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

Social Network Analysis LAB
Assessment - 1 REPORT

Faculty Name: Durgesh Kumar

Student Name: Sarthak Tripathi
Reg. No.: 22MCB0030
E-mail: sarthak.tripathi2022@vitstudent.ac.in
Mobile No.: 8630054046

Table of Contents

Overview

Graph

In the context of social network analysis, a graph is a visual representation of a social network, which consists of a set of actors (also called nodes or vertices) and their connections (also called ties or edges). Graphs can be used to model and analyse various aspects of social networks, such as the structure of relationships between actors, the flow of information or resources, and the formation of groups or communities.

In a graph, each actor is represented by a node, which is typically depicted as a point or circle, and each connection between actors is represented by an edge, which is typically depicted as a line or arrow. The edges may be directed (indicating the direction of the relationship) or undirected (indicating a mutual relationship).

Graphs can be analysed using various measures and algorithms, such as degree centrality (which measures the number of connections an actor has), betweenness centrality (which measures the extent to which an actor lies on the shortest paths between other actors), and clustering coefficient (which measures the degree to which actors tend to form groups or clusters). Graphs can also be visualized using various layouts and styles to highlight different aspects of the social network.

Types of Graphs:

The main types of graphs are:

1) Directed Graphs: Also known as digraphs, these graphs have edges with a direction indicating a specific relationship between two nodes. For example, in a social network where A follows B on a social media platform, there is a directed edge from A to B, indicating that A follows B. Directed graphs can be used to represent asymmetric relationships between nodes.
2) Undirected Graphs: These graphs have edges without a direction indicating that the relationship between two nodes is symmetrical or reciprocal. For example, in a social network where A and B are friends on a social media platform, there is an undirected edge between A and B, indicating that they are friends. Undirected graphs can be used to represent symmetric relationships between nodes.

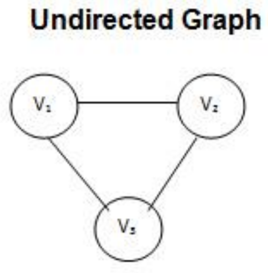


Figure 1: An Undirected Graph

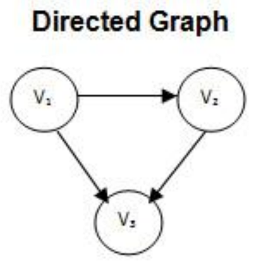


Figure 2: A Directed Graph

3) Weighted Graphs: These graphs have edges with weights that represent the strength, intensity, or frequency of the relationship between two nodes. For example, in a social network where A and B have interacted frequently on a social media platform, the weight of the edge between A and B may be high, indicating a strong relationship. Weighted graphs can be used to represent the strength or importance of relationships between nodes.
4) Unweighted Graphs: These graphs have edges without weights, indicating that all relationships between nodes have equal importance. For example, in a social network where A and B are connected on a social media platform, the edge between A and B may not have a weight, indicating that their relationship is not stronger or weaker than any other relationship in the network. Unweighted graphs can be used to represent the topology or structure of a social network without considering the strength or importance of relationships between nodes.

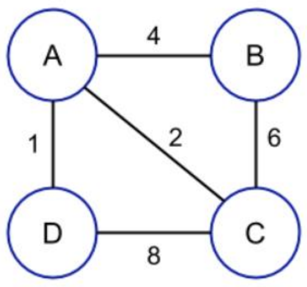


Figure: Weighted Graph
(also weighted undirected graph)

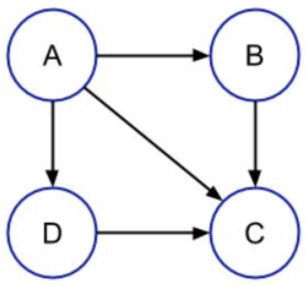


Figure: Unweighted Graph
(also unweighted directed graph)

Libraries Used

1) NumPy: NumPy (Numerical Python) is a Python library that is widely used for scientific computing and data analysis. It provides various data structures, functions, and tools for working with large multi-dimensional arrays and matrices. NumPy is particularly useful for performing mathematical operations and statistical analysis on large datasets, making it an essential tool for data scientists, engineers, and researchers.

   Social Network Analysis (SNA) is a field of study that analyses social networks using mathematical and computational techniques. SNA can be used to study various social structures, such as communication patterns, information flow, and influence networks. NumPy can be used in SNA to perform various tasks, such as data cleaning, graph generation, and statistical analysis. For example, NumPy can be used to generate adjacency matrices, which represent the connections between nodes in a social network. These matrices can then be analysed using various statistical techniques to identify patterns and relationships within the network. NumPy can also be used to generate visualizations of social networks, which can be useful for understanding the structure and dynamics of the network.

2) Pandas: Pandas is a popular open-source library for data manipulation and analysis in Python. It provides powerful tools for data cleaning, structuring, and analysis, making it a valuable tool for data scientists, analysts, and researchers. Pandas is particularly useful for working with tabular and time-series data, which makes it a natural fit for many applications in social network analysis.

   Pandas can be used in SNA to perform various tasks such as data cleaning, pre-processing, and analysis. For example, Pandas can be used to import, clean, and pre-process data from various sources, including CSV files, JSON files, and databases. Pandas can also be used to perform exploratory data analysis, such as calculating summary statistics, identifying outliers, and visualizing data. Additionally, Pandas can be used to prepare data for machine learning models, such as clustering and classification algorithms, which can be used to analyse social networks and identify patterns and relationships between nodes.

3) NetworkX: NetworkX is a Python package that provides tools for the creation, manipulation, and analysis of complex networks. It is a popular tool for social network analysis (SNA) because it provides various algorithms and data structures that are specifically designed for network analysis tasks. NetworkX provides an intuitive interface for creating and analysing networks, making it a valuable tool for researchers and data scientists who want to explore the structure and dynamics of social networks.

   NetworkX can be used in SNA to perform various tasks, such as generating and visualizing graphs, calculating network statistics, and identifying patterns and structures within the network. For example, NetworkX can be used to calculate various network measures, such as degree centrality, betweenness centrality, and clustering coefficient. These measures can be used to identify important nodes in the network, such as key influencers or hubs. NetworkX can also be used to generate visualizations of social networks, which can be used to gain insights into the structure and dynamics of the network. Additionally, NetworkX provides algorithms for community detection and link prediction, which can be used to identify sub-communities within the network and predict future connections between nodes. In summary, NetworkX is a powerful tool for the creation, manipulation, and analysis of complex networks, and it can be used in SNA to perform various tasks related to network analysis. By leveraging the capabilities of NetworkX, researchers and data scientists can gain valuable insights into the structure and dynamics of social networks and identify important nodes and patterns within the network.

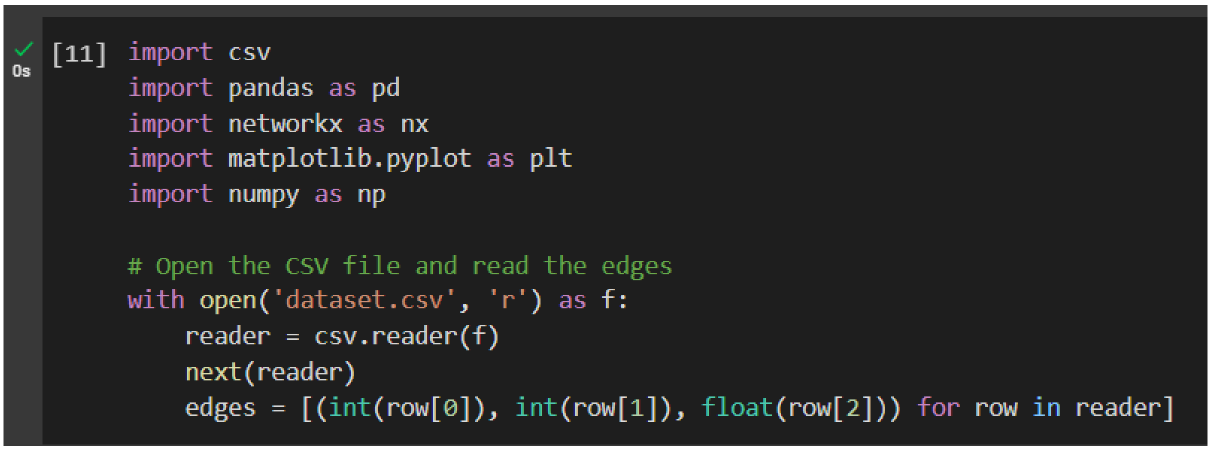
4) Matplotlib: Matplotlib is a popular data visualization library for Python that provides various tools for creating high-quality plots, charts, and graphs. It is particularly useful for social network analysis (SNA) because it allows data scientists and researchers to visualize the structure and dynamics of social networks in a clear and concise way. Matplotlib provides a range of customizable visualization options, making it a valuable tool for exploring and presenting network data.

   Matplotlib can be used in SNA to create various types of network visualizations, such as node-link diagrams, matrix plots, and heat maps. These visualizations can be used to represent the structure and dynamics of social networks, such as communication patterns, information flow, and influence networks. For example, node-link diagrams can be used to represent the connections between nodes in a network, while matrix plots can be used to visualize the strength of connections between nodes. Heat maps can be used to identify patterns and clusters within the network, such as sub-communities or groups of highly connected nodes. Matplotlib also provides options for customizing the appearance of visualizations, such as changing colour schemes, adding labels, and adjusting layout.

5) Tabulate: Tabulate is a Python library that provides a simple way to create formatted tables in various output formats. It is particularly useful in social network analysis (SNA) for presenting tabular data in a structured and easy-to-read format. Tabulate can be used to format data such as node attributes, edge weights, and other metrics for analysis of social network data.

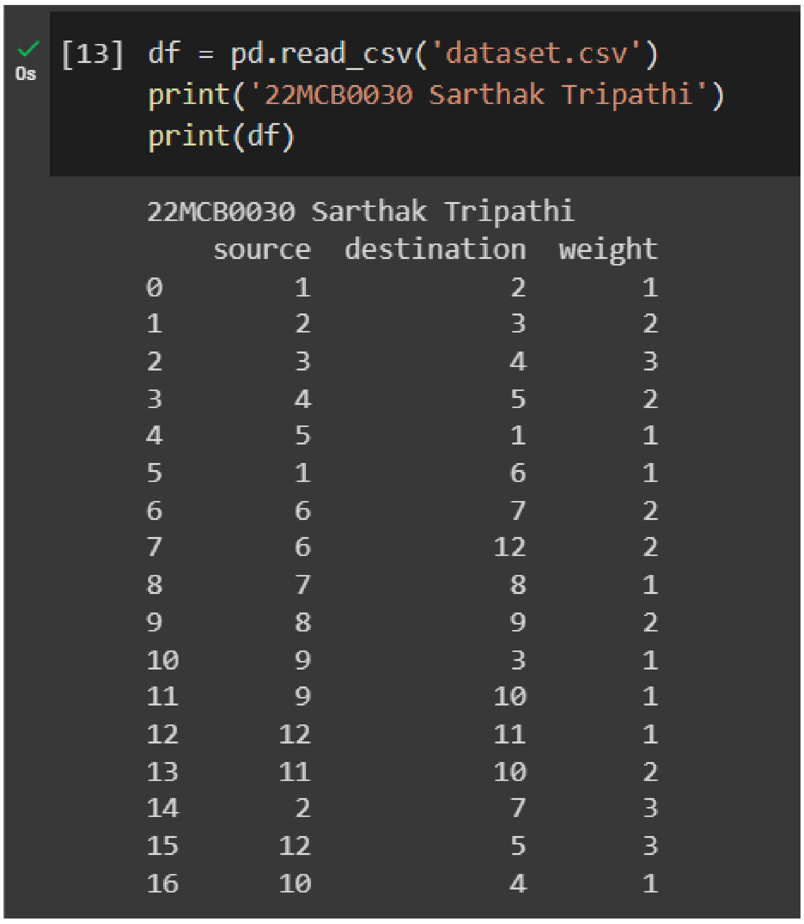
   Tabulate provides several options for customizing the appearance of tables, such as changing the alignment, adding headers, and specifying column widths. This makes it easy to present social network data in a clear and organized manner. In addition, tabulate supports several output formats, including plain text, HTML, and LaTeX, which makes it possible to create high-quality tables that can be easily shared and incorporated into research papers, reports, and presentations. Overall, Tabulate is a useful tool for any data scientist, researcher, or analyst working with social network data, as it provides an efficient and customizable way to present and share data in a structured format.

