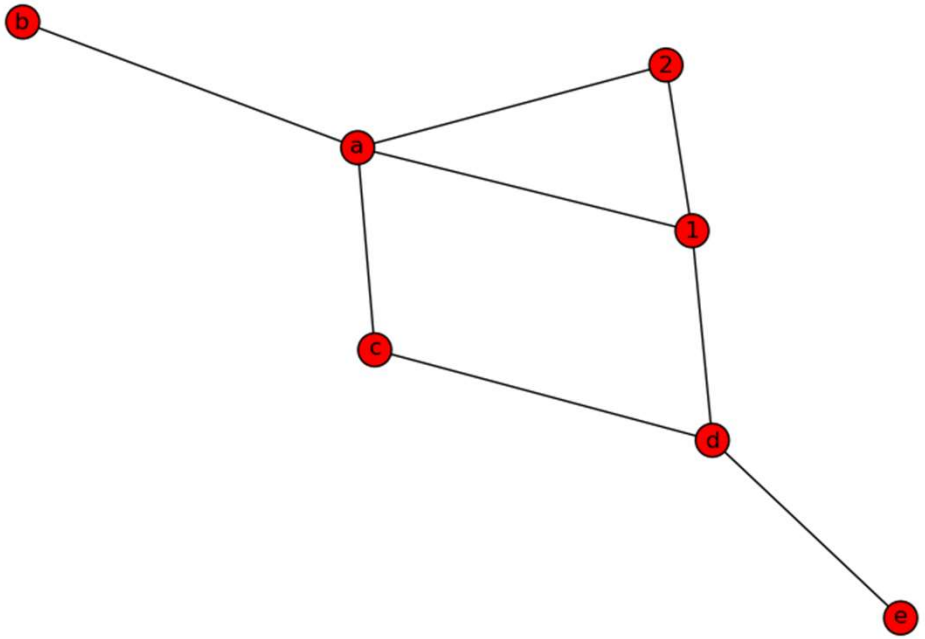
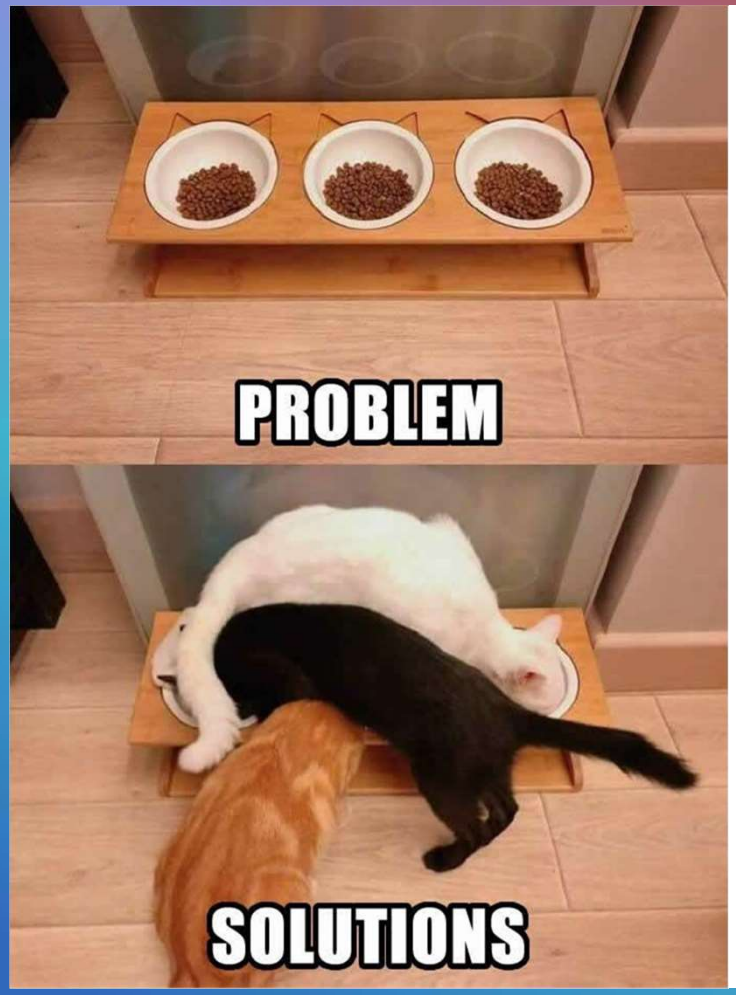
# Working with Trees and Networks

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

# Why Graph theory?

- Graph theory is an incredibly potent data science tool that allows you to visualize and understand complex interactions.






# I will never use Graphs...

# Graph applications

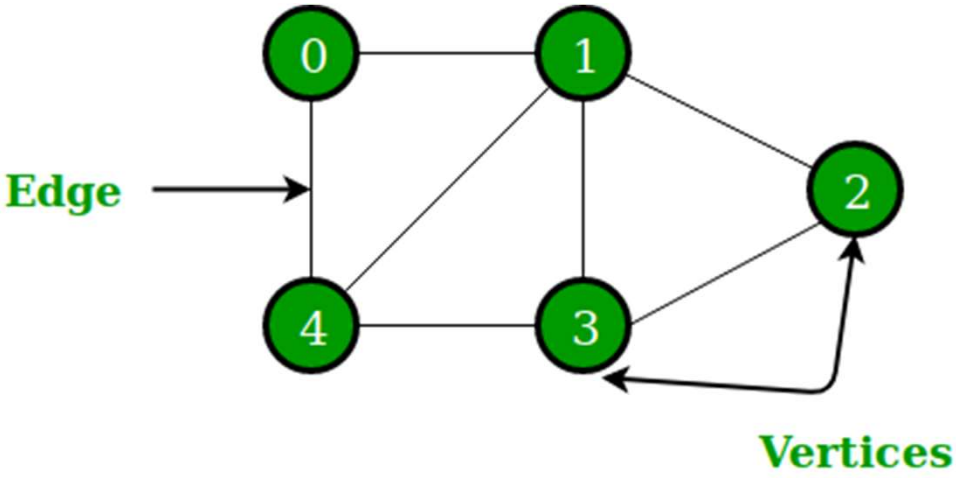
- In **Computer science** graphs are used to represent the flow of computation.
- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.

# Graph applications

- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

# What is a graph?

- A graph is a non-linear data structure, which consists of vertices (or nodes) connected by edges (or arcs) where edges may be directed or undirected.



