DTSC 5501-001

Group Project 3

On

Finding the fastest route between two points on a map

By:

Group Name: “Group18”

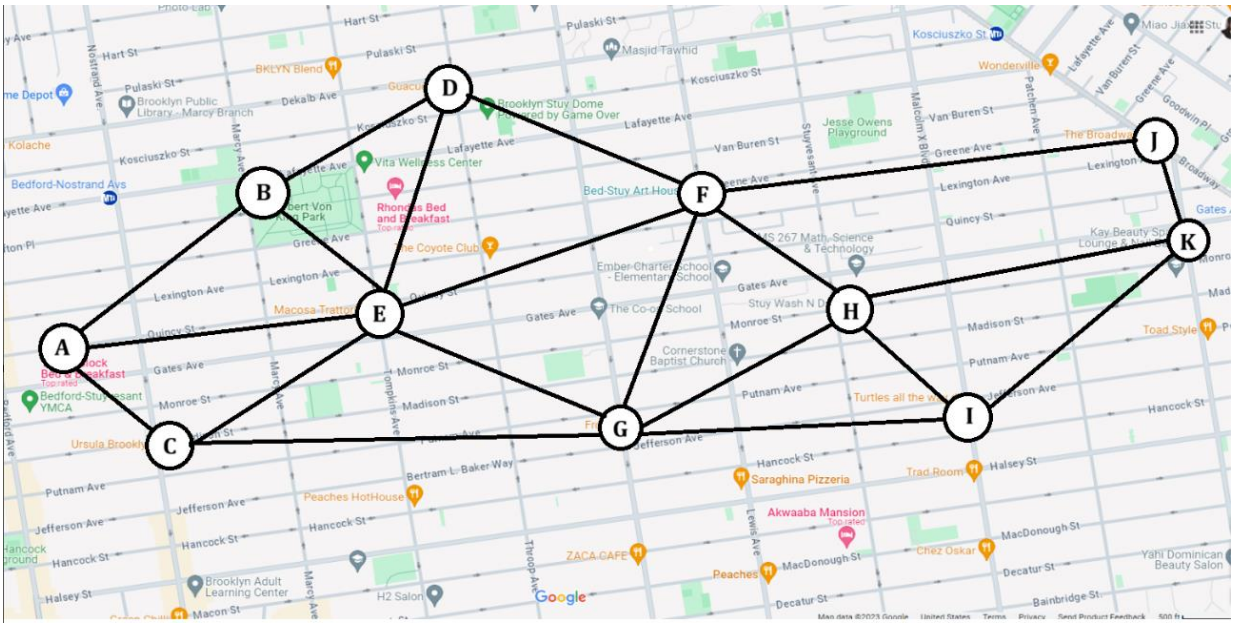
Arihant Sikarwar – arsi3239

Lohith Ramesh – lora4841

Shreyash Sahare – shsa7246

Real-life example taken into consideration:

In this project, we have chosen specific locations from **New York City** as our nodes. Starting from 'A' as the starting node and 'K' as the final node, there are a total of **9** internal nodes interconnected with each other to make a graph as seen below:



Approach:

- **Dijkstra's Algorithm:**

This is a popular algorithm for finding the shortest path in a weighted graph. Below is the break down for each part of the code for the function `dijkstra()`:

1. *Initialization:*
   i) Initialize dictionaries `distances` and `paths` to keep track of the shortest distances and paths from the start node to each node in the graph.
   ii) Set all initial distances to infinity except for the start node, which is set to 0.
   iii) Initialize an empty set `visited` to keep track of visited nodes.
   iv) Create a priority queue `priority_queue` with the initial distance and the start node.

2. *Main loop:*

While the priority queue is not empty, repeat the following steps:

i) Pop the node with the minimum distance from the priority queue. This is done using `heapq.heappop()`, which automatically maintains the min-heap property.
ii) If the node is already visited, skip to the next iteration.
iii) Mark the current node as visited.
iv) For each neighboring node of the current node, calculate the tentative distance from the start node to the neighbor through the current node.
v) The calculated distance is shorter than the current known distance to the neighbor, update the distance and the path.
vi) Push the neighbor into the priority queue with the updated distance.