Step 1: Importing Libraries



```
[11] import csv
     import pandas as pd
     import networkx as nx
     import matplotlib.pyplot as plt
     import numpy as np

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

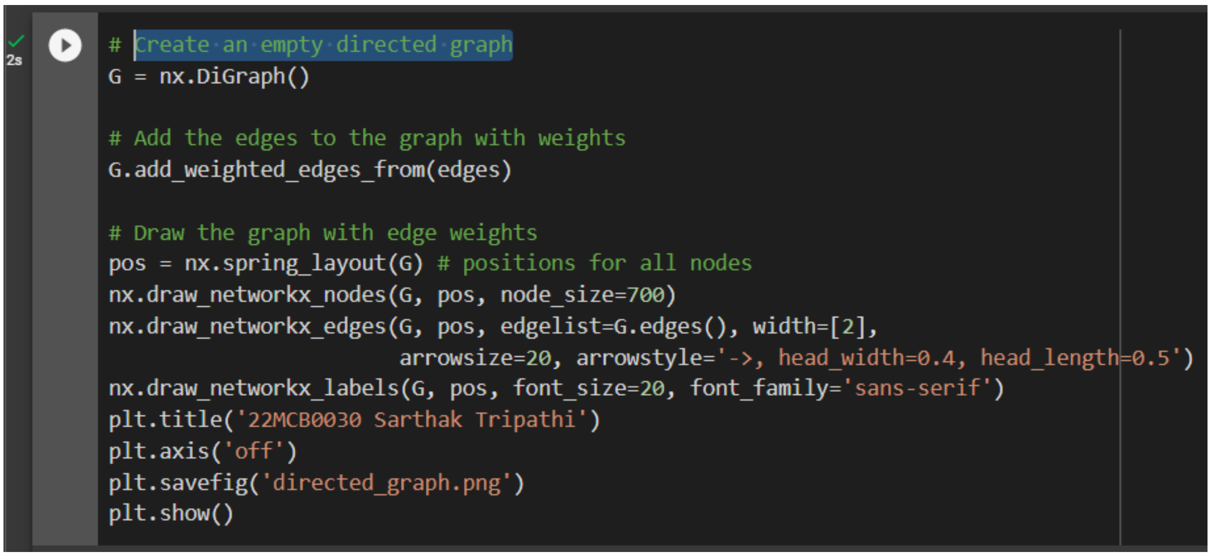
Step 2: Show the description of the dataset taken:



```
[13] df = pd.read_csv('dataset.csv')
     print('22MCB0030 Sarthak Tripathi')
     print(df)

     22MCB0030 Sarthak Tripathi
         source  destination  weight
     0        1            2       1
     1        2            3       2
     2        3            4       3
     3        4            5       2
     4        5            1       1
     5        1            6       1
     6        6            7       2
     7        6           12       2
     8        7            8       1
     9        8            9       2
     10       9            3       1
     11       9           10       1
     12      12           11       1
     13      11           10       2
     14       2            7       3
     15      12            5       3
     16      10            4       1
```

PART- 1

Create an empty directed graph



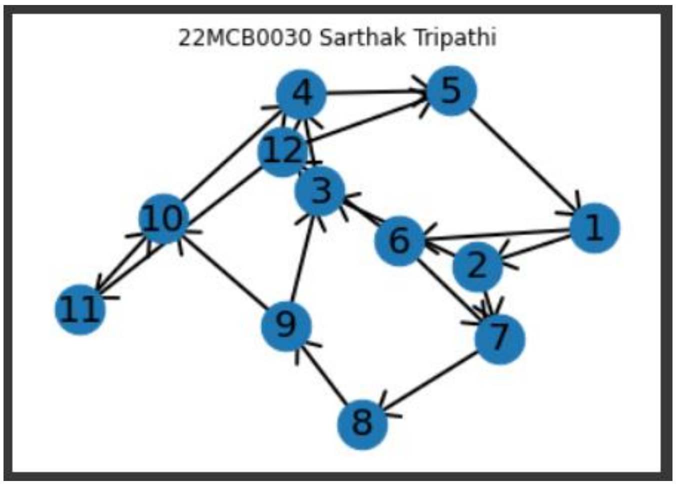
```
# Create an empty directed graph
G = nx.DiGraph()

# 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.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.savefig('directed_graph.png')
plt.show()
```

Explanation of the code: The above code creates an empty directed graph using the NetworkX library, adds weighted edges to it, and then visualizes it using Matplotlib. The edges are represented as arrows with different thicknesses based on their weights.

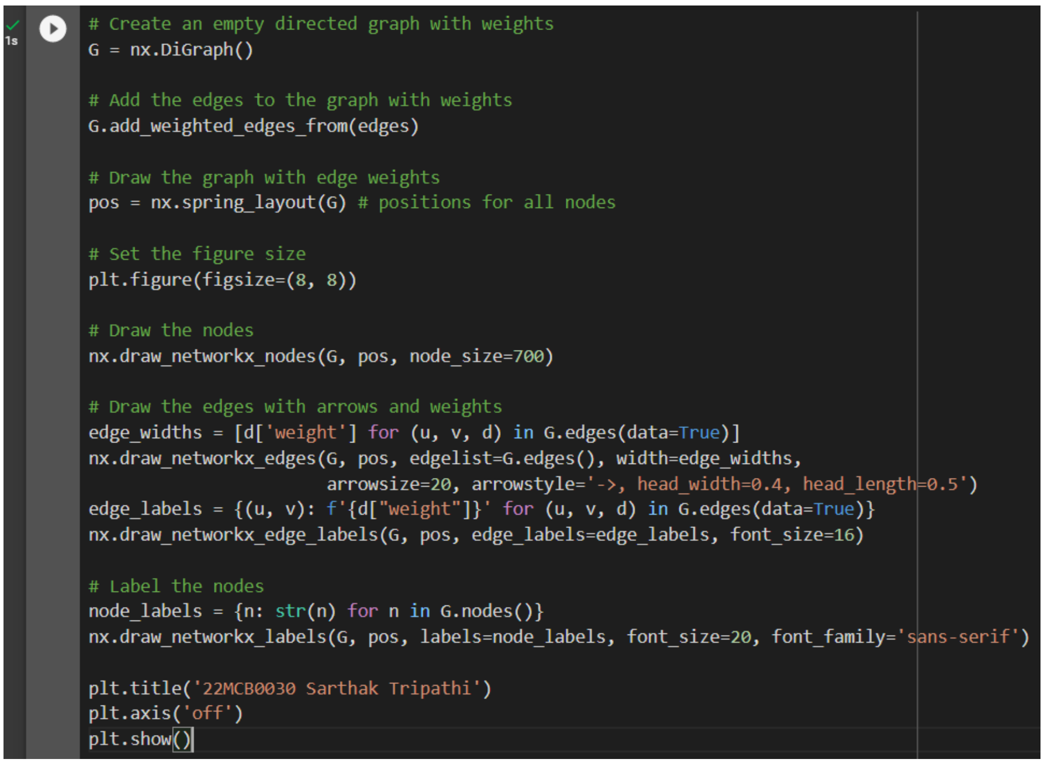
The code first creates an empty directed graph using the nx.DiGraph() function from the NetworkX library. The edges are then added to the graph using the G.add_weighted_edges_from(edges) function, where edges is a list of tuples representing the edges and their weights. Next, the code uses the nx.spring_layout() function to determine the positions of the nodes in the graph. The nx.draw_networkx_nodes(), nx.draw_networkx_edges(), and nx.draw_networkx_labels() functions are then used to draw the nodes, edges, and labels of the graph, respectively. The width argument of the nx.draw_networkx_edges() function is set to a list of values representing the thickness of the edges based on their weights.

Finally, the code turns off the axis labels using plt.axis('off'), saves the graph as an image using plt.savefig(), and displays it using plt.show().

OUTPUT:



Create an empty directed graph with weights:



```
# Create an empty directed graph with weights
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')

plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.show()
```

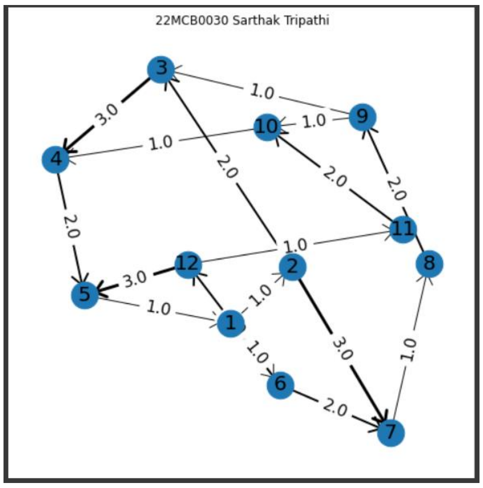
Explanation of the Code: The code first creates an empty directed graph using the nx.DiGraph() function from the NetworkX library. The edges are then added to the graph using the G.add_weighted_edges_from(edges) function, where edges is a list of tuples representing the edges and their weights.

Next, the code uses the nx.spring_layout() function to determine the positions of the nodes in the graph. The plt.figure(figsize=(8, 8)) function is then used to set the size of the figure to be displayed.

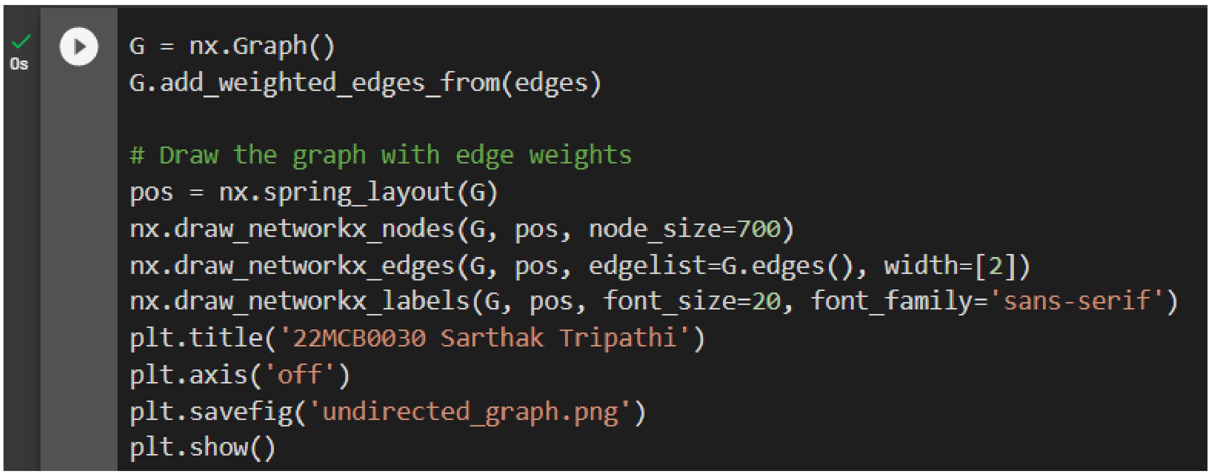
The nodes are drawn using the nx.draw_networkx_nodes() function with a specified node size. The edges are then drawn using the nx.draw_networkx_edges() function. The thickness of each edge is set to its corresponding weight using a list comprehension. The arrowstyle argument is set to '->, head_width=0.4, head_length=0.5' to indicate that the edges should be drawn as arrows.

The edge labels are drawn using the nx.draw_networkx_edge_labels() function. The edge_labels argument is set to a dictionary comprehension that maps each edge to its weight. The node labels are drawn using the nx.draw_networkx_labels() function. The labels argument is set to a dictionary comprehension that maps each node to its label.

OUTPUT:



Creates an undirected graph (without Weights):



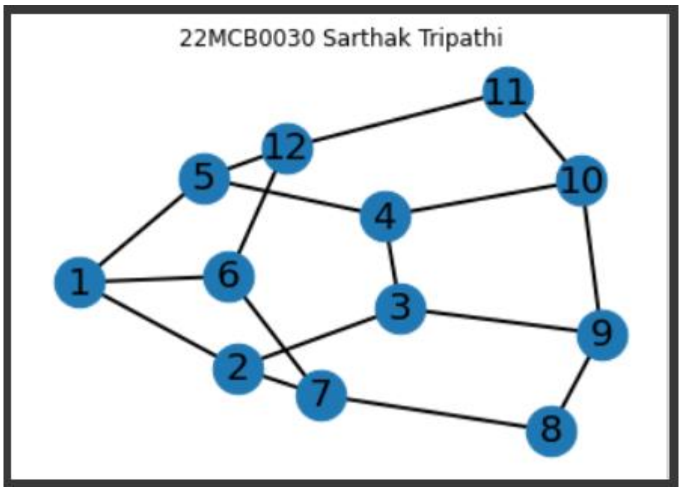
```
G = nx.Graph()
G.add_weighted_edges_from(edges)