# Requirements

- pip install networkx

- Before use:
- import networkx as nx

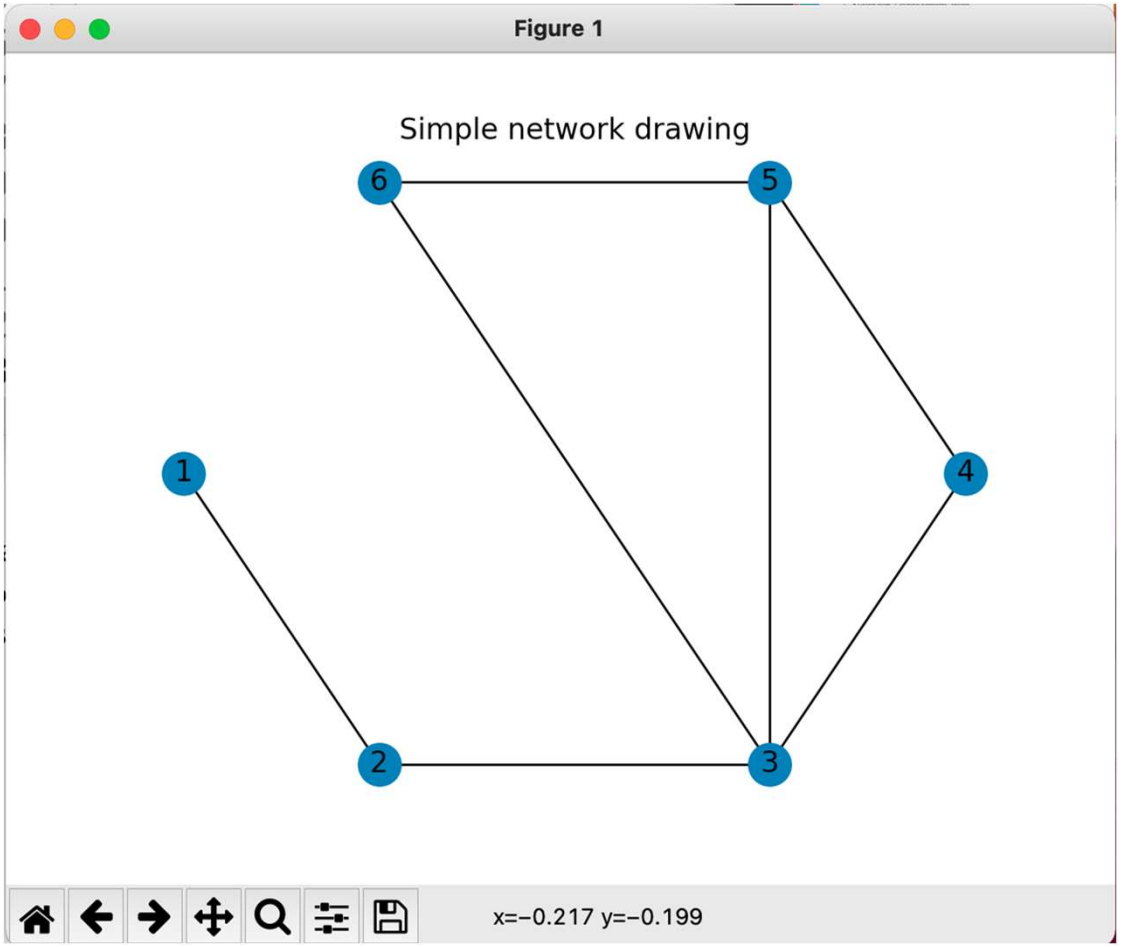
# Create an empty Graph

- import networkx as nx
- G=nx.Graph()
- print(G.nodes()) # []
- print(G.edges()) # []
- print(type(G.nodes())) #<class 'networkx.classes.reportviews.NodeView'>
- print(type(G.edges())) #<class 'networkx.classes.reportviews.EdgeView'>

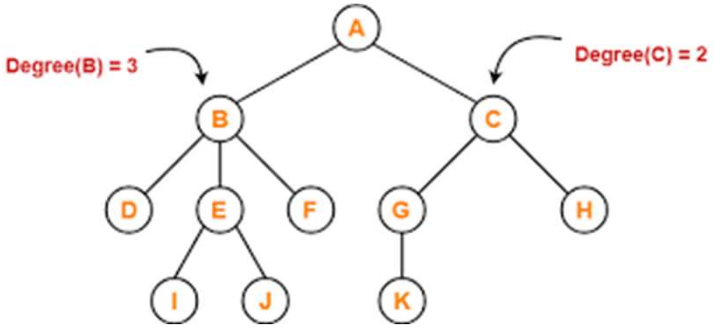
# Creating a Graph

- import networkx as nx
- G = nx.Graph()
- G.add_node(1)
- G.add_node(2)
- G.add_nodes_from([3, 4, 5, 6])
- G.add_edge(1, 2)
- G.add_edges_from([(2, 3), (3, 4), (3, 5), (3, 6), (4, 5), (5, 6)])
- print(G.nodes) # [1, 2, 3, 4, 5, 6]
- print(G.edges) # [(1, 2), (2, 3), (3, 4), (3, 5), (3, 6), (4, 5), (5, 6)]

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


# Graph from "Creating networks" recipe
G = nx.Graph()
G.add_nodes_from(range(1, 7))
G.add_edges_from([
    (1, 2), (2, 3), (3, 4), (3, 5),
    (3, 6), (4, 5), (5, 6)
])
fig, ax = plt.subplots()
layout = nx.shell_layout(G)
nx.draw(G, ax=ax, pos=layout, with_labels=True)
ax.set_title("Simple network drawing")
plt.show()
```



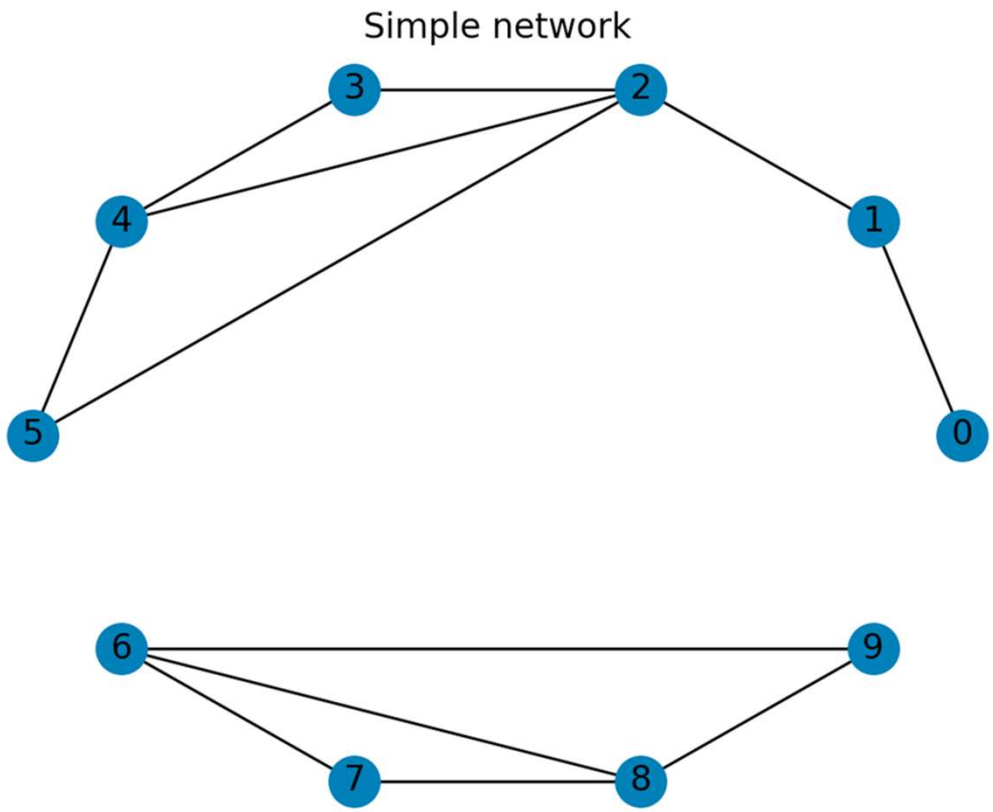
# Degree of a node




- The degree of a node is the number od edges that start (or end) at that node.
- Higher degree shows that the node is better connected to the rest of the network.

- **Let's compute that information.**

