Assignment – 1

REPORT

Social Network Analytics

Name: Aditya Panditrao

Registration No.: 22MCB0032

Course Name: Social Network Analytics

Course Code: MCSE618L

Table of Contents

Introduction

Graph -

The Social Network graph helps you to visualize the relationships between the selected entity and all entities that the selected entity is linked to. Using this unique graph, you get another way to see "who knows who".

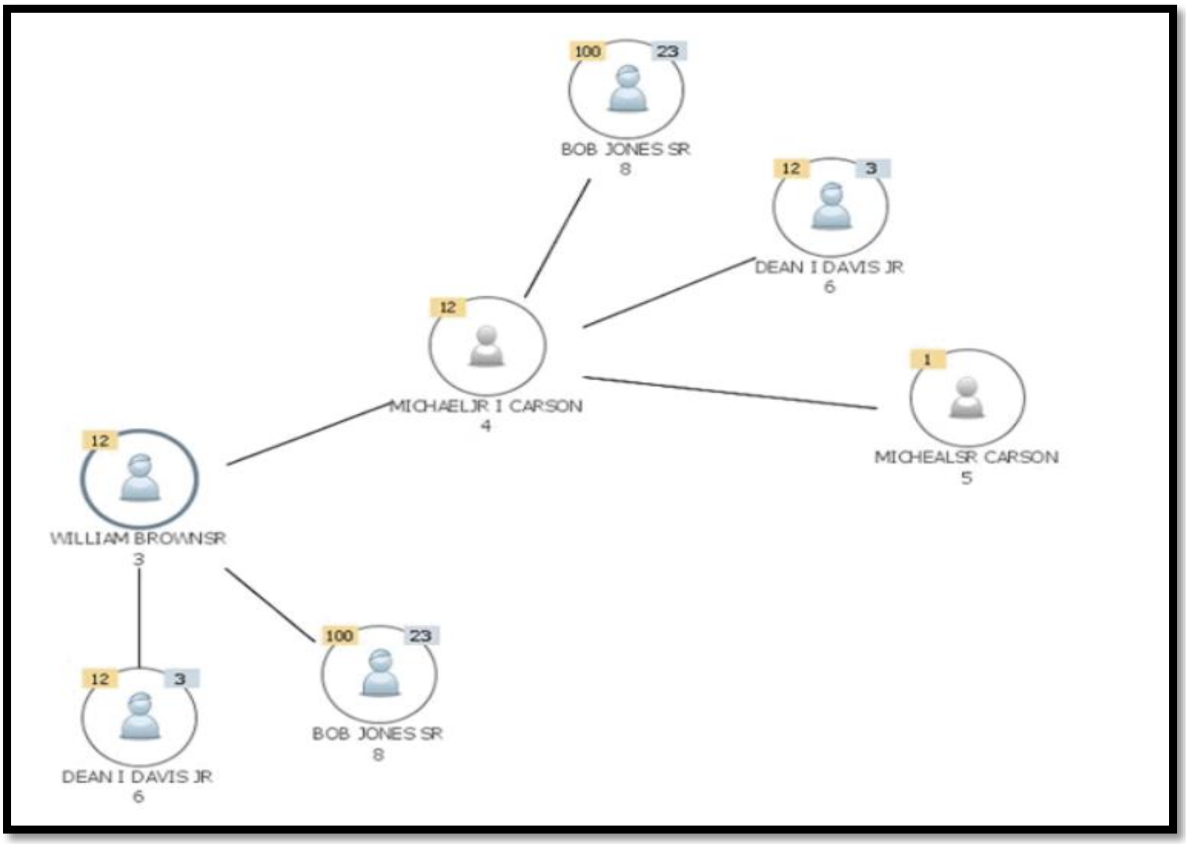


Fig. Graph

The Social Network graph shows:

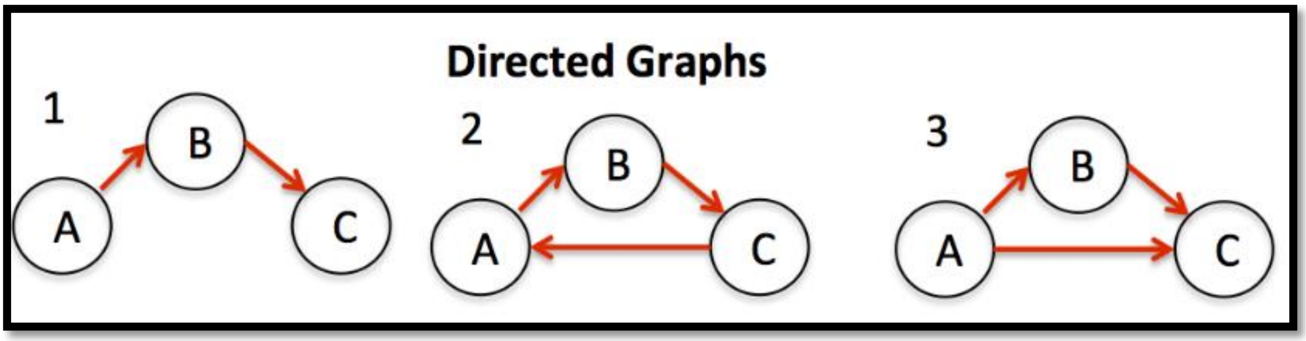
- **Entity-to-entity links:** You see all the entities related to the main (hub) entity. However, the attributes that link the entities do not display on the graph but are accessible by using the Attribute Explorer in combination with the graph.

- **Relationship clusters:** The Social Network graph is unique in that it displays the related entities in groups or clusters. This graph can help you see all the relationship clusters a particular entity belongs to and look for patterns in among the clusters and relationships.

Types of Graph-

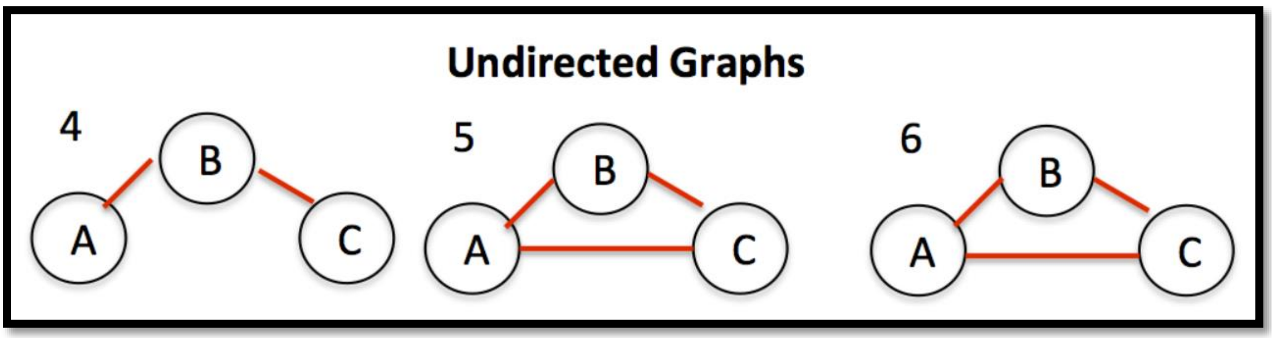
Directed Graph:

The directed graph is also known as the digraph, which is a collection of set of vertices edges. Here the edges will be directed edges, and each edge will be connected with order pair of vertices. In a graph, the directed edge or arrow points from the first/ original vertex to the second/ destination vertex in the pair. In the V-vertex graph, we will represent vertices by the name 0 through V-1. If there are two vertices, x and y, connected with an edge (x, y) in a directed graph, it is not necessary that the edge (y, a) is also available in that graph.