# Draw the graph with edge weights
pos = nx.spring_layout(G)
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.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.savefig('undirected_graph.png')
plt.show()
```

Explanation of Code: The code first creates an empty undirected graph using the nx.Graph() function from the NetworkX library. The edges are then added to the graph using the G.add_weighted_edges_from(edges) function, where edges is a list of tuples representing the edges and their weights.

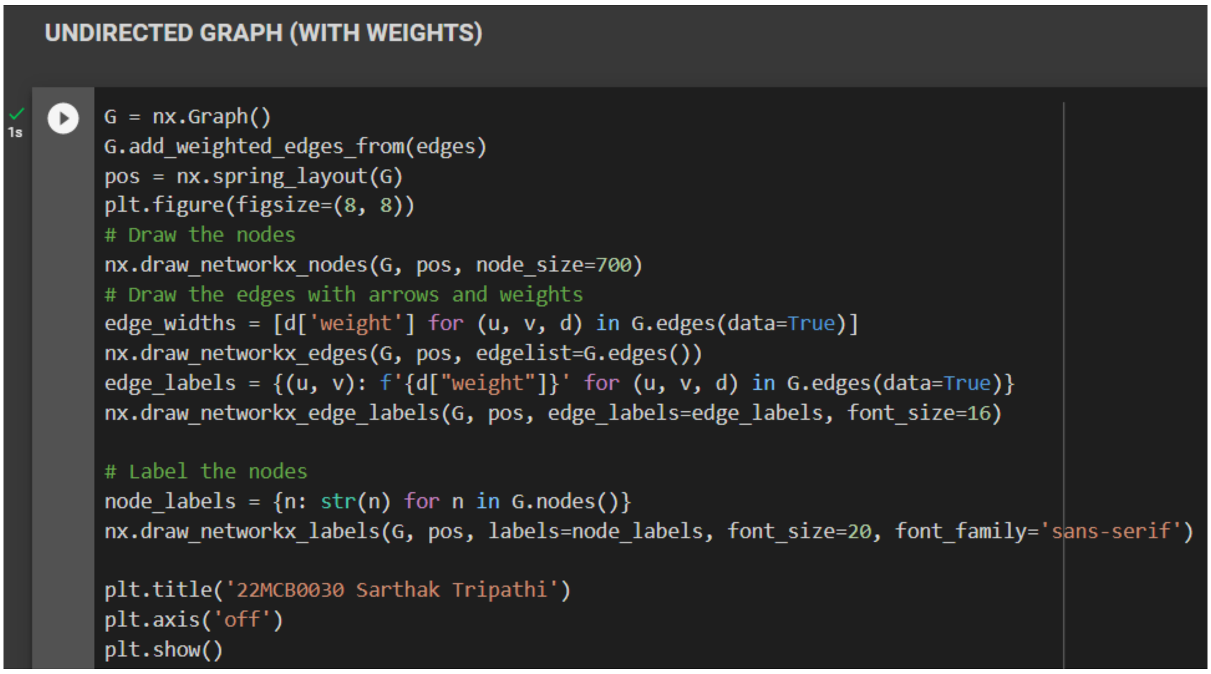
Next, the code uses the nx.spring_layout() function to determine the positions of the nodes in the graph. The nx.draw_networkx_nodes() function is then used to draw the nodes, with a specified node size. The nx.draw_networkx_edges() function is used to draw the edges, with a fixed thickness of 2. The edge list argument is set to G.edges() to indicate that all edges should be drawn.

The node labels are drawn using the nx.draw_networkx_labels() function. The font_size argument is set to 20 and the font_family argument is set to 'sans-serif'.

OUTPUT:



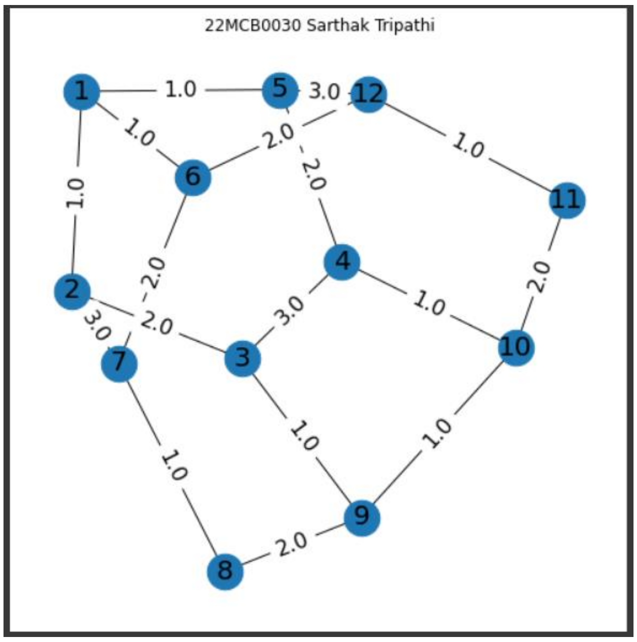
Creating an Undirected graph (with weights)


UNDIRECTED GRAPH (WITH WEIGHTS)

```
G = nx.Graph()
G.add_weighted_edges_from(edges)
pos = nx.spring_layout(G)
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')

plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.show()
```

Explanation of Code: This code creates an undirected graph using the networkx library and visualizes it using matplotlib. The graph is created by first initializing an empty graph using nx.Graph(), and then adding the edges to the graph with their respective weights using G.add_weighted_edges_from(edges).The graph is then drawn using the spring layout algorithm for node positioning, with the nodes being drawn as circles of size 700 using nx.draw_networkx_nodes(). The edges are drawn as lines using nx.draw_networkx_edges() with the edge widths set to a default value of 1. The node labels are also added using nx.draw_networkx_labels(). Finally, the plot is displayed using plt.show().Additionally, edge labels are added to the graph by using nx.draw_networkx_edge_labels(). The weights of the edges are stored in edge_labels as a dictionary with (source_node, target_node) tuples as keys and the weight as the value. The labels are then added to the graph using nx.draw_networkx_edge_labels(), which takes the graph, the node positions, and the edge labels as input.

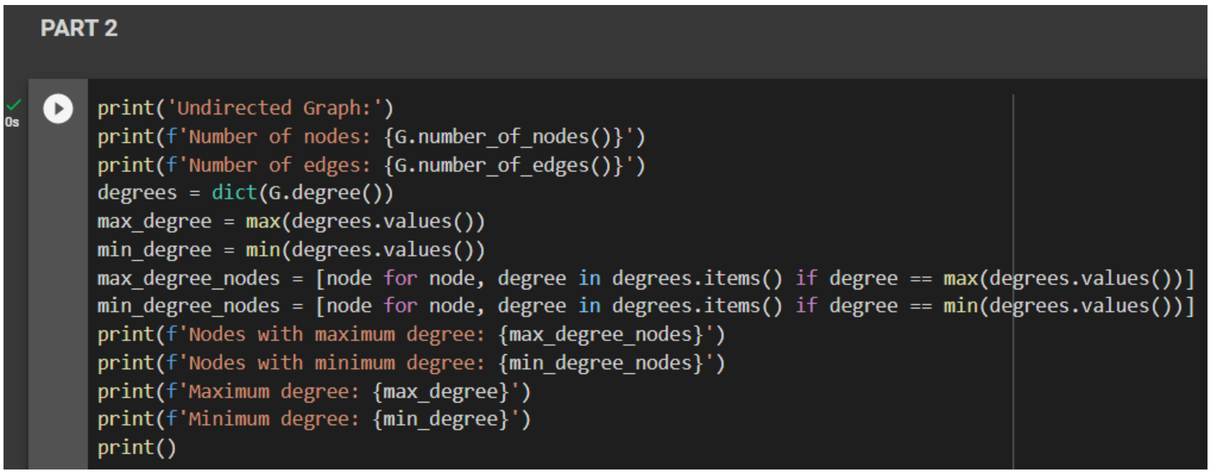
OUTPUT:



PART – 2

The degree of a node in an undirected graph is the number of edges incident on it; for directed graphs the indegree of a node is the number of edges leading into that node and its outdegree, the number of edges leading away from it.

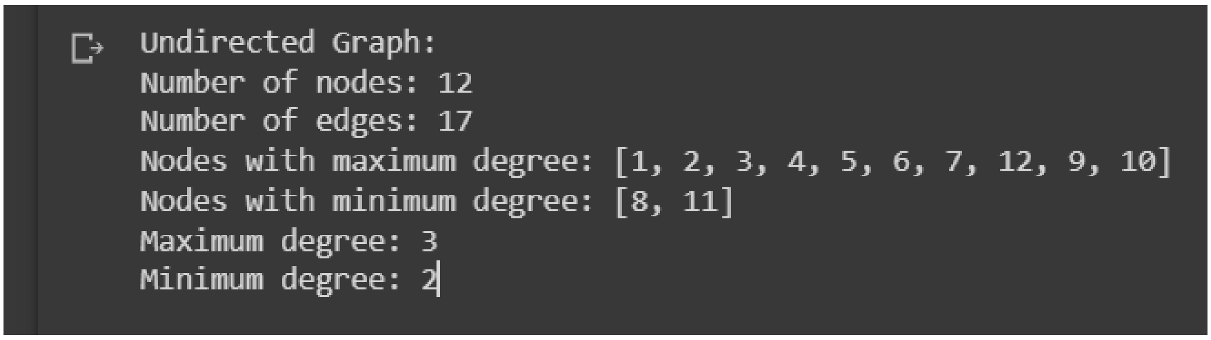
Finding the min and max degree for the nodes in Undirected Graph:



PART 2

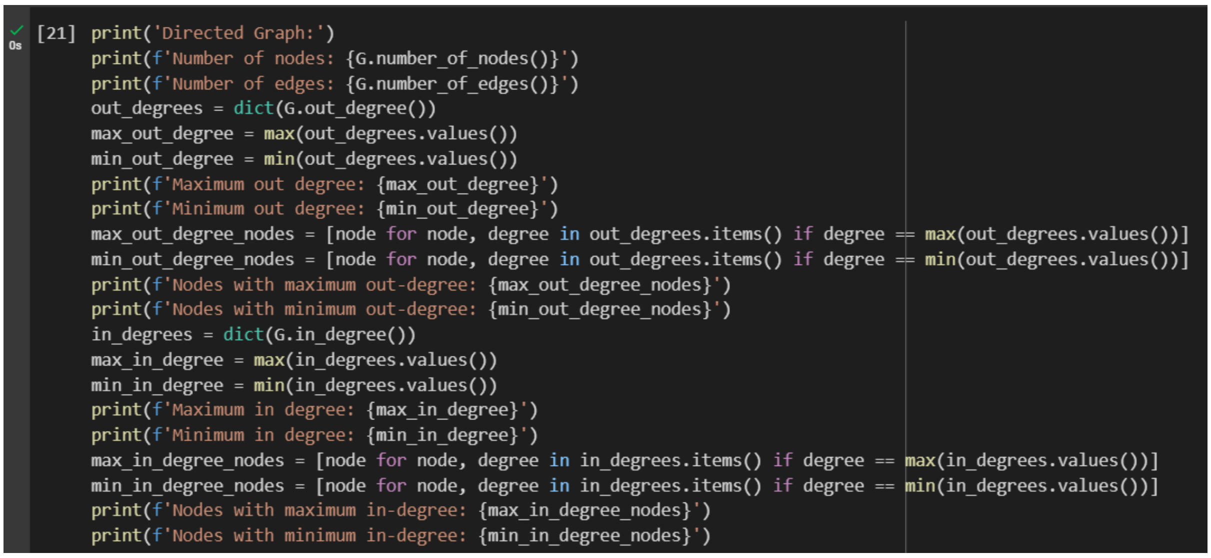
```
print('Undirected Graph:')
print(f'Number of nodes: {G.number_of_nodes()}')
print(f'Number of edges: {G.number_of_edges()}')
degrees = dict(G.degree())
max_degree = max(degrees.values())
min_degree = min(degrees.values())
max_degree_nodes = [node for node, degree in degrees.items() if degree == max(degrees.values())]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