```
import networkx as nx
import matplotlib.pyplot as plt
G = nx.Graph()
G.add_nodes_from(range(10))
G.add_edges_from([
    (0, 1), (1, 2), (2, 3), (2, 4),
    (2, 5), (3, 4), (4, 5), (6, 7),
    (6, 8), (6, 9), (7, 8), (8, 9)
])
fig, ax = plt.subplots()
nx.draw_circular(G, ax=ax, with_labels=True)
ax.set_title("Simple network")
plt.show()
```



# Get general information – deprecated version >0.99

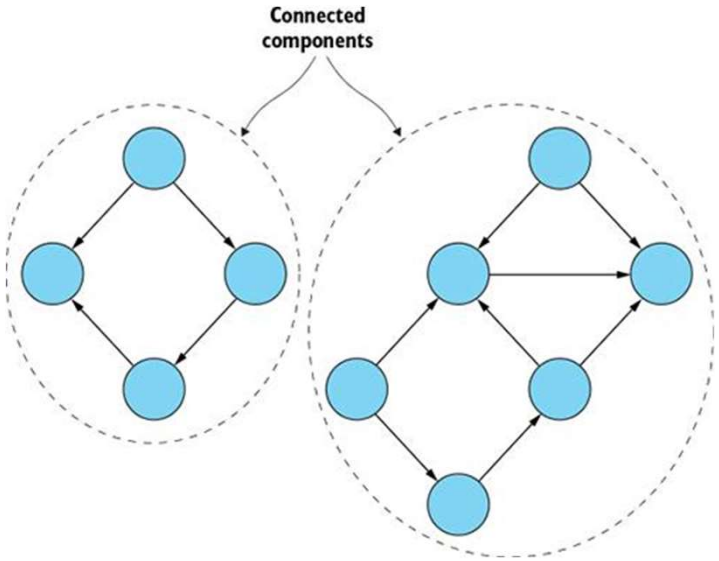
+ print(nx.info(G))
+ # Name:
+ # Type: Graph
+ # Number of nodes: 10
+ # Number of edges: 12
+ # Average degree: 2.4000

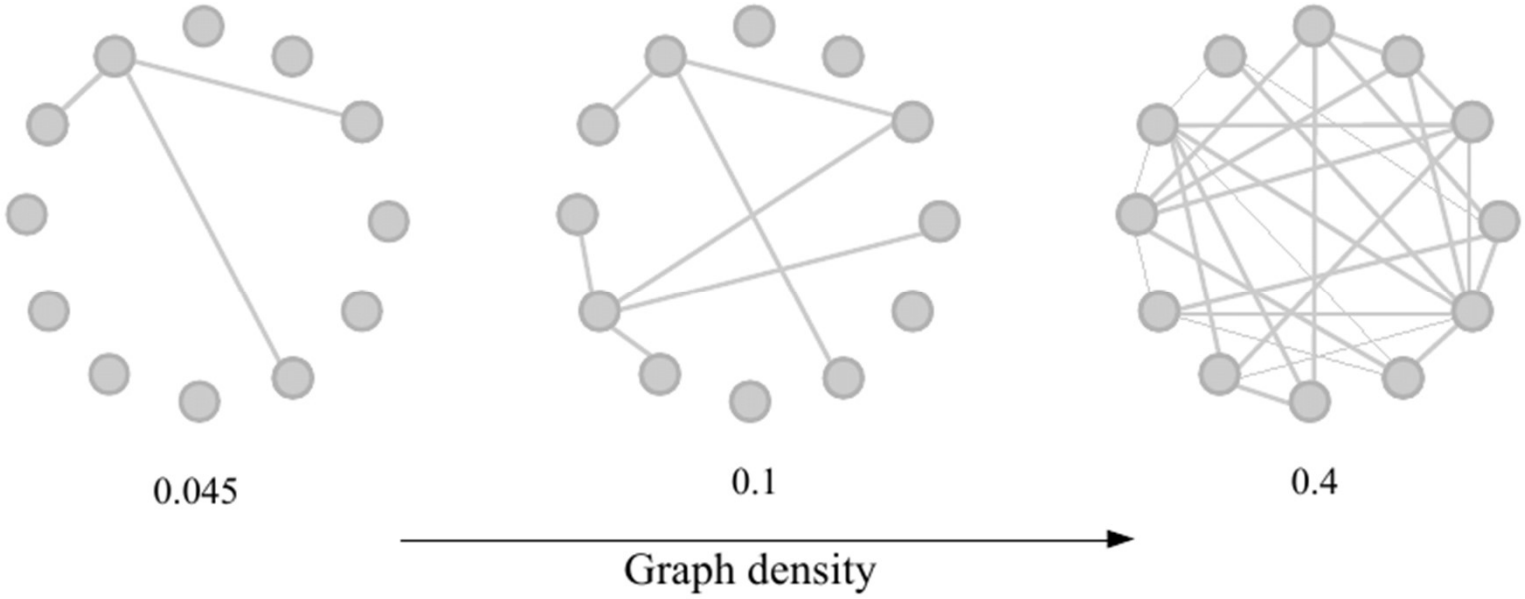
# Get the degree of several nodes

