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: Ketan Kolte

Reg No: 22MCB0016

Email: ketansanjay.kolte2022@vitstudent.ac.in

Mobile: 8329324714

Faculty Name: DURGESH KUMAR

Subject Name with code: SOCIAL NETWROK ANALYTICS LAB – MCSE618P

Date : 21/03/2023

PART – 1 :

PART – 1.1

Creation of directed graph :

a) Weighted

b) Unweighted

PART – 1.2

Creation of Undirected graph :

a) Weighted

b) Unweighted

PART – 1.1

THEORY :

Directed Graph :

A directed graph, also known as a digraph, is a type of graph in which the edges have a direction. That is, each edge connects two nodes or vertices, but it has a specific direction indicating that one vertex is the source and the other is the destination. In a directed graph, each node can have zero or more incoming edges and zero or more outgoing edges. The opposite of a directed graph is an undirected graph, where edges have no direction and can be traversed in either direction between nodes. Directed graphs are commonly used to represent relationships between objects in various fields, including computer science, social networks, and transportation systems.

Undirected Graph:

An undirected graph is a collection of vertices (also known as nodes) connected by edges, where the edges have no direction or orientation. In other words, an edge between two vertices in an undirected graph represents a symmetric relationship, indicating that there is a connection between the two vertices with no intrinsic direction. Additionally, each pair of vertices can have at most one edge connecting them. This means that if there is an edge connecting vertex A to vertex B, there cannot be another edge connecting vertex A to vertex B. This property ensures that the graph is uniquely defined and allows for efficient algorithms to be applied when working with the graph.

Weighted Graph :

A weighted graph is a mathematical structure that consists of a set of vertices (or nodes) connected by edges, where each edge is assigned a numerical weight or cost. The weights represent some meaningful quantity associated with the edges, such as distance, cost, or probability. Each edge in a weighted graph can have a different weight, and the weights can be positive, negative, or zero. In contrast to an unweighted graph, where all edges have equal weight, weighted graphs are used to model complex relationships where the strength or importance of connections between vertices varies. A unique weighted graph is a graph where each edge has a distinct weight, meaning that no two edges have the same weight.

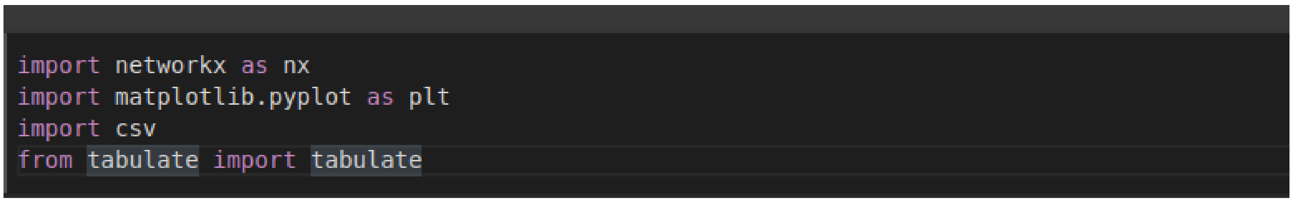
Unweighted Graph :

An unweighted graph is a type of graph where each edge has the same weight or cost associated with traversing it. In other words, an unweighted graph is a graph in which every edge has a weight of 1. This means that when calculating metrics such as shortest paths or optimal routes, only the connectivity between vertices is taken into account, without considering any additional complexities or nuances that might be associated with traversing certain edges. Unweighted graphs are often used to represent simple systems or relationships, where the focus is on the topology of the graph rather than any specific values or costs associated with its edges.

CODE :

STEP – 1 : Importing required libraries

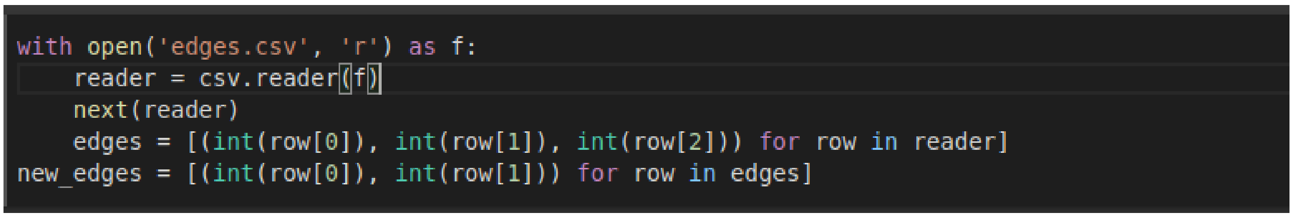
```
#networkx to plot the graph.
import networkx as nx

#matplotlib for visualizations purpose.
import matplotlib.pyplot as plt

#csv library for reading the .csv file using reader() function.
import csv

#tabulate to print the data in tabular format.
from tabulate import tabulate
```



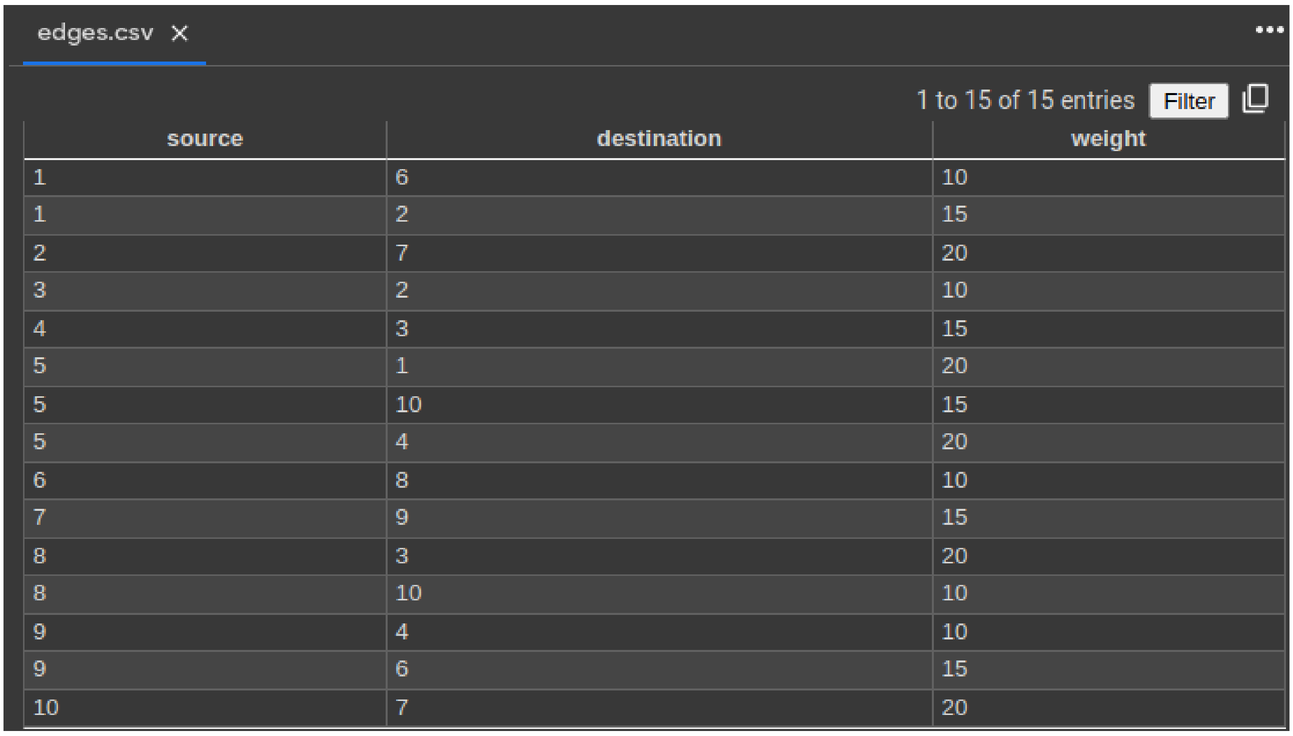
```
import networkx as nx
import matplotlib.pyplot as plt
import csv
from tabulate import tabulate
```

STEP – 2 : Reading the CSV file (edges.csv)

```
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), int(row[2])) for row in reader]
new_edges = [(int(row[0]), int(row[1])) for row in edges]
```



```
with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), int(row[2])) for row in reader]
new_edges = [(int(row[0]), int(row[1])) for row in edges]
```



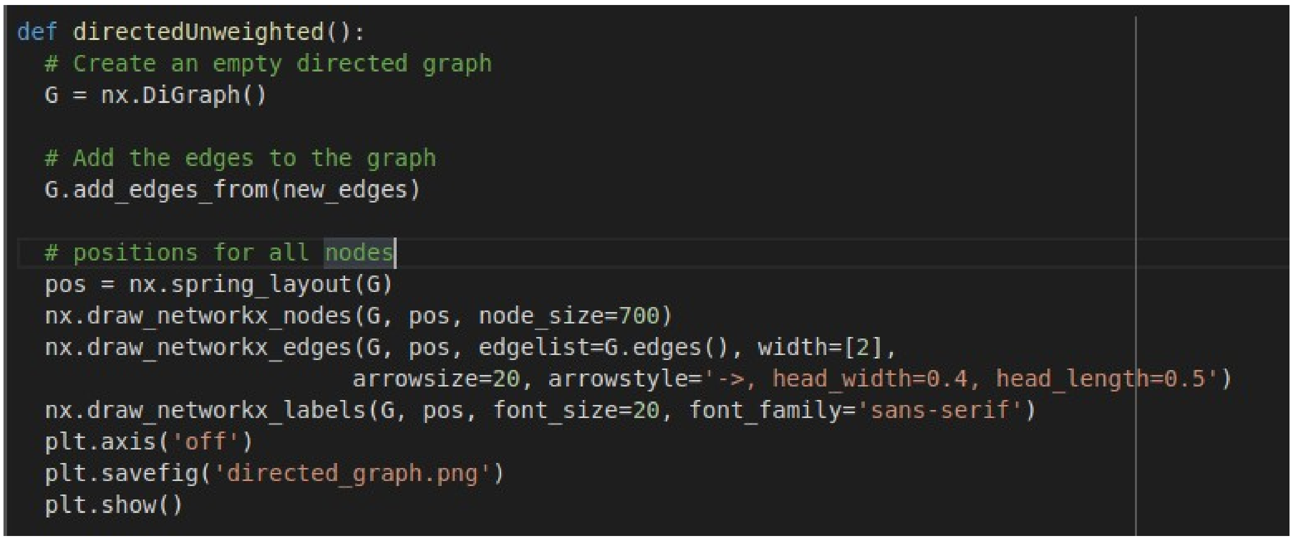
edges.csv

1 to 15 of 15 entries

source	destination	weight
1	6	10
1	2	15
2	7	20
3	2	10
4	3	15
5	1	20
5	10	15
5	4	20
6	8	10
7	9	15
8	3	20
8	10	10
9	4	10
9	6	15
10	7	20

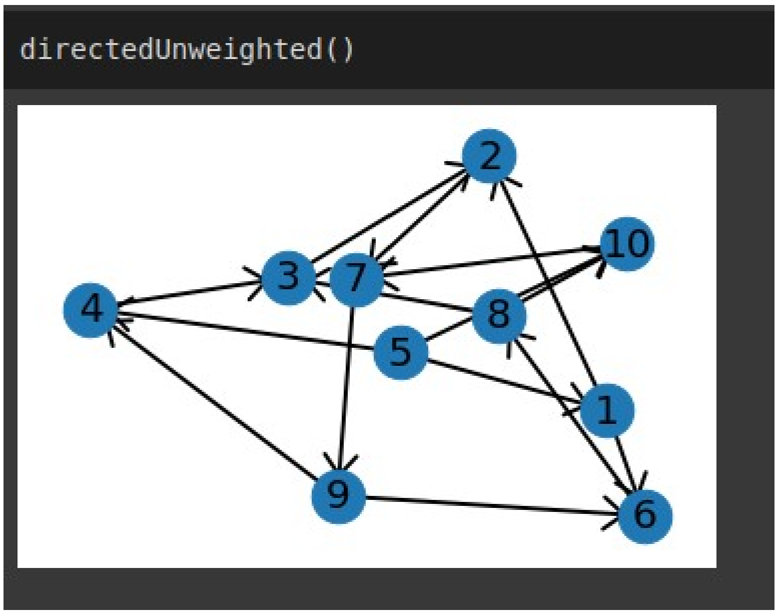
STEP – 3 : Create a function to return png file for directed and unweighted graph.