3. *Result:*
   i) Once the algorithm is complete, the `distances` dictionary contains the shortest distances from the start node to all other nodes in the graph.
   ii) The `paths` dictionary contains the shortest paths from the start node to all other nodes.
   iii) The function returns a tuple containing the shortest distance and the corresponding path from the start node to the specified end node.

- **`uncertainty_simulation` Function:**

This function simulates uncertainty in a graph by introducing roadblocks or disruptions in certain paths. The uncertainty is introduced randomly based on a specified probability.

1. Inputs:

i) `graph`: The original graph represented as an adjacency list.
ii) `uncertainty_probability`: The probability of introducing uncertainty for each path (default is set to 0.2 or 20%).

2. Initialization:

Create a deep copy of the original graph to avoid modifying the original graph unintentionally.

3. Decide Whether to Introduce Uncertainty:

i) Generate a random number between 0 and 1 using `random.random()`.
ii) Check if the generated random number is less than the specified `uncertainty_probability`. If true, uncertainty will be introduced; otherwise, no uncertainty will be introduced.

4. Introduce Uncertainty:

If uncertainty is to be introduced:

i) Select a random subset of nodes (`nodes_with_uncertainty`) from the original graph. The number of nodes is determined by the `random.sample` function, which samples without replacement.
ii) For each selected node, identify all paths leading to that node.
iii) Choose one random path (`chosen_path`) among the paths leading to the current node.
iv) Introduce uncertainty by modifying the weights of edges in the chosen path. In this case, the weights are increased by a factor of 10, simulating a high impact or a roadblock.

5. Return Modified Graph:

Return the modified graph with introduced uncertainty.

- **`simulate_scenario` Function:**

This function simulates different scenarios by modifying the input graph based on a specified scenario name. The two scenarios implemented in this function are the 'Base' scenario and the 'RushHour' scenario.

1. Inputs:

i) `graph`: The original graph represented as an adjacency list.
ii) `scenario_name`: The name of the scenario to simulate (default is 'Base').

2. Base Scenario:

If the `scenario_name` is 'Base', the function does not introduce any modifications to the graph. Instead, it uses the average times from mapping services like Google Maps or Apple Maps. This implies that the graph remains unchanged in terms of travel times.

3. Rush Hour Scenario:

If the `scenario_name` is 'RushHour', we apply a traffic factor by multiplying the weights of edges with a random factor between 1 and 3. This simulates high traffic during rush hour, increasing the time taken to travel between nodes.

4. Uncertainty Simulation:

After applying the scenario-specific modifications, the function calls the `uncertainty_simulation` function to further introduce uncertainty in the graph. The `uncertainty_simulation` function was previously explained to introduce disruptions or roadblocks in certain paths.

5. Return Modified Graph:

Return the modified graph after applying the scenario-specific modifications and introducing uncertainty.

Data Structures used and Algorithm implemented:

- ## Data Structures Used:

1. *Graph Representation:*

The graph is represented as an adjacency list, which is a collection of dictionaries. Each node in the graph is a key in the dictionary, and the corresponding value is another dictionary representing the neighbors and their edge weights.

2. *Dijkstra's Algorithm Data Structures:*

(i) distances: A dictionary to store the shortest known distance from the start node to each node in the graph.

(ii) paths: A dictionary to store the paths from the start node to each node.

(iii) visited: A set to keep track of visited nodes.

(iv) priority_queue: A min-heap implemented using heapq to keep track of nodes with minimum distances. It is a list of tuples where each tuple contains the distance and the node.

- ## Algorithm implemented:

Dijkstra's method plays a crucial role in the transportation network simulation by helping to identify the best paths between nodes in different ways. Applying the method to a graph that depicts the network gives each edge a weight based on how long it takes to traverse between two nodes. The main goal is to determine the fastest route and time between a chosen start node and a specified finish node.

The graph is first built in the base case scenario, using edge weights that correspond to average journey times as seen in mapping services like Apple Maps and Google Maps. Without any extra traffic jams or uncertainties, these weights reflect the initial situation. By executing Dijkstra's algorithm on this unaltered graph, the simulation captures the optimal routes and corresponding travel times, providing a reference point for comparison.