```
for i in [0, 2, 7]:
    degree = G.degree[i]
    print(f"Degree of {i}: {degree}")
# Degree of 0: 1
# Degree of 2: 4
# Degree of 7: 2
```

# Get connected components

- components = list(nx.connected_components(G))
- print(components)
- [{0, 1, 2, 3, 4, 5}, {8, 9, 6, 7}]






# Density

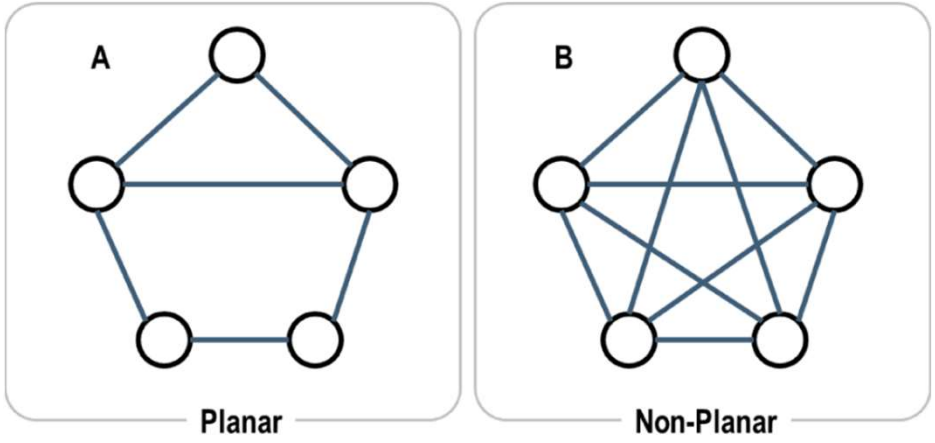
- Represents the proportion of edges meeting the node to the total number of possible edges at the node.

# Compute density

+ density = nx.density(G)
+ print("Density", density)
+ # Density 0.26666666666666666

# Planar and Non-Planar Graphs

- Graph A is planar since no link is overlapping with another.
- Graph B is non-planar since many links are overlapping. Also, the links of graph B cannot be reconfigured in a manner that would make it planar.

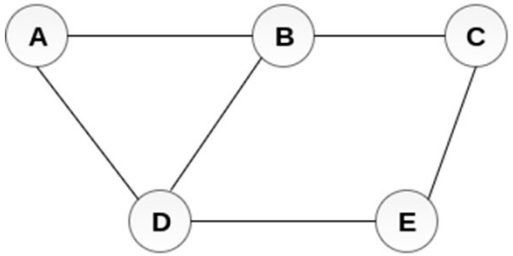


# Compute Planarity

+ is_planar, _ = nx.check_planarity(G)
+ print("Is planar", is_planar)
+ # Is planar True

# Adjacency Matrix

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

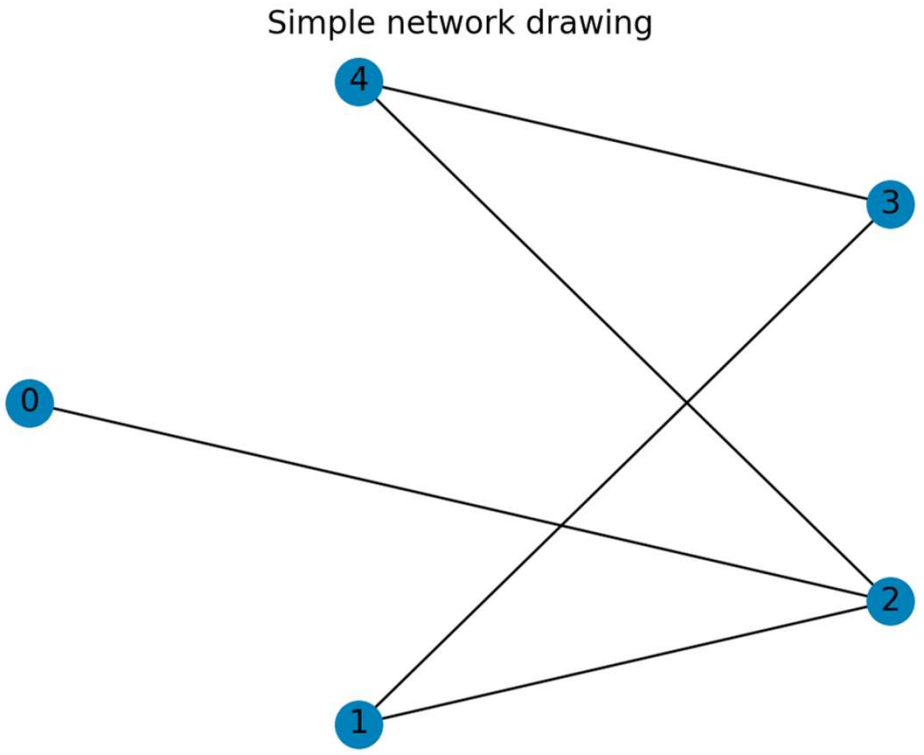



Undirected Graph

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 | 0 |
| B | 1 | 0 | 1 | 1 | 0 |
| C | 0 | 1 | 0 | 0 | 1 |
| D | 1 | 1 | 0 | 0 | 1 |
| E | 0 | 0 | 1 | 1 | 0 |

Adjacency Matrix

```
+ import numpy as np
+ import networkx as nx
+ import matplotlib.pyplot as plt
+ G = nx.dense_gnm_random_graph(5, 5, seed=12345)
+ fig, ax = plt.subplots()
+ layout = nx.shell_layout(G)
+ nx.draw(G, ax=ax, pos=layout, with_labels=True)
+ ax.set_title("Simple network drawing")
+ plt.show()
```



# Compute adjacency matrix

