Richard Boysen
2/16/2024

Project 1 Part B

Colab Link to Code

1. **R2**
   Answer to Question: I would recommend (1,4) and (0,3) for construction.

   Code Output:
   Benefit values for constructing new roads:
   Road (0, 2): Benefit -19
   Road (0, 3): Benefit -16
   Road (1, 2): Benefit -25
   Road (1, 4): Benefit -15
   Road (2, 3): Benefit -17

   Recommended roads for construction:
   (1, 4)
   (0, 3)

2. **R3**
   a) The k roads that need to be constructed:
   Start: 0, End: 2
   Start: 1, End: 2
   Start: 2, End: 3

   b) The benefit value for each of the 3 roads recommended for construction:
   Road (0, 2): Benefit -5
   Road (1, 2): Benefit -11
   Road (2, 3): Benefit -12

3. **R4**
   a) The k roads that need to be constructed:
   Start: 2, End: 3
   Start: 0, End: 3
   Start: 1, End: 2

   b) The benefit value for each of the 3 roads recommended for construction:
   Road (2, 3): Benefit -10
   Road (0, 3): Benefit -12
   Road (1, 2): Benefit -14

   c)The recommendations changed due to the increase in f.

Richard Boysen
2/16/2024

4. **<u>R5</u>**

   a) Benefit values for constructing new roads:
   Road (0, 2): Benefit -19
   Road (0, 3): Benefit -16
   Road (1, 2): Benefit -25
   Road (1, 4): Benefit -15
   Road (2, 3): Benefit -17

   b) Recommended roads for construction:
   (1, 4) = -15
   (0, 3) = -16
   (2, 3) = -17

   c) Top 1 value changed by 5, 2nd changed by 4, third changed by 3. These values changed due to the change in connectivity impacting the simulation.

#See next pages for code in order of requirement

Richard Boysen
2/16/2024

Requirement 2

```
#R2
import networkx as nx

class Road:
    def __init__(self, start_node, end_node, weight):
        self.start_node = start_node
        self.end_node = end_node
        self.weight = weight
        self.benefit = None  # Initial benefit is set to None

    def contains(self, nodes):
        return self.start_node in nodes or self.end_node in nodes

    def update_benefit(self, new_benefit):
        self.benefit = new_benefit

class Graph:
    def __init__(self):
        self.roads = []

    def add_road(self, road):
        self.roads.append(road)

    def calculate_shortest_paths(self):
        G = nx.Graph()
        for road in self.roads:
            G.add_edge(road.start_node, road.end_node, weight=road.weight)
        self.shortest_paths = dict(nx.all_pairs_dijkstra_path_length(G))

    def update_benefits(self):
        for road in self.roads:
            # Assuming calculate_new_benefit() computes the new benefit
based on changes in the graph
            new_benefit = calculate_new_benefit(self, road)
            road.update_benefit(new_benefit)

def compute_benefit_values(graph, new_roads):
    # Compute the benefit values for the specified new roads
```

Richard Boysen
2/16/2024

```
    benefit_matrix = {}
    for road in new_roads:
        start, end = road
        weight = graph.shortest_paths[start][end]
        if weight is not None:
            benefit_matrix[road] = -weight
        else:
            benefit_matrix[road] = "Not computed"
    return benefit_matrix

def recommend_new_roads(k, benefit_matrix):
    # Recommend the top k roads based on the benefit values
    sorted_benefits = sorted(benefit_matrix.items(), key=lambda x: x[1],
reverse=True)[:k]
    recommended_roads = [road for road, _ in sorted_benefits]
    return recommended_roads

def calculate_new_benefit(graph, road):
    # function for computing the new benefit
    # let's assume traffic volume as a proxy for benefit
    traffic_volume = calculate_traffic_volume(graph, road)
    new_benefit = 1 / (1 + traffic_volume)  # Inverse relationship: as
traffic volume increases, benefit decreases
    return new_benefit

def calculate_traffic_volume(graph, road):
    # function for computing traffic volume
    # let's assume traffic volume is based on the shortest path weight
    start = road.start_node
    end = road.end_node
    shortest_path_weight = graph.shortest_paths[start][end]
    if shortest_path_weight is not None:
        # Assuming traffic volume is inversely proportional to shortest
path weight
        traffic_volume = 1 / (1 + shortest_path_weight)
    else:
        # If there's no shortest path, assume high traffic volume
        traffic_volume = float('inf')
    return traffic_volume
```

Richard Boysen
2/16/2024

```
def main():
    graph = Graph()
    # Construct the network
    graph.add_road(Road(1, 3, 11))
    graph.add_road(Road(3, 4, 7))
    graph.add_road(Road(4, 2, 10))
    graph.add_road(Road(4, 0, 9))
    graph.add_road(Road(0, 1, 6))

    graph.calculate_shortest_paths()

    new_roads = [(0, 2), (0, 3), (1, 2), (1, 4), (2, 3)]
    k = 2  # Budget

    benefit_matrix = compute_benefit_values(graph, new_roads)
    recommended_roads = recommend_new_roads(k, benefit_matrix)

    print("Benefit values for constructing new roads:")
    for road, benefit in benefit_matrix.items():
        if benefit == "Not computed":
            print(f"Road {road}: Benefit Not computed")
        else:
            print(f"Road {road}: Benefit {benefit}")

    print("\nRecommended roads for construction:")
    for recommended_road in recommended_roads:
        print(recommended_road)

    # Update benefits
    graph.update_benefits()

if __name__ == "__main__":
    main()
```