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

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

   # positions for all nodes
   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],
                                    arrowsize=20, arrowstyle='->, head_width=0.4,
head_length=0.5')
   nx.draw_networkx_labels(G, pos, font_size=20, font_family='sans-serif')
   plt.axis('off')
   plt.savefig('directed_graph.png')
   plt.show()
```



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

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

  # positions for all nodes
  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],
                         arrowsize=20, arrowstyle='->, head_width=0.4, head_length=0.5')
  nx.draw_networkx_labels(G, pos, font_size=20, font_family='sans-serif')
  plt.axis('off')
  plt.savefig('directed_graph.png')
  plt.show()
```

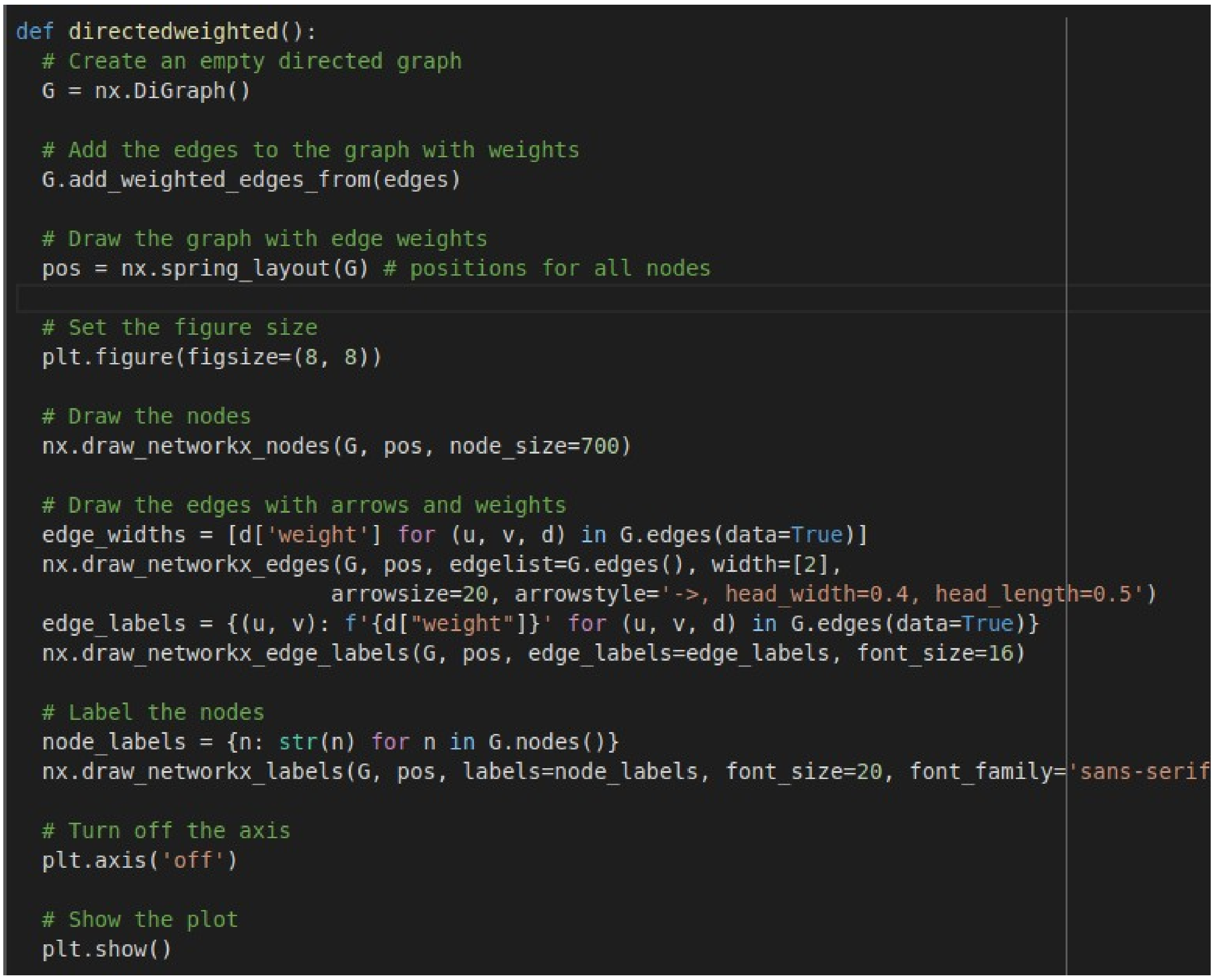
OUTPUT :



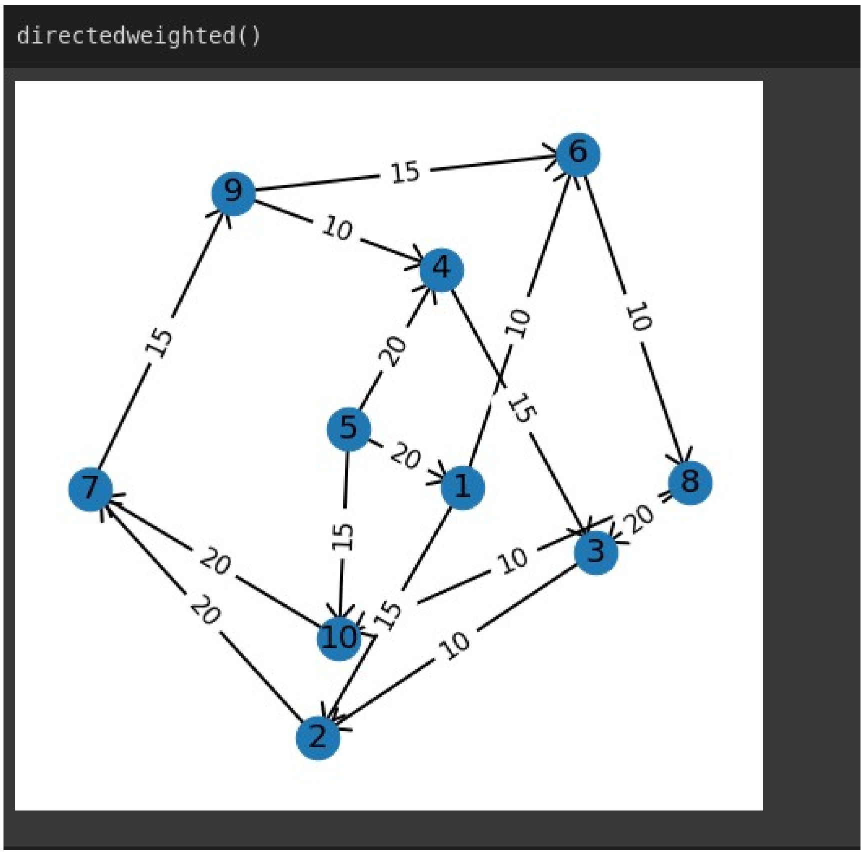
```
directedUnweighted()
```

STEP – 4 : Create a function to return png file for directed and Weighted graph.

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

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

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

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

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

    # Draw the edges with arrows and weights
    edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
    nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2],
                                  arrowsize=20, arrowstyle='->, head_width=0.4,
head_length=0.5')
    edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

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

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

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



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

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

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

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

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

  # Draw the edges with arrows and weights
  edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
  nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2],
                         arrowsize=20, arrowstyle='->, head_width=0.4, head_length=0.5')
  edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
  nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

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

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

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

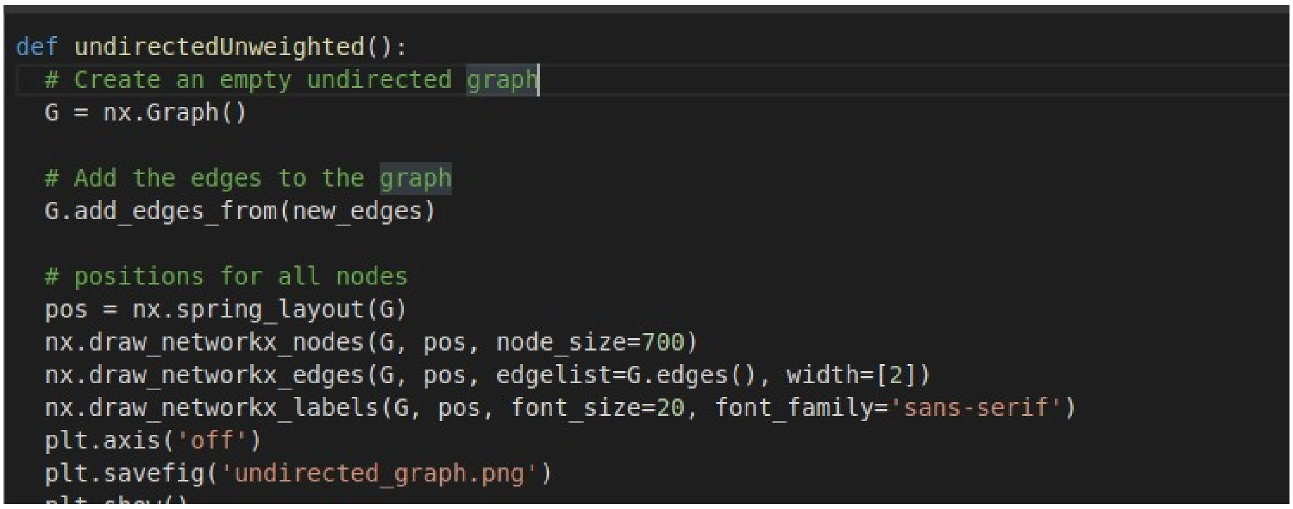


```
directedweighted()
```

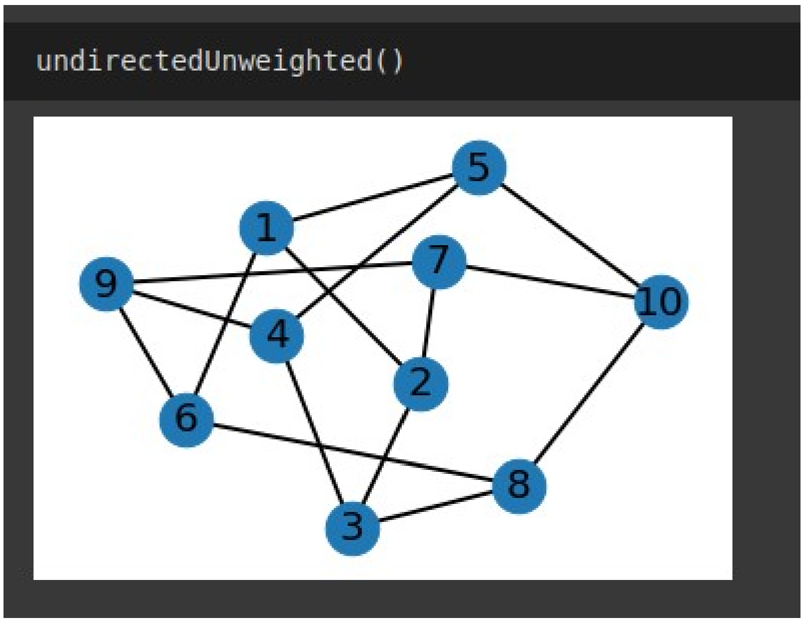
PART – 1.2

STEP – 5 : Create a function to return png file for Undirected and Unweighted graph.

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

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

   # positions for all nodes
   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.axis('off')
   plt.savefig('undirected_graph.png')
   plt.show()
```



```python
def undirectedUnweighted():
  # Create an empty undirected graph
  G = nx.Graph()

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

  # positions for all nodes
  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.axis('off')
  plt.savefig('undirected_graph.png')
  plt.show()
```