OUTPUT:



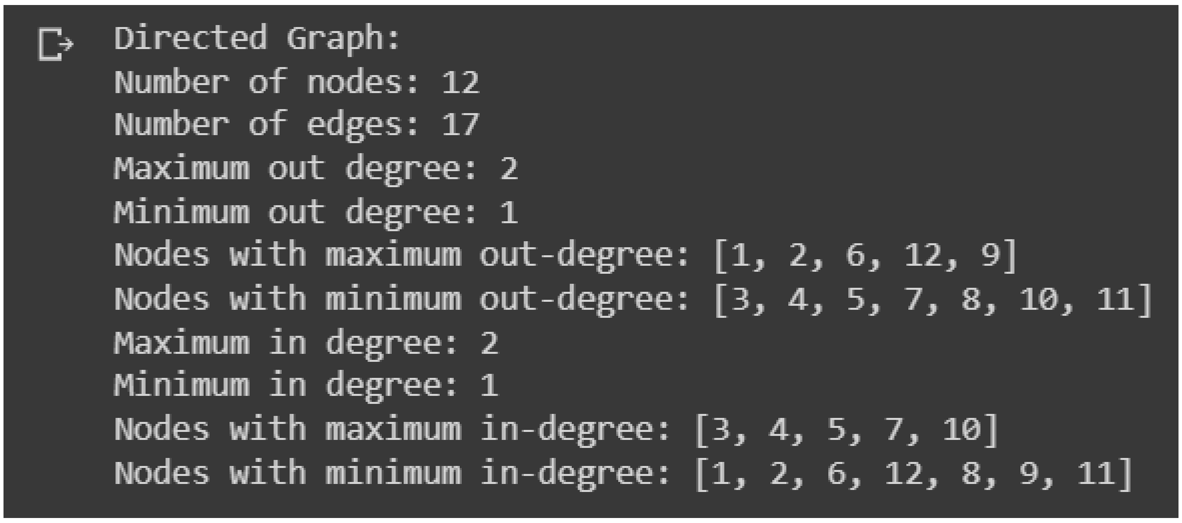
```
Undirected Graph:
Number of nodes: 12
Number of edges: 17
Nodes with maximum degree: [1, 2, 3, 4, 5, 6, 7, 12, 9, 10]
Nodes with minimum degree: [8, 11]
Maximum degree: 3
Minimum degree: 2
```

Finding the min and max Indegree and Outdegree for the nodes in directed Graph:



```
[21] 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}')
```

OTUPUT:



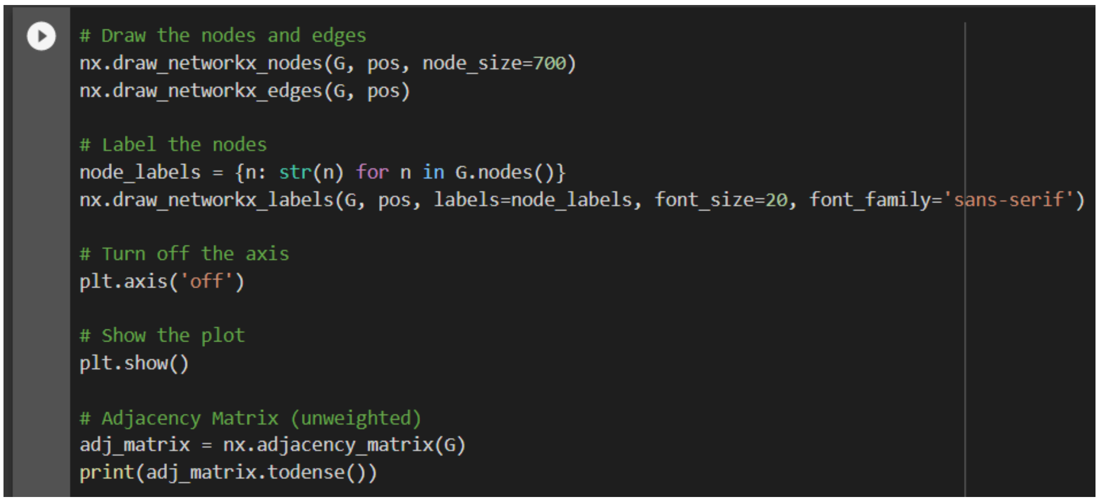
```
Directed Graph:
Number of nodes: 12
Number of edges: 17
Maximum out degree: 2
Minimum out degree: 1
Nodes with maximum out-degree: [1, 2, 6, 12, 9]
Nodes with minimum out-degree: [3, 4, 5, 7, 8, 10, 11]
Maximum in degree: 2
Minimum in degree: 1
Nodes with maximum in-degree: [3, 4, 5, 7, 10]
Nodes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

PART – 3

Adjacency Matrix: An adjacency matrix is a square matrix that represents a graph by showing the connections between nodes. In other words, it is a matrix in which the rows and columns represent the nodes of a graph, and the values in the matrix represent whether there is an edge connecting the nodes. If there is an edge between node i and node j, then the entry in the i-th row and j-th column (or j-th row and i-th column, if the graph is undirected) of the matrix will be non-zero, usually with a value of 1. If there is no edge between the nodes, then the entry will be 0.

The adjacency matrix can be used to perform various operations on graphs, such as calculating the degree of each node, finding paths between nodes, and identifying connected components. It can also be used for visualization and analysis purposes. For instance, the matrix can be used to calculate the number of paths of length k between two nodes in a graph, or to identify subgraphs with certain properties. Additionally, the matrix can be used to represent and analyse networks in various domains, including social networks, transportation networks, and biological networks.

Adjacency Matrix For Undirected Graph (without weights)



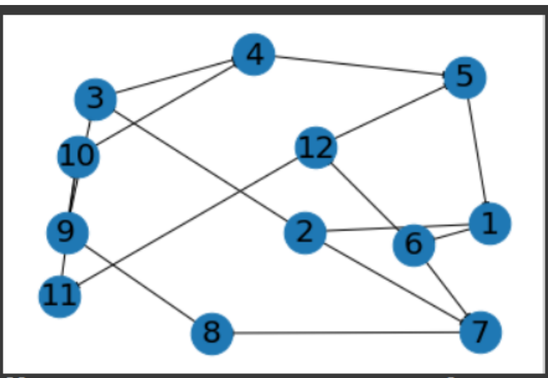
```
# Draw the nodes and edges
nx.draw_networkx_nodes(G, pos, node_size=700)
nx.draw_networkx_edges(G, pos)

# 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())
```

Explanation of the Code: This code first draws the nodes and edges of the graph using the networkx library. It then labels the nodes with their respective node numbers. Finally, it turns off the axis and shows the plot using the matplotlib library.

The last part of the code creates the adjacency matrix of the graph. The adjacency matrix is a square matrix of size N x N (where N is the number of nodes in the graph) and is used to represent the edges between nodes. The entry in the ith row and jth column of the matrix is 1 if there is an edge between node i and node j, and 0 otherwise. In this case, the adjacency matrix is unweighted, meaning that all edges are considered to have a weight of 1.

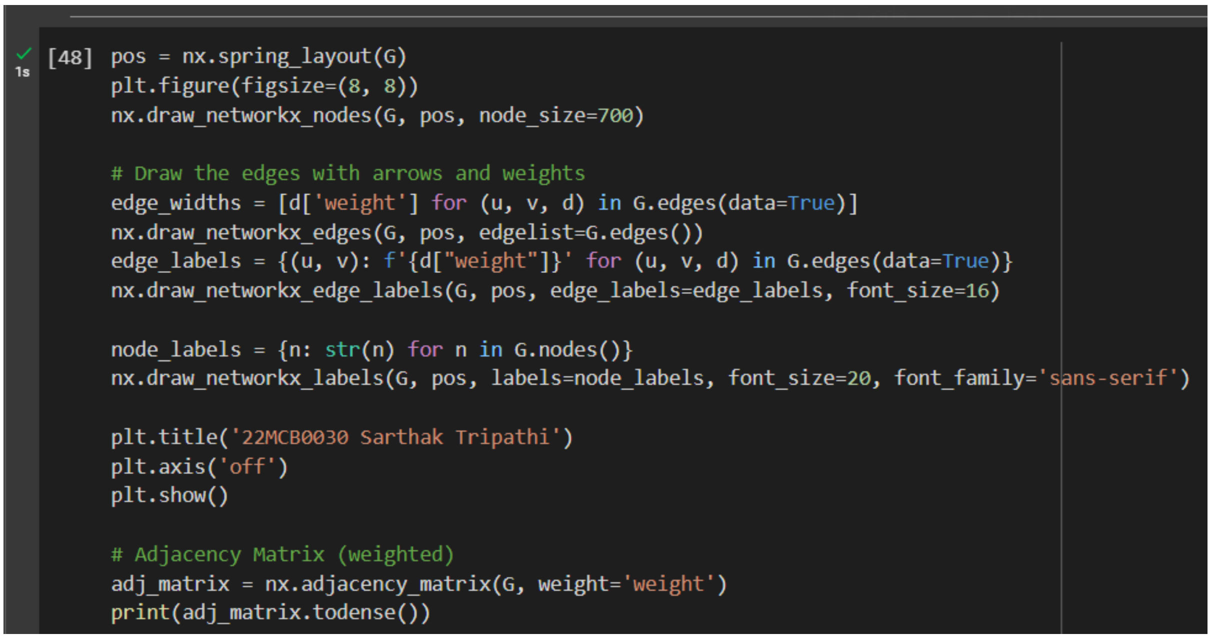
The adj_matrix.todense() function converts the sparse matrix representation of the adjacency matrix to a dense matrix representation and prints it. The dense matrix representation contains all the entries of the matrix, including the zeros.

OUTPUT:



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

Adjacency Matrix For Undirected Graph (with weights)

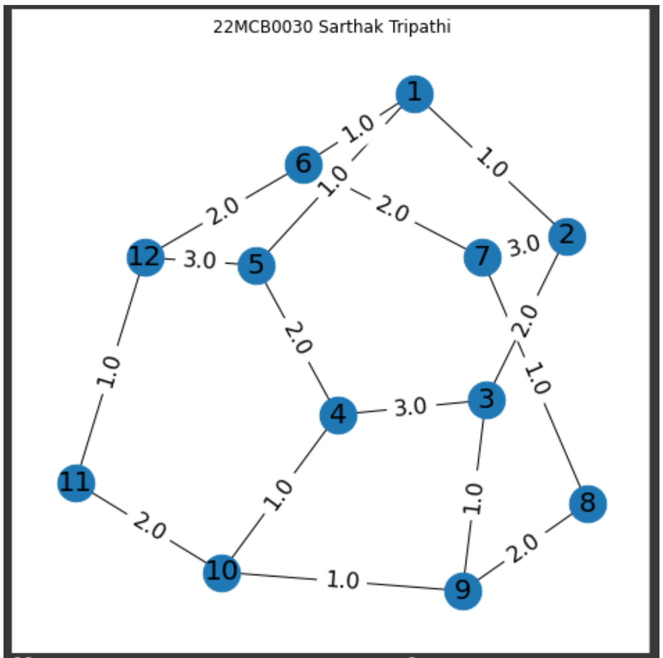


```
[48] pos = nx.spring_layout(G)
     plt.figure(figsize=(8, 8))
     nx.draw_networkx_nodes(G, pos, node_size=700)

     # 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)

     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')

     plt.title('22MCB0030 Sarthak Tripathi')
     plt.axis('off')
     plt.show()

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

Explanation of Code: First, the spring layout is used to position the nodes in the graph, and the nodes are drawn using nx.draw_networkx_nodes().Next, the edges are drawn with weights using nx.draw_networkx_edges(), and edge labels are added using nx.draw_networkx_edge_labels(). The edge weights are obtained from the weight attribute of the edges using a list comprehension.