Richard Boysen
2/16/2024

Requirement 3

```
import networkx as nx
import random

# Define constants
N = 60  # Number of nodes
p = 5   # Connectivity parameter (smaller value for a small network)
L_RANGE = (5, 25)  # Range for road length
T = 100  # Number of trips
k = 3    # Budget parameter
f = 0.6  # Shrinkage factor

class Road:
    def __init__(self, start_node, end_node, weight):
        self.start_node = start_node
        self.end_node = end_node
        self.weight = weight
        self.benefit = None  # Initial benefit is set to None

    def contains(self, nodes):
        return self.start_node in nodes or self.end_node in nodes

    def update_benefit(self, new_benefit):
        self.benefit = new_benefit

class Graph:
    def __init__(self):
        self.roads = []

    def add_road(self, road):
        self.roads.append(road)

    def calculate_shortest_paths(self):
        G = nx.Graph()
        for road in self.roads:
            G.add_edge(road.start_node, road.end_node, weight=road.weight)
        self.shortest_paths = dict(nx.all_pairs_dijkstra_path_length(G))

    def update_benefits(self):
```

Richard Boysen
2/16/2024

```
        for road in self.roads:
            # Assuming calculate_new_benefit() computes the new benefit
based on changes in the graph
            new_benefit = calculate_new_benefit(self, road)
            road.update_benefit(new_benefit)

def generate_random_graph(N, p, L_range):
    G = nx.fast_gnp_random_graph(N, p)
    graph = Graph()
    for edge in G.edges():
        weight = random.randint(*L_range)
        graph.add_road(Road(edge[0], edge[1], weight))
    return graph


def main():
    graph = generate_random_graph(N, p, L_RANGE)
    graph.calculate_shortest_paths()

    new_roads = [(0, 2), (0, 3), (1, 2), (1, 4), (2, 3)]
    k = 3  # Budget

    benefit_matrix = compute_benefit_values(graph, new_roads)
    recommended_roads = recommend_new_roads(k, benefit_matrix)

    print("\nThe k roads that need to be constructed:")
    for road in recommended_roads:
        print(f"Start: {road[0]}, End: {road[1]}")

    print("\nThe benefit value for each of the 3 roads recommended for
construction:")
    for road in recommended_roads:
        print(f"Road {road}: Benefit {benefit_matrix[road]}")

    # Update benefits
    graph.update_benefits()

if __name__ == "__main__":
    main()
```

Richard Boysen
2/16/2024

Requirement 4

```
#R4
import networkx as nx
import random

# Define constants
N = 60  # Number of nodes
p = 5   # Connectivity parameter (smaller value for a small network)
L_RANGE = (5, 25)  # Range for road length
T = 100  # Number of trips
k = 3    # Budget parameter
f = 0.8  # Shrinkage factor

class Road:
    def __init__(self, start_node, end_node, weight):
        self.start_node = start_node
        self.end_node = end_node
        self.weight = weight
        self.benefit = None  # Initial benefit is set to None

    def contains(self, nodes):
        return self.start_node in nodes or self.end_node in nodes

    def update_benefit(self, new_benefit):
        self.benefit = new_benefit

class Graph:
    def __init__(self):
        self.roads = []

    def add_road(self, road):
        self.roads.append(road)

    def calculate_shortest_paths(self):
        G = nx.Graph()
        for road in self.roads:
            G.add_edge(road.start_node, road.end_node, weight=road.weight)
        self.shortest_paths = dict(nx.all_pairs_dijkstra_path_length(G))
```

Richard Boysen
2/16/2024

```
    def update_benefits(self):
        for road in self.roads:
            # Assuming calculate_new_benefit() computes the new benefit
based on changes in the graph
            new_benefit = calculate_new_benefit(self, road)
            road.update_benefit(new_benefit)

def generate_random_graph(N, p, L_range):
    G = nx.fast_gnp_random_graph(N, p)
    graph = Graph()
    for edge in G.edges():
        weight = random.randint(*L_range)
        graph.add_road(Road(edge[0], edge[1], weight))
    return graph

def compute_benefit_values(graph, new_roads):
    # Compute the benefit values for the specified new roads
    benefit_matrix = {}
    for road in new_roads:
        start, end = road
        weight = graph.shortest_paths[start][end]
        if weight is not None:
            benefit_matrix[road] = -weight  # Negate the weight as higher
weight implies lower benefit
        else:
            benefit_matrix[road] = "Not computed"
    return benefit_matrix

def recommend_new_roads(k, benefit_matrix):
    # Recommend the top k roads based on the benefit values
    sorted_benefits = sorted(benefit_matrix.items(), key=lambda x: x[1],
reverse=True)[:k]
    recommended_roads = [road for road, _ in sorted_benefits]
    return recommended_roads

def calculate_new_benefit(graph, road):
    # Calculate the new benefit for the road
    # Here, let's assume the new benefit is inversely proportional to the
weight of the road
    weight = graph.shortest_paths[road.start_node][road.end_node]
```