```
undirectedUnweighted()
```

STEP – 6 : Create a function to return png file for Undirected and Weighted graph.

def undirectedweighted():

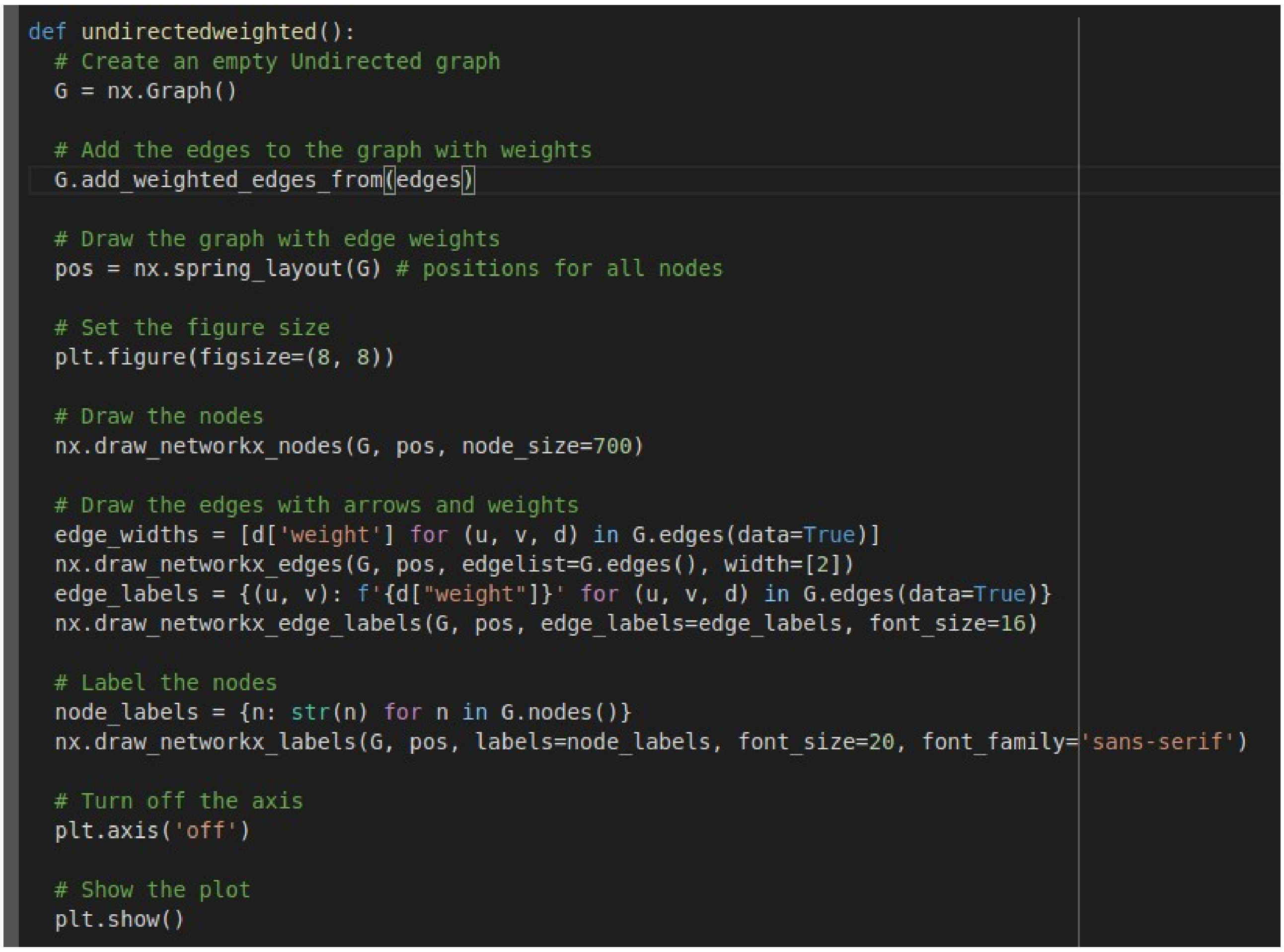
 # Create an empty Undirected graph

 G = nx.Graph()

 # Add the edges to the graph with weights

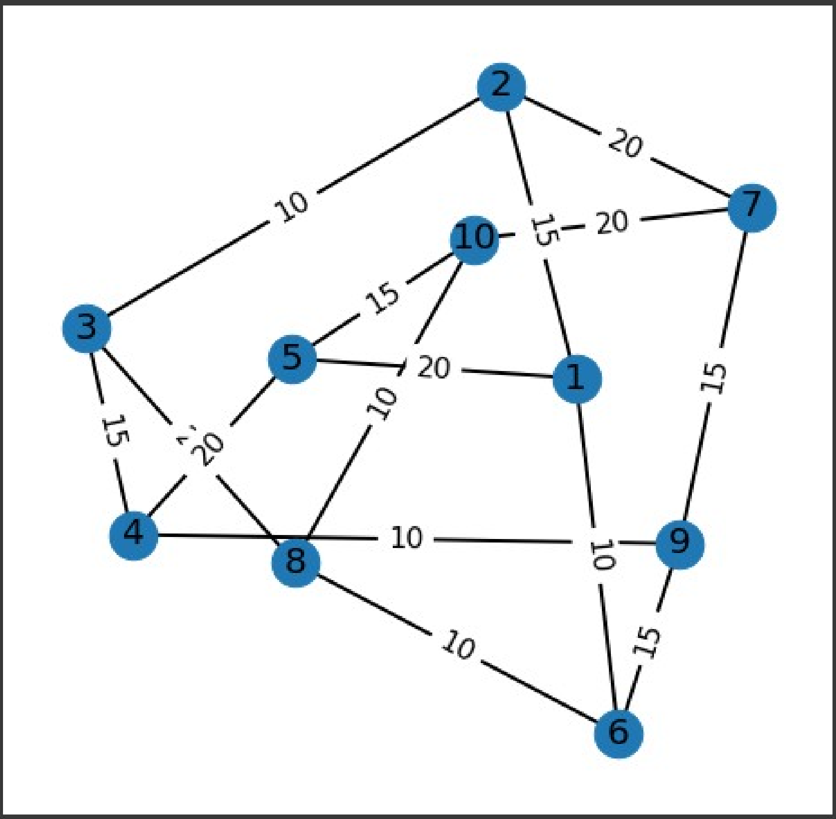
```
   G.add_weighted_edges_from(edges)

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

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

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

   # Draw the edges with arrows and weights
   edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
   nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2])
   edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in
G.edges(data=True)}
   nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_size=16)

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

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

   # Show the plot
```

plt.show()



```python
def undirectedweighted():
  # Create an empty Undirected graph
  G = nx.Graph()

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

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

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

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

  # Draw the edges with arrows and weights
  edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
  nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2])
  edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
  nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

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

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

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

```python
undirectedweighted()
```



PART – 2 :

Task : Printing the details of the directed and undirected graph such as

a)

Undirected Graph :

1) Number of nodes
2) Number of edges
3) Nodes with maximum degree
4) Nodes with minimum degree
5) Maximum degree value
6) Minimum degree value

b)

Directed Graph :

1) Number of nodes
2) Number of edges
3) Maximum out degree
4) Minimum out degree
5) Nodes with maximum out degree
6) Nodes with minimum out degree

7) Maximum IN degree

8) Minimum IN degree

9) Nodes with maximum IN degree

10) Nodes with minimum IN degree

PART – 2a :

THEORY :

IN degree :

In a directed graph or digraph, the in-degree of a vertex is the number of incoming edges to that vertex. In other words, it represents the number of edges that are directed towards the vertex.

For any given vertex in a directed graph, the in-degree is unique because each incoming edge originates from a unique vertex. However, multiple vertices in the graph may have the same in-degree value.

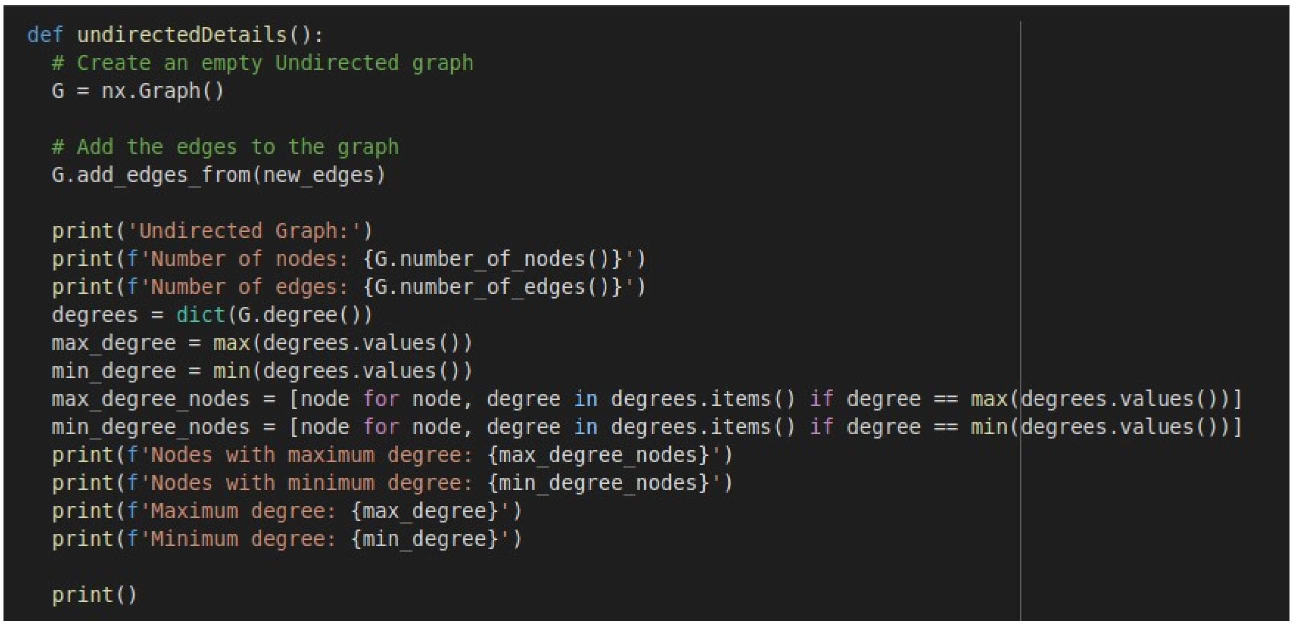
OUT degree :

In a directed graph or digraph, the out-degree of a vertex is the number of outgoing edges from that vertex. In other words, it represents the number of edges that are directed away from the vertex.

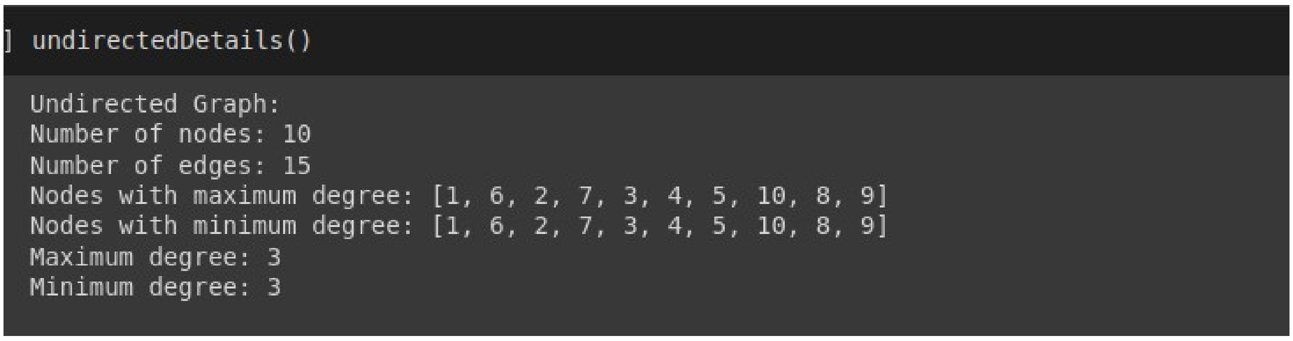
For any given vertex in a directed graph, the out-degree is unique because each outgoing edge goes to a unique vertex. However, multiple vertices in the graph may have the same out-degree value.

CODE :

For Undirected Graph :



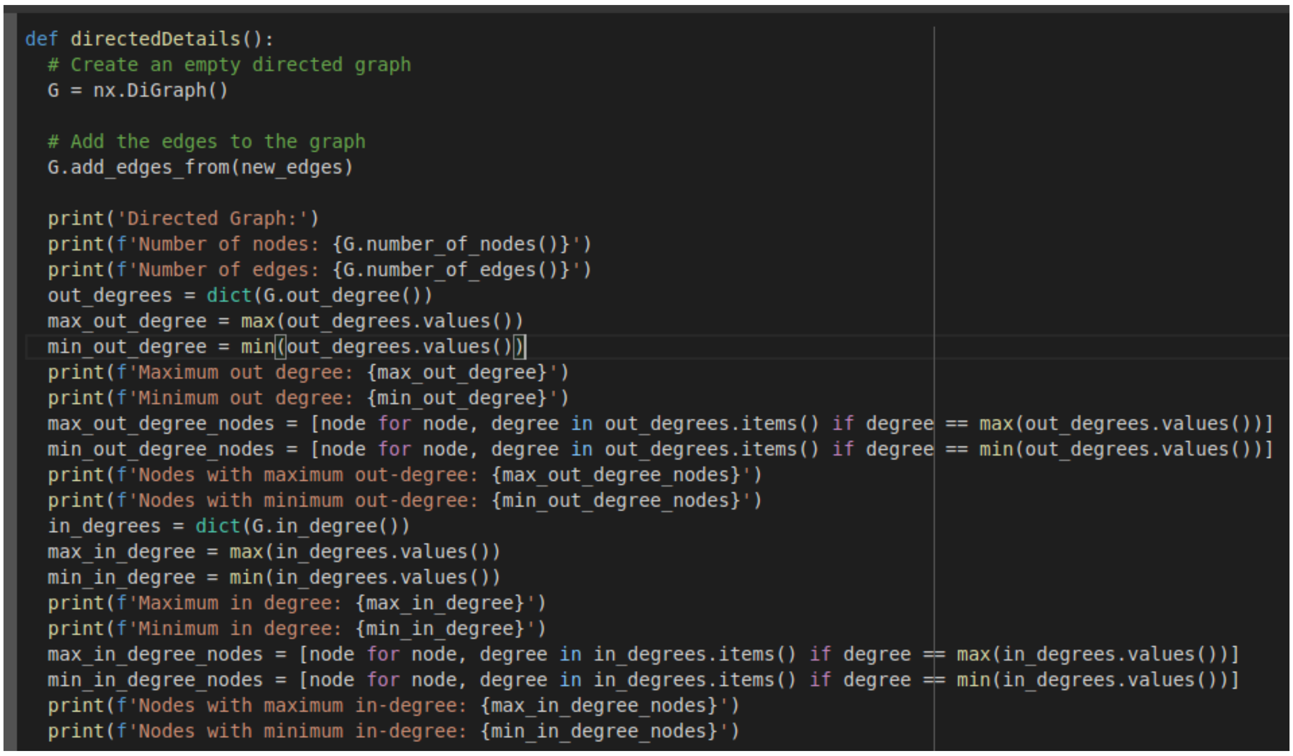
```
def undirectedDetails():
  # Create an empty Undirected graph
  G = nx.Graph()

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

  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()
```