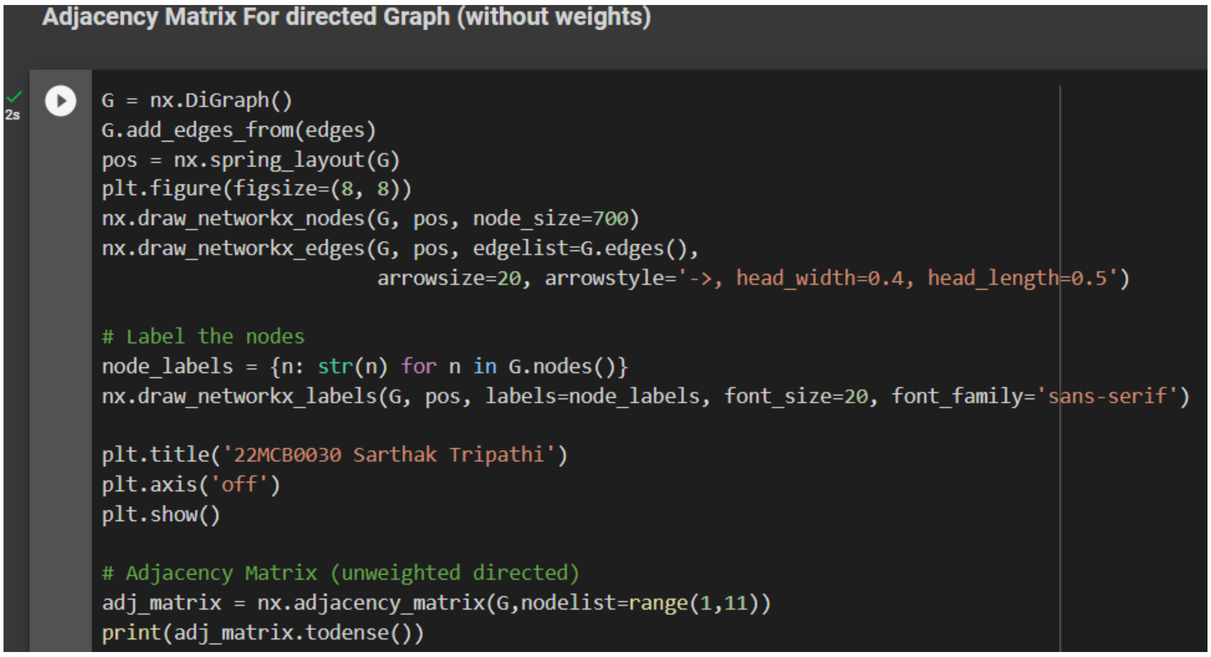
The node labels are added using nx.draw_networkx_labels().The plt.title() function sets the title of the plot, and plt.axis('off') turns off the axis.The adjacency matrix is computed using nx.adjacency_matrix(). The default is an unweighted adjacency matrix, but since the graph is weighted, the weight parameter is set to 'weight' to get a weighted adjacency matrix. The todense() method is used to convert the sparse matrix to a dense matrix, which is printed to the console.

OUTPUT:



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

Adjacency Matrix For directed Graph (without weights)



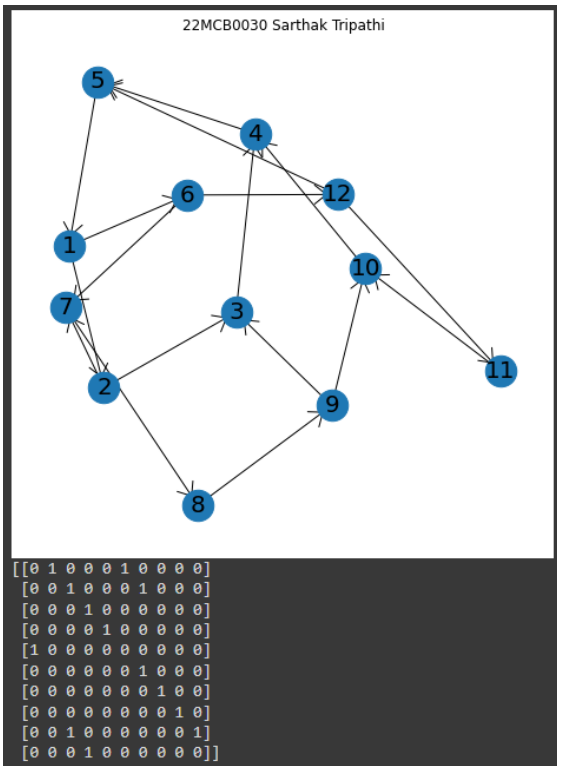
Adjacency Matrix For directed Graph (without weights)

```
G = nx.DiGraph()
G.add_edges_from(edges)
pos = nx.spring_layout(G)
plt.figure(figsize=(8, 8))
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')

plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.show()

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

Explanation of Code:

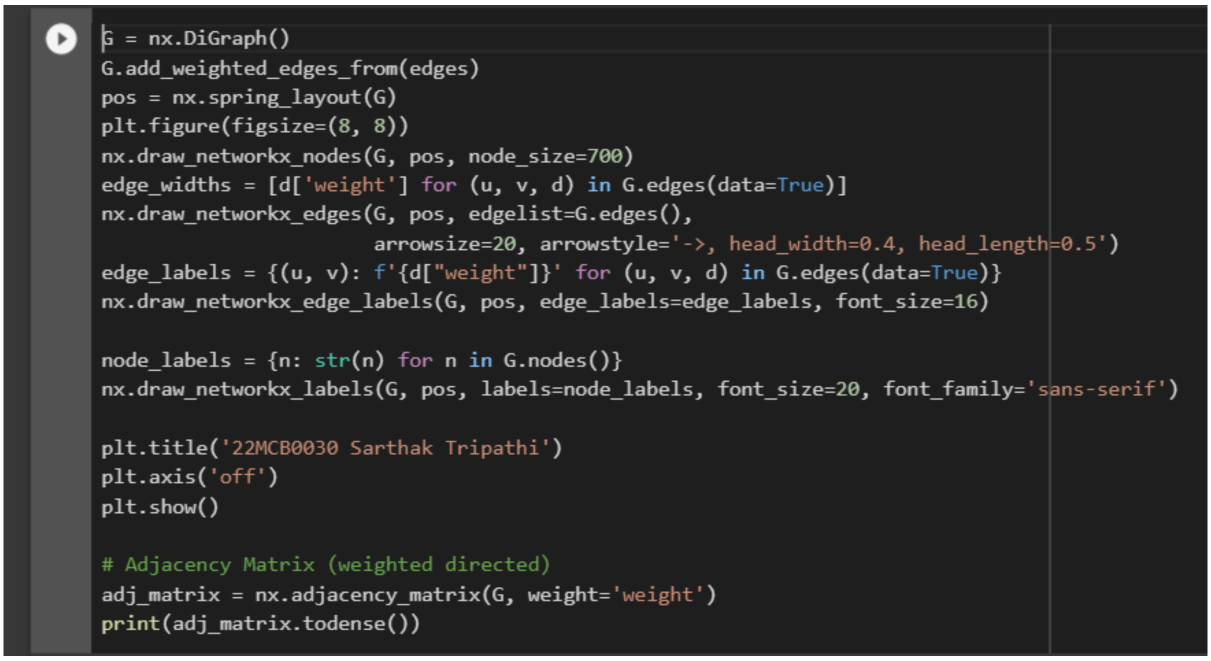
The adjacency matrix of an unweighted directed graph can be obtained using the nx.adjacency_matrix() function in NetworkX with the nodelist parameter to specify the order of nodes in the matrix.

OUTPUT:



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

Adjacency Matrix For directed Graph (with weights)

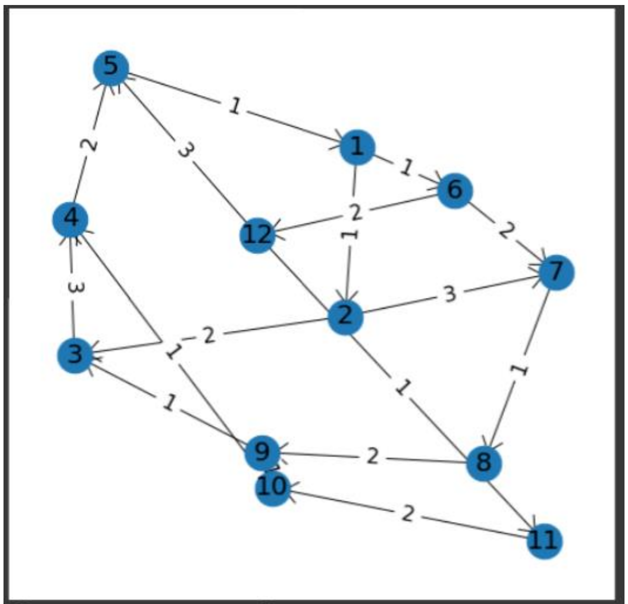


```python
G = nx.DiGraph()
G.add_weighted_edges_from(edges)
pos = nx.spring_layout(G)
plt.figure(figsize=(8, 8))
nx.draw_networkx_nodes(G, pos, node_size=700)
edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
nx.draw_networkx_edges(G, pos, edgelist=G.edges(),
                       arrowsize=20, 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)

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')

plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.show()

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

Explanation of Code:

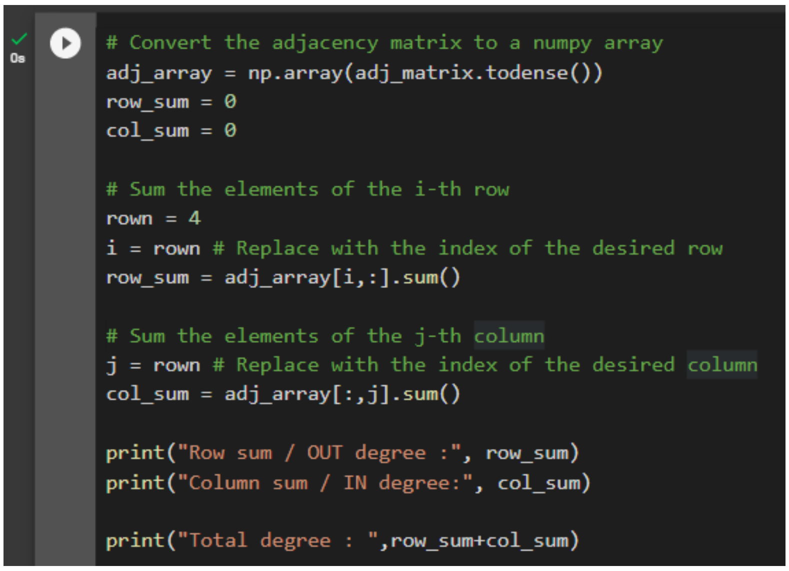
The output of the code above shows the adjacency matrix of the weighted directed graph represented by the variable G.

