VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering
(WINTER 2022-2023)

Student Name: **Harsha Dulhani**
Reg No: **22MCB0012**

Email: harsha.dulhani2022@vitstudent.ac.in

Mobile: 9479484361

Faculty Name: **DURGESH KUMAR**

SUBJECT NAME WITH CODE: SOCIAL NETWORK ANALYSIS LAB – MCSE506P

Assessment Report

Date of Submission : 23/03/2023

Table Of Content

GRAPHS

A graph is a visual representation of the connections between individuals or entities in a social network. A graph consists of nodes (also known as vertices) and edges (also known as ties or links).

Nodes represent the individuals or entities in the social network, and edges represent the relationships or interactions between them.

For example, in a social network of Facebook users, nodes could represent individual users, and edges could represent friend connections between them.

Types of Graphs : -

Directed Graph – In a directed graph, edges have a specific direction, indicating that the relationship or interaction between two nodes is asymmetric.

Undirected Graph – In an undirected graph, edges have no direction, indicating that the relationship or interaction between two nodes is symmetric.

Weighted Graph – In a weighted graph, each edge has a numerical value, indicating the strength or intensity of the relationship or interaction between two nodes.

Unweighted Graph - In an unweighted graph, all edges are considered to have equal importance.

LIBRARIES USED

1. **NumPy:** NumPy is a Python library used for numerical computing. It provides support for multi-dimensional arrays and matrices, along with a large collection of mathematical functions to operate on these arrays. It's used in a wide range of scientific and engineering applications, such as data analysis, image processing, machine learning, and more.
2. **Matplotlib:** Matplotlib is a plotting library for Python that provides a wide range of 2D and 3D visualizations. It enables users to create a variety of plots, including line plots, scatter plots, bar plots, histograms, and more. It's widely used in data visualization and scientific research, as well as in teaching and learning.
3. **Pandas:** Pandas is a library for data manipulation and analysis in Python. It provides data structures for efficiently storing and manipulating large datasets, as well as a wide range of functions for working with data. Pandas is often used in data science, finance, and other fields where large datasets need to be analysed and processed.
4. **NetworkX:** NetworkX is a Python library for the creation, manipulation, and analysis of complex networks. It provides tools for creating networks, visualizing them, and analysing their properties. NetworkX is often used in social network analysis, biological network analysis, and other fields where network analysis is required.

Importing the necessary libraries

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

Reading the csv file

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

Description Of csv file

```
# Read the CSV file into a pandas dataframe
df = pd.read_csv('edges_list.csv')

# Print the entire dataframe
print(df)
```

	Source	Destination	Weight
0	1	2	10
1	1	4	20
2	1	5	20
3	1	6	5
4	1	7	15
5	2	3	5
6	2	4	10
7	3	2	15
8	3	4	5
9	4	5	10
10	5	6	5
11	7	6	10
12	8	1	5
13	8	7	5
14	9	2	15
15	9	8	20
16	9	10	10
17	8	2	20
18	10	2	5
19	10	3	15

PART-A

- Creating an empty directed graph and adding edges
  To generate a directed graph using NetworkX, you can use the ***DiGraph()*** method.

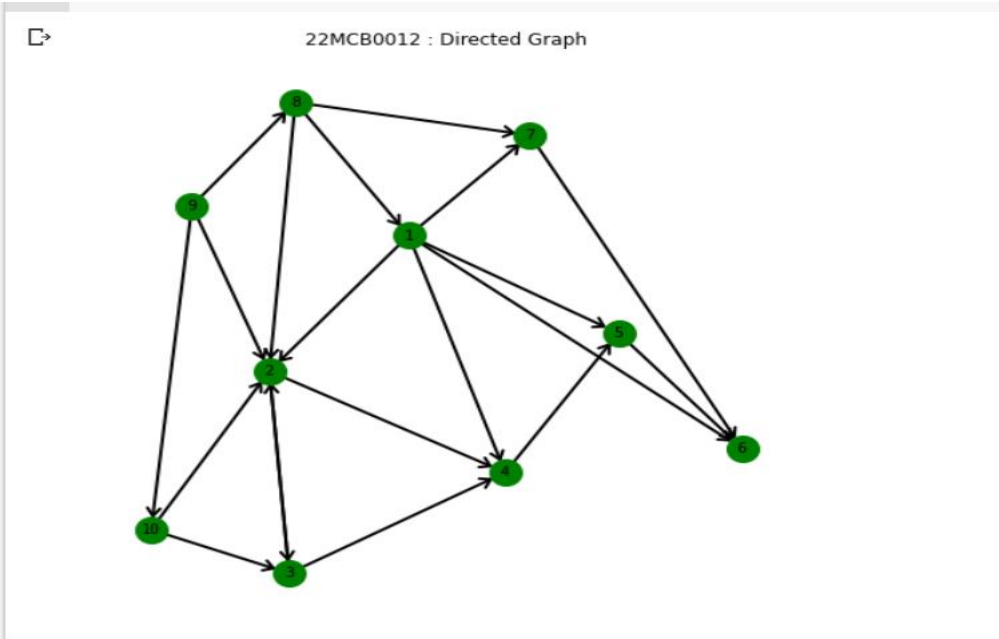
```
[36] # Create an empty directed graph
     G = nx.DiGraph()
```

```
[37] # Add the edges to the graph with weights
     G.add_weighted_edges_from(edges)
```