```
matrix = nx.adjacency_matrix(G).todense()
Matrix = nx.to_numpy_array(G)
print(matrix)
# [[0 0 1 0 0]
#  [0 0 1 1 0]
#  [1 1 0 0 1]
#  [0 1 0 0 1]
#  [0 0 1 1 0]]
```

# Simple Graphs

- A **simple graph** is a notation that is used to represent the connection between pairs of objects. It consists of:
- A set of **vertices**, which are also known as nodes. We denote a set of vertices with a *V*.
- A set of **edges**, which are the links that connect the vertices. We denote the edges set with an *E*.

# Weighted Graphs

- A **weighted graph** refers to a simple graph that has weighted edges. These weighted edges can be used to compute the shortest path. It consists of:
- A set of vertices *V*.
- A set of edges *E*.
- And a number *w* (with *w* meaning weight) that's assigned to each edge. Weights might represent things such as costs, lengths, or capacities.
- In a simple graph, the assumption is that the sum of all the weights is equal to 1.

# Graph Types

- n addition to simple and weighted descriptions, there are two types of graphs:
- A **directed graph**, where edges have direction (meaning that edges with arrows connect one vertex to another).
- An **undirected graph**, where edges have no direction (meaning arrowless connections). It's basically the same as a directed graph but has bi-directional connections between nodes.

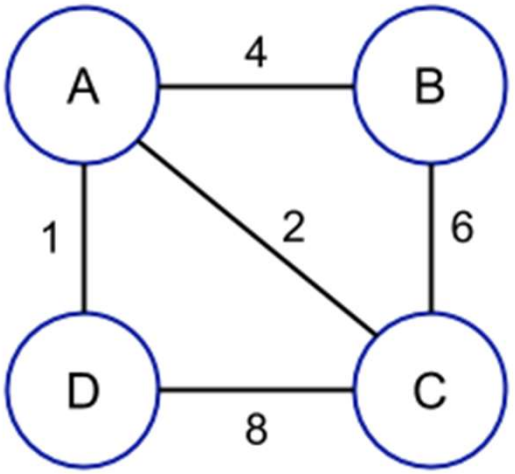



**Figure:** Weighted Graph
(also weighted undirected graph)

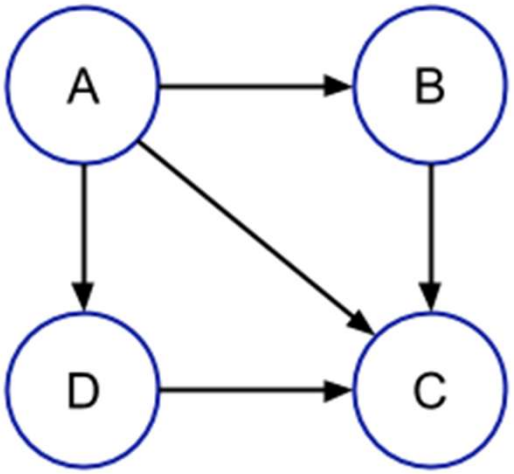
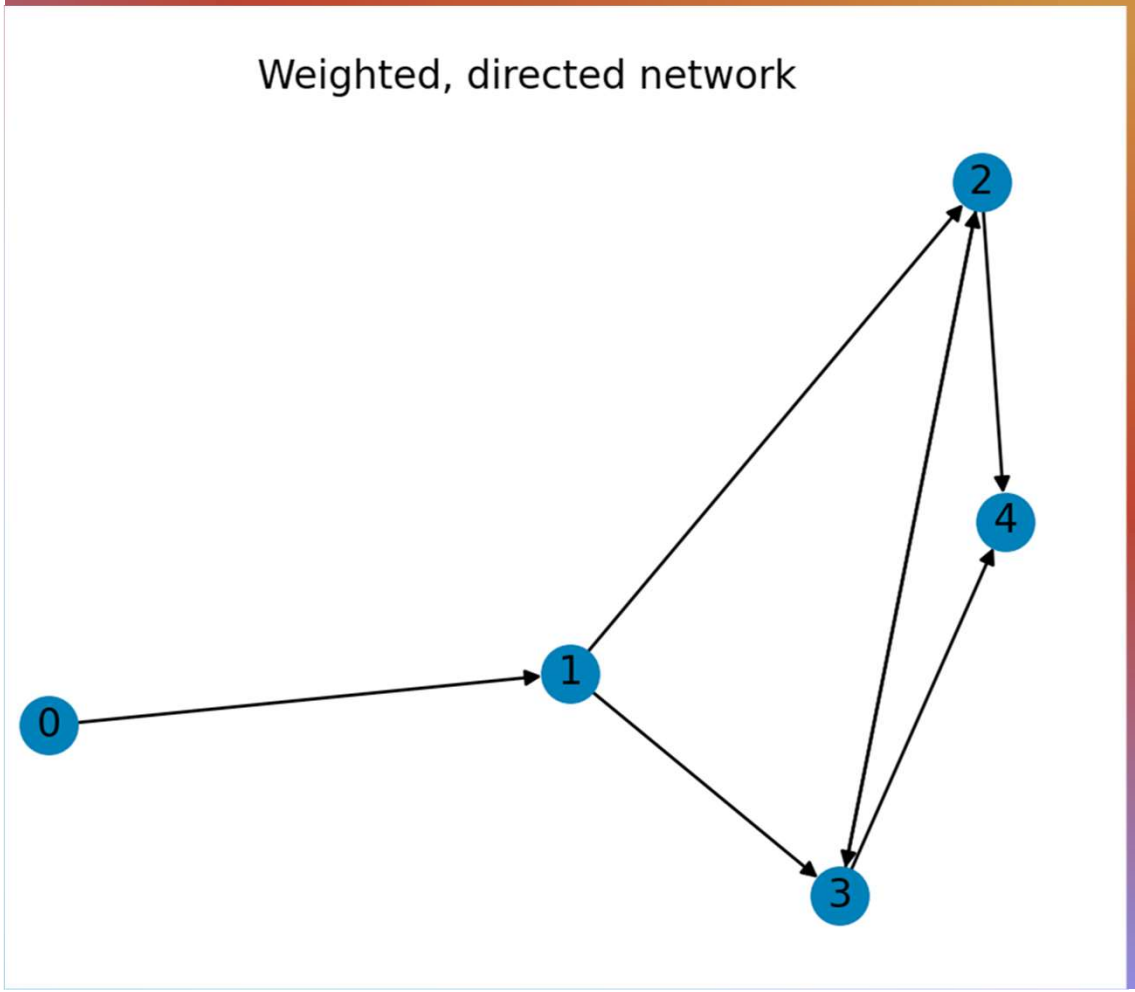


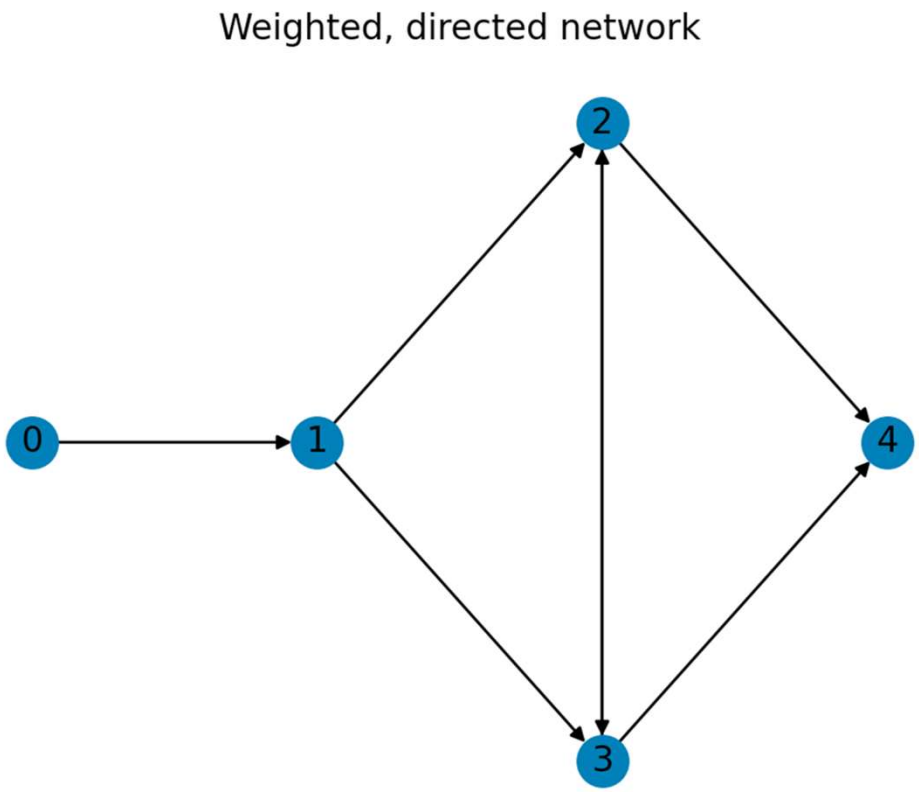

**Figure:** Unweighted Graph
(also unweighted directed graph)

# Compute Weighted and directed Network