Undirected Graph:

The undirected graph is also referred to as the bidirectional. It is a set of objects (also called vertices or nodes), which are connected together. Here the edges will be bidirectional. The two nodes are connected with a line, and this line is known as an edge. The undirected graph will be represented as G = (N, E). Where N is used to show the set of edges and E is used to show the set of edges, which are unordered pairs of elements N. The main difference between the directed and undirected graph is that the directed graph uses the arrow or directed edge to connect the two nodes. The arrow points from the original vertex to destination vertex in the directed graph. While in the undirected graph, the two nodes are connected with the two direction edges.



Weighted graph -

A **weighted graph** is a special type of graph in which the edges are assigned some weights which represent cost, distance and many other relative measuring units.

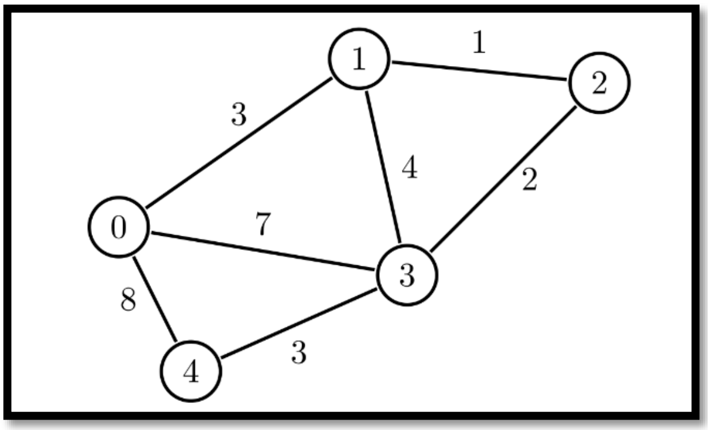


Fig. **Weighted graph**

Unweighted graph -

An unweighted graph **is a graph in which the edges do not have weights or costs associated with them. Instead, they simply represent the presence of a connection between two vertices.**

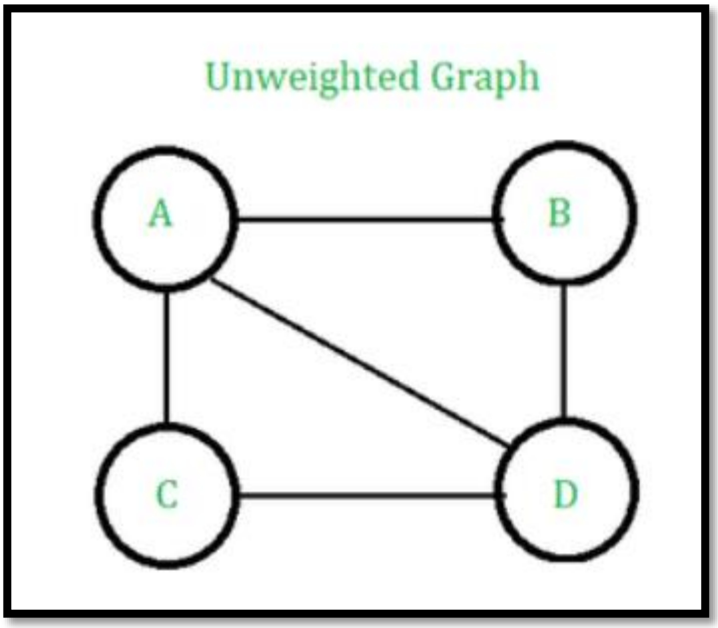


Fig. **Unweighted graph**

Used Libraries –

1) NetworkX-

NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. It is used to study large complex networks represented in form of graphs with nodes and edges. Using networkx we can load and store complex networks. We can generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms and draw networks.

Installation of the package:

```
pip install networkx
```

2) Matplotlib-

Matplotlib is easy to use and an amazing visualizing library in Python. It is built on NumPy arrays and designed to work with the broader SciPy stack and consists of several plots like line, bar, scatter, histogram, etc.

Installation of the package:

```
import matplotlib.pyplot as plt
```

3) NumPy-

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, Fourier transform, and matrices. NumPy stands for Numerical Python.

Installation of the package:

```
import numpy as np
```

4) Tabulate -

The **tabulate()** method is a method present in the **tabulate** module which creates a text-based table output inside the python program using any given inputs. It can be installed using the below command.

Installation of the package:

```
pip install tabulate
```

Code –

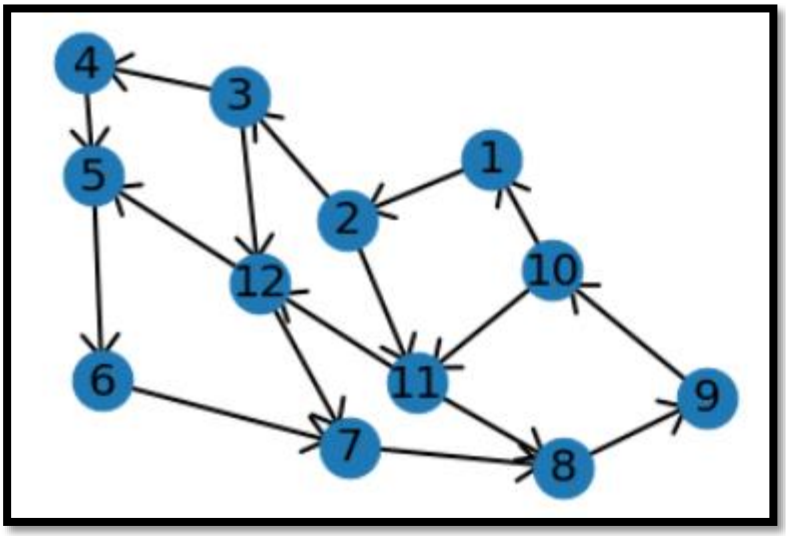
Part –1

1) **Creating a Directed Graph**

```
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
import csv

# Open the CSV file and read the edges
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), float(row[2])) for row in reader]


    df=pd.read_csv('edges.csv')
    print(df)

# Create an empty directed graph
G = nx.DiGraph()

# Add the edges to the graph with weights
G.add_weighted_edges_from(edges)

# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes
nx.draw_networkx_nodes(G, pos, node_size=700)
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2],
                       arrowsize=20, arrowstyle='-
>, head_width=0.4, head_length=0.5')
nx.draw_networkx_labels(G, pos, font_size=20, font_family='sans-serif')
plt.axis('off')
plt.savefig('directed_graph.png')
plt.show()
```

Output -

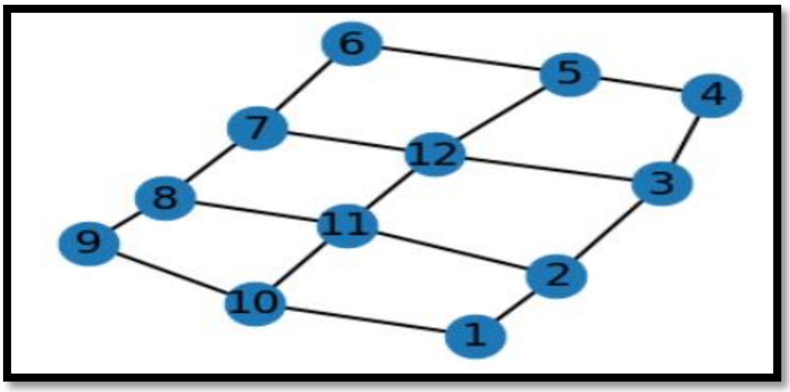
	source	destination	weight
0	1	2	2
1	2	3	1
2	3	4	1
3	4	5	3
4	5	6	1
5	6	7	1
6	7	8	2
7	8	9	4
8	9	10	1
9	10	1	1
10	10	11	1
11	11	12	2
12	2	11	1
13	3	12	1
14	11	8	2
15	12	7	2
16	12	5	1