Richard Boysen
2/16/2024

```
    if weight is not None:
        new_benefit = 1 / (1 + weight)  # Inverse relationship: as weight
increases, benefit decreases
    else:
        new_benefit = 0  # If there's no path, benefit is set to 0
    return new_benefit

def main():
    graph = generate_random_graph(N, p, L_RANGE)
    graph.calculate_shortest_paths()

    new_roads = [(0, 2), (0, 3), (1, 2), (1, 4), (2, 3)]
    k = 3  # Budget

    benefit_matrix = compute_benefit_values(graph, new_roads)
    recommended_roads = recommend_new_roads(k, benefit_matrix)

    print("\nThe k roads that need to be constructed:")
    for road in recommended_roads:
        print(f"Start: {road[0]}, End: {road[1]}")

    print("\nThe benefit value for each of the 3 roads recommended for
construction:")
    for road in recommended_roads:
        print(f"Road {road}: Benefit {benefit_matrix[road]}")

    # Update benefits
    graph.update_benefits()

if __name__ == "__main__":
    main()
```

Richard Boysen
2/16/2024

Requirement 5

```
#R5
import networkx as nx

class Road:
    def __init__(self, start_node, end_node, weight):
        self.start_node = start_node
        self.end_node = end_node
        self.weight = weight
        self.benefit = None  # Initial benefit is set to None

    def contains(self, nodes):
        return self.start_node in nodes or self.end_node in nodes

    def update_benefit(self, new_benefit):
        self.benefit = new_benefit

class Graph:
    def __init__(self):
        self.roads = []

    def add_road(self, road):
        self.roads.append(road)

    def calculate_shortest_paths(self):
        G = nx.Graph()
        for road in self.roads:
            G.add_edge(road.start_node, road.end_node, weight=road.weight)
        self.shortest_paths = dict(nx.all_pairs_dijkstra_path_length(G))

    def update_benefits(self):
        for road in self.roads:
            # Assuming calculate_new_benefit() computes the new benefit
based on changes in the graph
            new_benefit = calculate_new_benefit(self, road)
            road.update_benefit(new_benefit)

def compute_benefit_values(graph, new_roads):
    # Compute the benefit values for the specified new roads
```

Richard Boysen
2/16/2024

```
    benefit_matrix = {}
    for road in new_roads:
        start, end = road
        weight = graph.shortest_paths[start][end]
        if weight is not None:
            benefit_matrix[road] = -weight
        else:
            benefit_matrix[road] = "Not computed"
    return benefit_matrix

def recommend_new_roads(k, benefit_matrix):
    # Recommend the top k roads based on the benefit values
    sorted_benefits = sorted(benefit_matrix.items(), key=lambda x: x[1],
reverse=True)[:k]
    recommended_roads = [road for road, _ in sorted_benefits]
    return recommended_roads

def calculate_new_benefit(graph, road):
    # function for computing the new benefit
    # let's assume traffic volume as a proxy for benefit
    traffic_volume = calculate_traffic_volume(graph, road)
    new_benefit = 1 / (1 + traffic_volume)  # Inverse relationship: as
traffic volume increases, benefit decreases
    return new_benefit

def calculate_traffic_volume(graph, road):
    # function for computing traffic volume
    # let's assume traffic volume is based on the shortest path weight
    start = road.start_node
    end = road.end_node
    shortest_path_weight = graph.shortest_paths[start][end]
    if shortest_path_weight is not None:
        # Assuming traffic volume is inversely proportional to shortest
path weight
        traffic_volume = 1 / (1 + shortest_path_weight)
    else:
        # If there's no shortest path, assume high traffic volume
        traffic_volume = float('inf')
    return traffic_volume
```

Richard Boysen
2/16/2024

```
def main():
    graph = Graph()
    # Construct the network
    graph.add_road(Road(1, 3, 11))
    graph.add_road(Road(3, 4, 7))
    graph.add_road(Road(4, 2, 10))
    graph.add_road(Road(4, 0, 9))
    graph.add_road(Road(0, 1, 6))

    graph.calculate_shortest_paths()

    new_roads = [(0, 2), (0, 3), (1, 2), (1, 4), (2, 3)]
    k = 3  # Budget

    benefit_matrix = compute_benefit_values(graph, new_roads)
    recommended_roads = recommend_new_roads(k, benefit_matrix)

    print("Benefit values for constructing new roads:")
    for road, benefit in benefit_matrix.items():
        if benefit == "Not computed":
            print(f"Road {road}: Benefit Not computed")
        else:
            print(f"Road {road}: Benefit {benefit}")

    print("\nRecommended roads for construction:")
    for recommended_road in recommended_roads:
        print(recommended_road)

    # Update benefits
    graph.update_benefits()

if __name__ == "__main__":
    main()
```

Richard Boysen
2/16/2024