing across these 3 centrality values
  - Generally, the 3 centrality types will be positively correlated
  - When they are not (low) correlated, it probably tells you something interesting about the network.

	Low Degree	Low Closeness	Low Betweenness
High Degree		Embedded in cluster that is far from the rest of the network	Ego's connections are redundant - communication bypasses him/her
High Closeness	Key player tied to important important/active alters		Probably multiple paths in the network, ego is near many people, but so are many others
High Betweenness	Ego's few ties are crucial for network flow	Very rare cell. Would mean that ego monopolizes the ties from a small number of people to many others.	

- An extension of degree centrality
  - Centrality increases with number of neighbors
- Not all neighbors are equal
  - Having connection to more central nodes increases importance
  - For example, a node with 300 relatively unpopular friends on Facebook would have lower eigenvector centrality than someone with 300 very popular friends.

- Computing Eigenvector centrality:

$$x_i(t) = \sum_j A_{ij} x_j(t-1)$$

with the centrality at time  $t=0$  being  $x_i(0) = 1, \forall j$


$A$  denotes the adjacency matrix.

- Define the centrality  $x'_i$  of  $i$  recursively in terms of the centrality of its neighbours:

$$x'_i = \sum_{v_j \in N(v_i)} A_{ij} x_j \quad \text{with the initial node centrality } x_j = 1, \forall j$$

- That is equivalent to:

$$x_i(t) = \sum_{v_j \in N(v_i)} A_{ij} x_j(t-1) \quad \text{with the centrality at time } t=0 \text{ being } x_j(0) = 1, \forall j$$



The centrality of nodes  $x_i$  and  $x_j$  at time  $t$  and  $(t-1)$ , respectively.

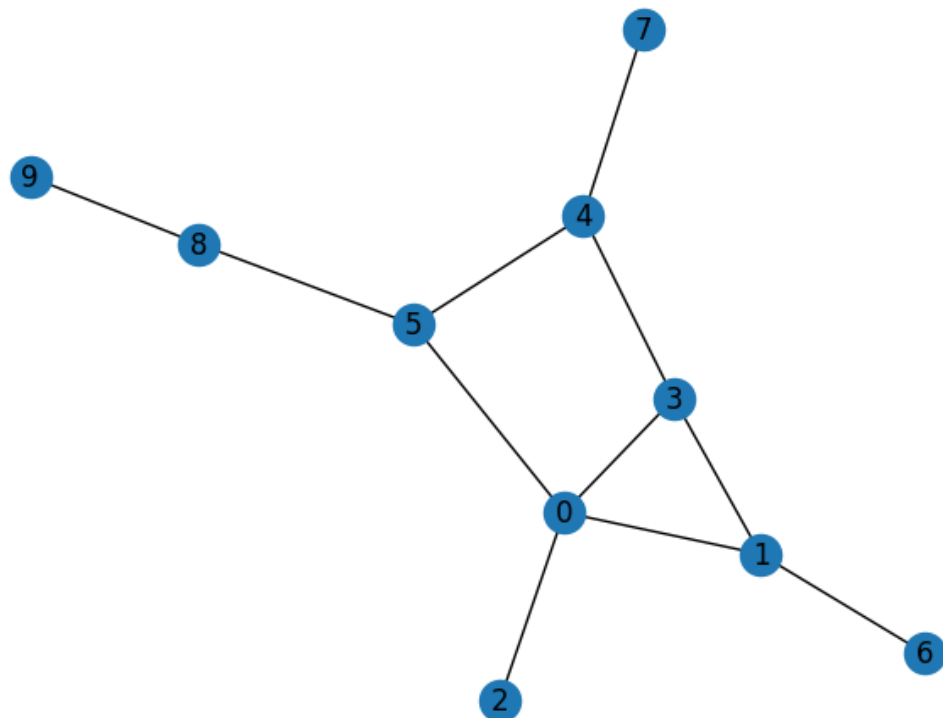


```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

# Create a graph
G = nx.Graph()
# add node/edge pairs
G.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)])

# Calculate eigenvector centrality
eigenvector_centrality = nx.eigenvector_centrality(G)
# Sort for identifying most influential nodes using eigenvector centrality
for node in sorted(eigenvector_centrality, key=eigenvector_centrality.get, reverse=True):
    print(node, eigenvector_centrality[node])
```

```
0 0.5163311132778224
3 0.4605589097129744
1 0.40781116737108153
5 0.3684701926326299
4 0.34608812874643025
2 0.18721071680314055
8 0.15382393197189426
6 0.14786314470285283
7 0.12548504820698178
9 0.05577442762250838
```



- Eigenvector Centrality is an algorithm that measures the transitive influence of nodes.
- Relationships originating from high-scoring nodes contribute more to the score of a node than connections from low-scoring nodes.
- A high eigenvector score means that a node is connected to many nodes who themselves have high scores.