2) Creating Undirected Graph

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

# Open the CSV file and read the edges with weights
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), float(row[2])) for row in reader]

# Create an empty undirected graph
G = nx.Graph()

# Add the edges to the graph with weights
G.add_weighted_edges_from(edges)

# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes
nx.draw_networkx_nodes(G, pos, node_size=700)
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2])
nx.draw_networkx_labels(G, pos, font_size=20, font_family='sans-serif')
plt.axis('off')
plt.savefig('undirected_graph.png')
plt.show()
```

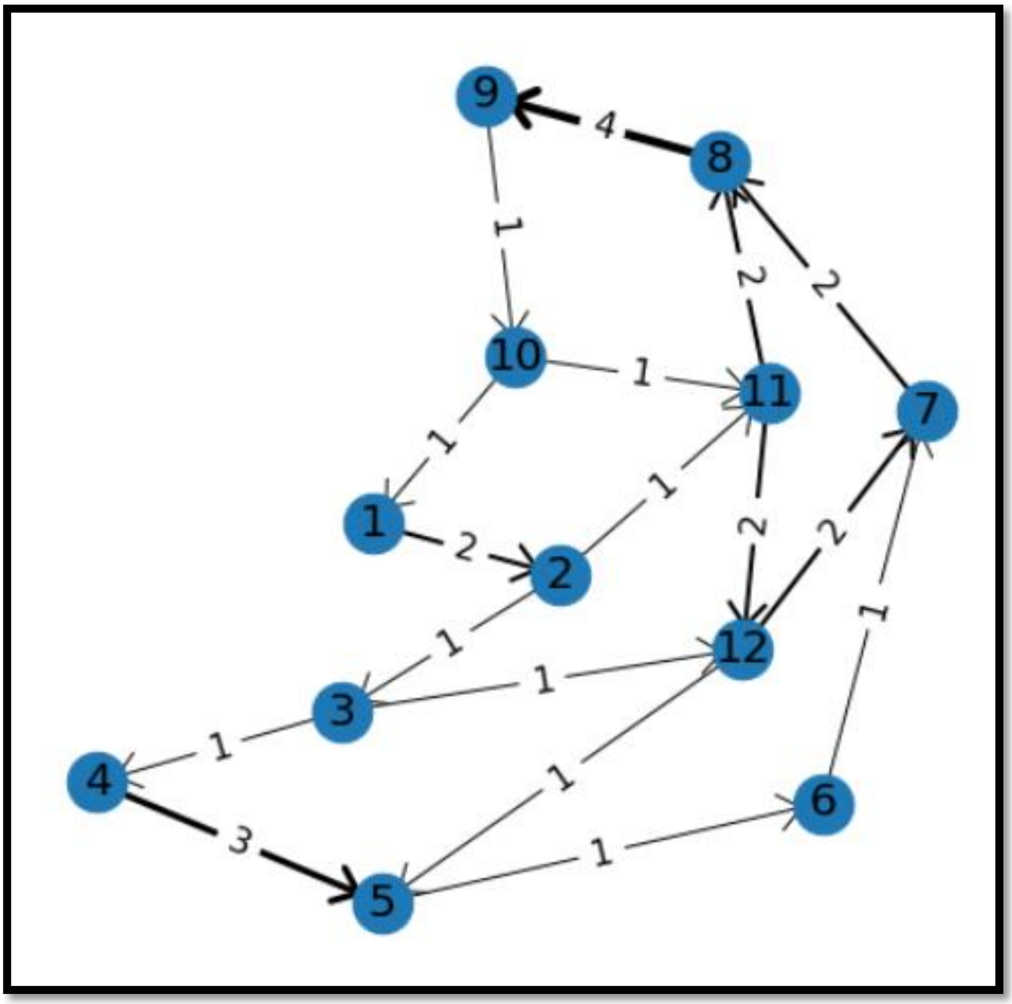
Output -



3) Creating Directed Graph with Weight

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

# Open the CSV file and read the edges
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), int(row[2])) for row in reader]

# Create an empty directed graph
G = nx.DiGraph()

# Add the edges to the graph with weights
G.add_weighted_edges_from(edges)

# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes

# Set the figure size
plt.figure(figsize=(8, 8))

# Draw the nodes
nx.draw_networkx_nodes(G, pos, node_size=700)

# Draw the edges with arrows and weights
edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=edge_widths,
                       arrowsize=20, arrowstyle='-
>, head_width=0.4, head_length=0.5')
edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=20, font_family
='sans-serif')

# Turn off the axis
plt.axis('off')

# Show the plot
plt.show()
```

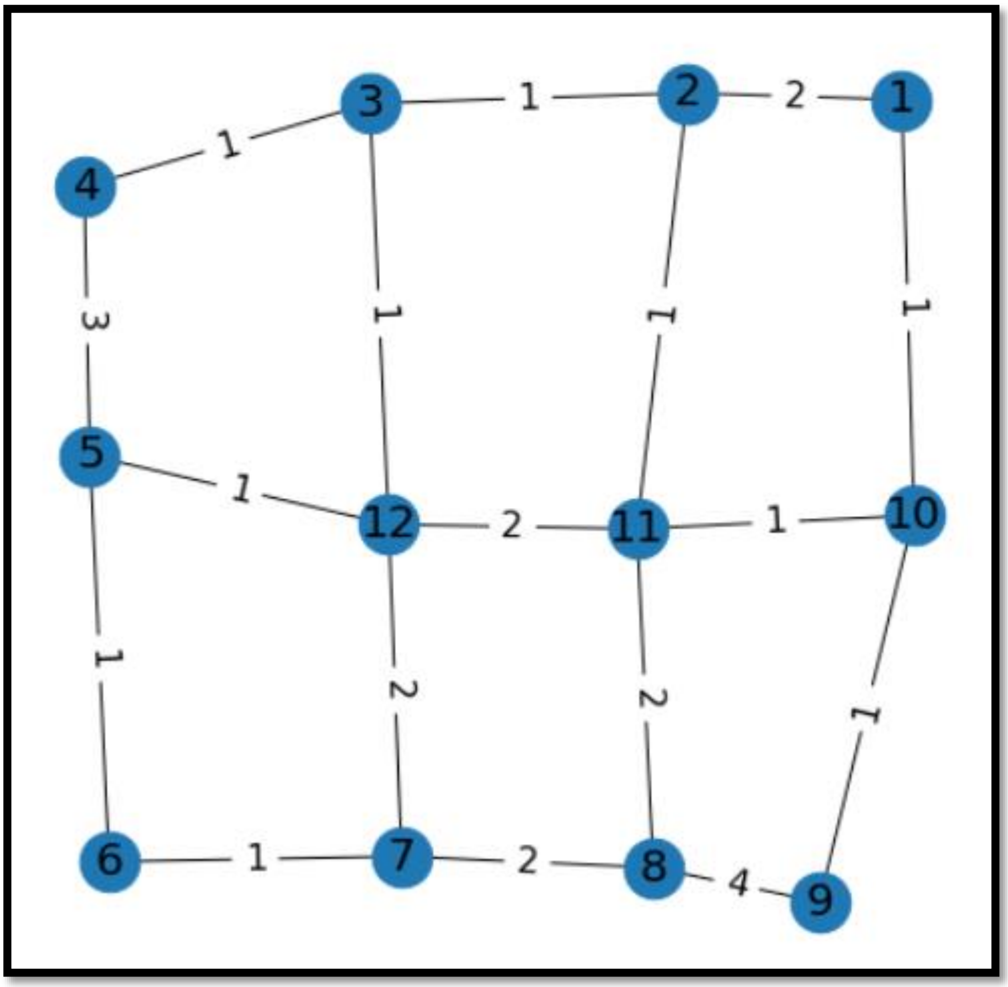
Output -



4) Create Graph with undirected with weight

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

# Open the CSV file and read the edges
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), int(row[2])) for row in reader]