- Plotting a Directed Graph without weights

```
# Define a list of colors for the nodes
node_colors = ['red', 'blue', 'green', 'orange', 'purple']
plt.figure(figsize=(8, 8))
# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes
nx.draw_networkx_nodes(G, pos, node_size=400,node_color='green')
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2],
                       arrowsize=20,arrows=True, arrowstyle='->, head_width=0.2, head_length=0.3')
nx.draw_networkx_labels(G, pos, font_size=10, font_family='monospace')
plt.title('22MCB0012 : Directed Graph')
plt.axis('off')
plt.savefig('Directed_Graph.png')
plt.show()
```

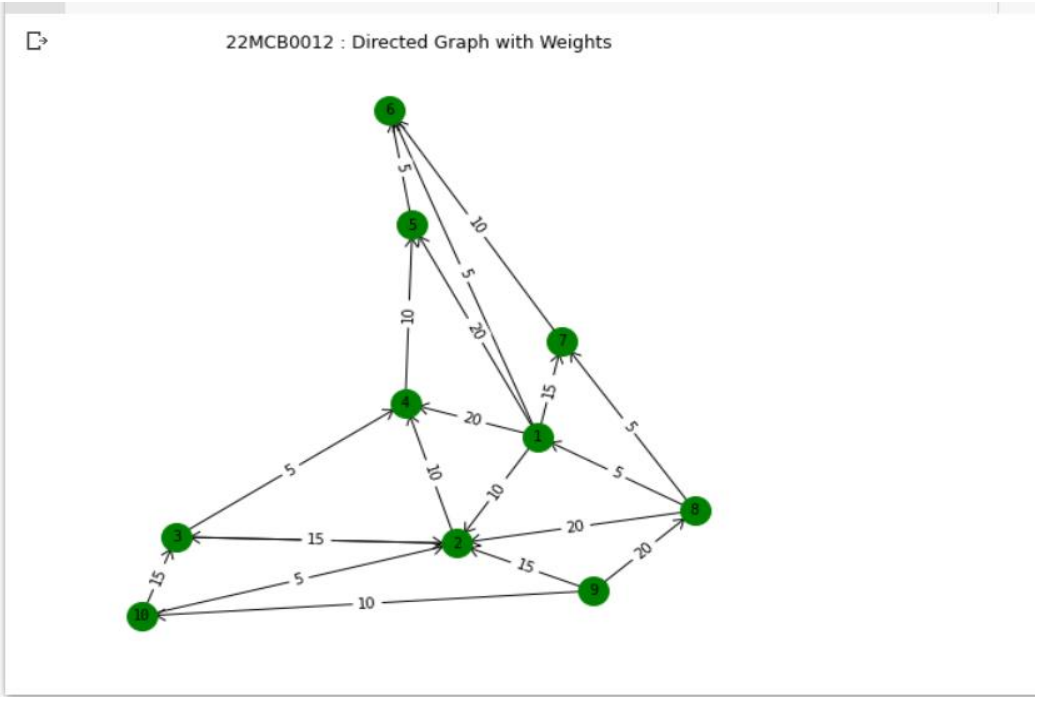
Explanation of above code: This code generates a directed graph plot using the NetworkX and Matplotlib libraries in Python. The positions for all nodes are generated using the spring_layout function of the NetworkX library. The nodes of the graph are drawn using the draw_networkx_nodes function of the NetworkX library. The edges of the graph are drawn using the draw_networkx_edges function of the NetworkX library. The labels for the nodes of the graph are drawn using the draw_networkx_labels function. The plot is displayed using the show function of the Matplotlib library.