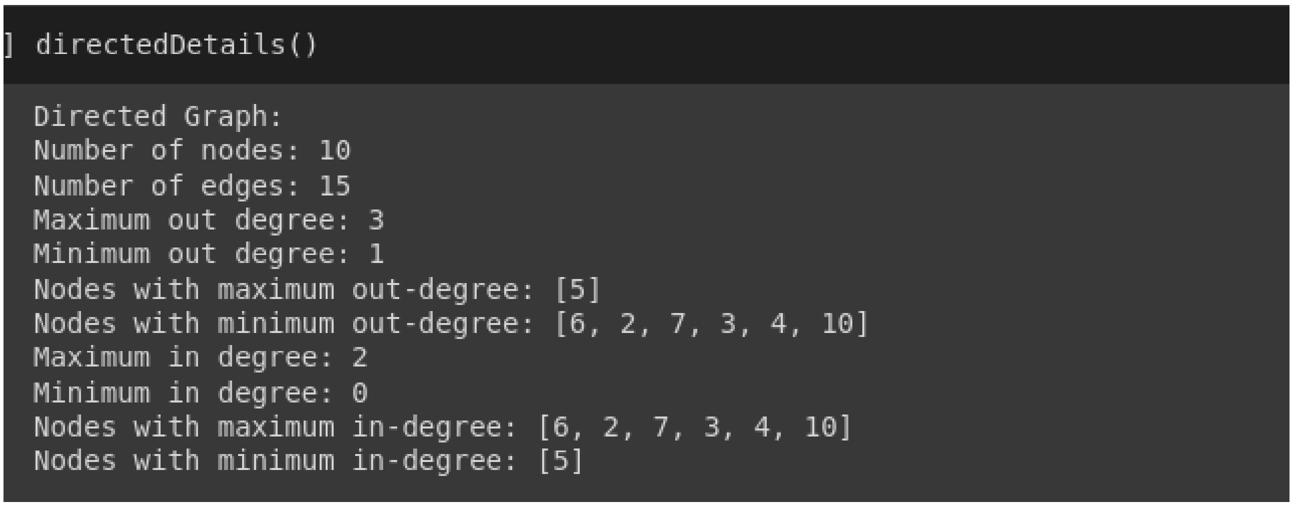
```
] undirectedDetails()

Undirected Graph:
Number of nodes: 10
Number of edges: 15
Nodes with maximum degree: [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]
Nodes with minimum degree: [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]
Maximum degree: 3
Minimum degree: 3
```

PART – 2b :

For Directed Graph :



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

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

  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}')
```



```
] directedDetails()

Directed Graph:
Number of nodes: 10
Number of edges: 15
Maximum out degree: 3
Minimum out degree: 1
Nodes with maximum out-degree: [5]
Nodes with minimum out-degree: [6, 2, 7, 3, 4, 10]
Maximum in degree: 2
Minimum in degree: 0
Nodes with maximum in-degree: [6, 2, 7, 3, 4, 10]
Nodes with minimum in-degree: [5]
```

PART – 3 :

a) Adjacency Matrix for Directed Unweighted Graph

b) Adjacency Matrix for Directed Weighted Graph

c) Adjacency Matrix for Undirected Unweighted Graph

d) Adjacency Matrix for Undirected Weighted Graph

THEORY :

Adjacency Matrix :

An adjacency matrix is a square matrix that represents the connections or edges between vertices in an undirected or directed graph. The rows and columns of the matrix correspond to the vertices of the graph, and each element of the matrix represents the presence or absence of an edge between two vertices.

In an unweighted graph, the elements are usually 0 or 1, indicating whether there is an edge or not. In a weighted graph, the elements represent the weight of the corresponding edge.

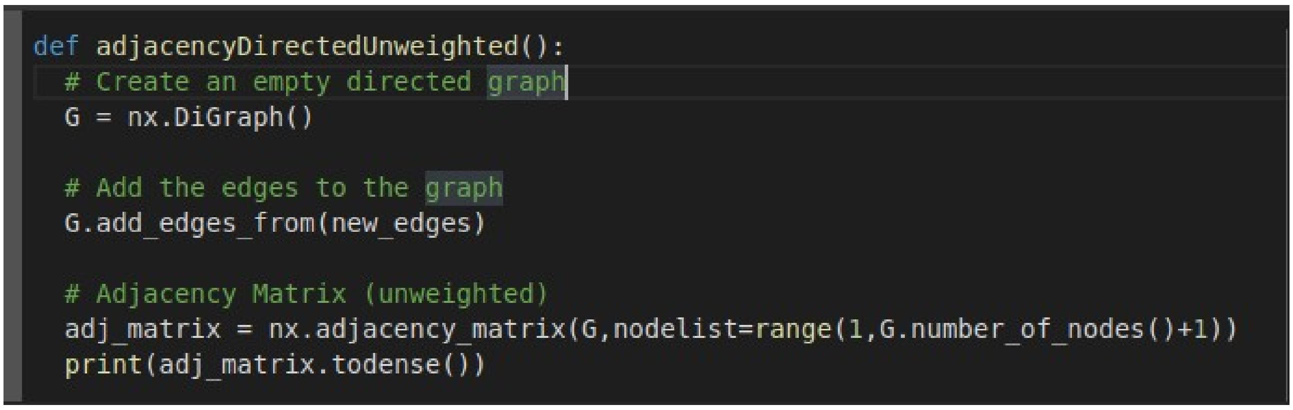
If the graph is undirected, the adjacency matrix is symmetric with respect to the main diagonal. If the graph is directed, the matrix may not be symmetric.

The diagonal entries of the adjacency matrix for a simple graph are always zero, since there are no self-loops in a simple graph.

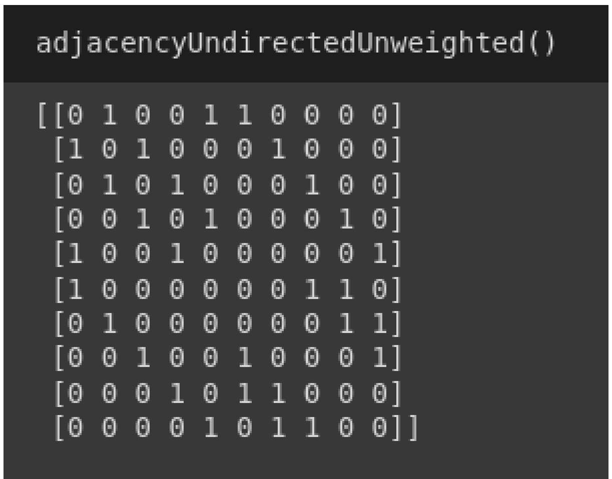
CODE :

PART – 3a :

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

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

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



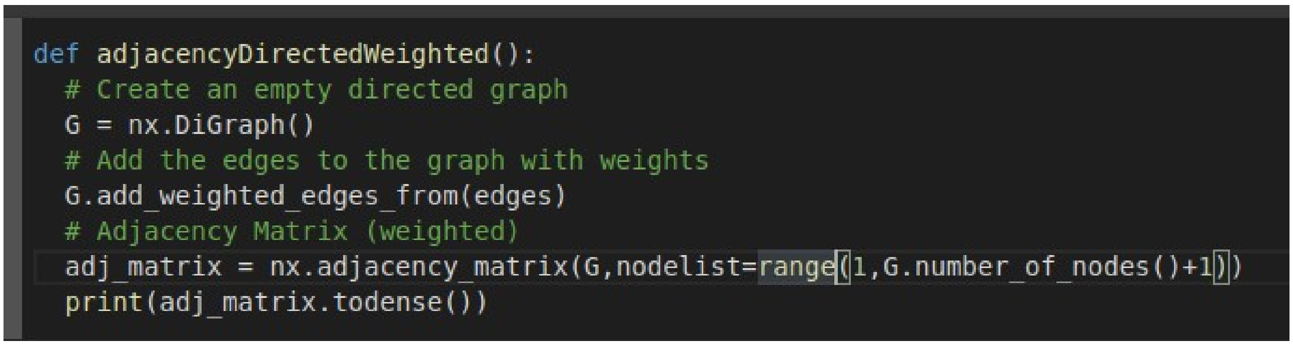
```
def adjacencyDirectedUnweighted():
  # Create an empty directed graph
  G = nx.DiGraph()

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

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



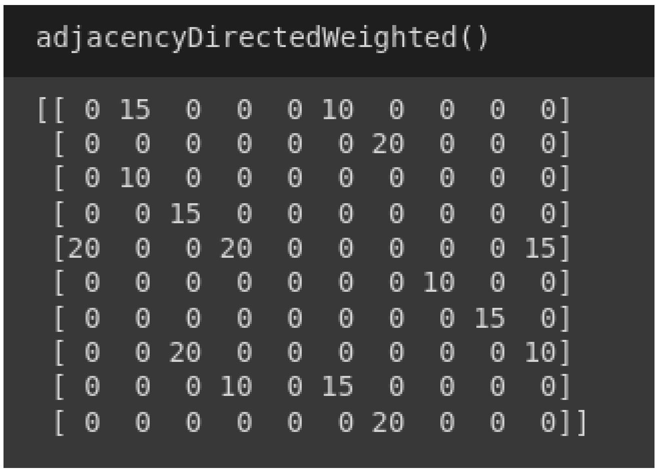
```
adjacencyUndirectedUnweighted()

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

PART – 3b :

```
def adjacencyDirectedWeighted():
   # Create an empty directed graph
   G = nx.DiGraph()
   # Add the edges to the graph with weights
   G.add_weighted_edges_from(edges)
   # Adjacency Matrix (weighted)
   adj_matrix =
nx.adjacency_matrix(G,nodelist=range(1,G.number_of_nodes()+1))
   print(adj_matrix.todense())
```



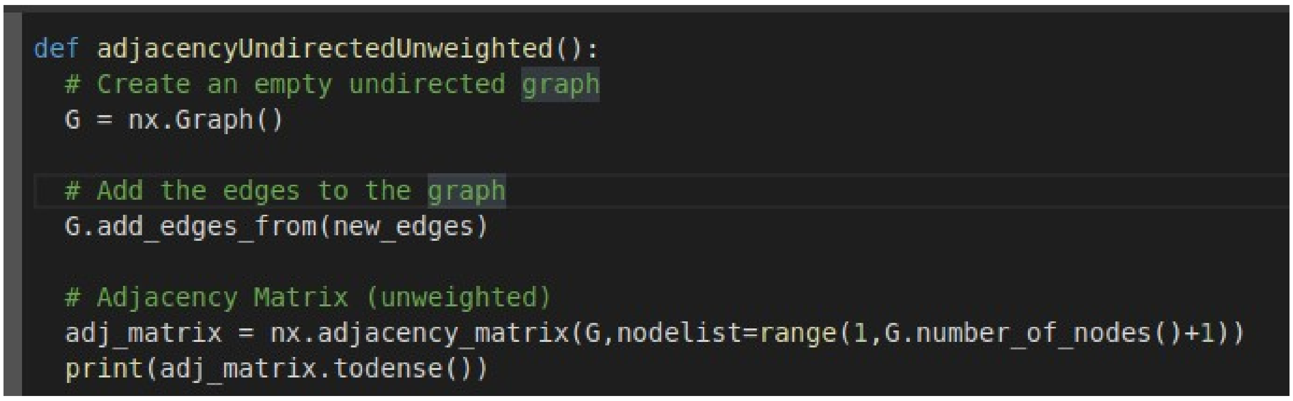
```
def adjacencyDirectedWeighted():
  # Create an empty directed graph
  G = nx.DiGraph()
  # Add the edges to the graph with weights
  G.add_weighted_edges_from(edges)
  # Adjacency Matrix (weighted)
  adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,G.number_of_nodes()+1))
  print(adj_matrix.todense())
```



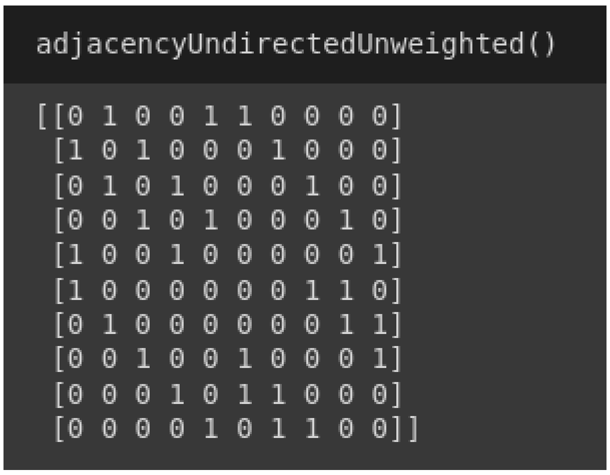
```
adjacencyDirectedWeighted()