# Create an empty directed graph
G = nx.Graph()

# Add the edges to the graph with weights
G.add_weighted_edges_from(edges)

# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes

# Set the figure size
plt.figure(figsize=(8, 8))

# Draw the nodes
nx.draw_networkx_nodes(G, pos, node_size=700)

# Draw the edges with arrows and weights
edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
nx.draw_networkx_edges(G, pos, edgelist=G.edges())
edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=20, font_family
='sans-serif')

# Turn off the axis
plt.axis('off')

# Show the plot
plt.show()
```

Output -



Part-2

1) Calculation of number of nodes, edges, node with maximum & minimum degree for undirected graph.

```
print('Undirected Graph:')
print(f'Number of nodes: {G.number_of_nodes()}')
print(f'Number of edges: {G.number_of_edges()}')
degrees = dict(G.degree())
max_degree = max(degrees.values())
min_degree = min(degrees.values())
max_degree_nodes = [node for node, degree in degrees.items() if degree == max(degrees.values())]
min_degree_nodes = [node for node, degree in degrees.items() if degree == min(degrees.values())]
print(f'Nodes with maximum degree: {max_degree_nodes}')
print(f'Nodes with minimum degree: {min_degree_nodes}')
print(f'Maximum degree: {max_degree}')
print(f'Minimum degree: {min_degree}')
```

Output -

```
Undirected Graph:
Number of nodes: 12
Number of edges: 17
Nodes with maximum degree: [11, 12]
Nodes with minimum degree: [1, 4, 6, 9]
Maximum degree: 4
Minimum degree: 2
```

2) Calculation of number of nodes, edges, maximum out degree, minimum out degree, Nodes with maximum out degree, Nodes with minimum out degree, maximum in degree, minimum in degree, Nodes with maximum in degree, nodes with minimum in degree for Directed Graph

```
print('Directed Graph:')
print(f'Number of nodes: {G.number_of_nodes()}')
print(f'Number of edges: {G.number_of_edges()}')
out_degrees = dict(G.out_degree())
max_out_degree = max(out_degrees.values())
min_out_degree = min(out_degrees.values())
print(f'Maximum out degree: {max_out_degree}')
print(f'Minimum out degree: {min_out_degree}')
max_out_degree_nodes = [node for node, degree in out_degrees.items() if degree == max(out_degrees.values())]
min_out_degree_nodes = [node for node, degree in out_degrees.items() if degree == min(out_degrees.values())]
print(f'Nodes with maximum out-degree: {max_out_degree_nodes}')
print(f'Nodes with minimum out-degree: {min_out_degree_nodes}')
in_degrees = dict(G.in_degree())
max_in_degree = max(in_degrees.values())
min_in_degree = min(in_degrees.values())
print(f'Maximum in degree: {max_in_degree}')
print(f'Minimum in degree: {min_in_degree}')
max_in_degree_nodes = [node for node, degree in in_degrees.items() if degree == max(in_degrees.values())]
min_in_degree_nodes = [node for node, degree in in_degrees.items() if degree == min(in_degrees.values())]
print(f'Nodes with maximum in-degree: {max_in_degree_nodes}')
print(f'Nodes with minimum in-degree: {min_in_degree_nodes}')
```

Output -

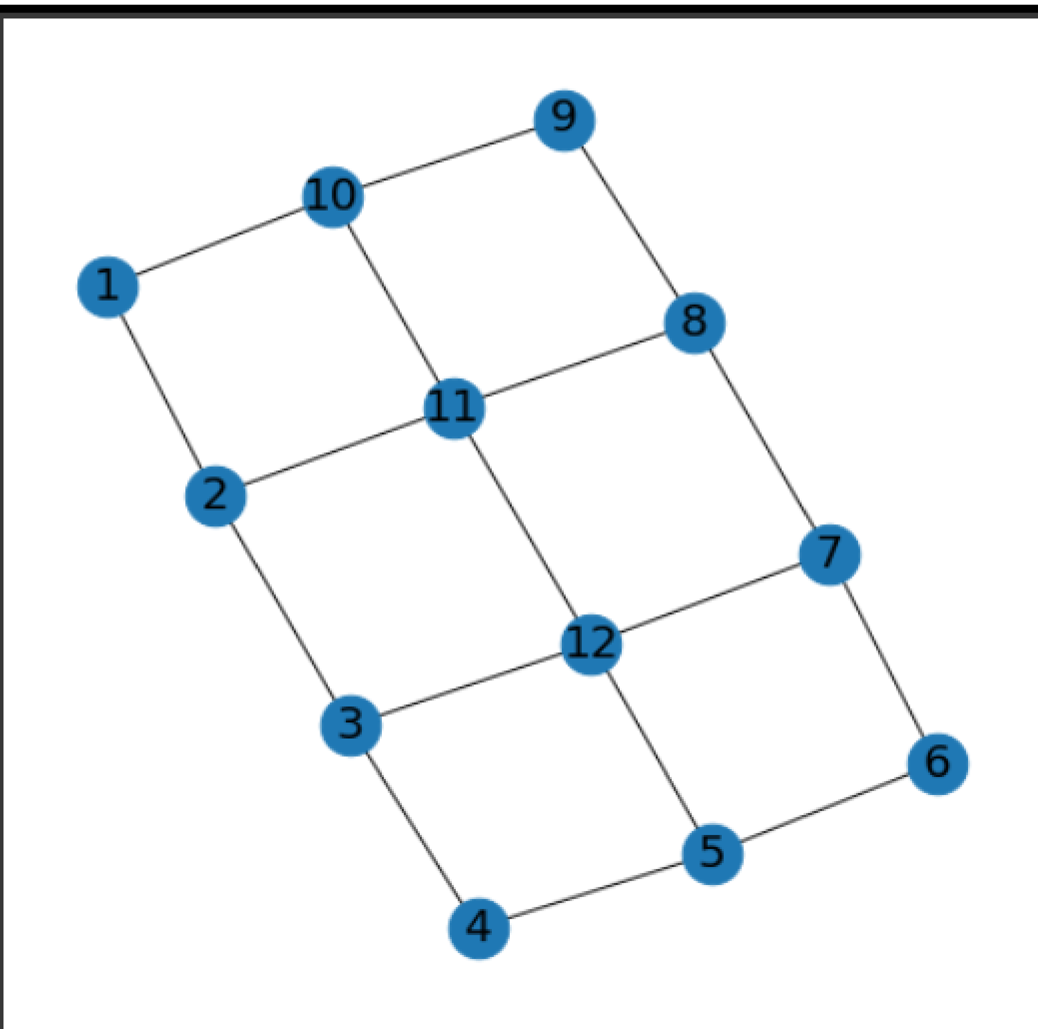
```
Directed Graph:
Number of nodes: 12
Number of edges: 17
Maximum out degree: 2
Minimum out degree: 1
Nodes with maximum out-degree: [2, 3, 10, 11, 12]
Nodes with minimum out-degree: [1, 4, 5, 6, 7, 8, 9]
Maximum in degree: 2
Minimum in degree: 1
Nodes with maximum in-degree: [5, 7, 8, 11, 12]
Nodes with minimum in-degree: [1, 2, 3, 4, 6, 9, 10]
```

Part-3

1) Creating Adjacency Matrix for undirected and unweighted Graph.

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

# Open the CSV file and read the edges
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1])) for row in reader]

# Create an empty undirected graph
G = nx.Graph()

# Add the edges to the graph
G.add_edges_from(edges)

# Draw the graph
pos = nx.spring_layout(G) # positions for all nodes

# Set the figure size
plt.figure(figsize=(8, 8))

# Draw the nodes and edges
nx.draw_networkx_nodes(G, pos, node_size=700)
nx.draw_networkx_edges(G, pos)

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=20, font_family
='sans-serif')

# Turn off the axis
plt.axis('off')

# Show the plot
plt.show()

# Adjacency Matrix (unweighted)
adj_matrix = nx.adjacency_matrix(G)
print(adj_matrix.todense())
```