Conversely, the rush hour scenario adds a dynamic component to the graph by including a traffic factor. Before Dijkstra's algorithm is executed, a random factor with a range of 1 to 3 is multiplied by the edge weights. This change replicates the longer travel times that occur during rush hour as a result of more traffic. As a result, the algorithm is then applied to this modified graph in order to examine how traffic fluctuations affect the best route and trip time.

Testing the algorithm with manual calculation:

The optimal path with the shortest travel time from point A to point K, derived by applying Dijkstra's algorithm to the given graph in the base case without uncertainty, is ['A', 'E', 'F', 'J', 'K'] with a time value of 12.2. To validate the algorithm's effectiveness, manual exploration of alternative paths was conducted, aiming to identify potential routes that might outperform the algorithmically generated path.

For instance, a manually considered alternative path, ['A', 'E', 'F', 'H', 'K'], was evaluated, revealing a total travel time of 13.1. Despite initial speculation, this path proved to be less efficient than the path identified by Dijkstra's algorithm. Subsequent consideration of another alternative path, [‘A’, ‘C’, ‘G’, ‘I’, ‘K’], similarly resulted in a higher travel time of 14.2.

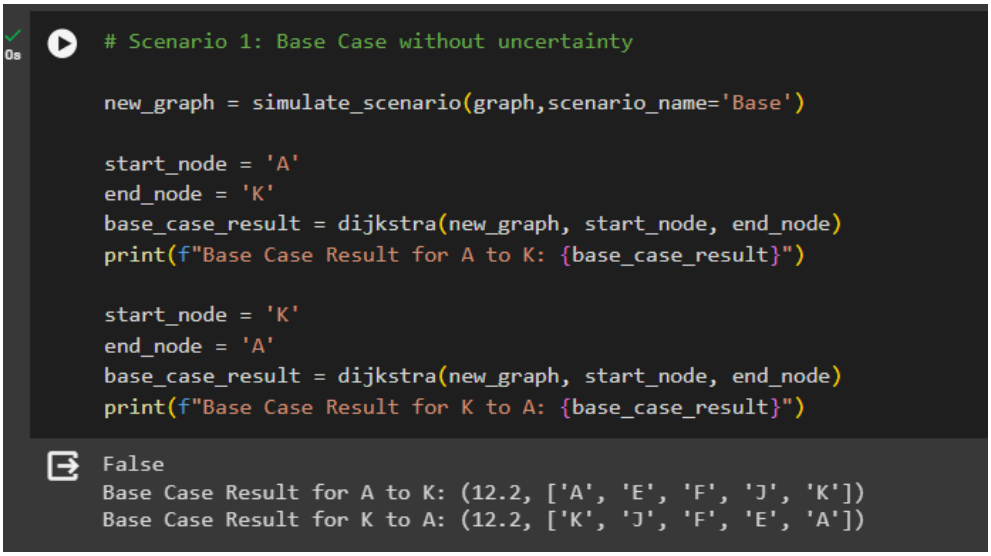
Through a thorough examination of multiple manually proposed paths, it became evident that none of these alternatives surpassed the efficiency of the path generated by Dijkstra's algorithm. The algorithm consistently provided the most optimal route in terms of minimizing travel time, confirming its accuracy and reliability for route optimization in the given transportation network.

Results for each scenario:

1. Base case scenario:

(i) *Without uncertainty:*

In this case, it is observed that there do not exist any nodes in the network that have any uncertainty of a roadblock or any other disruption and the uncertainty is shown as **False**.



```
# Scenario 1: Base Case without uncertainty

new_graph = simulate_scenario(graph,scenario_name='Base')

start_node = 'A'
end_node = 'K'
base_case_result = dijkstra(new_graph, start_node, end_node)
print(f"Base Case Result for A to K: {base_case_result}")

start_node = 'K'
end_node = 'A'
base_case_result = dijkstra(new_graph, start_node, end_node)
print(f"Base Case Result for K to A: {base_case_result}")
```

```
False
Base Case Result for A to K: (12.2, ['A', 'E', 'F', 'J', 'K'])
Base Case Result for K to A: (12.2, ['K', 'J', 'F', 'E', 'A'])
```