- Creating an directed graph and adding edges with weights
- Plotting a Directed Graph with weights

```python
# 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=400,node_color='green')

# 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=[1],
                       arrowsize=20, arrowstyle='->, head_width=0.2, head_length=0.3')
edge_labels = {(u, v): f'{int(d["weight"])}' for (u, v, d) in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=10)

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=10, font_family='monospace')

plt.title('22MCB0012 : Directed Graph with Weights')

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

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

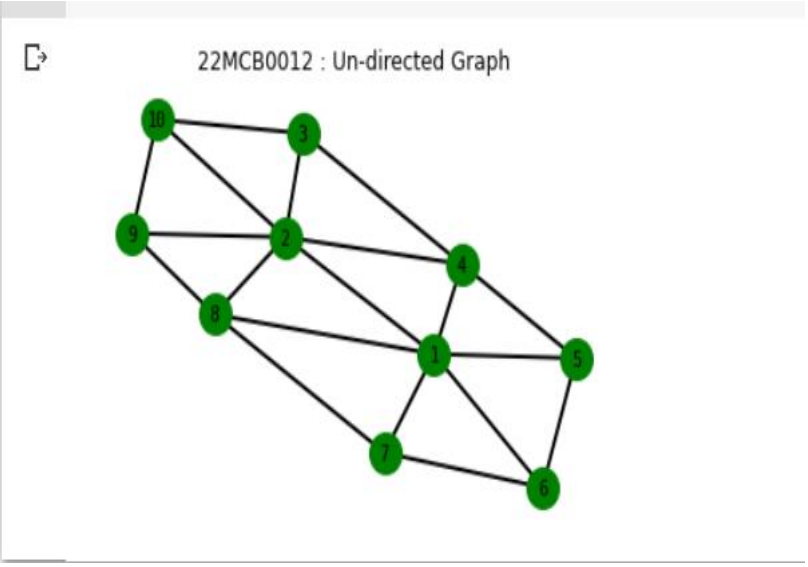
Explanation of above code : This code creates a directed graph with weighted edges using NetworkX and matplotlib libraries in Python. It adds the edges with weights to the graph using the add_weighted_edges_from() function of NetworkX. It uses the spring_layout() function of NetworkX to compute the positions of the nodes in the graph.



- Creating an empty Un-directed graph and adding edges without weight
  An empty undirected graph is created using ***nx.Graph()***.
- Plotting a Un- Directed Graph without weights

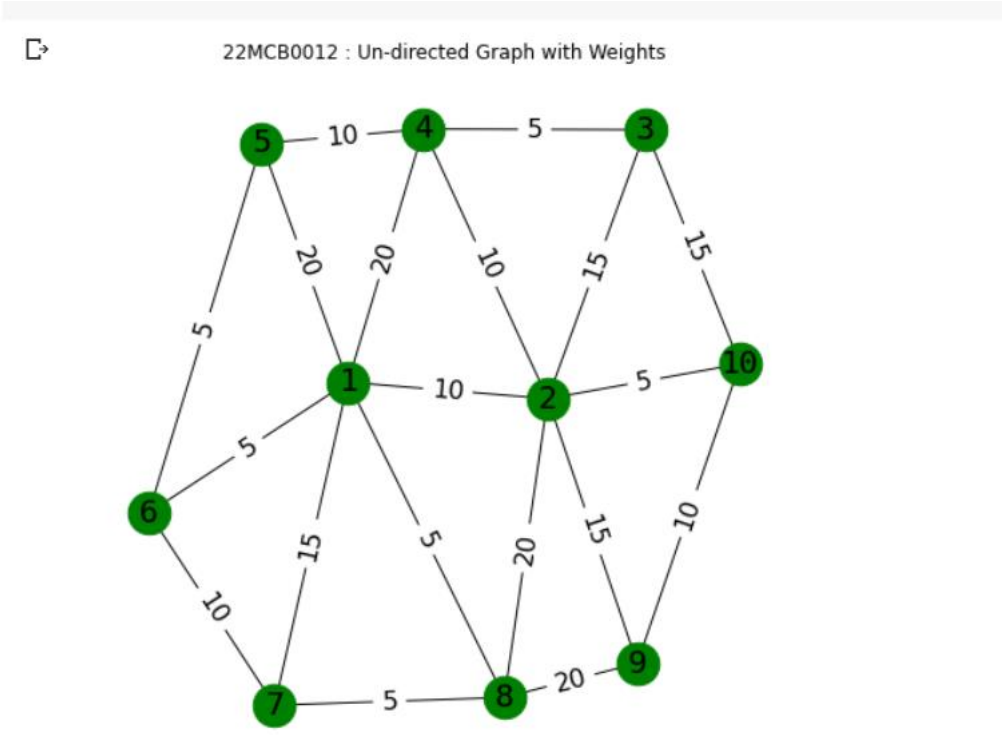
```
# Create an empty undirected graph
G = nx.Graph()

# 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=400,node_color='green')
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2])
nx.draw_networkx_labels(G, pos, font_size=10, font_family='monospace')
plt.title('22MCB0012 : Un-directed Graph')
plt.axis('off')
plt.savefig('Undirected_Graph.png')
plt.show()
```

Explanation Of above code : The graph is then drawn using nx.draw_networkx_nodes(), nx.draw_networkx_edges(), and nx.draw_networkx_labels().



- Creating an Un-directed graph and adding edges with weights
  The edges are then added to the graph using **G.add_weighted_edges_from(edges).**
- Plotting a Directed Graph with weights

```python
# 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,node_color='green')

# 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='monospace')

plt.title('22MCB0012 : Un-directed Graph with Weights')

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

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



PART-B

Indegree Of a Graph : The indegree of a vertex in a directed graph is the number of incoming edges to that vertex. That is, the indegree of a vertex v is the number of edges that have v as their destination vertex.

Outdegree of a Graph : The outdegree of a vertex in a directed graph is the number of outgoing edges from that vertex. That is, the outdegree of a vertex v is the number of edges that start from that vertex.

Find no of nodes, edges, node with maximum degree, node with minimum degree in an **"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}')

print()
```

Explanation of above code : The number_of_nodes() function of the graph object G to find the number of nodes in the graph. The number_of_edges() function of the graph object G to find the number of edges in the graph. The degree of each node in the graph using the degree() function of the graph object G and creates a dictionary called degrees where the keys are nodes and the values are their degree. The max() function to find the maximum degree of the nodes in the graph and assigns it to a variable called max_degree. The min() function to find the minimum degree of the nodes in the graph and assigns it to a variable called min_degree. Then it creates a list called max_degree_nodes that contains the nodes in the graph with the maximum degree by using a list comprehension. The comprehension iterates over the degrees dictionary and adds nodes to the list if their degree equals the maximum degree found. Same for min_degree_nodes.

```
Undirected Graph :
Number of nodes: 10
Number of edges: 19
Nodes with maximum degree: [1, 2]
Nodes with minimum degree: [5, 6, 7, 3, 9, 10]
Maximum degree: 6
Minimum degree: 3
```

Find no of nodes, edges, node with maximum indegree, node with minimum indegree, node with maximum outdegree, node with minimum outdegree in an **"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}')
```