Output -

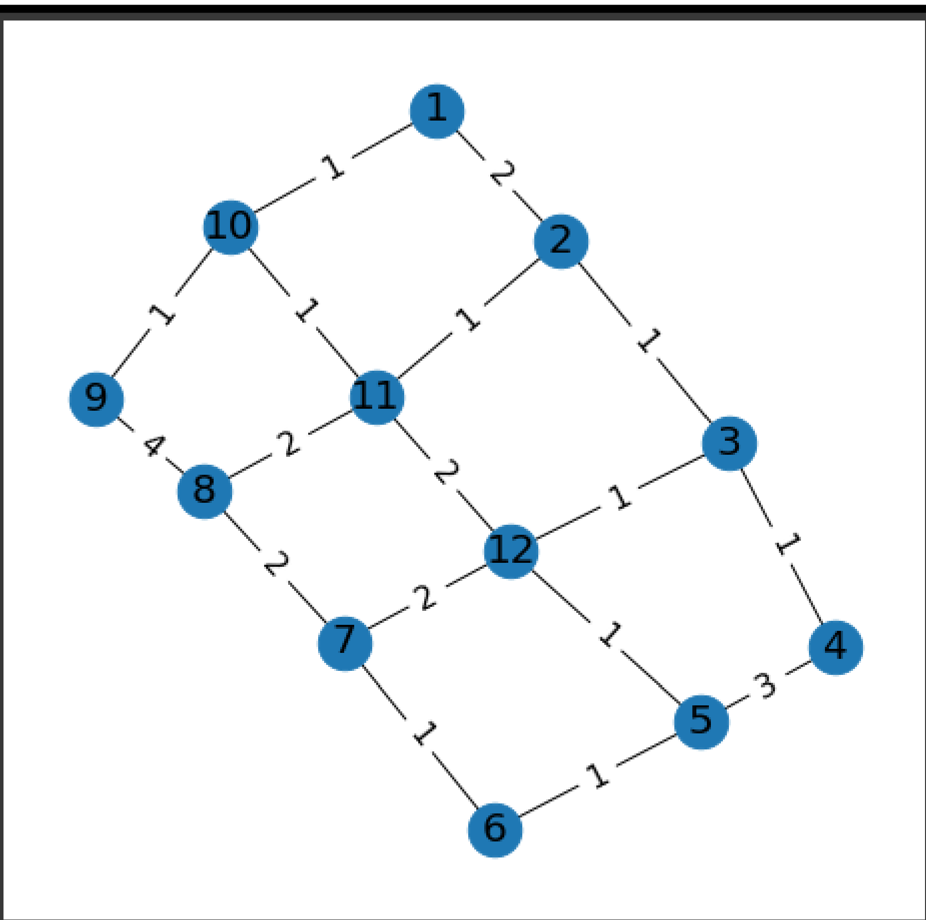



```
[[0 1 0 0 0 0 0 0 0 1 0 0]
 [1 0 1 0 0 0 0 0 0 0 1 0]
 [0 1 0 1 0 0 0 0 0 0 0 1]
 [0 0 1 0 1 0 0 0 0 0 0 0]
 [0 0 0 1 0 1 0 0 0 0 0 1]
 [0 0 0 0 1 0 1 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 0 0 0 1]
 [0 0 0 0 0 0 1 0 1 0 1 0]
 [0 0 0 0 0 0 0 1 0 1 0 0]
 [1 0 0 0 0 0 0 0 1 0 1 0]
 [0 1 0 0 0 0 0 1 0 1 0 1]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

2) Creating Adjacency Matrix for undirected and weighted Graph.

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

# Open the CSV file and read the edges
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), int(row[2])) for row in reader]

# Create an empty undirected graph
G = nx.Graph()

# Add the edges to the graph with weights
G.add_weighted_edges_from(edges)

# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes

# Set the figure size
plt.figure(figsize=(8, 8))

# Draw the nodes
nx.draw_networkx_nodes(G, pos, node_size=700)

# Draw the edges with arrows and weights
edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
nx.draw_networkx_edges(G, pos, edgelist=G.edges())
edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=20, font_family
='sans-serif')

# Turn off the axis
plt.axis('off')

# Show the plot
plt.show()

# Adjacency Matrix (weighted)
adj_matrix = nx.adjacency_matrix(G, weight='weight')
print(adj_matrix.todense())
```

Output -

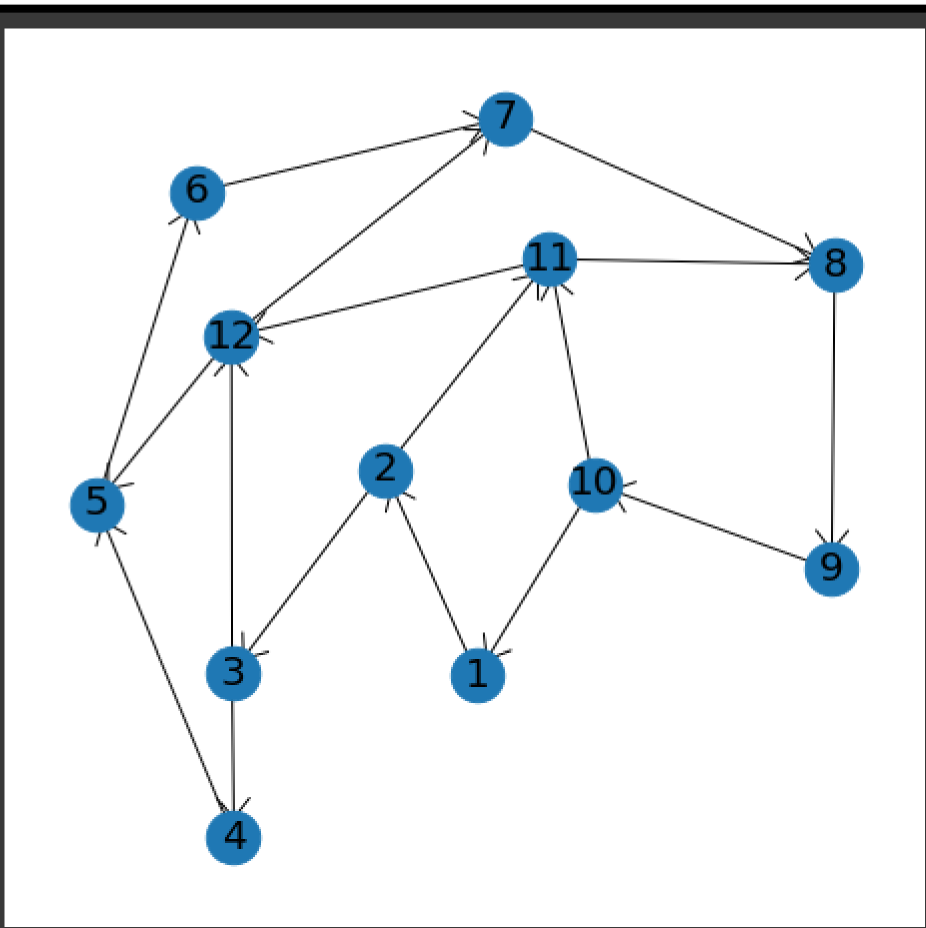



```
[[0 2 0 0 0 0 0 0 0 1 0 0]
 [2 0 1 0 0 0 0 0 0 0 1 0]
 [0 1 0 1 0 0 0 0 0 0 0 1]
 [0 0 1 0 3 0 0 0 0 0 0 0]
 [0 0 0 3 0 1 0 0 0 0 0 1]
 [0 0 0 0 1 0 1 0 0 0 0 0]
 [0 0 0 0 0 1 0 2 0 0 0 2]
 [0 0 0 0 0 0 2 0 4 0 2 0]
 [0 0 0 0 0 0 0 4 0 1 0 0]
 [1 0 0 0 0 0 0 0 1 0 1 0]
 [0 1 0 0 0 0 0 2 0 1 0 2]
 [0 0 1 0 1 0 2 0 0 0 2 0]]
```