- Katz centrality computes the centrality for a node based on the centrality of its neighbours. It is a generalization of the eigenvector centrality.
- The Katz centrality for node  $v_i$  is:

$$x_i = \alpha \sum_j A_{ij} x_j + \beta,$$

where:

$\alpha$  is a constant called damping factor, and  $\beta$  is a bias constant,  
 $A$  is the adjacency matrix.

- When  $\alpha = 1 / \lambda_{\max}$ ,  $\beta = 0$ , Katz centrality is the same as eigenvector centrality

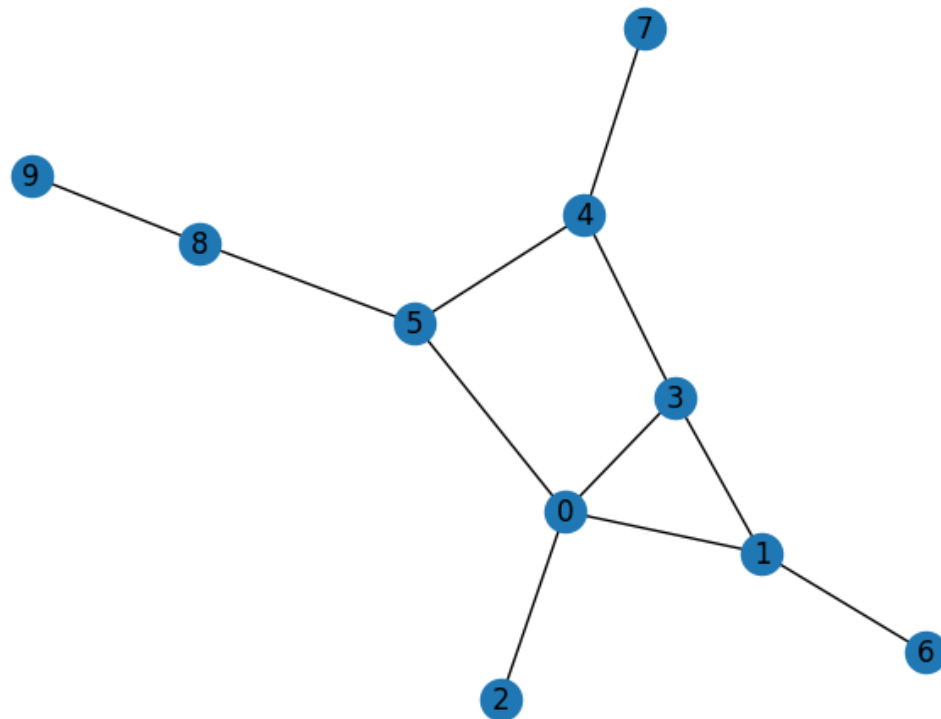
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

# Create a graph
G = nx.Graph()
# add node/edge pairs
G.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)])
nx.draw(G, with_labels = True)
#Calculate kat centrality

kat centrality = nx.katz centrality(G)

#Sort for identifying most influential nodes using kat centrality
for node in sorted(kat centrality, key=kat centrality.get, reverse=True):
    print(node, kat centrality[node])
```