Explanation of above code : To calculate the out-degree of each node in the graph using the out_degree() function of the graph object G and creates a dictionary called out_degrees where the keys are nodes and the values are their out-degree. The max() function used to find the maximum out-degree of the nodes in the graph and assigns it to a variable called max_out_degree. The min() function used to find the minimum out-degree of the nodes in the graph and assigns it to a variable called min_out_degree. Then it creates a list called max_out_degree_nodes that contains the nodes in the graph with the maximum out-degree by using a list comprehension. The comprehension iterates over the out_degrees dictionary and adds nodes to the list if their out-degree equals the maximum out-degree. Then it creates a list called min_out_degree_nodes that contains the nodes in the graph with the minimum out-degree by using a similar list comprehension. The comprehension iterates over the out_degrees dictionary and adds nodes to the list if their out-degree equals the minimum out-degree. Same for indegree.

```
Directed Graph:
Number of nodes: 10
Number of edges: 20
Maximum out degree: 5
Minimum out degree: 0
Nodes with maximum out-degree: [1]
Nodes with minimum out-degree: [6]
Maximum in degree: 5
Minimum in degree: 0
Nodes with maximum in-degree: [2]
Nodes with minimum in-degree: [9]
```

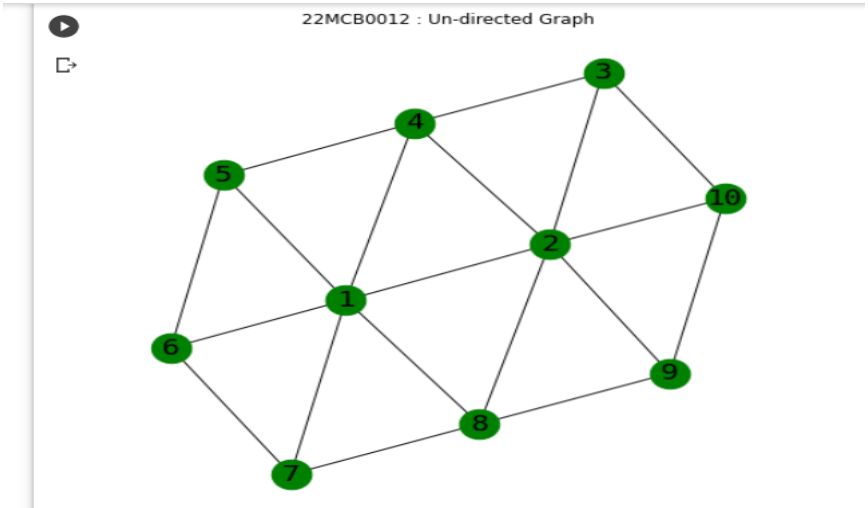
PART-C

Adjacency Matrix : An adjacency matrix is a matrix representation of a graph. It is a square matrix, where the rows and columns represent the nodes in the graph, and the elements of the matrix represent the edges between the nodes. If there is an edge between two nodes i and j, then the corresponding element in the matrix is 1, otherwise it is 0.

- ### Adjacency Matrix for Un-directed Graph without weight

```
# Draw the nodes and edges
nx.draw_networkx_nodes(G, pos, node_size=700,node_color='green')
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='monospace')

plt.title('22MCB0012 : Un-directed Graph')

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

# Show the plot
plt.show()

print('Adjancency Matrix for Un-directed Graph :')

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

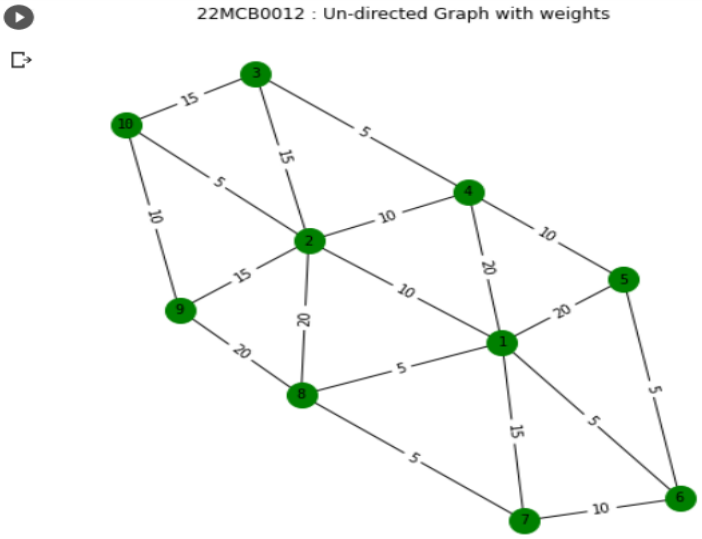
Explanation of above code : creates an adjacency matrix object adj_matrix using the nx.adjacency_matrix() function of NetworkX. uses the todense() function to convert the sparse matrix adj_matrix to a dense matrix, which is a standard NumPy matrix. The todense() function returns the dense matrix representation of the sparse matrix adj_matrix.