(ii) *With uncertainty:*

In this case, it is observed that there do exist nodes in the network that have the uncertainty of a roadblock, or any other disruption and the uncertainty is shown as **True**, along with the nodes that are having the uncertainty.

```
# Scenario 2: Base Case with uncertainty

new_graph = simulate_scenario(graph,scenario_name='Base')

start_node = 'A'
end_node = 'K'
base_case_result = dijkstra(new_graph, start_node, end_node)
print(f"Base Case Result for A to K: {base_case_result}")

start_node = 'K'
end_node = 'A'
base_case_result = dijkstra(new_graph, start_node, end_node)
print(f"Base Case Result for K to A: {base_case_result}")
```

```
True
The nodes randomly selected to incorporate uncertainty are as follows: ['G', 'A', 'J', 'C', 'E', 'I']
Base Case Result for A to K: (41.2, ['A', 'B', 'E', 'F', 'H', 'K'])
Base Case Result for K to A: (41.2, ['K', 'H', 'F', 'E', 'B', 'A'])
```

2. Rush hour scenario:

(i) *Without uncertainty:*

In this case, it is observed that there do not exist any nodes in the network that have any uncertainty of a roadblock, or any other disruption and the uncertainty is shown as **False** but there is heavy traffic present in all of the routes.

```
# Scenario 3: Rush Hour without uncertainty

new_graph = simulate_scenario(graph,scenario_name='RushHour')

start_node = 'A'
end_node = 'K'
base_case_result = dijkstra(new_graph, start_node, end_node)
print(f"Rush Hour Result for A to K: {base_case_result}")

start_node = 'K'
end_node = 'A'
base_case_result = dijkstra(new_graph, start_node, end_node)
print(f"Rush Hour Result for K to A: {base_case_result}")
```

```
False
Rush Hour Result for A to K: (20.827051300847078, ['A', 'E', 'F', 'J', 'K'])
Rush Hour Result for K to A: (20.827051300847078, ['K', 'J', 'F', 'E', 'A'])
```

(ii) *With uncertainty:*

In this case, it is observed that there do exist nodes in the network that have the uncertainty of a roadblock, or any other disruption and the uncertainty is shown as **True**, along with the nodes that are having the uncertainty and heavy traffic.

```
# Scenario 4: Rush Hour with uncertainty

new_graph = simulate_scenario(graph,scenario_name='RushHour')

start_node = 'A'
end_node = 'K'
base_case_result = dijkstra(new_graph, start_node, end_node)
print(f"Rush Hour Result for A to K: {base_case_result}")

start_node = 'K'
end_node = 'A'
base_case_result = dijkstra(new_graph, start_node, end_node)
print(f"Rush Hour Result for K to A: {base_case_result}")
```

```
True
The nodes randomly selected to incorporate uncertainty are as follows: ['F', 'K', 'J', 'I', 'G', 'B']
Rush Hour Result for A to K: (86.91599731480285, ['A', 'E', 'F', 'H', 'I', 'K'])
Rush Hour Result for K to A: (86.91599731480287, ['K', 'I', 'H', 'F', 'E', 'A'])
```

Analysis of the results:

- As observed from the results, we can infer that the time taken to traverse the best route for the base case scenario is lesser than the time taken to traverse the same route but during rush hour since there is a factor of traffic involved at that time.
- We can also observe the fact that the time taken when there is no uncertainty is significantly lesser than the time taken when there are uncertainties (roadblocks) in the best path traversed in both the cases – base case and rush hour case. From this, we can infer that in case of a roadblock, the person has to take a longer route than expected and hence, a greater number of nodes (places on the map) are covered leading to a significant rise in the amount of time taken to reach the destination.
- The observation that the amount of time taken to traverse the path from point A to point K and back from point K to point A is the same signifies that the transportation network is symmetric in terms of travel times. In other words, the travel time between any two nodes in the network is the same regardless of the direction of travel.

  This symmetry in travel times is a characteristic often found in real-world road networks where the time it takes to travel between two locations remains consistent regardless of the starting and ending points. It implies that the road conditions, traffic patterns, and other factors influencing travel time are uniform in both directions along the edges of the graph.

  For practical purposes, symmetric travel times simplify route planning and analysis since the time required to reach a destination and return is equivalent. This observation is crucial in scenarios where round-trip considerations, such as commuting or delivery routes, are essential, as it implies that the travel time for a complete journey can be determined by examining only one direction.