```
+ import numpy as np
+ import networkx as nx
+ import matplotlib.pyplot as plt
+ G = nx.DiGraph()
+ G.add_nodes_from(range(5))
+ G.add_edge(0, 1, weight=1.0)
+ G.add_weighted_edges_from([
+     (1, 2, 0.5), (1, 3, 2.0), (2, 3, 0.3), (3, 2, 0.3),
+     (2, 4, 1.2), (3, 4, 0.8)
+ ])
+ fig, ax = plt.subplots()
+ nx.draw(G, ax=ax, with_labels=True)
+ ax.set_title("Weighted, directed network")
+ plt.show()
```

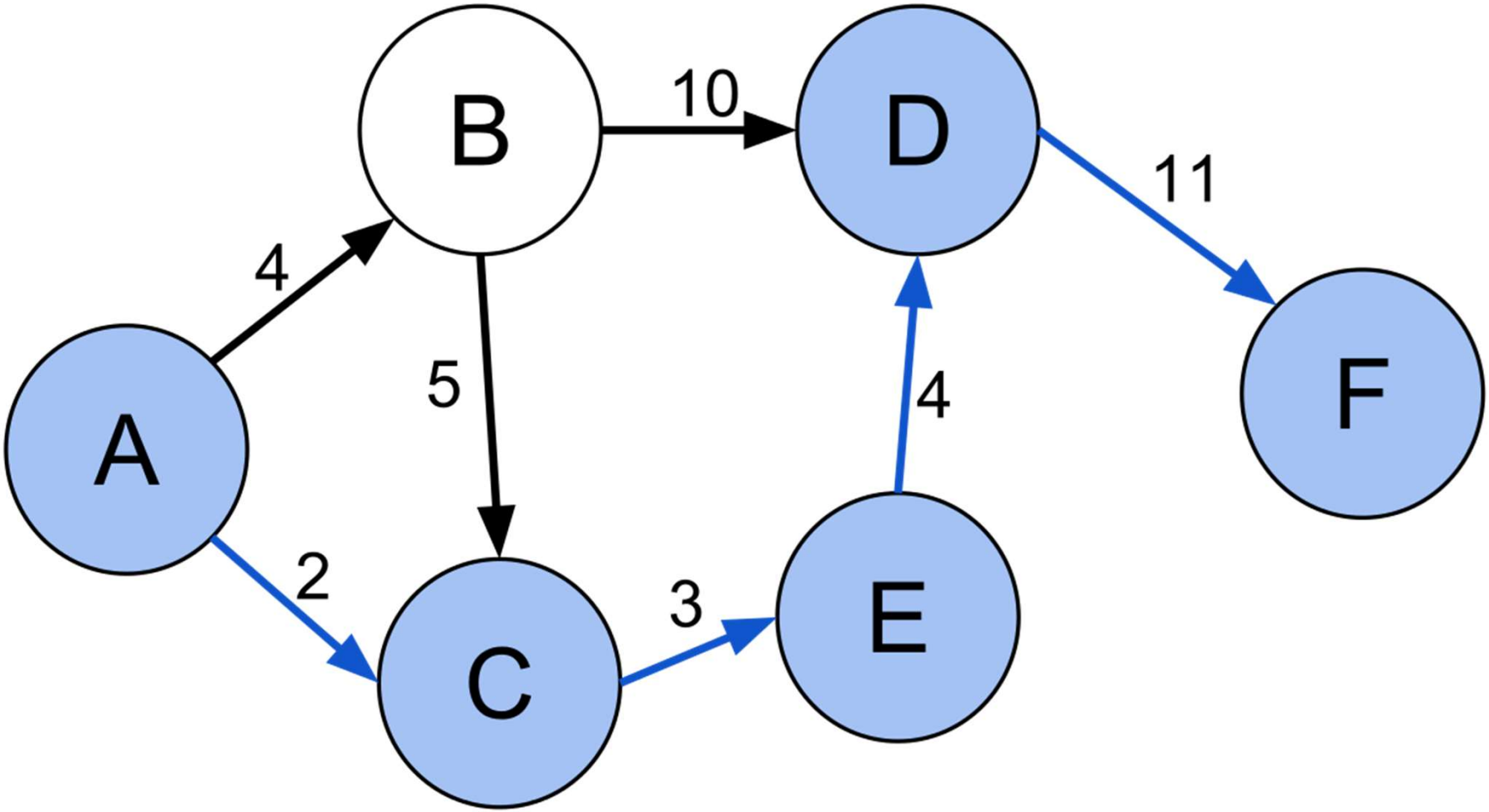


```
+ import numpy as np
+ import networkx as nx
+ import matplotlib.pyplot as plt