```
Adjancency Matrix for Un-directed Graph :
[[0 1 1 1 1 1 0 1 0 0]
 [1 0 1 0 0 0 1 1 1 1]
 [1 1 0 1 0 0 1 0 0 0]
 [1 0 1 0 1 0 0 0 0 0]
 [1 0 0 1 0 1 0 0 0 0]
 [1 0 0 0 1 0 0 1 0 0]
 [0 1 1 0 0 0 0 0 0 1]
 [1 1 0 0 0 1 0 0 1 0]
 [0 1 0 0 0 0 0 1 0 1]
 [0 1 0 0 0 0 1 0 1 0]]
```

○ Adjacency Matrix for Un-directed Graph with weights

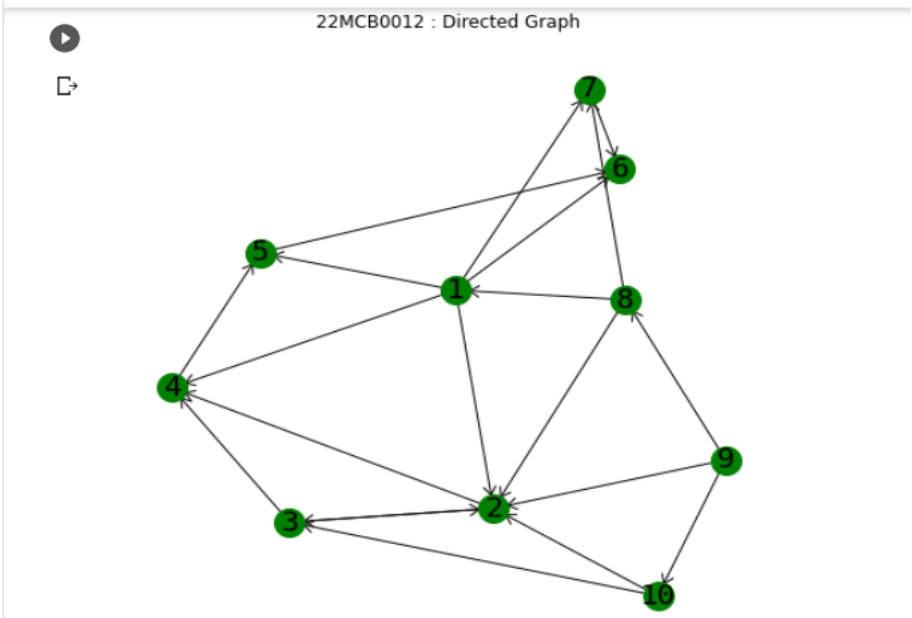
```
# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes

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

# Draw the nodes
nx.draw_networkx_nodes(G, pos, node_size=400,node_color='green')

# 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=10)

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=10, font_family='monospace')

plt.title('22MCB0012 : Un-directed Graph with weights')

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

# Show the plot
plt.show()

print('Adjancency Matrix for Un-directed Graph with weights :')
# Adjacency Matrix (weighted)
adj_matrix = nx.adjacency_matrix(G, weight='weight')
print(adj_matrix.todense())
```




```
Adjancency Matrix for Un-directed Graph with weights :
[[ 0 10 20 20  5 15  0  5  0  0]
 [10  0 10  0  0  0 15 20 15  5]
 [20 10  0 10  0  0  5  0  0  0]
 [20  0 10  0  5  0  0  0  0  0]
 [ 5  0  0  5  0 10  0  0  0  0]
 [15  0  0  0 10  0  0  5  0  0]
 [ 0 15  5  0  0  0  0  0  0 15]
 [ 5 20  0  0  0  5  0  0 20  0]
 [ 0 15  0  0  0  0  0 20  0 10]
 [ 0  5  0  0  0  0 15  0 10  0]]
```

- **Adjacency Matrix for Directed Graph without weights**

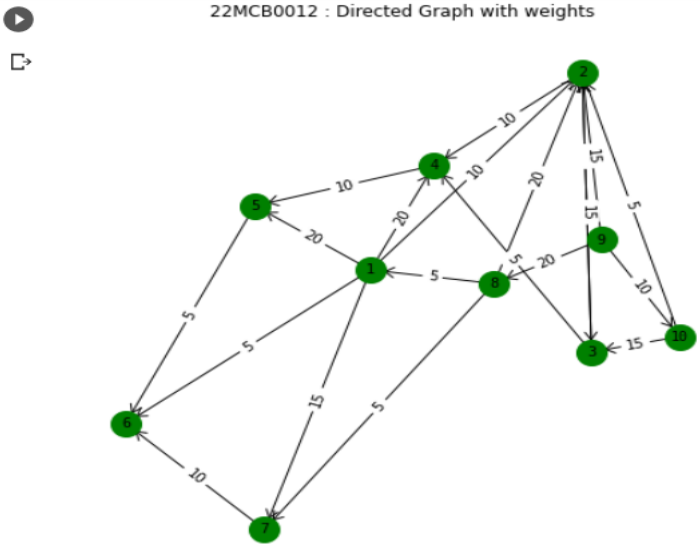
```
# Create an empty directed graph
G = nx.DiGraph()

# 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=400,node_color='green')
nx.draw_networkx_edges(G, pos, edgelist=G.edges(),
                       arrowsize=20, arrowstyle='->, head_width=0.2, head_length=0.3')

# 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='monospace')

plt.title('22MCB0012 : Directed Graph')
# Turn off the axis
plt.axis('off')

# Show the plot
plt.show()

print('Adjancency Matrix for Directed Graph :')
# Adjacency Matrix (unweighted directed)
adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,11))
print(adj_matrix.todense())
```




```
Adjancency Matrix for Directed Graph :
[[0 1 0 1 1 1 1 0 0 0]
 [0 0 1 1 0 0 0 0 0 0]
 [0 1 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 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0]
 [1 1 0 0 0 0 1 0 0 0]
 [0 1 0 0 0 0 0 1 0 1]
 [0 1 1 0 0 0 0 0 0 0]]
```

Adjacency Matrix for Directed Graph with weights

```
# 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=400, node_color='green')

# 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=0.2, head_length=0.3')
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=10)

# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=10, font_family='monospace')

plt.title('22MCB0012 : Directed Graph with weights')
# Turn off the axis
plt.axis('off')

# Show the plot
plt.show()

print('Adjancency Matrix for Directed Graph with weights :')
# Adjacency Matrix (weighted directed)
adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,11))
adj_matrix = nx.adjacency_matrix(G, weight='weight')
print(adj_matrix.todense())
```