```
3 0.39837116164016045
2 0.3979957464558509
4 0.3912885477686667
1 0.3627994945952694
5 0.3266180854671848
6 0.32248848309558037
7 0.3218787410123867
8 0.28962990709484226
```



- Katz centrality is a network measure that considers all path between nodes, not just shortest ones. It penalizes connections with distant nodes
- Pros:
  - Considering all paths between nodes, not just the shortest ones, which can provide more comprehensive measure of influence within a network
  - It is like Google's PageRank and the eigenvector centrality, which are widely used and recognized measures
- Cons:
  - It can be computationally intensive for large networks as it involves calculating the total number of walks between all pairs of nodes
  - The attenuation factor, which penalizes connections with distant nodes, need to be chosen carefully. If not, it could skew the centrality measures

- Eigenvector centrality:  $i$ 's Rank score  $x_i$  is the sum of the Rank scores  $x_j$  of all pages  $j$  that point to  $i$ :

$$x_i = \sum_{(j,i) \in E} x_j$$

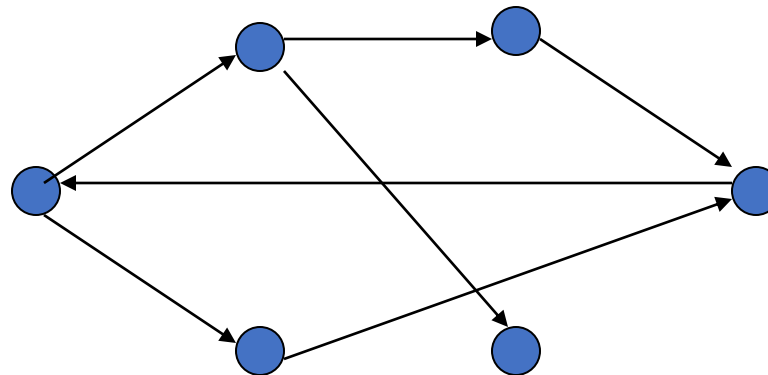
- Then Katz centrality adds **the teleportation** by adding a small weight edge to each node (using a weight of  $\beta$ ):

$$x_i = \sum_{(j,i) \in E} x_j + \beta$$

- BUT, since a page  $j$  may point to many other pages, its prestige score should be shared among these pages.  
(For example one website can point to many sites)

$$x_i = \sum_{(j,i) \in E} \frac{x_j}{\text{out deg } x_j} + \beta$$

- Web pages are organized in a network.
  - Each webpage is represented as a node.
  - Each hyperlink is a directed edge