+ G = nx.DiGraph()
+ G.add_nodes_from(range(5))
+ G.add_edge(0, 1, weight=1.0)
+ G.add_weighted_edges_from([
+     (1, 2, 0.5), (1, 3, 2.0), (2, 3, 0.3), (3, 2, 0.3),
+     (2, 4, 1.2), (3, 4, 0.8)
+ ])
+ fig, ax = plt.subplots()
+ pos = {0: (-1, 0), 1: (0, 0), 2: (1, 1), 3: (1, -1), 4: (2, 0)}
+ nx.draw(G, ax=ax, pos=pos, with_labels=True)
+ ax.set_title("Weighted, directed network")
+ plt.show()
```

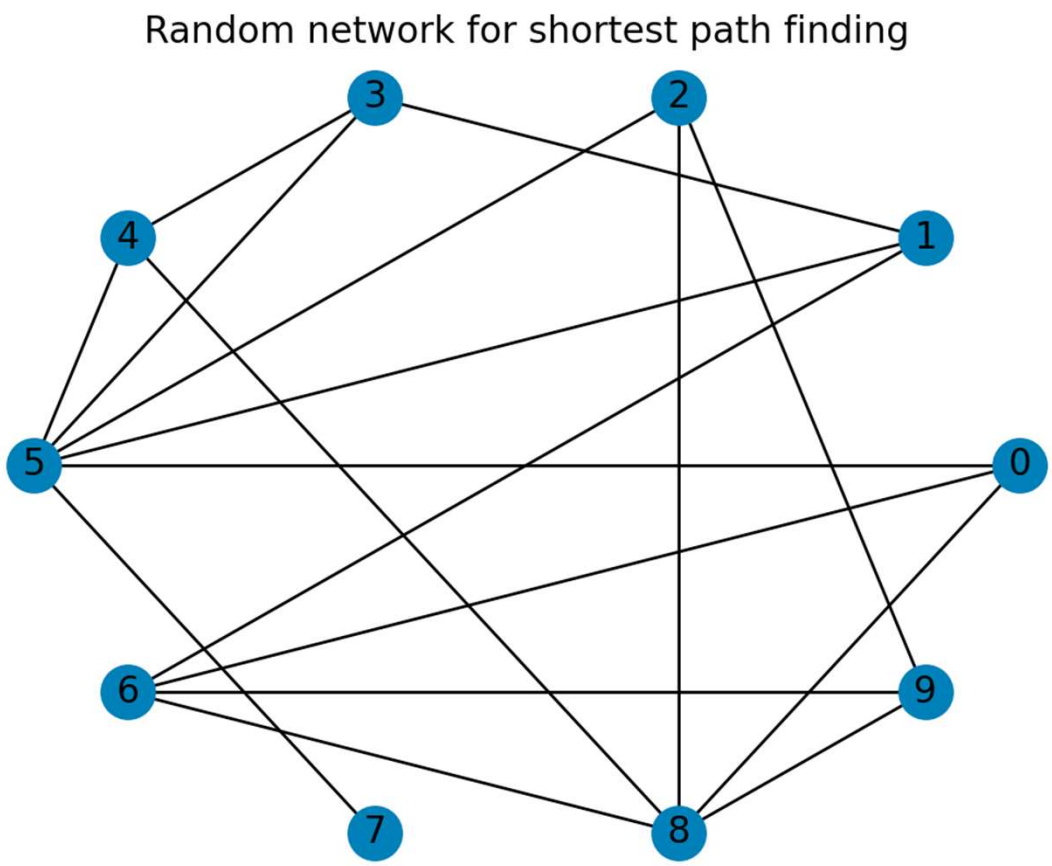


# Finding the shortest paths in a network

- In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
- The problem of finding the shortest path between two intersections on a road map may be modeled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.


A
B
C
D
E
F
4
10
11
5
4
2
3

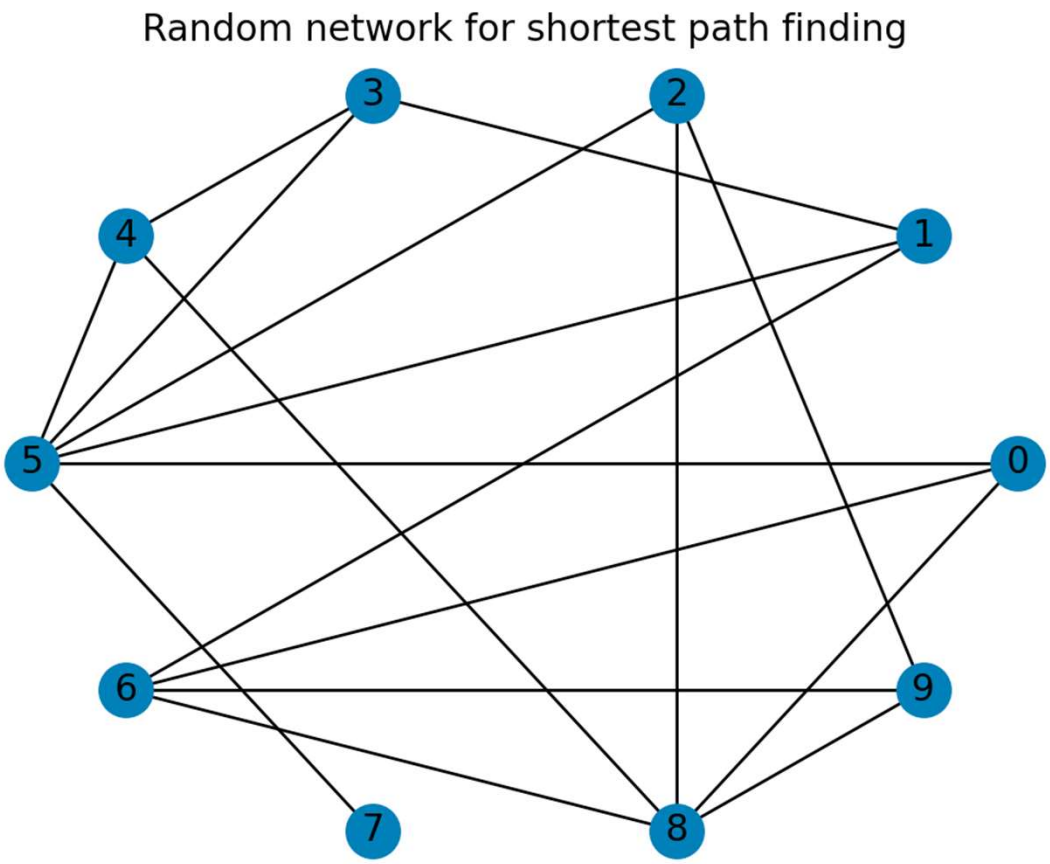
```
+ import networkx as nx
+ import matplotlib.pyplot as plt

+ from numpy.random import default_rng
+ rng = default_rng(12345)
+ G = nx.gnm_random_graph(10, 17, seed=12345)
+ fig, ax = plt.subplots()
+ nx.draw_circular(G, ax=ax, with_labels=True)
+ ax.set_title("Random network for shortest path finding")

+ plt.show()
```



# Shortest path from 7 to 9

- 7 and 9 are not connected.
- So, how can we calculate the shortest path from7 to 9 since they are several paths available?

Note: paths can have different cost



# Add weight to the deges

- We need to add a weight to each of the edges so that routes are preferable to others:

```
for u, v in G.edges:
    G.edges[u, v]["weight"] = rng.integers(5, 15)
```

# Compute the shortest path

- path = nx.shortest_path(G, 7, 9, weight="weight")
- print(path)
- # [7, 5, 2, 9]

# Compute the cost of the shortest path

+ length = nx.shortest_path_length(G, 7, 9, weight="weight")
+ print("Length", length)
+ # Length 32

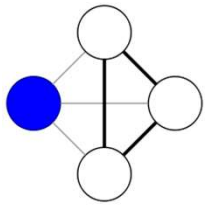
# networkx.algorithms.shortest_paths.generic.shortest_path

- shortest_path(*G*, *source=None*, *target=None*, *weight=None*, *method='dijkstra'*)

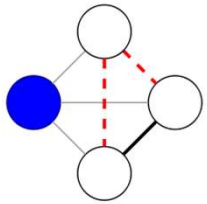
- *There are several methods for finding the shortest path.*

# Clustering coefficient

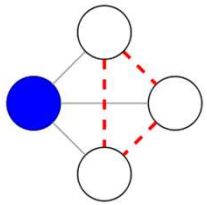
- In graph theory, a clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. Evidence suggests that in most real-world networks, and in particular social networks, nodes tend to create tightly knit groups characterized by a relatively high density of ties; this likelihood tends to be greater than the average probability of a tie randomly established between two nodes (Holland and Leinhardt, 1971; Watts and Strogatz, 1998).



c = 1



c = 1/3



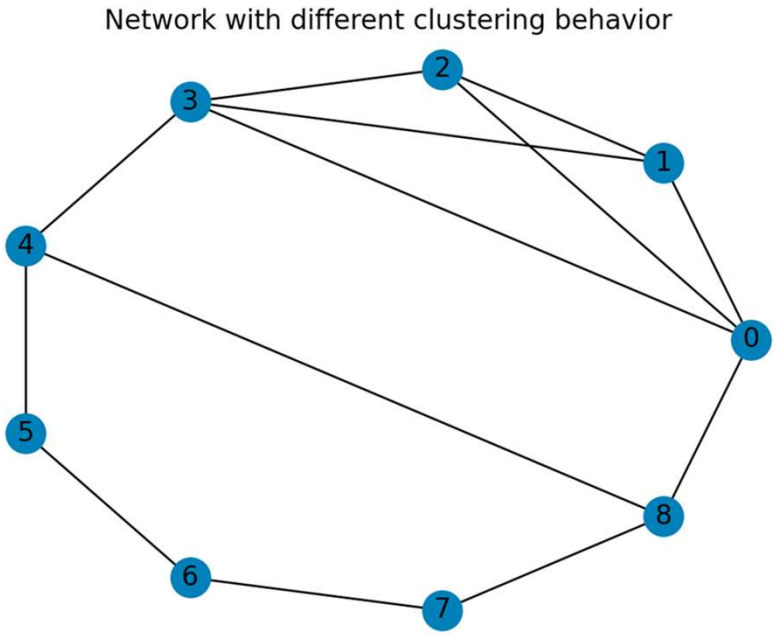
c = 0

- Example local clustering coefficient on an undirected graph.
- The local clustering coefficient of the blue node is computed as the proportion of connections among its neighbours which are actually realised compared with the number of all possible connections.
- In the figure, the blue node has three neighbours, which can have a maximum of 3 connections among them.
- In the top part of the figure all three possible connections are realised (thick black segments), giving a local clustering coefficient of 1.
- In the middle part of the figure only one connection is realised (thick black line) and 2 connections are missing (dotted red lines), giving a local cluster coefficient of 1/3.
- Finally, none of the possible connections among the neighbours of the blue node are realised, producing a local clustering coefficient value of 0.

# Compute clustering coefficients

- 1 – Draw a random network
- 2 – Compute the coefficients for specified nodes
- 3 – Compute the average coefficients

# Draw the Graph




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

G = nx.Graph()¢
complete_part = nx.complete_graph(4)
cycle_part = nx.cycle_graph(range(4, 9))
G.update(complete_part)
G.update(cycle_part)
G.add_edges_from([(0, 8), (3, 4)])
fig, ax = plt.subplots()
nx.draw_circular(G, ax=ax, with_labels=True)
ax.set_title("Network with different clustering behavior")
plt.show()
```

# 2 – Compute the coefficients for specified nodes

```
+ cluster_coeffs = nx.clustering(G)

+ for i in [0, 2, 6]:
+     print(f"Node {i}, clustering {cluster_coeffs[i]}")
+ # Node 0, clustering 0.5
+ # Node 2, clustering 1.0
+ # Node 6, clustering 0
```

# 3 – Compute the average coefficients

+ av_clustering = nx.average_clustering(G)
+ print(av_clustering)
+ # 0.3333333333333333

# Minimal spanning trees

- A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
- That is, it is a spanning tree whose sum of edge weights is as small as possible.
- Any edge-weighted undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of the minimum spanning trees for its connected components.

```
+ import networkx as nx
+ import matplotlib.pyplot as plt
+ G = nx.gnm_random_graph(15, 22, seed=12345)
+ fig, ax = plt.subplots()
+ pos = nx.circular_layout(G)
+ nx.draw(G, pos=pos, ax=ax, with_labels=True)
+ ax.set_title("Network with minimum spanning tree overlaid")
+ min_span_tree = nx.minimum_spanning_tree(G)
+ print(list(min_span_tree.edges))
+ # [(0, 13), (0, 7), (0, 5), (1, 13), (1, 11),
+ #   (2, 5), (2, 9), (2, 8), (2, 3), (2, 12),
+ #   (3, 4), (4, 6), (5, 14), (8, 10)]
+ nx.draw_networkx_edges(min_span_tree, pos=pos, ax=ax, width=1.5,
  edge_color="r")
+ plt.show()
```

# Result

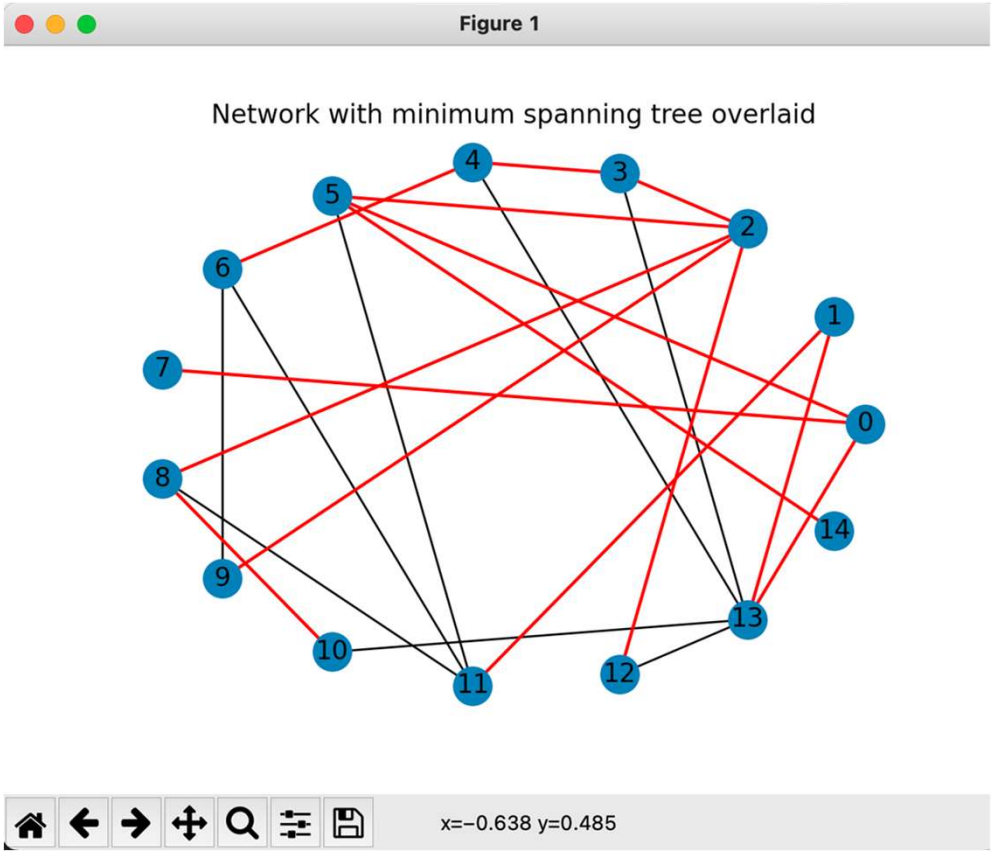


Figure 1
Network with minimum spanning tree overlaid
4
3
5
2
6
1
7
0
8
14
9
13
10
12
11
x=−0.638 y=0.485