3) Creating Adjacency Matrix for directed graph and unweighted Graph.

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

# Open the CSV file and read the edges
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1])) for row in reader]

# Create an empty directed graph
G = nx.DiGraph()

# Add the edges to the graph
G.add_edges_from(edges)

# Draw the graph
pos = nx.spring_layout(G) # positions for all nodes

# Set the figure size
plt.figure(figsize=(8, 8))

# Draw the nodes and edges
nx.draw_networkx_nodes(G, pos, node_size=700)
nx.draw_networkx_edges(G, pos, edgelist=G.edges(),
                       arrowsize=20, arrowstyle='-
>, head_width=0.4, head_length=0.5')

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=20, font_family
='sans-serif')

# Turn off the axis
plt.axis('off')

# Show the plot
plt.show()

# Adjacency Matrix (unweighted directed)
adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,11))
print(adj_matrix.todense())
```

Output -




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

4) Calculate the sum of particular row from adjacency matix.

```
# Convert the adjacency matrix to a numpy array
adj_array = np.array(adj_matrix.todense())
row_sum = 0
col_sum = 0
# Sum the elements of the i-th row
rown = 5
i = rown # Replace with the index of the desired row
row_sum = adj_array[i,:].sum()

# Sum the elements of the j-th column
j = rown # Replace with the index of the desired column
col_sum = adj_array[:,j].sum()

print("Row sum / OUT degree :", row_sum)
print("Column sum / IN degree:", col_sum)

print("Total degree : ",row_sum+col_sum)
```

Output -

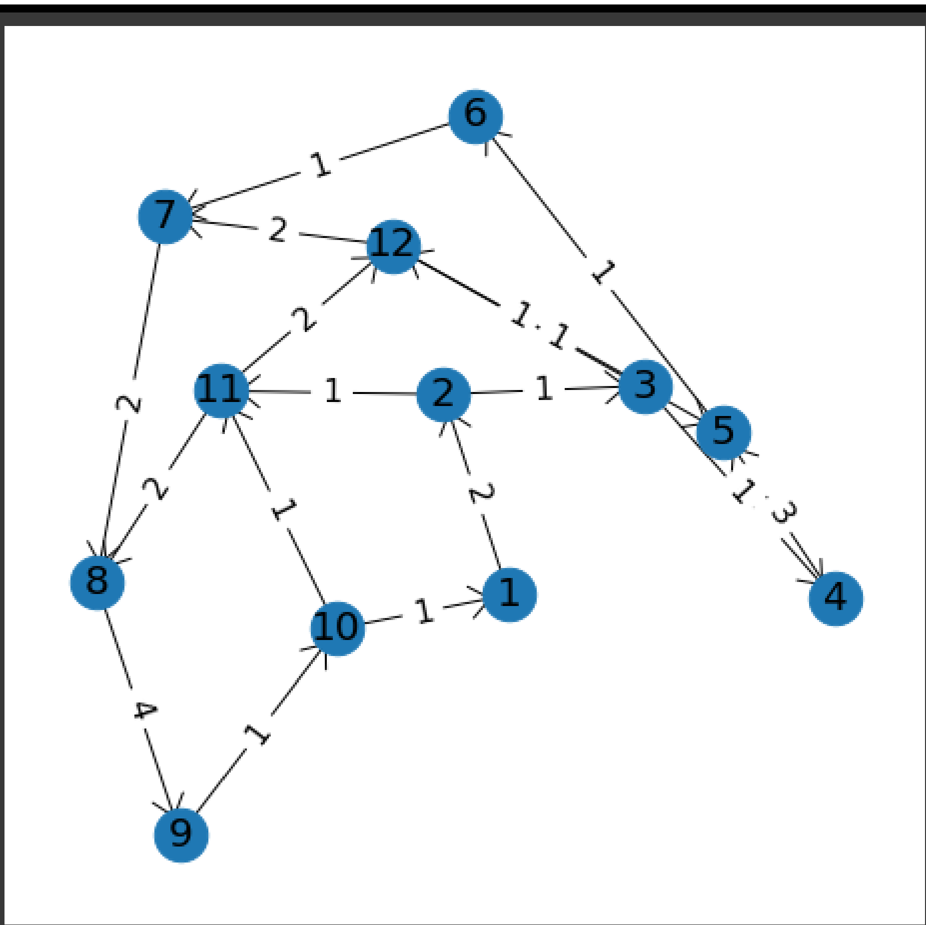
```
Row sum / OUT degree : 1
Column sum / IN degree: 1
Total degree :  2
```

5) Creating Adjacency Matrix for directed graph and weighted Graph.

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

# Open the CSV file and read the edges
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), int(row[2])) for row in reader]

# Create an empty directed graph
G = nx.DiGraph()

# Add the edges to the graph with weights
G.add_weighted_edges_from(edges)

# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes

# Set the figure size
plt.figure(figsize=(8, 8))

# Draw the nodes
nx.draw_networkx_nodes(G, pos, node_size=700)

# Draw the edges with arrows and weights
edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
nx.draw_networkx_edges(G, pos, edgelist=G.edges(),
                       arrowsize=20, arrowstyle='-
>, head_width=0.4, head_length=0.5')
edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=20, font_family
='sans-serif')

# Turn off the axis
plt.axis('off')

# Show the plot
plt.show()

# Adjacency Matrix (weighted directed)
```

```python
adj_matrix = nx.adjacency_matrix(G, weight='weight')
print(adj_matrix.todense())
```

Output -




```
[[0 2 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 0 0 3 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 2 0 0 0 0]
 [0 0 0 0 0 0 0 0 4 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0]
 [1 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 2 0 0 0 2]
 [0 0 0 0 1 0 2 0 0 0 0 0]]
```