OUTPUT:



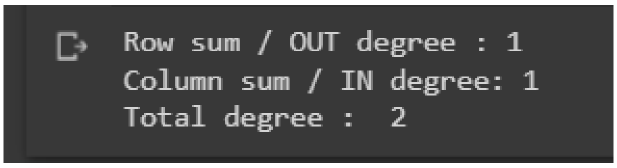
```
[[0 1 0 0 0 1 0 0 0 0 0 0]
 [0 0 2 0 0 0 3 0 0 0 0 0]
 [0 0 0 3 0 0 0 0 0 0 0 0]
 [0 0 0 0 2 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 2 2 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 3 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 2 0 0]
 [0 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 1 0 0 0 0 0 0 0 0]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

Find the sum of particular row from the Adjacency matrix (For Directed Graph)



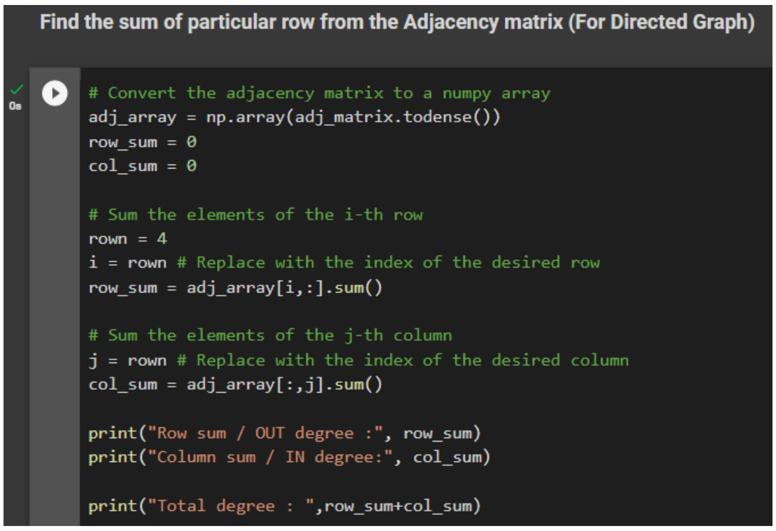
```
# Convert the adjacency matrix to a numpy array
adj_array = np.array(adj_matrix.todense())
row_sum = 0
col_sum = 0

# Sum the elements of the i-th row
rown = 4
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
```

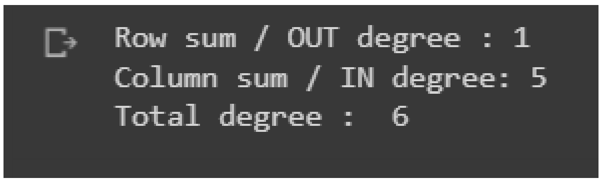
Find the sum of particular row from the Adjacency matrix (For Undirected Graph)



Find the sum of particular row from the Adjacency matrix (For Directed Graph)

```
# 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 = 4
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: 5
Total degree :  6
```

PART – 4

Centrality measures are a set of quantitative metrics used in social network analysis (SNA) to identify and evaluate the importance, influence, or prominence of individual actors or nodes within a network. There are several types of centrality measures, including:

1) Degree Centrality: This measure is based on the number of connections or edges that a node has. A node with a high degree centrality is considered to be well connected to other nodes in the network.Degree centrality measures the number of edges connected to a node i, also known as its degree. The formula for degree centrality is:

$$C\_d(i) = k\_i / (n-1)$$

where k_i is the degree of node i and n is the total number of nodes in the network.

2) Betweenness Centrality: This measure is based on the number of times a node acts as a bridge or intermediary between other nodes in the network. A node with a high betweenness centrality is considered to be strategically positioned within the network. Betweenness centrality measures the number of times a node i lies on the shortest path between other nodes in the network. The formula for betweenness centrality is:

$$C\_b(i) = \sum(s,t \in V)\ \sigma(s,t|i) / \sigma(s,t)$$

where V is the set of all nodes, σ(s,t) is the total number of shortest paths from node s to node t, and σ(s,t|i) is the number of shortest paths from node s to node t that pass through node i.

3) Closeness Centrality: This measure is based on the distance between a node and all other nodes in the network. A node with a high closeness centrality is considered to be more closely connected to other nodes in the network. Closeness centrality measures the inverse of the sum of the shortest distances from a node i to all other nodes in the network. The formula for closeness centrality is:

$$C\_c(i) = (n-1) / \sum(j \in V)\ d(i,j)$$

where d(i,j) is the shortest distance between nodes i and j, and n is the total number of nodes in the network.

4) Eigenvector Centrality: This measure is based on the idea that a node is important if it is connected to other important nodes in the network. A node with a high eigenvector centrality is considered to be connected to other nodes that are themselves well-connected. Eigenvector centrality measures the importance of a node i based on the importance of its neighbors. The formula for eigenvector centrality is:

$$Ax = \lambda x$$

where A is the adjacency matrix of the network, x is the eigenvector of A, and λ is the eigenvalue corresponding to x.

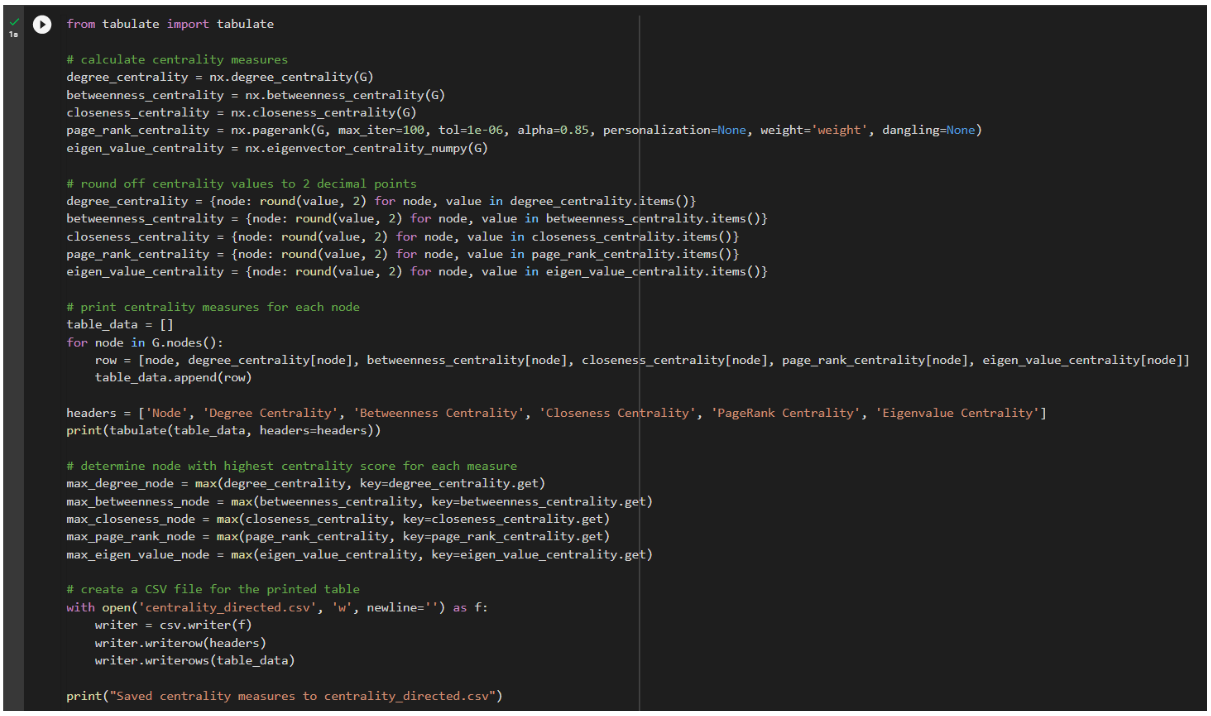
5) PageRank: This measure is based on the idea of the importance of web pages in the Google search engine. In SNA, PageRank is used to identify nodes that are connected to other important nodes in the network.PageRank measures the importance of a node i based on the importance of its incoming neighbors. The formula for PageRank is:

$$PR(i) = (1-d) / N + d \sum(j \in In(i)) PR(j) / Out(j)$$

where N is the total number of nodes in the network, In(i) is the set of nodes that point to node i, Out(j) is the total number of outgoing links from node j, and d is the damping factor, typically set to 0.85.

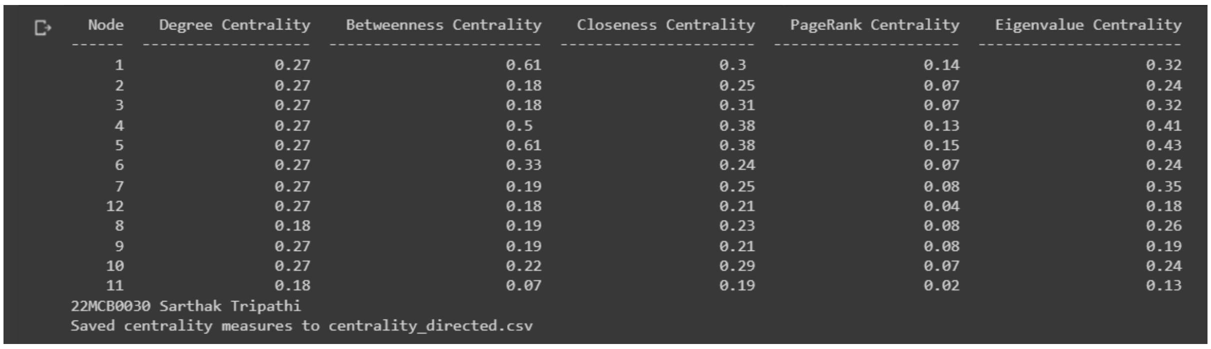
Each of these measures provides a different perspective on the importance of nodes in a network, and they can be used to identify key players, influential nodes, or nodes that may be critical to the functioning of the network.

Finding centrality measures for directed graph



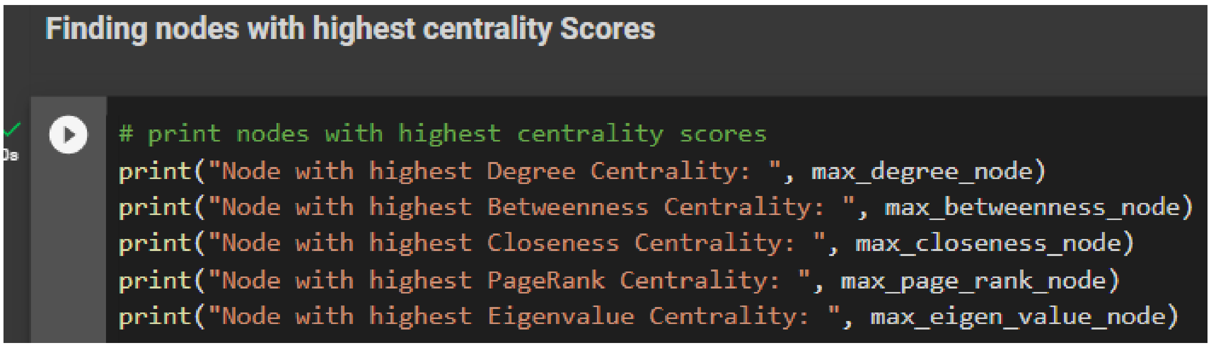
```
from tabulate import tabulate