  - The entire web can be viewed as a directed graph.



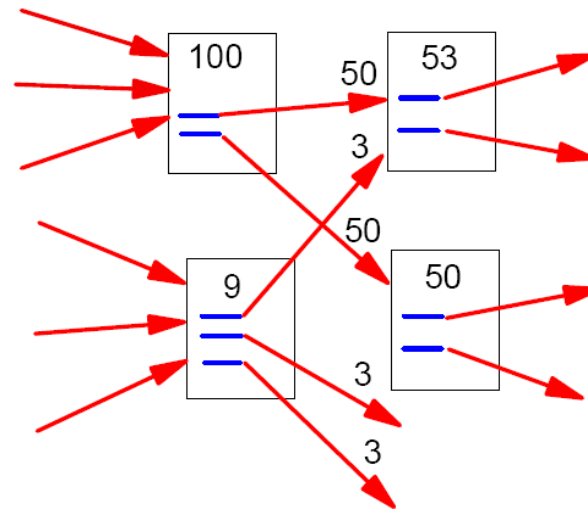
PageRank is a numeric value that represents how important a page is on the web.

- Webpage importance
  - One page links to another page = A vote for the other page A link from page A to page B is a vote on A to B.
  - If page A is more important itself, then the vote of A to B should carry more weight.
  - More votes = More important the page must be
- How can we model this importance?



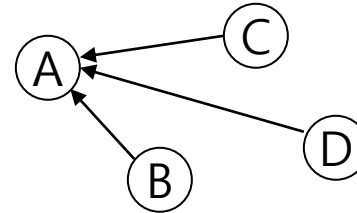
## ➤ Importance Computation

- The importance of a page is distributed to pages that it points to.
- The importance of a page is the aggregation of the importance shares of the pages that points to it.
  - If a page has 5 outlinks, the importance of the page is divided into 5 and each link receives one fifth share of the importance.

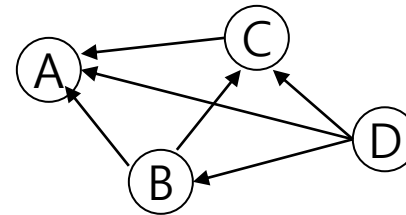


- Assume four web pages: A, B, C and D. Let each page would begin with an estimated PageRank of 0.25.

$$PR(A) = PR(B) + PR(C) + PR(D)$$



$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}$$



- $L(A)$  is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

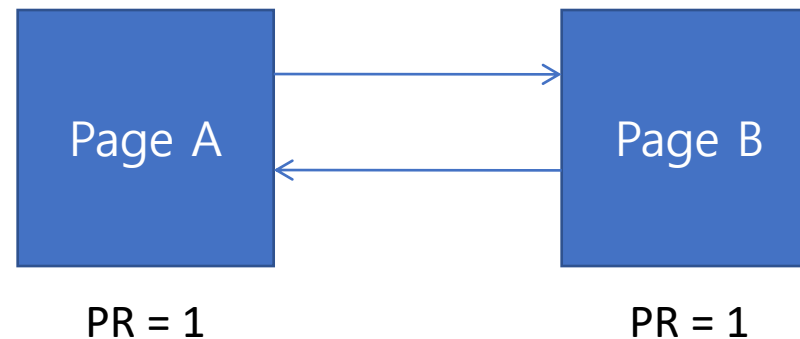
$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}$$

- Assume page A has pages B, C, D ..., which point to it.
- The parameter d is a damping factor which can be set between 0 and 1.
- Usually set d to 0.85.
- The PageRank of a page A is given as follows:

$$PR(A) = 1 - d + d \left( \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \dots \right)$$

- Where:
  - d is a damping factor which can be set between 0 and 1 (usually set to 0.85)