Part–4

1) Calculating the centrality measure for directed graph.

```
import csv
import networkx as nx
from tabulate import tabulate

# create the graph from the edges.csv file
G = nx.DiGraph()
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)  # skip header row
    for row in reader:
        source, dest, weight = row
        G.add_edge(source, dest, weight=float(weight))

# calculate centrality measures
degree_centrality = nx.degree_centrality(G)
betweenness_centrality = nx.betweenness_centrality(G)
closeness_centrality = nx.closeness_centrality(G)
page_rank_centrality = nx.pagerank(G, max_iter=100, tol=1e-
06, alpha=0.85, personalization=None, weight='weight', dangling=None)
eigen_value_centrality = nx.eigenvector_centrality_numpy(G)

# round off centrality values to 2 decimal points
degree_centrality = {node: round(value, 2) for node, value in degree_centrali
ty.items()}
betweenness_centrality = {node: round(value, 2) for node, value in betweennes
s_centrality.items()}
closeness_centrality = {node: round(value, 2) for node, value in closeness_ce
ntrality.items()}
page_rank_centrality = {node: round(value, 2) for node, value in page_rank_ce
ntrality.items()}
eigen_value_centrality = {node: round(value, 2) for node, value in eigen_valu
e_centrality.items()}
```

```
# print centrality measures for each node
table_data = []
for node in G.nodes():
    row = [node, degree_centrality[node], betweenness_centrality[node], close
ness_centrality[node], page_rank_centrality[node], eigen_value_centrality[nod
e]]
    table_data.append(row)
```

```
headers = ['Node', 'Degree Centrality', 'Betweenness Centrality', 'Closeness
Centrality', 'PageRank Centrality', 'Eigenvalue Centrality']
print(tabulate(table_data, headers=headers))
```

```
# determine node with highest centrality score for each measure
max_degree_node = max(degree_centrality, key=degree_centrality.get)
max_betweenness_node = max(betweenness_centrality, key=betweenness_centrality
.get)
max_closeness_node = max(closeness_centrality, key=closeness_centrality.get)
max_page_rank_node = max(page_rank_centrality, key=page_rank_centrality.get)
max_eigen_value_node = max(eigen_value_centrality, key=eigen_value_centrality
.get)

# create a CSV file for the printed table
with open('centrality_directed.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(headers)
    writer.writerows(table_data)

print("Saved centrality measures to centrality_directed.csv")
```

Output -

Node	Degree Centrality	Betweenness Centrality	Closeness Centrality	PageRank Centrality	Eigenvalue Centrality
1	0.18	0.25	0.23	0.07	0.2
2	0.27	0.25	0.2	0.07	0.15
3	0.27	0.13	0.19	0.04	0.11
4	0.18	0.02	0.18	0.03	0.08
5	0.27	0.17	0.29	0.06	0.28
6	0.18	0.17	0.26	0.06	0.21
7	0.27	0.37	0.37	0.11	0.38
8	0.27	0.53	0.42	0.14	0.5
9	0.18	0.53	0.32	0.13	0.37
10	0.27	0.53	0.26	0.13	0.27
11	0.36	0.29	0.26	0.1	0.31
12	0.36	0.25	0.26	0.07	0.31

Saved centrality measures to centrality_directed.csv

```
# print nodes with highest centrality scores
print("Node with highest Degree Centrality: ", max_degree_node)
print("Node with highest Betweenness Centrality: ", max_betweenness_node)
print("Node with highest Closeness Centrality: ", max_closeness_node)
print("Node with highest PageRank Centrality: ", max_page_rank_node)
print("Node with highest Eigenvalue Centrality: ", max_eigen_value_node)
```

Output -

```
Node with highest Degree Centrality:  11
Node with highest Betweenness Centrality:  8
Node with highest Closeness Centrality:  8
Node with highest PageRank Centrality:  8
Node with highest Eigenvalue Centrality:  8
```

2) Calculating Highest Centrality score for Nodes.

```
import csv
import networkx as nx
from tabulate import tabulate

# create the graph from the edges.csv file
G = nx.DiGraph()
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)  # skip header row
    for row in reader:
        source, dest, weight = row
        G.add_edge(source, dest, weight=float(weight))

# calculate centrality measures
degree_centrality = nx.degree_centrality(G)
betweenness_centrality = nx.betweenness_centrality(G)
closeness_centrality = nx.closeness_centrality(G)
page_rank_centrality = nx.pagerank(G, max_iter=100, tol=1e-
06, alpha=0.85, personalization=None, weight='weight', dangling=None)
eigenvector_centrality = nx.eigenvector_centrality_numpy(G)

# round off centrality values to 2 decimal points
```

```
degree_centrality = {node: round(value, 2) for node, value in degree_centrali
ty.items()}
betweenness_centrality = {node: round(value, 2) for node, value in betweennes
s_centrality.items()}
closeness_centrality = {node: round(value, 2) for node, value in closeness_ce
ntrality.items()}
page_rank_centrality = {node: round(value, 2) for node, value in page_rank_ce
ntrality.items()}
eigenvector_centrality = {node: round(value, 2) for node, value in eigenvecto
r_centrality.items()}
```

```
# determine nodes with highest and lowest centrality scores for each measure
max_degree = max(degree_centrality.values())
max_degree_nodes = [node for node, value in degree_centrality.items() if valu
e == max_degree]
min_degree = min(degree_centrality.values())
min_degree_nodes = [node for node, value in degree_centrality.items() if valu
e == min_degree]

max_betweenness = max(betweenness_centrality.values())
max_betweenness_nodes = [node for node, value in betweenness_centrality.items
() if value == max_betweenness]
min_betweenness = min(betweenness_centrality.values())
min_betweenness_nodes = [node for node, value in betweenness_centrality.items
() if value == min_betweenness]

max_closeness = max(closeness_centrality.values())
max_closeness_nodes = [node for node, value in closeness_centrality.items() i
f value == max_closeness]
min_closeness = min(closeness_centrality.values())
min_closeness_nodes = [node for node, value in closeness_centrality.items() i
f value == min_closeness]

max_page_rank = max(page_rank_centrality.values())
max_page_rank_nodes = [node for node, value in page_rank_centrality.items() i
f value == max_page_rank]
min_page_rank = min(page_rank_centrality.values())
min_page_rank_nodes = [node for node, value in page_rank_centrality.items() i
f value == min_page_rank]

max_eigenvector = max(eigenvector_centrality.values())
max_eigenvector_nodes = [node for node, value in eigenvector_centrality.items
() if value == max_eigenvector]
min_eigenvector = min(eigenvector_centrality.values())
min_eigenvector_nodes = [node for node, value in eigenvector_centrality.items
() if value == min_eigenvector]
# store the results in a table
table_data = [
```

```
    ['Degree Centrality', max_degree, max_degree_nodes, min_degree, min_degre
e_nodes],
    ['Betweenness Centrality', max_betweenness, max_betweenness_nodes, min_be
tweenness, min_betweenness_nodes],
    ['Closeness Centrality', max_closeness, max_closeness_nodes, min_closenes
s, min_closeness_nodes],
    ['PageRank Centrality', max_page_rank, max_page_rank_nodes, min_page_rank
, min_page_rank_nodes],
    ['Eigenvector Centrality', max_eigenvector, max_eigenvector_nodes, min_ei
genvector, min_eigenvector_nodes]
]

print(tabulate(table_data, headers=['Centrality', 'Highest Centrality Score',
 'Highest Centrality Nodes', 'lowest Centrality Score', 'lowest Centrality No
des']))
```

Output -

```
Centrality                Highest Centrality Score  Highest Centrality Nodes      lowest Centrality Score  lowest Centrality Nodes
----------------------  --------------------------  --------------------------  -------------------------  --------------------------
Degree Centrality                             0.36  ['11', '12']                                    0.18  ['1', '4', '6', '9']
Betweenness Centrality                        0.53  ['8', '9', '10']                                0.02  ['4']
Closeness Centrality                          0.42  ['8']                                           0.18  ['4']
PageRank Centrality                           0.14  ['8']                                           0.03  ['4']
Eigenvector Centrality                        0.5   ['8']                                           0.08  ['4']
```