[[ 0 15  0  0  0 10  0  0  0  0]
 [ 0  0  0  0  0  0 20  0  0  0]
 [ 0 10  0  0  0  0  0  0  0  0]
 [ 0  0 15  0  0  0  0  0  0  0]
 [20  0  0 20  0  0  0  0  0 15]
 [ 0  0  0  0  0  0  0 10  0  0]
 [ 0  0  0  0  0  0  0  0 15  0]
 [ 0  0 20  0  0  0  0  0  0 10]
 [ 0  0  0 10  0 15  0  0  0  0]
 [ 0  0  0  0  0  0 20  0  0  0]]
```

PART – 3c :

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

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

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



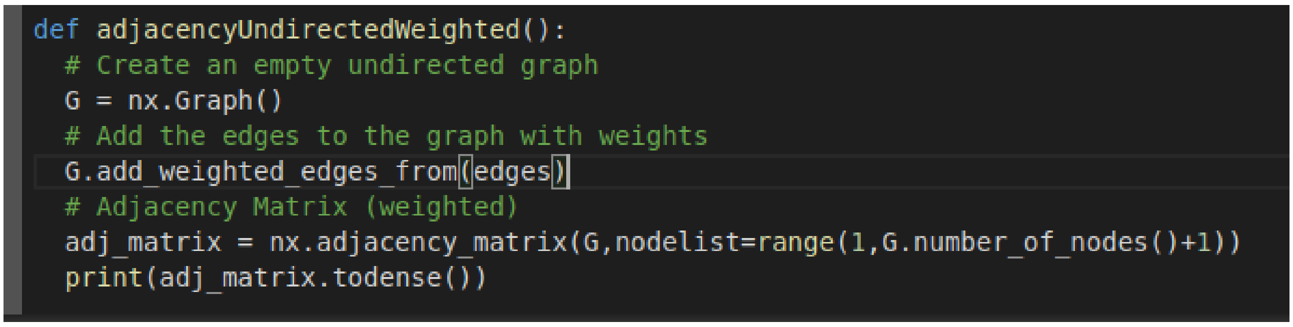
```
def adjacencyUndirectedUnweighted():
  # Create an empty undirected graph
  G = nx.Graph()

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

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



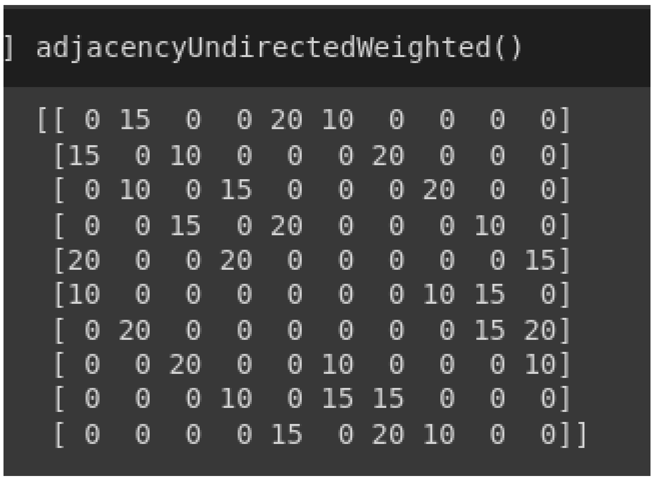
```
adjacencyUndirectedUnweighted()

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

PART – 3d :

```
def adjacencyUndirectedWeighted():
   # Create an empty undirected graph
   G = nx.Graph()
   # Add the edges to the graph with weights
   G.add_weighted_edges_from(edges)
   # Adjacency Matrix (weighted)
   adj_matrix =
nx.adjacency_matrix(G,nodelist=range(1,G.number_of_nodes()+1))
   print(adj_matrix.todense())
```



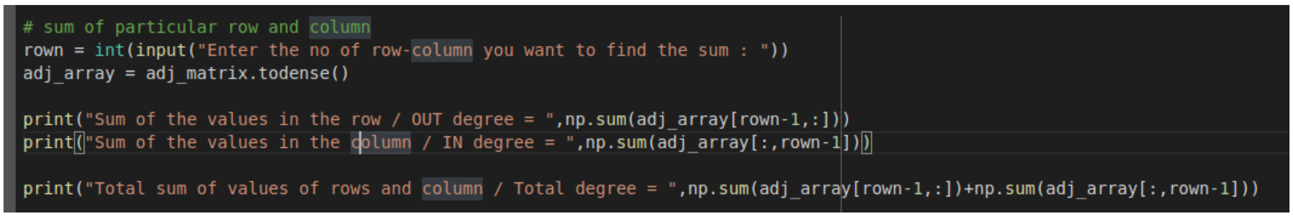
```
def adjacencyUndirectedWeighted():
  # Create an empty undirected graph
  G = nx.Graph()
  # Add the edges to the graph with weights
  G.add_weighted_edges_from(edges)
  # Adjacency Matrix (weighted)
  adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,G.number_of_nodes()+1))
  print(adj_matrix.todense())
```



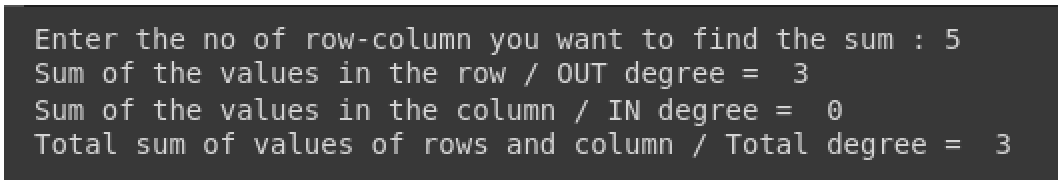
```
] adjacencyUndirectedWeighted()

[[ 0 15  0  0 20 10  0  0  0  0]
 [15  0 10  0  0  0 20  0  0  0]
 [ 0 10  0 15  0  0  0 20  0  0]
 [ 0  0 15  0 20  0  0  0 10  0]
 [20  0  0 20  0  0  0  0  0 15]
 [10  0  0  0  0  0  0 10 15  0]
 [ 0 20  0  0  0  0  0  0 15 20]
 [ 0  0 20  0  0 10  0  0  0 10]
 [ 0  0  0 10  0 15 15  0  0  0]
 [ 0  0  0  0 15  0 20 10  0  0]]
```

Find the Sum of the particular row :

```
# sum of particular row and column
rown = int(input("Enter the no of row-column you want to find the sum : "))
adj_array = adj_matrix.todense()

print("Sum of the values in the row / OUT degree = ",np.sum(adj_array[rown-1,:]))
print("Sum of the values in the column / IN degree = ",np.sum(adj_array[:,rown-1]))

print("Total sum of values of rows and column / Total degree = ",np.sum(adj_array[rown-1,:])+np.sum(adj_array[:,rown-1]))
```



```
# sum of particular row and column
rown = int(input("Enter the no of row-column you want to find the sum : "))
adj_array = adj_matrix.todense()

print("Sum of the values in the row / OUT degree = ",np.sum(adj_array[rown-1,:]))
print("Sum of the values in the column / IN degree = ",np.sum(adj_array[:,rown-1]))

print("Total sum of values of rows and column / Total degree = ",np.sum(adj_array[rown-1,:])+np.sum(adj_array[:,rown-1]))
```



```
Enter the no of row-column you want to find the sum : 5
Sum of the values in the row / OUT degree =  3
Sum of the values in the column / IN degree =  0
Total sum of values of rows and column / Total degree =  3
```

PART – 4 :

a)

For Directed Weighted Graph :

Print the data for all centralities for all the nodes.

Save all the data into CSV file.

b)

For Undirected Weighted Graph :

Print the data for all centralities for all the nodes.

Save all the data into CSV file.

PART – 4a :

THEORY :

1. Degree Centrality:

Degree centrality is a measure of the number of edges that are incident to a node (i.e., the number of connections a node has). It is calculated as the ratio of the number of nodes that are directly connected to a given node to the total number of nodes in the graph. The degree centrality for a node v in an undirected graph is given by:

degree_centrality(v) = degree(v) / (n-1) where degree(v) is the number of edges incident to node v, and n is the total number of nodes in the graph.

In NetworkX, you can calculate degree centrality using the nx.degree_centrality() function.

2. Betweenness Centrality:

Betweenness centrality is a measure of the extent to which a node lies on the shortest path between other nodes in the network. It is calculated as the ratio of the number of shortest paths that pass through a given node to the total number of shortest paths between all pairs of nodes in the graph. The betweenness centrality for a node v in an undirected graph is given by:

betweenness_centrality(v) = sum(sigma(s,t|v) / sigma(s,t))

where sigma(s,t) is the number of shortest paths between nodes s and t, and sigma(s,t|v) is the number of those shortest paths that pass through node v.

In NetworkX, you can calculate betweenness centrality using the nx.betweenness_centrality() function.

3. Closeness Centrality:

Closeness centrality is a measure of how close a node is to all other nodes in the network. It is calculated as the inverse of the average shortest path length between a given node and all other nodes in the graph. The closeness centrality for a node v in an undirected graph is given by:

closeness_centrality(v) = (n-1) / sum(shortest_path_length(v, u))

where shortest_path_length(v, u) is the length of the shortest path between node v and node u, and n is the total number of nodes in the graph.

In NetworkX, you can calculate closeness centrality using the nx.closeness_centrality() function.

4. PageRank Centrality:

PageRank centrality is a measure of the importance of a node in a network, based on the idea that important nodes are likely to be pointed to by other important nodes. It is calculated as the stationary distribution of a random walk on the graph, where at each step the walker has a probability alpha (the damping factor) of following an outgoing edge and a probability 1-alpha of jumping to a random node in the graph. The PageRank centrality for a node v in an undirected graph is given by:

page_rank_centrality(v) = (1 - alpha) * sum(page_rank_centrality(u) / out_degree(u)) + alpha * personalization[v]

where out_degree(u) is the number of edges originating from node u, and personalization[v] is the probability of starting at node v. By default, personalization is set to a uniform distribution over all nodes.

In NetworkX, you can calculate PageRank centrality using the nx.pagerank() function.

5. Eigenvector Centrality:

Eigenvector centrality is a measure of the influence of a node in a network, based on the idea that a node is important if it is connected to other important nodes. It is calculated as the principal eigenvector of the adjacency matrix of the graph. The eigenvector centrality for a node v in an undirected graph is given by:

eigen_value_centrality(v) = eigenvector_centrality(A)[v]

where A is the adjacency matrix of the graph.

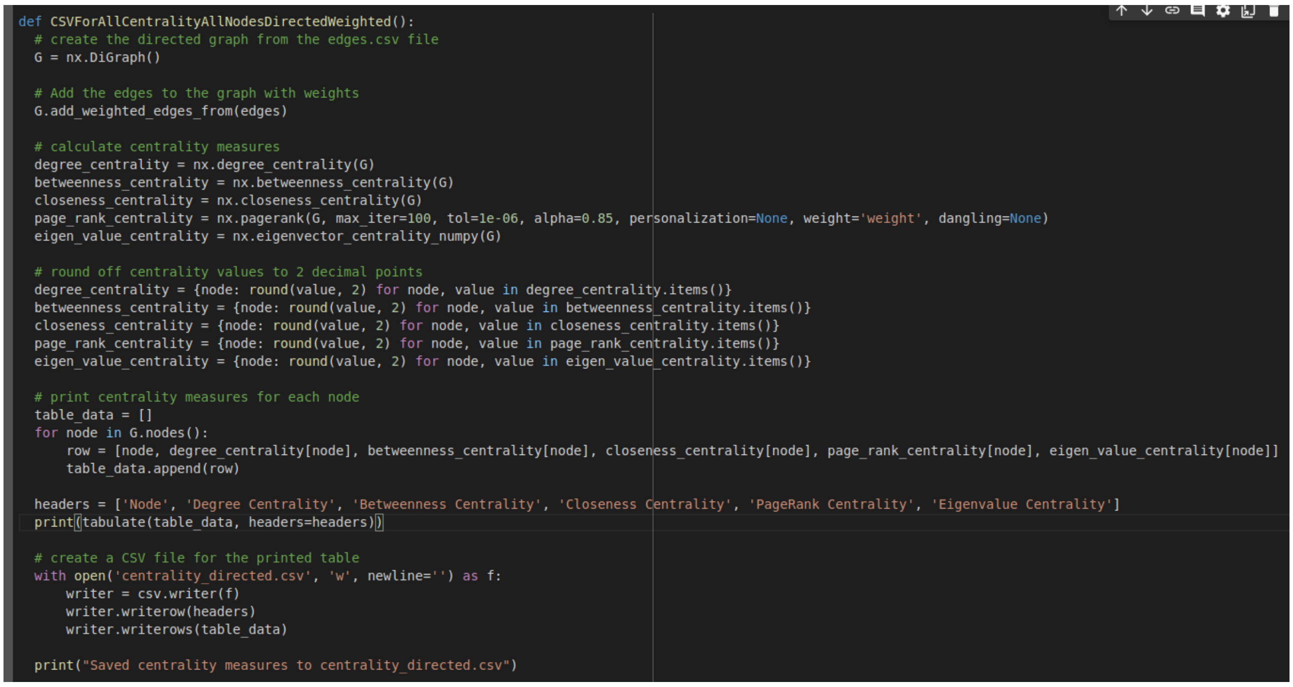
In NetworkX, you can calculate eigenvector centrality using the nx.eigenvector_centrality_numpy() function.

CODE :

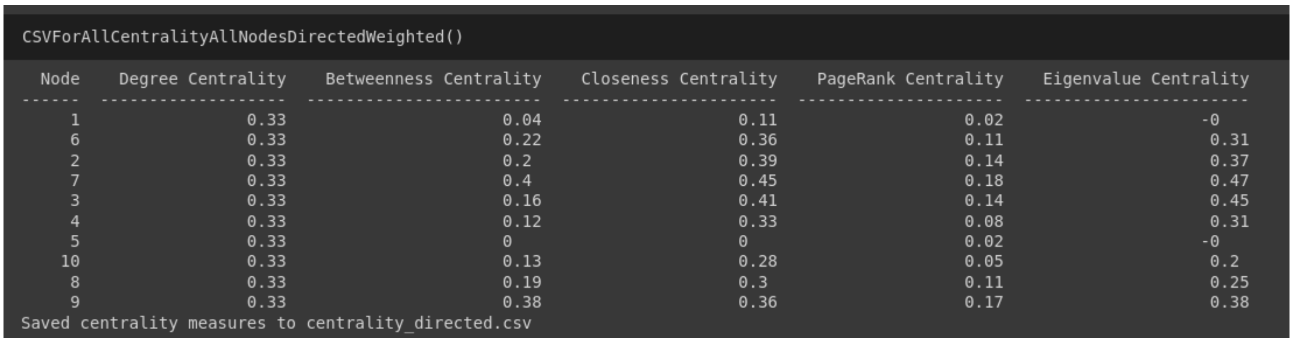
For Directed Graph