```
Adjancency Matrix for Directed Graph with weights :
[[ 0 10 20 20  5 15  0  0  0  0]
 [ 0  0 10  0  0  0  5  0  0  0]
 [ 0  0  0 10  0  0  0  0  0  0]
 [ 0  0  0  0  5  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0 10  0  0  0  0  0]
 [ 0 15  5  0  0  0  0  0  0  0]
 [ 5 20  0  0  0  5  0  0  0  0]
 [ 0 15  0  0  0  0  0 20  0 10]
 [ 0  5  0  0  0  0 15  0  0  0]]
```

PART-D

Centrality Measures : -

In social network analysis, centrality measures are used to quantify the importance or influence of nodes (i.e., individuals, organizations, etc.) in a network.

There are several centrality measures.

1. **Degree Centrality:** Degree centrality is a measure of the number of connections a node has in a network. It is calculated by taking the total number of edges connected to a node and dividing it by the maximum possible number of edges for that node.

Mathematically, degree centrality of node i in a network G can be expressed as:

$C_d(i) = degree(i) / (n-1)$

where degree(i) is the number of edges connected to node i, n is the total number of nodes in the network, and (n-1) is the maximum possible number of edges for node i.

2. **Betweenness Centrality**: Betweenness centrality is a measure of how often a node acts as a bridge or intermediary between other nodes in the network. It is calculated by finding the shortest path between every pair of nodes in the network and then determining the proportion of those paths that pass through a given node.

Mathematically, betweenness centrality of node i in a network G can be expressed as:

$C_b(i) = \Sigma(j,k)\ \sigma(j,k|i) / \sigma(j,k)$

where $\sigma(j,k)$ is the total number of shortest paths between nodes j and k, and $\sigma(j,k|i)$ is the number of those paths that pass through node i.

3. **Eigenvector Centrality:** Eigenvector centrality is a measure of the influence of a node in a network based on the centrality of its neighbors. It is calculated using the eigenvectors of the adjacency matrix of the network.

Mathematically, eigenvector centrality of node i in a network G can be expressed as:

$C_e(i) = (1/\lambda)\ \Sigma(j \in N(i))\ A_{ij}\ C_e(j)$

where λ is the largest eigenvalue of the adjacency matrix of the network, N(i) is the set of neighbors of node i, A_{ij} is the element of the adjacency matrix indicating whether nodes i and j are connected, and $C_e(j)$ is the eigenvector centrality of node j.

4. **Closeness Centrality**: Closeness centrality is a measure of how easily a node can access other nodes in a network. It is calculated by finding the shortest path between a node and all other nodes in the network, and then summing up the lengths of those paths. The reciprocal of this sum is the closeness centrality.

Mathematically, closeness centrality of node i in a network G can be expressed as:

$C_c(i) = 1 / \Sigma(j)\ d(i,j)$

where d(i,j) is the length of the shortest path between nodes i and j.

5. **PageRank Centrality:** PageRank centrality is a measure of the importance of a node in a network based on the concept of random walks. It is calculated by considering a random surfer moving through the network, who follows links from one node to another with a probability determined by the weight of the links. Nodes with more incoming links and/or links from more important nodes are assigned a higher PageRank score.

Mathematically, PageRank centrality of node i in a network G can be expressed as:

$C_p(i) = (1-d) / N + d \Sigma(j \in B_i) A_{ji} C_p(j) / d_j$

where N is the total number of nodes in the network, d is a damping factor that determines the probability of the surfer following a link, B_i is the set of nodes that link to node i, A_{ji} is the element of the adjacency matrix indicating whether nodes j and i are connected, $C_p(j)$ is the PageRank centrality of node j, and d_j is the out-degree of node j.

- **Calculating the centrality measures for directed graph**

```
print("Finding centrality measures for Directed Graph ")
# 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_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_directed.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(headers)
    writer.writerows(table_data)

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

```
Finding centrality measures for Directed Graph
```

Node	Degree Centrality	Betweenness Centrality	Closeness Centrality	PageRank Centrality	Eigenvalue Centrality
1	0.67	0.04	0.15	0.04	0
2	0.78	0.08	0.56	0.15	0.27
4	0.44	0.09	0.44	0.15	0.53
5	0.33	0.06	0.39	0.17	0.53
6	0.33	0	0.45	0.22	0.53
7	0.33	0.01	0.25	0.05	0
3	0.44	0.02	0.35	0.1	0.27
8	0.44	0.05	0.11	0.05	0
9	0.33	0	0	0.03	0
10	0.33	0.01	0.11	0.04	0

```
Saved centrality measures to Centrality_Directed_Graph.csv
```

- **Analysing node with highest centrality for all centrality measures**

```
# 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)
```

```
Node with highest Degree Centrality:  2
Node with highest Betweenness Centrality:  4
Node with highest Closeness Centrality:  2
Node with highest PageRank Centrality:  6
Node with highest Eigenvalue Centrality:  4
```