  - L(X) is the number of outbound links on page X

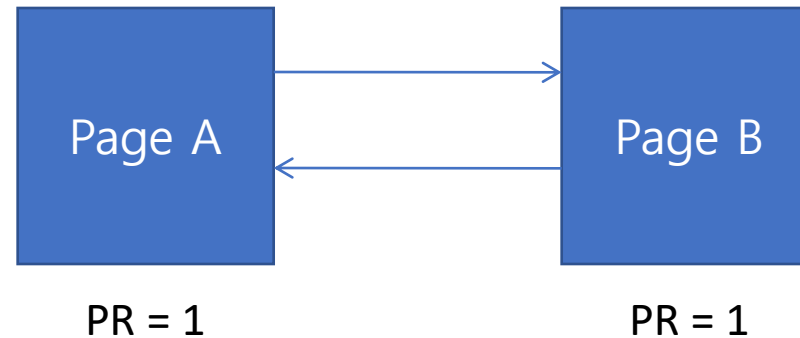
- We can't work out A's PageRank until we know B's PageRank, and we can't work out B's PageRank until we know A's PageRank.
- Iterations are necessary to calculate the most accurate values by using inaccurate values



$$PR(A) = 0.15 + 0.85 * PR(B)$$

$$PR(B) = 0.15 + 0.85 * PR(A)$$

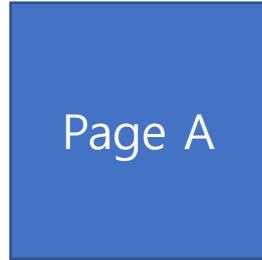
- A website has a maximum amount of PageRank that is distributed between its pages by internal links
- The maximum amount of PageRank in a site increases as the number of pages in the site increases
- By linking poorly, it is possible to fail to reach the site's maximum PageRank, but it is not possible to exceed it



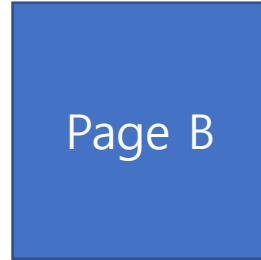
$$PR(A) = 0.15 + 0.85 * PR(B)$$

$$PR(B) = 0.15 + 0.85 * PR(A)$$

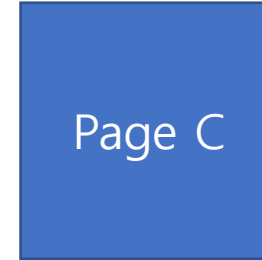
- Maximum PageRank is the amount of PageRank in the site.
- So this site's maximum PageRank is 3.



PR = 1



PR = 1



PR = 1

$$PR(A) = 0.15 + 0.85 * (0) = 0.15$$

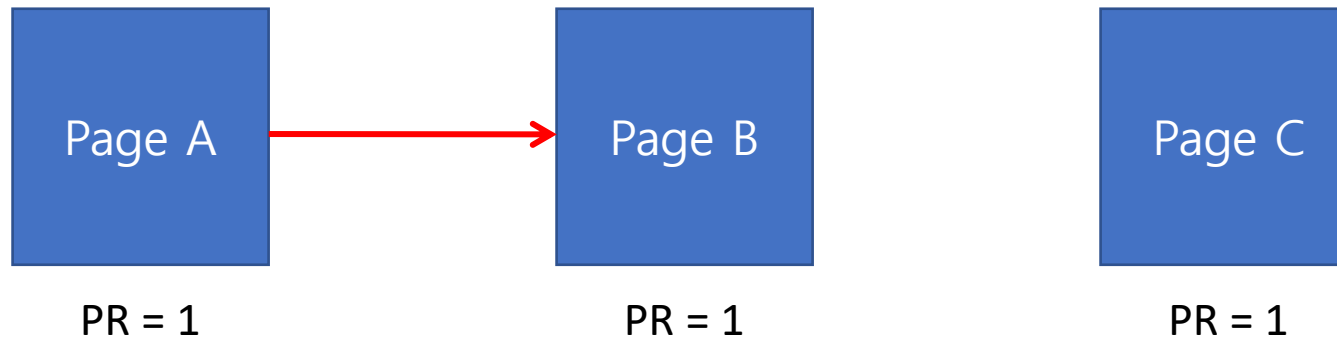
$$PR(B) = 0.15 + 0.85 * (0) = 0.15$$

$$PR(C) = 0.15 + 0.85 * (0) = 0.15$$

Total PageRank in this site = 0.45

Wasting most of its potential PageRank!

- Maximum PageRank is the amount of PageRank in the site.
- So this site's maximum PageRank is 3.



$$PR(A) = 0.15 + 0.85 * ( 0 ) = 0.15$$

$$PR(B) = 0.15 + 0.85 * ( PR(A)/1 ) = 0.15 + 0.85 * ( 1 ) = 1$$

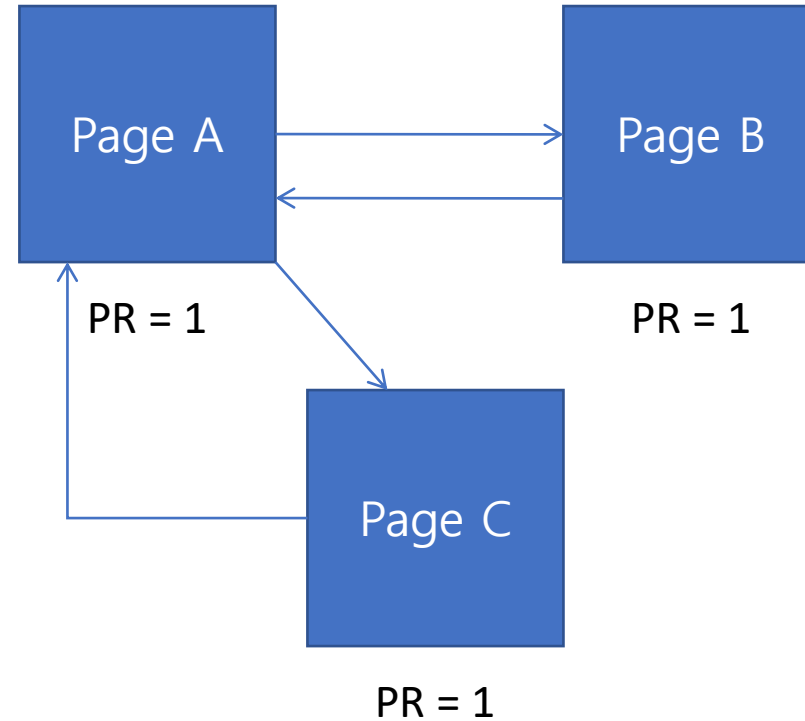
$$PR(C) = 0.15 + 0.85 * ( 0 ) = 0.15$$

After 100 iterations....