```
def CSVForAllCentralityAllNodesDirectedWeighted():
    # create the directed graph from the edges.csv file
    G = nx.DiGraph()

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

    # 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))

    # 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")
```



```
def CSVForAllCentralityAllNodesDirectedWeighted():
  # create the directed graph from the edges.csv file
  G = nx.DiGraph()

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

  # 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))

  # 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")
```



```
CSVForAllCentralityAllNodesDirectedWeighted()

  Node    Degree Centrality    Betweenness Centrality    Closeness Centrality    PageRank Centrality    Eigenvalue Centrality
------  -------------------  ------------------------  ----------------------  ---------------------  -----------------------
     1                 0.33                      0.04                    0.11                   0.02                       -0
     6                 0.33                      0.22                    0.36                   0.11                     0.31
     2                 0.33                      0.2                     0.39                   0.14                     0.37
     7                 0.33                      0.4                     0.45                   0.18                     0.47
     3                 0.33                      0.16                    0.41                   0.14                     0.45
     4                 0.33                      0.12                    0.33                   0.08                     0.31
     5                 0.33                      0                       0                      0.02                    -0
    10                 0.33                      0.13                    0.28                   0.05                     0.2
     8                 0.33                      0.19                    0.3                    0.11                     0.25
     9                 0.33                      0.38                    0.36                   0.17                     0.38
Saved centrality measures to centrality_directed.csv
```

PART _ 4b :

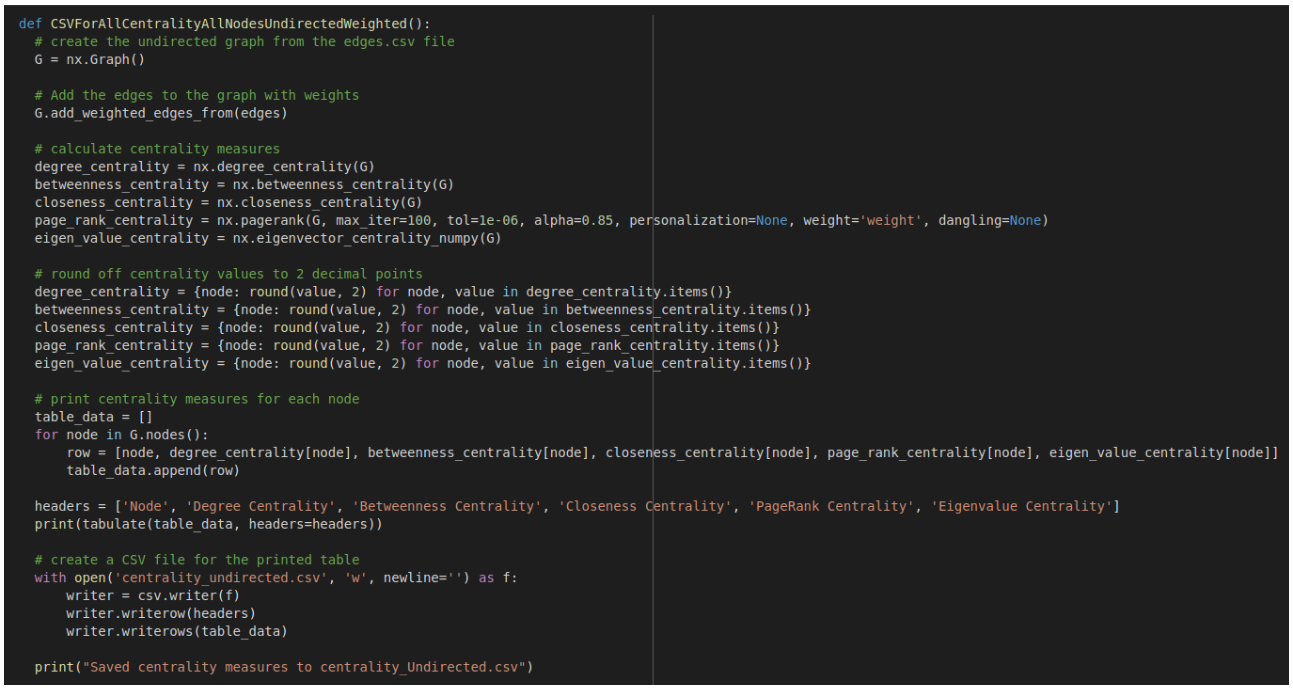
For undirected graph :

CODE :

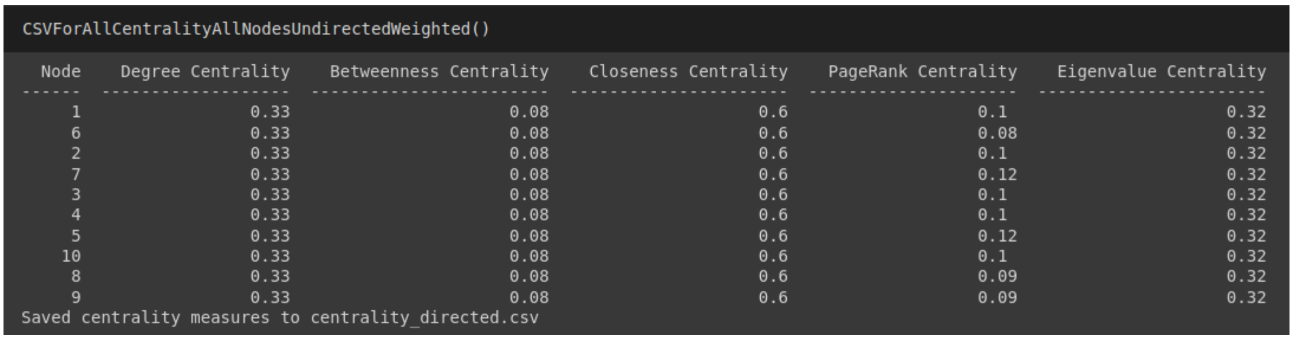
```
def CSVForAllCentralityAllNodesUndirectedWeighted():
  # create the undirected graph from the edges.csv file
```

```
   G = nx.Graph()

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

   # 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))

  # 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")
```



```python
def CSVForAllCentralityAllNodesUndirectedWeighted():
  # create the undirected graph from the edges.csv file
  G = nx.Graph()

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

  # 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))

  # 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")
```



```
CSVForAllCentralityAllNodesUndirectedWeighted()

  Node    Degree Centrality    Betweenness Centrality    Closeness Centrality    PageRank Centrality    Eigenvalue Centrality
------  -------------------  ------------------------  ----------------------  ---------------------  -----------------------
     1                 0.33                      0.08                     0.6                   0.1                      0.32
     6                 0.33                      0.08                     0.6                   0.08                     0.32
     2                 0.33                      0.08                     0.6                   0.1                      0.32
     7                 0.33                      0.08                     0.6                   0.12                     0.32
     3                 0.33                      0.08                     0.6                   0.1                      0.32
     4                 0.33                      0.08                     0.6                   0.1                      0.32
     5                 0.33                      0.08                     0.6                   0.12                     0.32
    10                 0.33                      0.08                     0.6                   0.1                      0.32
     8                 0.33                      0.08                     0.6                   0.09                     0.32
     9                 0.33                      0.08                     0.6                   0.09                     0.32
Saved centrality measures to centrality_directed.csv
```

PART – 5 :

a)

For Directed Graph calculate the maximum and minimum node according to each centrality.

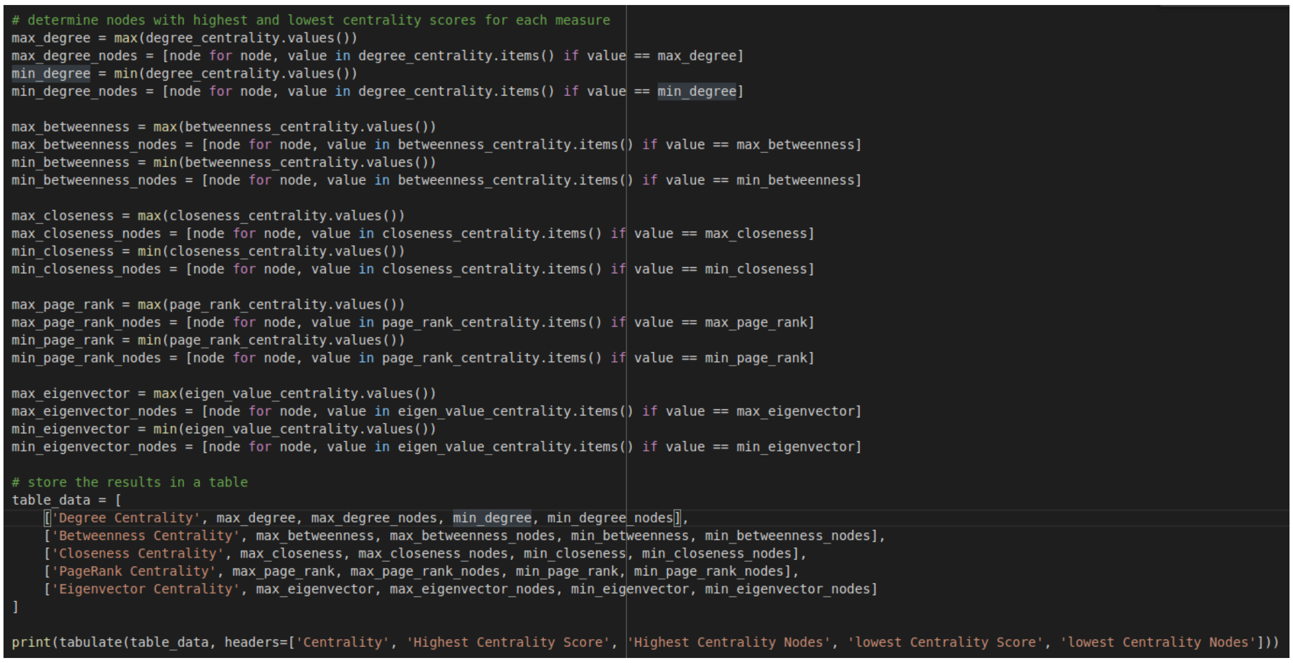
b)

For Undiirected Graph calculate the maximum and minimum node according to each centrality.

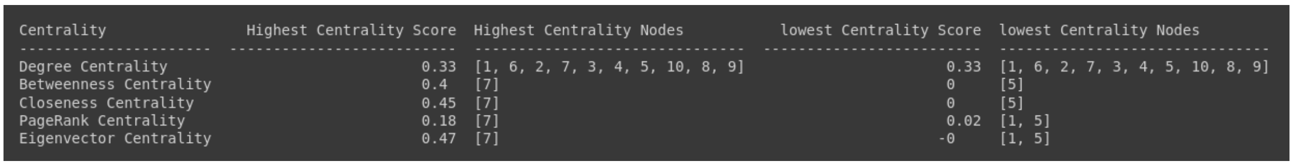
PART – 5a (Directed Graph) :

```
# 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(eigen_value_centrality.values())
    max_eigenvector_nodes = [node for node, value in
eigen_value_centrality.items() if value == max_eigenvector]
    min_eigenvector = min(eigen_value_centrality.values())
    min_eigenvector_nodes = [node for node, value in
eigen_value_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']))



```
# 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(eigen_value_centrality.values())
max_eigenvector_nodes = [node for node, value in eigen_value_centrality.items() if value == max_eigenvector]
min_eigenvector = min(eigen_value_centrality.values())
min_eigenvector_nodes = [node for node, value in eigen_value_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']))
```



```
Centrality                Highest Centrality Score  Highest Centrality Nodes            lowest Centrality Score  lowest Centrality Nodes
----------------------  --------------------------  --------------------------------  -------------------------  --------------------------------
Degree Centrality                             0.33  [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]                        0.33  [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]
Betweenness Centrality                        0.4   [7]                                                    0     [5]
Closeness Centrality                          0.45  [7]                                                    0     [5]
PageRank Centrality                           0.18  [7]                                                    0.02  [1, 5]
Eigenvector Centrality                        0.47  [7]                                                   -0     [1, 5]
```

