## Appendix

```
#!/usr/bin/env python3

import random
import networkx as nx
import numpy
from collections import deque
import matplotlib.pyplot as plt
nodenum=10
edgenum=12

import numpy as np
from collections import deque
def visualize_adjacency_matrix(adj_matrix):
        # Create a graph from the adjacency matrix
        G = nx.from_numpy_array(adj_matrix)

        # Draw the graph
        pos = nx.spring_layout(G)  # positions for all nodes
        nx.draw(G, pos, with_labels=True, cmap=plt.cm.Blues,
              node_color='skyblue', node_size=50)

        # Display the graph
        plt.show()

def visualize_graph_with_labels_and_colors(adjacency_matrix, color_array):
        # Create a graph from the adjacency matrix
        G = nx.Graph()
        for i in range(len(adjacency_matrix)):
                for j in range(i+1, len(adjacency_matrix)):
                       if adjacency_matrix[i][j] != 0: # Assuming 0 means
                            no edge
                               # Add an edge with a label, rounded to 2
                                   decimal places
                               G.add_edge(i, j,
                                   weight=round(adjacency_matrix[i][j], 2))

        # Determine node colors based on the color_array
        node_colors = ['pink' if color_array[i] == 1 else 'blue' for i in
              range(len(color_array))]

        # Position nodes using the spring layout
        pos = nx.spring_layout(G)
        # Draw the nodes with determined colors
        nx.draw(G, pos, with_labels=True, node_color=node_colors,
              edge_color='gray')
        # Draw edge labels
        edge_labels = nx.get_edge_attributes(G, 'weight')
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

        plt.show()

# Example usage
#visualize_graph_with_labels_and_colors([[0, 1.123456, 0], [1.123456, 0,
       2.654321], [0, 2.654321, 0]], [1, 0, 1])
def visualize_graph(graph):
        G = nx.Graph()

        # Add edges to the graph
        for i in range(len(graph)):
                for j in range(len(graph[i])):
```

```
                if graph[i][j] != 0:
                    label = "{:.2f}".format(graph[i][j])  # 
                        Round to 2 decimal places
                    G.add_edge(i, j, weight=label)

    # Draw the graph
    pos = nx.spring_layout(G)  # Positions nodes using Fruchterman-
        Reingold force-directed algorithm
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw(G, pos, with_labels=True, node_color='skyblue',
        node_size=150, font_size=10)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

    plt.show()


def is_fully_interconnected(adj_matrix):
    n = len(adj_matrix)
    visited = np.zeros(n, dtype=bool)

    # Function to perform BFS
    def bfs(start):
        visited[start] = True
        queue = deque([start])
        while queue:
            node = queue.popleft()
            for neighbor in np.where(adj_matrix[node] == 1)[0]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append(neighbor)

    # Start BFS from each node
    for i in range(n):
        visited.fill(False)
        bfs(i)
        if not all(visited):
            return False

    return True
def generate_random_graph(nodes, edges):
    if edges > nodes * (nodes - 1) / 2:
        raise ValueError("Too many edges for the given number of
            nodes")
    G = nx.Graph()
    G.add_nodes_from(range(nodes))
    edge_count = 0
    while edge_count < edges:
        node1 = random.randint(0, nodes - 1)
        node2 = random.randint(0, nodes - 1)
        if node1 != node2 and not G.has_edge(node1, node2):
            G.add_edge(node1, node2)
            edge_count += 1
    return G
firstStep={}
def bfs_with_distances(graph):
    distances = {}
    global firstStep
    for node in graph.nodes():
        distances[node] = {}
        firstStep[node] = {}

        #firstStep={}
        queue = [(node, 0)]
```

```
        queuefirststep=[-1]
        visited = set()
        while queue:
            current_node, distance = queue.pop(0)
            firstS=queuefirststep.pop(0)
            if current_node not in visited:
                visited.add(current_node)
                distances[node][current_node] = distance
                neighbors =
                    list(graph.neighbors(current_node))
                if distance==1:
                    firstS=current_node
                firstStep[node][current_node] = firstS

                for neighbor in neighbors:
                    if neighbor not in visited:
                        queue.append((neighbor,
                          distance + 1))

                          queuefirststep.append(firs
                          tS)
    return distances
def calcf(adjacency_matrix,distances,firstStep,particals):
    f={}
    for i in range (nodenum):
        f[i]={}
        for j in range (nodenum):
            f[i][j]=0
        for j in range (nodenum):
            if(i==j):
                continue
            fcon=-1
            if(particals[i]==particals[j]):
                fcon=1
            f_act=fcon/distances[i][j]**2
            f[i][int(firstStep[i][j])]+=f_act
    return f
def main():
    particals=[0,0,0,0,0,1,1,1,1,1]
    nodes = nodenum
    edges = edgenum
    graph = generate_random_graph(nodes, edges)
    adjacency_matrix = nx.to_numpy_array(graph)
    while(not is_fully_interconnected(adjacency_matrix)):
        graph = generate_random_graph(nodes, edges)
        adjacency_matrix = nx.to_numpy_array(graph)
        print("graph generation failure!")

    print("\ngraph generation success!\nGenerated Graph (Adjacency
          Matrix):")


    adjacency_matrix=numpy.matrix(adjacency_matrix)
    graph=nx.from_numpy_array(adjacency_matrix)

    print(adjacency_matrix)

    distances = bfs_with_distances(graph)
    print("\nDistances from each node to all other nodes:")
    for node, distances_from_node in distances.items():
        print(f"From node {node}: {distances_from_node}")
    print()
    for node, distances_from_node in firstStep.items():
```

```
                print(f"From node FS {node}: {distances_from_node}")
        lasttension=[]
        for i in range(nodenum):
                lasttension.append([])
                for j in range(nodenum):
                        lasttension[i].append(0)
        while 1:
                fcur=calcf(adjacency_matrix,distances,firstStep,particals)
                for node, distances_from_node in fcur.items():
                        print(f"From node Force {node}:
                             {distances_from_node}")
                print()
                inforce=[]
                for i in range(nodenum):
                        inforce.append(0)
                        for j in range(nodenum):
                                inforce[i]+=fcur[i][j];
                print(inforce)
                edgelist=graph.edges()
                maxtension=-10000
                maxtindex=0
                for edge in edgelist:
                        start_node, end_node =edge
                        fstart=inforce[start_node]-2*fcur[start_node]
                             [end_node]
                        fend=inforce[end_node]-2*fcur[end_node][start_node]
                        t=fstart+fend
                        print(f"Edge: {start_node} -> {end_node} tension:
                             {t}")
                        lasttension[start_node][end_node]=t
                        lasttension[end_node][start_node]=t
                        if(t>maxtension):
                                maxtension=t
                                maxtindex=(start_node, end_node)
                if maxtension>2:
                     particals[maxtindex[0]],particals[maxtindex[1]]=partic
                     als[maxtindex[1]],particals[maxtindex[0]]
                else:
                        break
                print('========================================')
                print(particals)
        print('========================================')
        print(particals)
        print(lasttension)
        visualize_graph_with_labels_and_colors(lasttension,particals)
if __name__ == "__main__":
        main()
```