$$\begin{aligned} \text{PR}(A) &= 0.15 + 0.85 * ( \text{PR}(B)/1 + \text{PR}(C) / 1 ) \\ &= 0.15 + 0.85 * ( 1 + 1 ) \\ &= 0.15 + 1.7 \\ &= 1.85 \end{aligned}$$

$$\begin{aligned} \text{PR}(B) &= 0.15 + 0.85 * ( \text{PR}(A)/2 ) \\ &= 0.15 + 0.85 * ( 0.5 ) \\ &= 0.15 + 0.425 \\ &= 0.575 \end{aligned}$$

$$\begin{aligned} \text{PR}(C) &= 0.15 + 0.85 * ( \text{PR}(A)/2 ) \\ &= 0.15 + 0.85 * ( 0.5 ) \\ &= 0.15 + 0.425 \\ &= 0.575 \end{aligned}$$



After 100 iterations...

Page A = 1.459459  
Page B = 0.7702703  
Page C = 0.7702703

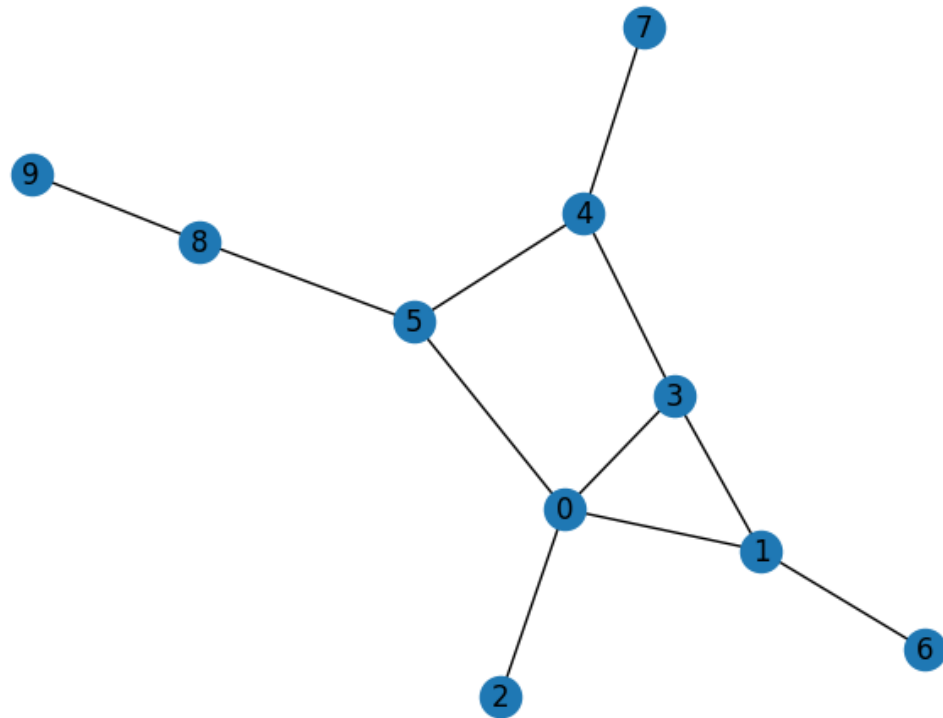


```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

# Create a graph
G = nx.Graph()
# add node/edge pairs
G.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)])

# Calculate pagerank
nx.pagerank(G, alpha=0.1)
```

```
{0: 0.10949439969135805,
1: 0.10541496772119344,
2: 0.09273734992283952,
3: 0.09978650527263376,
5: 0.10141741370884774,
6: 0.09351382767489713,
4: 0.10606037911522635,
7: 0.09353532767489713,
8: 0.1028950931069959,
9: 0.09514473611111113}
```



- Estimating Web Traffic
  - On analyzing the statistics, it was found that there are some sites that have a very high usage, but low PageRank.
  - Ex: Links to pirated software
- PageRank as Backlink Predictor
  - The goal is to try to crawl the pages in as close to the optimal order as possible i.e., in the order of their rank.
  - PageRank is a better predictor than citation counting
- User Navigation: The PageRank Proxy
  - The user receives some information about the link before they click on it
  - This proxy can help users decide which links are more likely to be interesting



네트워크 과학연구실  
NETWORK SCIENCE LAB



가톨릭대학교  
THE CATHOLIC UNIVERSITY OF KOREA