# 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.csv")
```

OUTPUT:



```
  Node    Degree Centrality    Betweenness Centrality    Closeness Centrality    PageRank Centrality    Eigenvalue Centrality
------  -------------------  ------------------------  ----------------------  ---------------------  -----------------------
     1                 0.27                      0.61                    0.3                    0.14                     0.32
     2                 0.27                      0.18                    0.25                   0.07                     0.24
     3                 0.27                      0.18                    0.31                   0.07                     0.32
     4                 0.27                      0.5                     0.38                   0.13                     0.41
     5                 0.27                      0.61                    0.38                   0.15                     0.43
     6                 0.27                      0.33                    0.24                   0.07                     0.24
     7                 0.27                      0.19                    0.25                   0.08                     0.35
    12                 0.27                      0.18                    0.21                   0.04                     0.18
     8                 0.18                      0.19                    0.23                   0.08                     0.26
     9                 0.27                      0.19                    0.21                   0.08                     0.19
    10                 0.27                      0.22                    0.29                   0.07                     0.24
    11                 0.18                      0.07                    0.19                   0.02                     0.13
22MCB0030 Sarthak Tripathi
Saved centrality measures to centrality_directed.csv
```

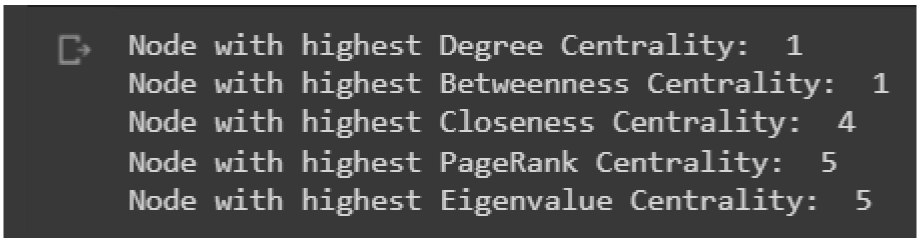
Finding nodes with Highest Centrality Scores



Finding nodes with highest centrality Scores

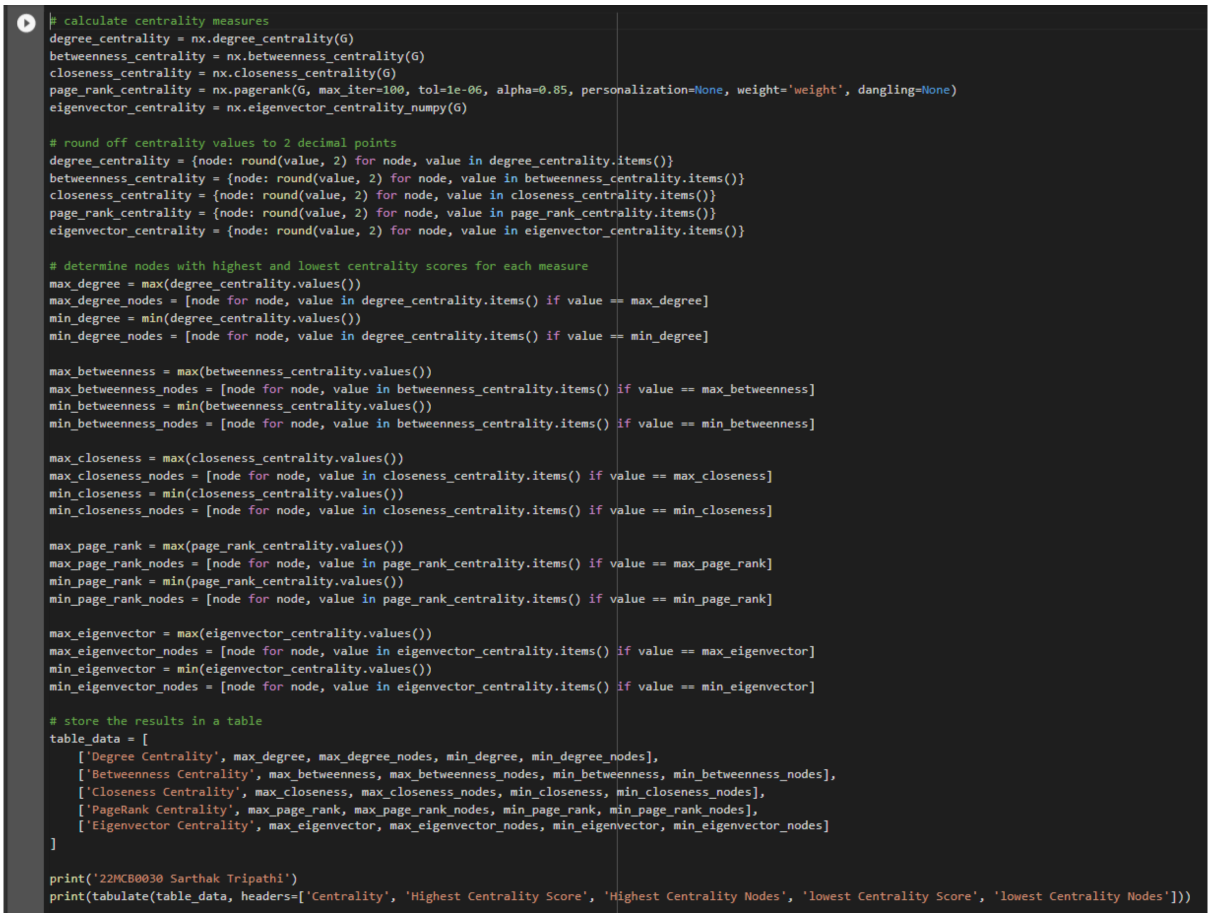
```
# print nodes with highest centrality scores
print("Node with highest Degree Centrality: ", max_degree_node)
print("Node with highest Betweenness Centrality: ", max_betweenness_node)
print("Node with highest Closeness Centrality: ", max_closeness_node)
print("Node with highest PageRank Centrality: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

OUTPUT:



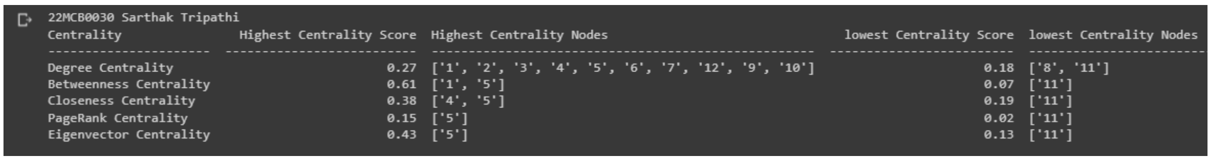
```
Node with highest Degree Centrality:  1
Node with highest Betweenness Centrality:  1
Node with highest Closeness Centrality:  4
Node with highest PageRank Centrality:  5
Node with highest Eigenvalue Centrality:  5
```

Finding node with highest centrality score and node for all centrality measures



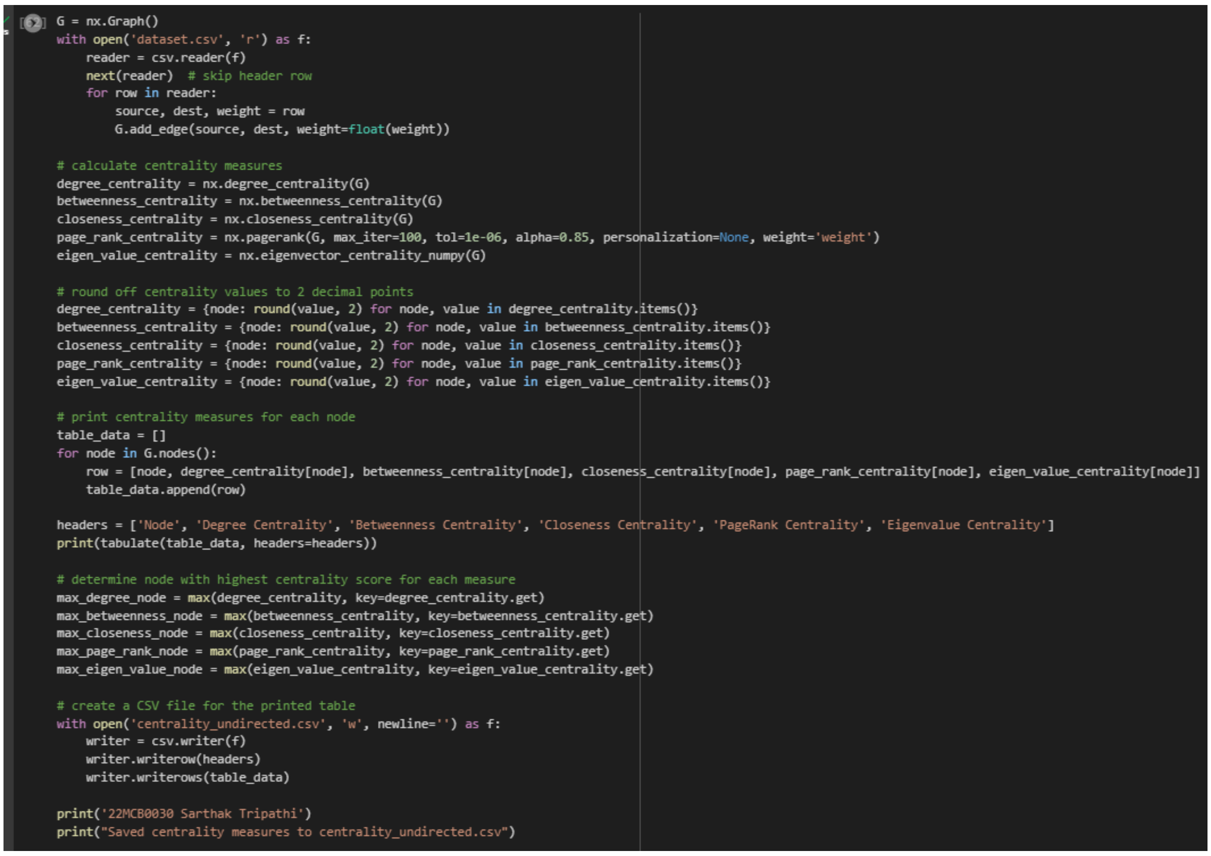
```
# calculate centrality measures
degree_centrality = nx.degree_centrality(G)
betweenness_centrality = nx.betweenness_centrality(G)
closeness_centrality = nx.closeness_centrality(G)
page_rank_centrality = nx.pagerank(G, max_iter=100, tol=1e-06, alpha=0.85, personalization=None, weight='weight', dangling=None)
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('22MCB0030 Sarthak Tripathi')
print(tabulate(table_data, headers=['Centrality', 'Highest Centrality Score', 'Highest Centrality Nodes', 'lowest Centrality Score', 'lowest Centrality Nodes']))
```

OUTPUT:



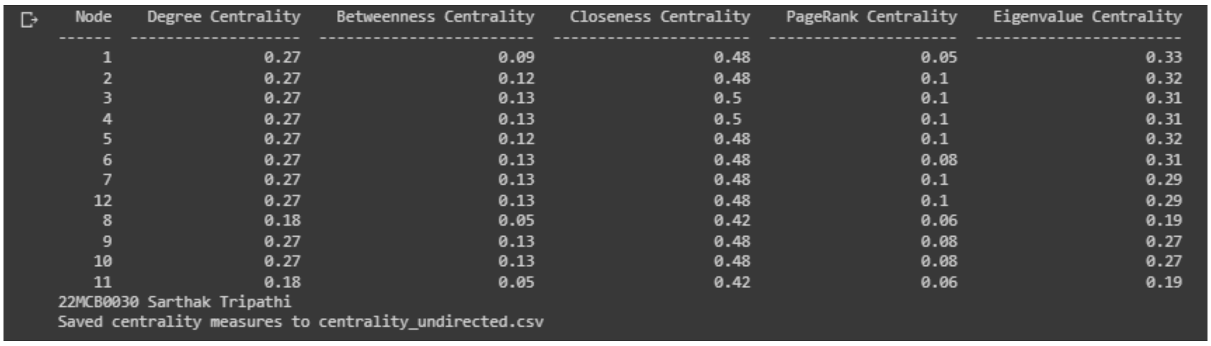
```
22MCB0030 Sarthak Tripathi
Centrality               Highest Centrality Score  Highest Centrality Nodes                                    lowest Centrality Score  lowest Centrality Nodes
---------------------  -------------------------  ----------------------------------------------------  -------------------------  ------------------------
Degree Centrality                           0.27  ['1', '2', '3', '4', '5', '6', '7', '12', '9', '10']                       0.18  ['8', '11']
Betweenness Centrality                      0.61  ['1', '5']                                                              0.07  ['11']
Closeness Centrality                        0.38  ['4', '5']                                                              0.19  ['11']
PageRank Centrality                         0.15  ['5']                                                                   0.02  ['11']
Eigenvector Centrality                      0.43  ['5']                                                                   0.13  ['11']
```