PART – 5b (Undirected Graph) :

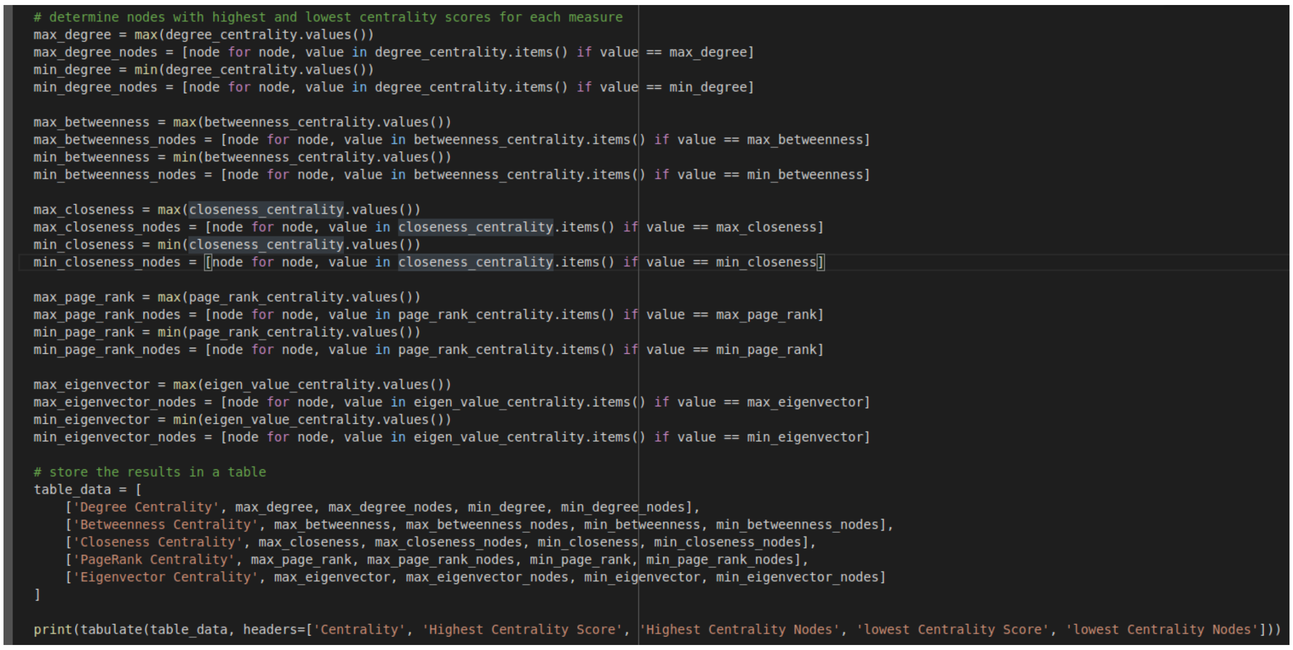
CODE :

determine nodes with highest and lowest centrality scores for each measure

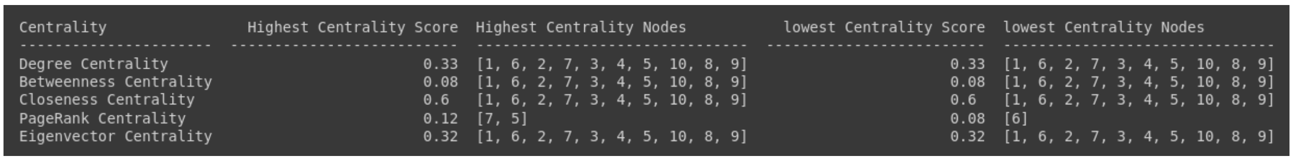
max_degree = max(degree_centrality.values())

```
    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(eigen_value_centrality.values())
    max_eigenvector_nodes = [node for node, value in
eigen_value_centrality.items() if value == max_eigenvector]
    min_eigenvector = min(eigen_value_centrality.values())
    min_eigenvector_nodes = [node for node, value in
eigen_value_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']))
```

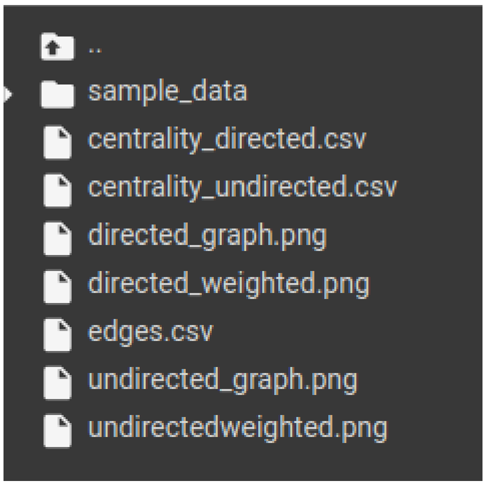


```
# determine nodes with highest and lowest centrality scores for each measure
max_degree = max(degree_centrality.values())
max_degree_nodes = [node for node, value in degree_centrality.items() if value == max_degree]
min_degree = min(degree_centrality.values())
min_degree_nodes = [node for node, value 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(eigen_value_centrality.values())
max_eigenvector_nodes = [node for node, value in eigen_value_centrality.items() if value == max_eigenvector]
min_eigenvector = min(eigen_value_centrality.values())
min_eigenvector_nodes = [node for node, value in eigen_value_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']))
```



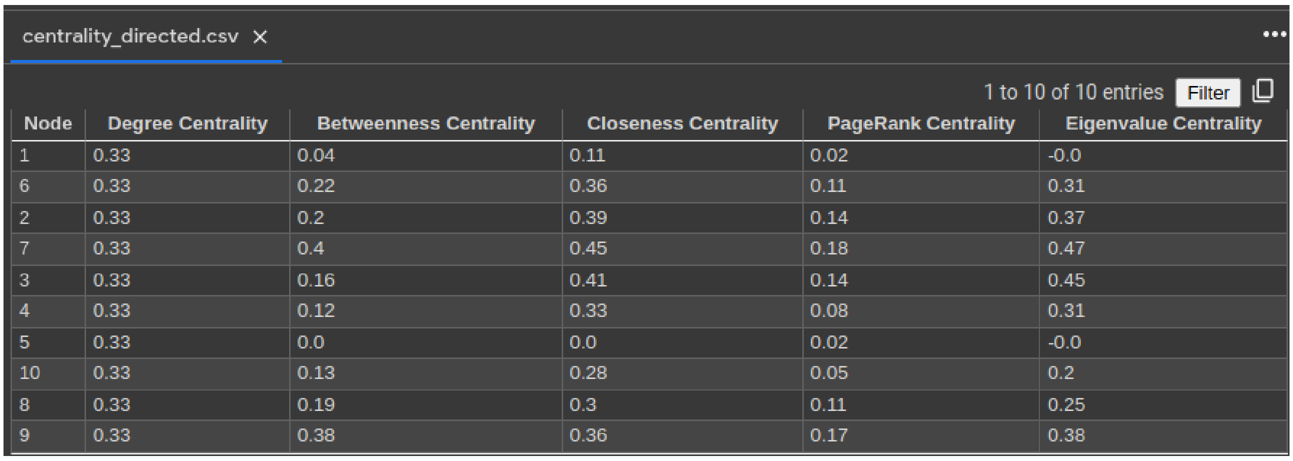
```
Centrality                Highest Centrality Score  Highest Centrality Nodes             lowest Centrality Score  lowest Centrality Nodes
----------------------  --------------------------  ----------------------------------  -------------------------  ----------------------------------
Degree Centrality                             0.33  [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]                          0.33  [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]
Betweenness Centrality                        0.08  [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]                          0.08  [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]
Closeness Centrality                          0.6   [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]                          0.6   [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]
PageRank Centrality                           0.12  [7, 5]                                                   0.08  [6]
Eigenvector Centrality                        0.32  [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]                          0.32  [1, 6, 2, 7, 3, 4, 5, 10, 8, 9]
```

FILES GENERATED DURING CODE EXECUTION :



```
..
sample_data
centrality_directed.csv
centrality_undirected.csv
directed_graph.png
directed_weighted.png
edges.csv
undirected_graph.png
undirectedweighted.png
```

1) centrality_directed.csv

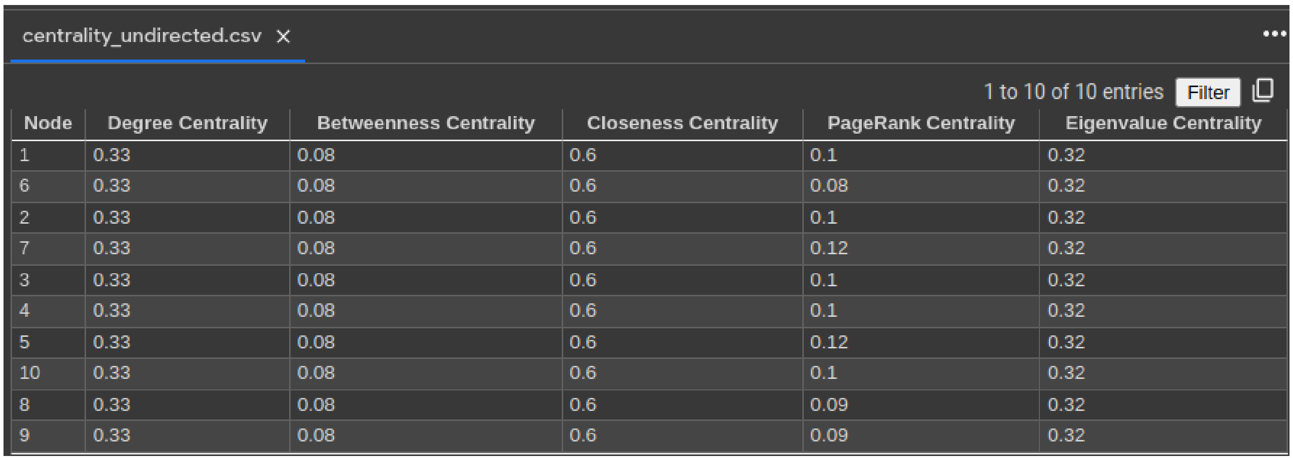


centrality_directed.csv ×

1 to 10 of 10 entries Filter

Node	Degree Centrality	Betweenness Centrality	Closeness Centrality	PageRank Centrality	Eigenvalue Centrality
1	0.33	0.04	0.11	0.02	-0.0
6	0.33	0.22	0.36	0.11	0.31
2	0.33	0.2	0.39	0.14	0.37
7	0.33	0.4	0.45	0.18	0.47
3	0.33	0.16	0.41	0.14	0.45
4	0.33	0.12	0.33	0.08	0.31
5	0.33	0.0	0.0	0.02	-0.0
10	0.33	0.13	0.28	0.05	0.2
8	0.33	0.19	0.3	0.11	0.25
9	0.33	0.38	0.36	0.17	0.38

2) centrality_undirected.csv

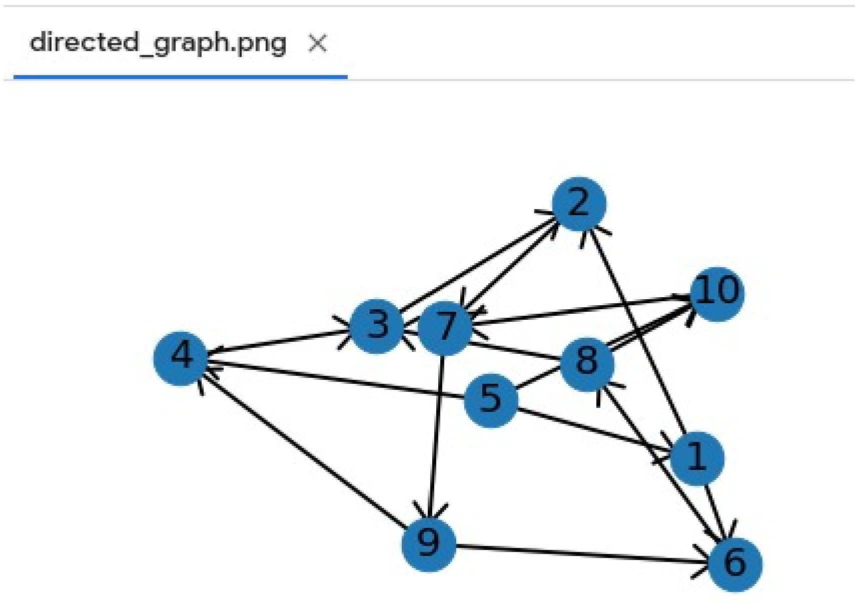


centrality_undirected.csv ×

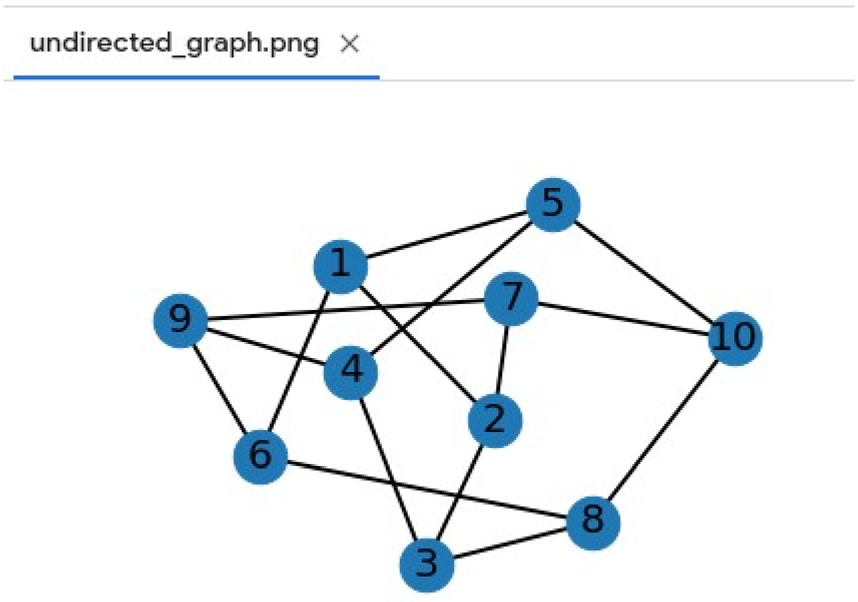
1 to 10 of 10 entries Filter

Node	Degree Centrality	Betweenness Centrality	Closeness Centrality	PageRank Centrality	Eigenvalue Centrality
1	0.33	0.08	0.6	0.1	0.32
6	0.33	0.08	0.6	0.08	0.32
2	0.33	0.08	0.6	0.1	0.32
7	0.33	0.08	0.6	0.12	0.32
3	0.33	0.08	0.6	0.1	0.32
4	0.33	0.08	0.6	0.1	0.32
5	0.33	0.08	0.6	0.12	0.32
10	0.33	0.08	0.6	0.1	0.32
8	0.33	0.08	0.6	0.09	0.32
9	0.33	0.08	0.6	0.09	0.32