3) Calculating the centrality measure for undirected graph.

```
import csv
import networkx as nx
from tabulate import tabulate

# create the graph from the edges.csv file
G = nx.Graph()
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)  # skip header row
    for row in reader:
        source, dest, weight = row
        G.add_edge(source, dest, weight=float(weight))

# calculate centrality measures
degree_centrality = nx.degree_centrality(G)
betweenness_centrality = nx.betweenness_centrality(G)
closeness_centrality = nx.closeness_centrality(G)
page_rank_centrality = nx.pagerank(G, max_iter=100, tol=1e-
06, alpha=0.85, personalization=None, weight='weight')
```

```
eigen_value_centrality = nx.eigenvector_centrality_numpy(G)

# round off centrality values to 2 decimal points
degree_centrality = {node: round(value, 2) for node, value in degree_centrality.items()}
betweenness_centrality = {node: round(value, 2) for node, value in betweenness_centrality.items()}
closeness_centrality = {node: round(value, 2) for node, value in closeness_centrality.items()}
page_rank_centrality = {node: round(value, 2) for node, value in page_rank_centrality.items()}
eigen_value_centrality = {node: round(value, 2) for node, value in eigen_value_centrality.items()}
```

```
# print centrality measures for each node
table_data = []
for node in G.nodes():
    row = [node, degree_centrality[node], betweenness_centrality[node], closeness_centrality[node], page_rank_centrality[node], eigen_value_centrality[node]]
    table_data.append(row)

headers = ['Node', 'Degree Centrality', 'Betweenness Centrality', 'Closeness Centrality', 'PageRank Centrality', 'Eigenvalue Centrality']
print(tabulate(table_data, headers=headers))

# determine node with highest centrality score for each measure
max_degree_node = max(degree_centrality, key=degree_centrality.get)
max_betweenness_node = max(betweenness_centrality, key=betweenness_centrality.get)
max_closeness_node = max(closeness_centrality, key=closeness_centrality.get)
max_page_rank_node = max(page_rank_centrality, key=page_rank_centrality.get)
max_eigen_value_node = max(eigen_value_centrality, key=eigen_value_centrality.get)

# create a CSV file for the printed table
with open('centrality_undirected.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(headers)
    writer.writerows(table_data)

print("Saved centrality measures to centrality_undirected.csv")
```

Output -

```
 Node    Degree Centrality    Betweenness Centrality    Closeness Centrality    PageRank Centrality    Eigenvalue Centrality
------  -------------------  ------------------------  ----------------------  ---------------------  -----------------------
     1                 0.18                      0.03                    0.37                   0.06                     0.19
     2                 0.27                      0.16                    0.46                   0.08                     0.3
     3                 0.27                      0.16                    0.46                   0.06                     0.3
     4                 0.18                      0.03                    0.37                   0.08                     0.19
     5                 0.27                      0.11                    0.42                   0.1                      0.26
     6                 0.18                      0.03                    0.37                   0.04                     0.19
     7                 0.27                      0.16                    0.46                   0.09                     0.3
     8                 0.27                      0.16                    0.46                   0.13                     0.3
     9                 0.18                      0.03                    0.37                   0.08                     0.19
    10                 0.27                      0.11                    0.42                   0.06                     0.26
    11                 0.36                      0.31                    0.55                   0.1                      0.43
    12                 0.36                      0.31                    0.55                   0.11                     0.43
Saved centrality measures to centrality_undirected.csv
```

4) Calculating Highest Centrality score for Nodes.

```python
import csv
import networkx as nx
from tabulate import tabulate

# create the graph from the edges.csv file
G = nx.Graph()
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)  # skip header row
    for row in reader:
        source, dest, weight = row
        G.add_edge(source, dest, weight=float(weight))

# calculate centrality measures
degree_centrality = nx.degree_centrality(G)
betweenness_centrality = nx.betweenness_centrality(G)
closeness_centrality = nx.closeness_centrality(G)
page_rank_centrality = nx.pagerank(G, max_iter=100, tol=1e-06, alpha=0.85, personalization=None, weight='weight')
eigenvector_centrality = nx.eigenvector_centrality_numpy(G)

# round off centrality values to 2 decimal points
degree_centrality = {node: round(value, 2) for node, value in degree_centrality.items()}
betweenness_centrality = {node: round(value, 2) for node, value in betweenness_centrality.items()}
closeness_centrality = {node: round(value, 2) for node, value in closeness_centrality.items()}
```

```
page_rank_centrality = {node: round(value, 2) for node, value in page_rank_ce
ntrality.items()}
eigenvector_centrality = {node: round(value, 2) for node, value in eigenvecto
r_centrality.items()}

# determine nodes with highest and lowest centrality scores for each measure
max_degree = max(degree_centrality.values())
max_degree_nodes = [node for node, value in degree_centrality.items() if valu
e == max_degree]
min_degree = min(degree_centrality.values())
min_degree_nodes = [node for node, value in degree_centrality.items() if valu
e == min_degree]
```

```
max_betweenness = max(betweenness_centrality.values())
max_betweenness_nodes = [node for node, value in betweenness_centrality.items
() if value == max_betweenness]
min_betweenness = min(betweenness_centrality.values())
min_betweenness_nodes = [node for node, value in betweenness_centrality.items
() if value == min_betweenness]

max_closeness = max(closeness_centrality.values())
max_closeness_nodes = [node for node, value in closeness_centrality.items() i
f value == max_closeness]
min_closeness = min(closeness_centrality.values())
min_closeness_nodes = [node for node, value in closeness_centrality.items() i
f value == min_closeness]

max_page_rank = max(page_rank_centrality.values())
max_page_rank_nodes = [node for node, value in page_rank_centrality.items() i
f value == max_page_rank]
min_page_rank = min(page_rank_centrality.values())
min_page_rank_nodes = [node for node, value in page_rank_centrality.items() i
f value == min_page_rank]

max_eigenvector = max(eigenvector_centrality.values())
max_eigenvector_nodes = [node for node, value in eigenvector_centrality.items
() if value == max_eigenvector]
min_eigenvector = min(eigenvector_centrality.values())
min_eigenvector_nodes = [node for node, value in eigenvector_centrality.items
() if value == min_eigenvector]

# store the results in a table
table_data = [
```

```
    ['Degree Centrality', max_degree, max_degree_nodes, min_degree, min_degree_nodes],
    ['Betweenness Centrality', max_betweenness, max_betweenness_nodes, min_betweenness, min_betweenness_nodes],
    ['Closeness Centrality', max_closeness, max_closeness_nodes, min_closeness, min_closeness_nodes],
    ['PageRank Centrality', max_page_rank, max_page_rank_nodes, min_page_rank, min_page_rank_nodes],
    ['Eigenvector Centrality', max_eigenvector, max_eigenvector_nodes, min_eigenvector, min_eigenvector_nodes]
]

print(tabulate(table_data, headers=['Centrality', 'Highest Centrality Score', 'Highest Centrality Nodes', 'lowest Centrality Score', 'lowest Centrality Nodes']))
```

Output -

Centrality	Highest Centrality Score	Highest Centrality Nodes	lowest Centrality Score	lowest Centrality Nodes
Degree Centrality	0.36	['11', '12']	0.18	['1', '4', '6', '9']
Betweenness Centrality	0.31	['11', '12']	0.03	['1', '4', '6', '9']
Closeness Centrality	0.55	['11', '12']	0.37	['1', '4', '6', '9']
PageRank Centrality	0.13	['8']	0.04	['6']
Eigenvector Centrality	0.43	['11', '12']	0.19	['1', '4', '6', '9']