Finding the centrality measures for undirected graph



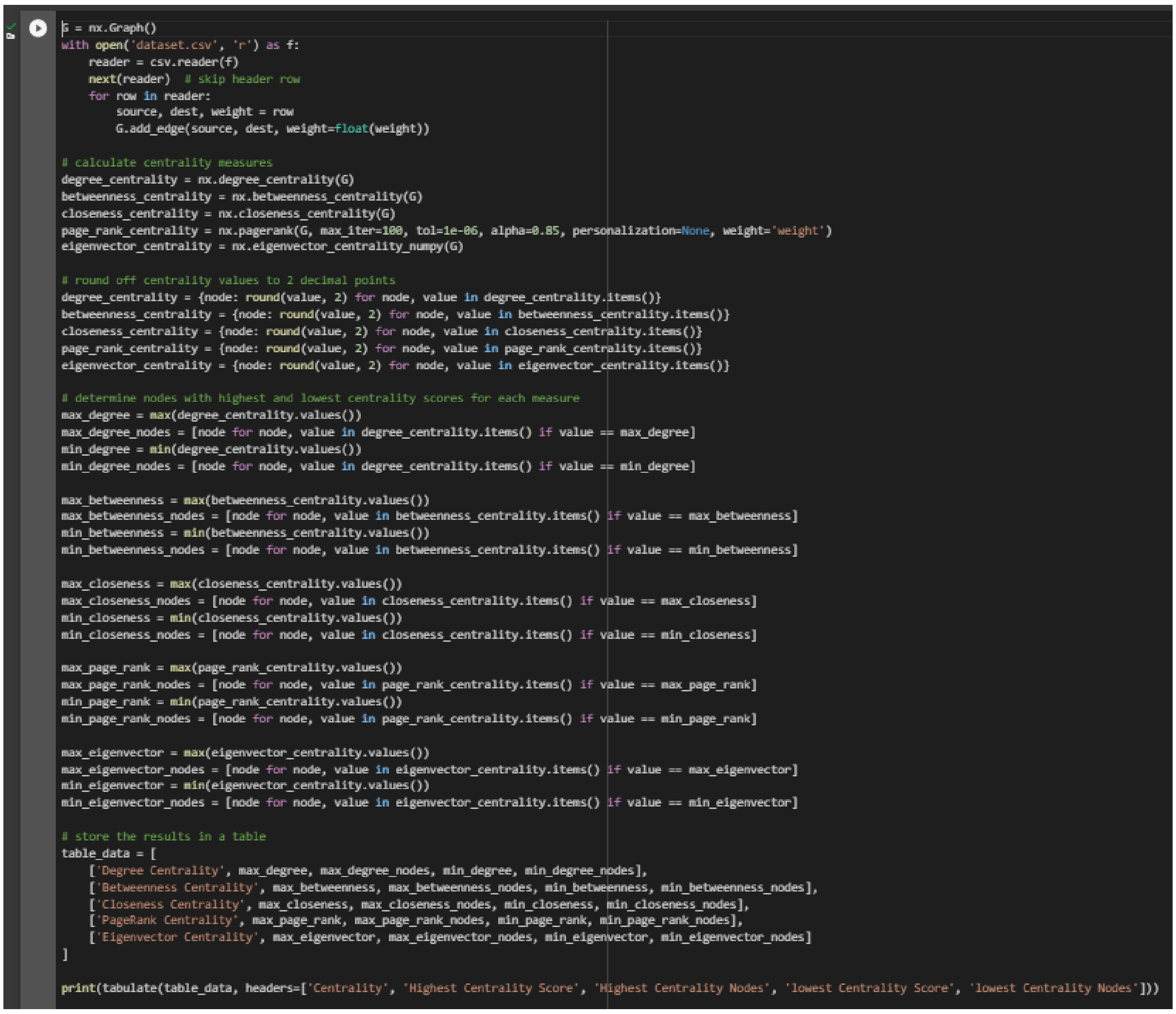
```python
G = nx.Graph()
with open('dataset.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)  # skip header row
    for row in reader:
        source, dest, weight = row
        G.add_edge(source, dest, weight=float(weight))

# 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('22MCB0030 Sarthak Tripathi')
print("Saved centrality measures to centrality_undirected.csv")
```

OUTPUT:



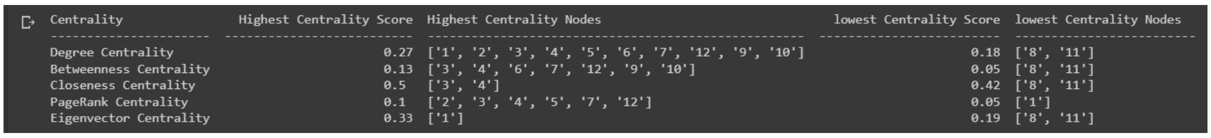
```
  Node    Degree Centrality    Betweenness Centrality    Closeness Centrality    PageRank Centrality    Eigenvalue Centrality
------  -------------------  ------------------------  ----------------------  ---------------------  -----------------------
     1                 0.27                      0.09                    0.48                   0.05                     0.33
     2                 0.27                      0.12                    0.48                   0.1                      0.32
     3                 0.27                      0.13                    0.5                    0.1                      0.31
     4                 0.27                      0.13                    0.5                    0.1                      0.31
     5                 0.27                      0.12                    0.48                   0.1                      0.32
     6                 0.27                      0.13                    0.48                   0.08                     0.31
     7                 0.27                      0.13                    0.48                   0.1                      0.29
    12                 0.27                      0.13                    0.48                   0.1                      0.29
     8                 0.18                      0.05                    0.42                   0.06                     0.19
     9                 0.27                      0.13                    0.48                   0.08                     0.27
    10                 0.27                      0.13                    0.48                   0.08                     0.27
    11                 0.18                      0.05                    0.42                   0.06                     0.19
22MCB0030 Sarthak Tripathi
Saved centrality measures to centrality_undirected.csv
```

Finding node with highest centrality score and node for all centrality measures



```python
G = nx.Graph()
with open('dataset.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_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']))
```

OUTPUT:



```
Centrality                Highest Centrality Score  Highest Centrality Nodes                                    lowest Centrality Score  lowest Centrality Nodes
----------------------  --------------------------  --------------------------------------------------  -------------------------  -------------------------
Degree Centrality                             0.27  ['1', '2', '3', '4', '5', '6', '7', '12', '9', '10']                        0.18  ['8', '11']
Betweenness Centrality                        0.13  ['3', '4', '6', '7', '12', '9', '10']                                       0.05  ['8', '11']
Closeness Centrality                          0.5   ['3', '4']                                                                  0.42  ['8', '11']
PageRank Centrality                           0.1   ['2', '3', '4', '5', '7', '12']                                             0.05  ['1']
Eigenvector Centrality                        0.33  ['1']                                                                       0.19  ['8', '11']
```

CODE

```
import csv
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
with open('dataset.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('dataset.csv')
print('22MCB0030 Sarthak Tripathi')
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.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.savefig('directed_graph.png')
plt.show()
# Create an empty directed graph with weights
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')
plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.show()
G = nx.Graph()
G.add_weighted_edges_from(edges)

# Draw the graph with edge weights
pos = nx.spring_layout(G)
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.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.savefig('undirected_graph.png')
plt.show()
G = nx.Graph()
```

```
G.add_weighted_edges_from(edges)
pos = nx.spring_layout(G)
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')
plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
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_degree
s.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.va
lues())]

min_in_degree_nodes = [node for node, degree in in_degrees.items() if degree == min(in_degrees.val
ues())]

print(f'Nodes with maximum in-degree: {max_in_degree_nodes}')

print(f'Nodes with minimum in-degree: {min_in_degree_nodes}')

# 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')

plt.axis('off')

plt.show()


# Adjacency Matrix (unweighted)

adj_matrix = nx.adjacency_matrix(G)

print(adj_matrix.todense())
```

```
pos = nx.spring_layout(G)
plt.figure(figsize=(8, 8))
nx.draw_networkx_nodes(G, pos, node_size=700)
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)
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')
plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.show()

# Adjacency Matrix (weighted)
adj_matrix = nx.adjacency_matrix(G, weight='weight')
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
rown = 4
i = rown # Replace with the index of the desired row
row_sum = adj_array[i,:].sum()
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)
G = nx.DiGraph()
G.add_edges_from(edges)
pos = nx.spring_layout(G)
plt.figure(figsize=(8, 8))
```

```
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')
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')
plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.show()

# 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
rown = 4
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)
G = nx.DiGraph()
G.add_weighted_edges_from(edges)
pos = nx.spring_layout(G)
plt.figure(figsize=(8, 8))
nx.draw_networkx_nodes(G, pos, node_size=700)
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)
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')
plt.title('22MCB0030 Sarthak Tripathi')
plt.axis('off')
plt.show()

# Adjacency Matrix (weighted directed)
adj_matrix = nx.adjacency_matrix(G, weight='weight')
print(adj_matrix.todense())
from tabulate import tabulate

# 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], pa
ge_rank_centrality[node], eigen_value_centrality[node]]

    table_data.append(row)

headers = ['Node', 'Degree Centrality', 'Betweenness Centrality', 'Closeness Centrality', 'PageRank Cen
trality', '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('22MCB0030 Sarthak Tripathi')

print("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)

# 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_b
etweenness]

max_closeness = max(closeness_centrality.values())
max_closeness_nodes = [node for node, value in closeness_centrality.items() if value == max_closene
ss]
min_closeness = min(closeness_centrality.values())
min_closeness_nodes = [node for node, value in closeness_centrality.items() if value == min_closene
ss]

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_eig
envector]

min_eigenvector = min(eigenvector_centrality.values())

min_eigenvector_nodes = [node for node, value in eigenvector_centrality.items() if value == min_eig
envector]


# 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_bet
weenness_nodes],

    ['Closeness Centrality', max_closeness, max_closeness_nodes, min_closeness, min_closeness_node
s],

    ['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_eigenv
ector_nodes]

]

print('22MCB0030 Sarthak Tripathi')

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


G = nx.Graph()

with open('dataset.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], pa
ge_rank_centrality[node], eigen_value_centrality[node]]

    table_data.append(row)


headers = ['Node', 'Degree Centrality', 'Betweenness Centrality', 'Closeness Centrality', 'PageRank Cen
trality', '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('22MCB0030 Sarthak Tripathi')
print("Saved centrality measures to centrality_undirected.csv")

G = nx.Graph()
with open('dataset.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_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_b
etweenness]

max_closeness = max(closeness_centrality.values())

max_closeness_nodes = [node for node, value in closeness_centrality.items() if value == max_closene
ss]

min_closeness = min(closeness_centrality.values())

min_closeness_nodes = [node for node, value in closeness_centrality.items() if value == min_closene
ss]

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_eig
envector]

min_eigenvector = min(eigenvector_centrality.values())

min_eigenvector_nodes = [node for node, value in eigenvector_centrality.items() if value == min_eig
envector]

# 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_bet
weenness_nodes],

   ['Closeness Centrality', max_closeness, max_closeness_nodes, min_closeness, min_closeness_node
s],

   ['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_eigenv
ector_nodes]

]
```

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