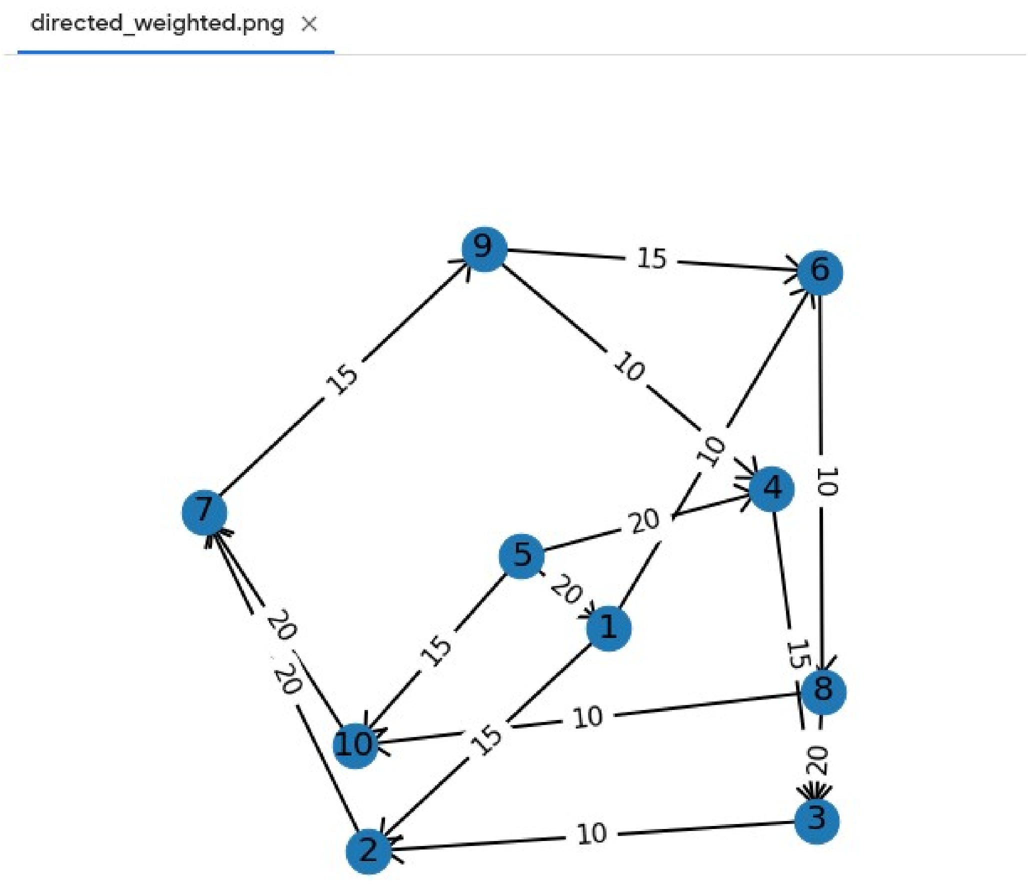
3) directed_graph.png



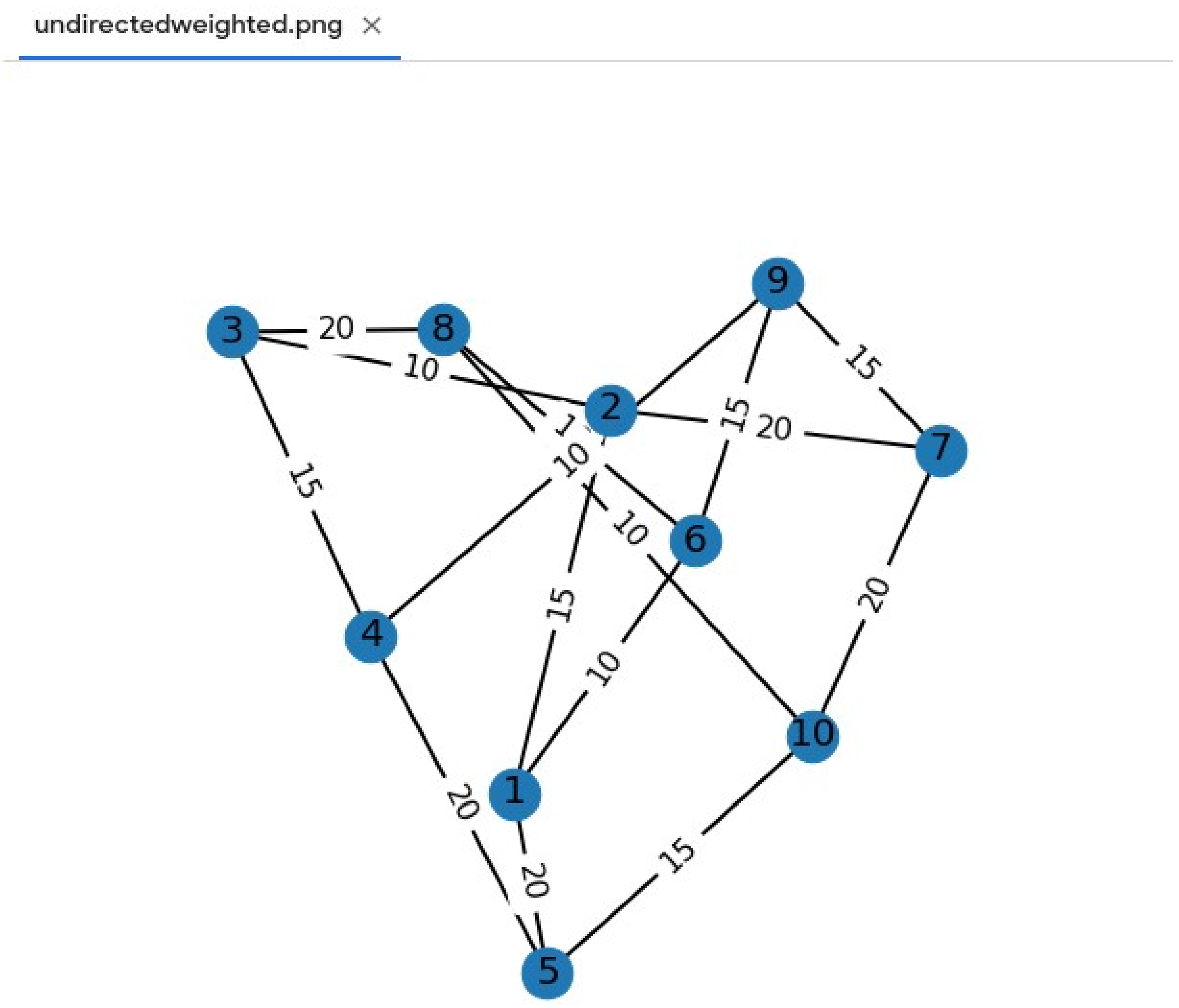
4) Undirected_graph.png



5) directed_weighted.png



6) Undirected_weighted.png



ALL CODE SNIPPET :

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

with open('edges.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)
    edges = [(int(row[0]), int(row[1]), int(row[2])) for row in reader]
new_edges = [(int(row[0]), int(row[1])) for row in edges]

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

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

  # positions for all nodes
  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],
                                   arrowsize=20, arrowstyle='->, head_width=0.4,
head_length=0.5')
  nx.draw_networkx_labels(G, pos, font_size=20, font_family='sans-serif')
  plt.axis('off')
  plt.savefig('directed_graph.png')
  plt.show()
```

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

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

    # positions for all nodes
    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.axis('off')
    plt.savefig('undirected_graph.png')
    plt.show()

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

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

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

    # Set the figure size
    plt.figure(figsize=(8, 8))
    # Draw the nodes
    nx.draw_networkx_nodes(G, pos, node_size=700)

    # Draw the edges with arrows and weights
    edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
    nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2],
                                 arrowsize=20, arrowstyle='->, head_width=0.4,
head_length=0.5')
    edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

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

    # Turn off the axis
    plt.axis('off')
    plt.savefig('directed_weighted.png')
    # Show the plot
    plt.show()

def undirectedweighted():
    # Create an empty Undirected graph
    G = nx.Graph()

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

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

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

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

    # Draw the edges with arrows and weights
    edge_widths = [d['weight'] for (u, v, d) in G.edges(data=True)]
    nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=[2])
    edge_labels = {(u, v): f'{d["weight"]}' for (u, v, d) in G.edges(data=True)}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=16)

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

    # Turn off the axis
    plt.axis('off')
    plt.savefig('undirectedweighted.png')
    # Show the plot
    plt.show()

def undirectedDetails():
    # Create an empty Undirected graph
    G = nx.Graph()

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

    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()

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

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

    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(fMaximum out degree: {max_out_degree}')
    print(fMinimum 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(fNodes with maximum out-degree: {max_out_degree_nodes}')
    print(fNodes 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(fMaximum in degree: {max_in_degree}')
    print(fMinimum 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(fNodes with maximum in-degree: {max_in_degree_nodes}')
    print(fNodes with minimum in-degree: {min_in_degree_nodes}')

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

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

    # Adjacency Matrix (unweighted)
    adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,G.number_of_nodes()+1))
    print(adj_matrix.todense())

def adjacencyUndirectedWeighted():
    # Create an empty undirected graph
    G = nx.Graph()
    # Add the edges to the graph with weights
    G.add_weighted_edges_from(edges)
    # Adjacency Matrix (weighted)
    adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,G.number_of_nodes()+1))
    print(adj_matrix.todense())

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

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

    # Adjacency Matrix (unweighted)
    adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,G.number_of_nodes()+1))
    print(adj_matrix.todense())

def adjacencyDirectedWeighted():
    # Create an empty directed graph
    G = nx.DiGraph()
    # Add the edges to the graph with weights
    G.add_weighted_edges_from(edges)
    # Adjacency Matrix (weighted)
    adj_matrix = nx.adjacency_matrix(G,nodelist=range(1,G.number_of_nodes()+1))
    print(adj_matrix.todense())

def CSVForAllCentralityAllNodesDirectedWeighted():
    # create the directed graph from the edges.csv file
    G = nx.DiGraph()

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

    # 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))

  # 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")

  print()

  # 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(eigen_value_centrality.values())
    max_eigenvector_nodes = [node for node, value in
eigen_value_centrality.items() if value == max_eigenvector]
    min_eigenvector = min(eigen_value_centrality.values())
    min_eigenvector_nodes = [node for node, value in
eigen_value_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']))

def CSVForAllCentralityAllNodesUndirectedWeighted():
   # create the undirected graph from the edges.csv file
   G = nx.Graph()

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

   # 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))

    # 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")

  print()

  # 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(eigen_value_centrality.values())
    max_eigenvector_nodes = [node for node, value in
eigen_value_centrality.items() if value == max_eigenvector]
    min_eigenvector = min(eigen_value_centrality.values())
    min_eigenvector_nodes = [node for node, value in
eigen_value_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']))
```

directedUnweighted()

undirectedUnweighted()

directedweighted()

undirectedweighted()

undirectedDetails()

directedDetails()

adjacencyUndirectedUnweighted()

adjacencyUndirectedWeighted()

adjacencyDirectedUnweighted()

adjacencyDirectedWeighted()

CSVForAllCentralityAllNodesDirectedWeighted()

CSVForAllCentralityAllNodesUndirectedWeighted()