- **Analysing node with highest centrality score for all centrality measures**

```
print("Finding highest centrality score and node for all the centrality measures in an Directed Graph :  ")
# 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_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()}
eigenvector_centrality = {node: round(value, 2) for node, value in eigenvector_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 value == max_degree]
min_degree = min(degree_centrality.values())
min_degree_nodes = [node for node, value in degree_centrality.items() if value == 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() if value == max_closeness]
min_closeness = min(closeness_centrality.values())
min_closeness_nodes = [node for node, value in closeness_centrality.items() if value == min_closeness]

max_page_rank = max(page_rank_centrality.values())
max_page_rank_nodes = [node for node, value in page_rank_centrality.items() if 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() if 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 Centrali
```

```
Finding highest centrality score and node for all the centrality measures in an Directed Graph :
Centrality              Highest Centrality Score  Highest Centrality Nodes     lowest Centrality Score  lowest Centrality Nodes
----------------------  ------------------------  -------------------------  -------------------------  -------------------------
Degree Centrality                           0.78  ['2']                                           0.33  ['5', '6', '7', '9', '10']
Betweenness Centrality                      0.09  ['4']                                           0     ['6', '9']
Closeness Centrality                        0.56  ['2']                                           0     ['9']
PageRank Centrality                         0.22  ['6']                                           0.03  ['9']
Eigenvector Centrality                      0.53  ['4', '5', '6']                                 0     ['1', '7', '8', '9', '10']
```

- **Calculating the centrality measures for Un- directed graph**

```python
print("Finding centrality measures for Un-Directed Graph ")

# 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")
```

```
Finding centrality measures for Un-Directed Graph
  Node    Degree Centrality    Betweenness Centrality    Closeness Centrality    PageRank Centrality    Eigenvalue Centrality
------  -------------------  ------------------------  ----------------------  ---------------------  -----------------------
     1                 0.67                      0.29                    0.75                    0.16                     0.45
     2                 0.67                      0.29                    0.75                    0.16                     0.45
     4                 0.44                      0.11                    0.64                    0.1                      0.34
     5                 0.33                      0.02                    0.53                    0.08                     0.25
     6                 0.33                      0.01                    0.5                     0.06                     0.23
     7                 0.33                      0.02                    0.53                    0.08                     0.25
     3                 0.33                      0.02                    0.53                    0.08                     0.25
     8                 0.44                      0.11                    0.64                    0.11                     0.34
     9                 0.33                      0.02                    0.53                    0.1                      0.25
    10                 0.33                      0.01                    0.5                     0.07                     0.23
Saved centrality measures to centrality_undirected.csv
```

- **Analysing node with highest centrality score and node for all centrality measures**

```
print("Finding highest centrality score and node for all the centrality measures in an Un-directed graph :  ")

# 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_centrality.items()}
eigenvector_centrality = {node: round(value, 2) for node, value in eigenvector_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 value == max_degree]
min_degree = min(degree_centrality.values())
min_degree_nodes = [node for node, value in degree_centrality.items() if value == 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() if value == max_closeness]
min_closeness = min(closeness_centrality.values())
min_closeness_nodes = [node for node, value in closeness_centrality.items() if value == min_closeness]

max_page_rank = max(page_rank_centrality.values())
max_page_rank_nodes = [node for node, value in page_rank_centrality.items() if 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() if 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']))
```

```
Finding highest centrality score and node for all the centrality measures in an Un-directed graph :
Centrality              Highest Centrality Score  Highest Centrality Nodes     lowest Centrality Score  lowest Centrality Nodes
--------------------  --------------------------  --------------------------  -------------------------  ----------------------------
Degree Centrality                           0.67  ['1', '2']                                      0.33  ['5', '6', '7', '3', '9', '10']
Betweenness Centrality                      0.29  ['1', '2']                                      0.01  ['6', '10']
Closeness Centrality                        0.75  ['1', '2']                                      0.5   ['6', '10']
PageRank Centrality                         0.16  ['1', '2']                                      0.06  ['6']
Eigenvector Centrality                      0.45  ['1', '2']                                      0.23  ['6', '10']
```

Appendix

```
!pip install --upgrade networkx
import networkx as nx
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import csv

# Open the CSV file and read the edges
with open('edges_list.csv', 'r') as f:
reader = csv.reader(f)
next(reader)
edges = [(int(row[0]), int(row[1]), float(row[2])) for row in reader]
# Read the CSV file into a pandas dataframe
df = pd.read_csv('edges_list.csv')
# Print the entire dataframe
print(df)

# Create an empty directed graph
G = nx.DiGraph()
# Add the edges to the graph with weights
G.add_weighted_edges_from(edges)
# Define a list of colors for the nodes
node_colors = ['red', 'blue', 'green', 'orange', 'purple']
plt.figure(figsize=(8, 8))
# Draw the graph with edge weights
pos = nx.spring_layout(G) # positions for all nodes
nx.draw_networkx_nodes(G, pos, node_size=400,node_color='green')
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2],
arrowsize=20,arrows=True, arrowstyle='->, head_width=0.2, head_length=0.3')
nx.draw_networkx_labels(G, pos, font_size=10, font_family='monospace')
plt.title('22MCB0012 : Directed Graph')
plt.axis('off')
plt.savefig('Directed_Graph.png')
```

```
plt.show()


import networkx as nx
import matplotlib.pyplot as plt
import csv
# Open the CSV file and read the edges with weights
with open('edges_list.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=400,node_color='green')
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2])
nx.draw_networkx_labels(G, pos, font_size=10, font_family='monospace')
plt.title('22MCB0012 : Un-directed Graph')
plt.axis('off')
plt.savefig('Undirected_Graph.png')
plt.show()


import networkx as nx
import matplotlib.pyplot as plt
import csv
# Open the CSV file and read the edges with weights
with open('edges_list.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 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=400,node_color='green')
# 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=[1],
arrowsize=20, arrowstyle='->, head_width=0.2, head_length=0.3')
edge_labels = {(u, v): f'{int(d["weight"])}' for (u, v, d) in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=10)
# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=10, font_family='monospace')
plt.title('22MCB0012 : Directed Graph with Weights')
# Turn off the axis
plt.axis('off')
# Show the plot
plt.show()

import networkx as nx
import matplotlib.pyplot as plt
import csv
# Open the CSV file and read the edges
with open('edges_list.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,node_color='green')
# 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='monospace')
plt.title('22MCB0012 : Un-directed Graph with Weights')
# Turn off the axis
plt.axis('off')
# Show the plot
plt.show()

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}')
print()
```

```
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}')
```

```
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_list.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,node_color='green')
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='monospace')
plt.title('22MCB0012 : Un-directed Graph')
# Turn off the axis
plt.axis('off')
# Show the plot
plt.show()
print('Adjancency Matrix for Un-directed Graph :')
# Adjacency Matrix (unweighted)
adj_matrix = nx.adjacency_matrix(G)
print(adj_matrix.todense())

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_list.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=400,node_color='green')
# 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=10)
# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=10, font_family='monospace')
plt.title('22MCB0012 : Un-directed Graph with weights')
# Turn off the axis
plt.axis('off')
# Show the plot
plt.show()
print('Adjancency Matrix for Un-directed Graph with weights :')
# Adjacency Matrix (weighted)
adj_matrix = nx.adjacency_matrix(G, weight='weight')
print(adj_matrix.todense())


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_list.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=400,node_color='green')
nx.draw_networkx_edges(G, pos, edgelist=G.edges(),
arrowsize=20, arrowstyle='->, head_width=0.2, head_length=0.3')
# 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='monospace')
plt.title('22MCB0012 : Directed Graph')
# Turn off the axis
plt.axis('off')
# Show the plot
plt.show()
print('Adjancency Matrix for Directed Graph :')
# Adjacency Matrix (unweighted directed)
adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,11))
print(adj_matrix.todense())



# 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)

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_list.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=400, node_color='green')
# 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=0.2, head_length=0.3')
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=10)
# Label the nodes
node_labels = {n: str(n) for n in G.nodes()}
```

```
nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=10, font_family='monospace')
plt.title('22MCB0012 : Directed Graph with weights')
# Turn off the axis
plt.axis('off')
# Show the plot
plt.show()
print('Adjancency Matrix for Directed Graph with weights :')
# Adjacency Matrix (weighted directed)
adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,11))
adj_matrix = nx.adjacency_matrix(G, weight='weight')
print(adj_matrix.todense())
import csv
import networkx as nx
from tabulate import tabulate
# create the graph from the edges.csv file
G = nx.DiGraph()
with open('edges_list.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))
print("Finding centrality measures for Directed Graph ")
# 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_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_val

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_Graph.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)


import csv

import networkx as nx

from tabulate import tabulate

# create the graph from the edges.csv file

G = nx.DiGraph()
```

```
with open('edges_list.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))
print("Finding highest centrality score and node for all the centrality measures in an Directed Graph : ")
# 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_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()}
eigenvector_centrality = {node: round(value, 2) for node, value in eigenvector_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 value == max_degree]
min_degree = min(degree_centrality.values())
min_degree_nodes = [node for node, value in degree_centrality.items() if value == 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() if value == max_closeness]
min_closeness = min(closeness_centrality.values())
min_closeness_nodes = [node for node, value in closeness_centrality.items() if value == min_closeness]
```

```
max_page_rank = max(page_rank_centrality.values())
max_page_rank_nodes = [node for node, value in page_rank_centrality.items() if 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() if 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’]))


import csv
import networkx as nx
from tabulate import tabulate
# create the graph from the edges.csv file
G = nx.Graph()
with open('edges_list.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))
print("Finding centrality measures for Un-Directed Graph ")
# 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_val

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")
```

```
Saved centrality measures to centrality_undirected.csv
```

```
import·csv

import·networkx·as·nx
```

```
from·tabulate·import·tabulate

#·create·the·graph·from·the·edges.csv·file

G·=·nx.Graph()

with·open('edges_list.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))

print("Finding·highest·centrality·score·and·node·for·all·the·centrality·measures·in·an·Un-directed·graph·:··")

#·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_centrality.items()}

eigenvector_centrality·=·{node:·round(value,·2)·for·node,·value·in·eigenvector_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·value·==·max_degree]

min_degree·=·min(degree_centrality.values())

min_degree_nodes·=·[node·for·node,·value·in·degree_centrality.items()·if·value·==·min_degree]

max_betweenness·=·max(betweenness_centrality.values())

max_betweenness_nodes·=·[node·for·node,·value·in·betweenness_centrality.items()·if·value·==·max_between
ness]

min_betweenness·=·min(betweenness_centrality.values())

min_betweenness_nodes·=·[node·for·node,·value·in·betweenness_centrality.items()·if·value·==·min_betweenn
ess]

max_closeness·=·max(closeness_centrality.values())
```

```
max_closeness_nodes·=·[node·for·node,·value·in·closeness_centrality.items()·if·value·==·max_closeness]

min_closeness·=·min(closeness_centrality.values())

min_closeness_nodes·=·[node·for·node,·value·in·closeness_centrality.items()·if·value·==·min_closeness]

max_page_rank·=·max(page_rank_centrality.values())

max_page_rank_nodes·=·[node·for·node,·value·in·page_rank_centrality.items()·if·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()·if·value·==·min_page_rank]

max_eigenvector·=·max(eigenvector_centrality.values())

max_eigenvector_nodes·=·[node·for·node,·value·in·eigenvector_centrality.items()·if·value·==·max_eigenvecto
r]

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_betweennes
s_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_no
des]

]

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