MSDS 689 Final Project: Shortest Path Analysis on Real-World Transportation Networks

Ricky Miura, John Green, Eli Mecinas Cruz

In [2]:
```
import osmnx as ox
import networkx as nx
import matplotlib.pyplot as plt
import pickle
import numpy as np
import heapq
from scipy.sparse import csr_matrix
from geopy.geocoders import Nominatim
```

Problem Statement/Graph Initialization

In this notebook, we'll walk through the background and construction of this project while doing some sample path-mapping in San Franciso. In addition to this static analysis, we also built an interactive web app which allows you to visualize the shortest path between two addresses in SF. The app can be found here and the code can be found in this repo.

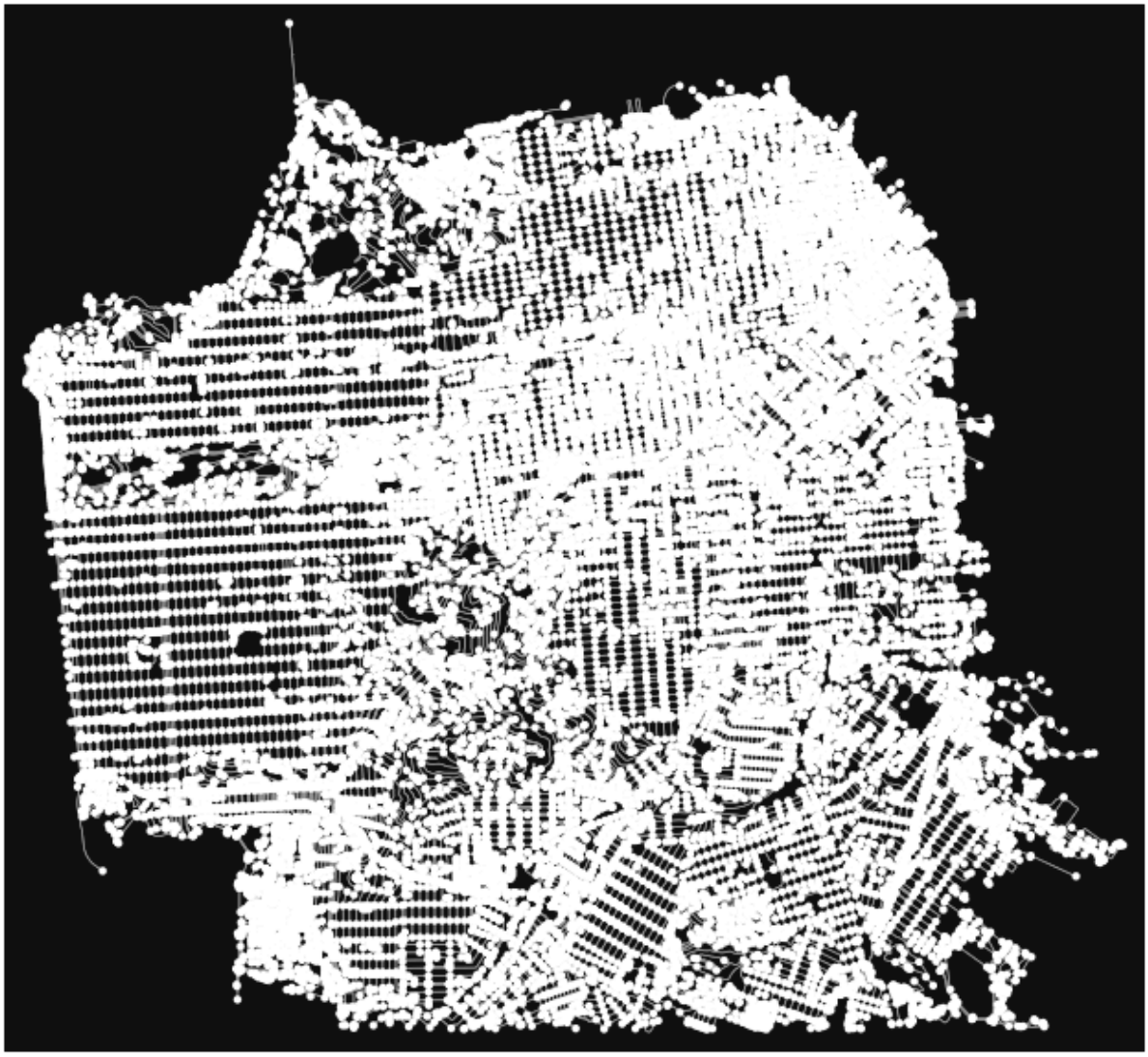
Let's start by defining our area of focus: San Francisco, a city well-known for its notorious traffic. Many of us have experienced the frustration of sitting through endless traffic jams. To improve our commutes, we'll use Dijkstra's algorithm to find the shortest paths right here in San Francisco.

In [3]:
```
place_name = "San Francisco, California, USA"
G = ox.graph_from_place(place_name, network_type='walk')

# Ensure edges have distance as a weight
for u, v, data in G.edges(data=True):
    data['weight'] = data.get('length', 1)  # Default to 1 if no length
```

We initialize our San Francisco transportation network by using OSMnx, a Python package which allows us to model and analyze street networks from OpenStreetMap. We also add weights to the edges on our graph to represent the distance between nodes.

Here's a look at our map of San Francisco:

In [4]:
```
fig, ax = ox.plot_graph(G, node_size=10, edge_linewidth=0.5)
```



In [5]:
```
print(f"Number of nodes: {len(G.nodes)}")
print(f"Number of edges: {len(G.edges)}")
```

```
Number of nodes: 48638
Number of edges: 153774
```

We define our own function which turns our graph into a sparse adjacency matrix. This matrix will be necessary when computing the shortest paths later on.

In [7]:
```
# Function to convert NetworkX graph to a sparse adjacency matrix
def graph_to_sparse_matrix(G):
    # Map nodes to indices
    node_list = list(G.nodes)
    node_to_idx = {node: idx for idx, node in enumerate(node_list)}

    # Create row, col, and weight arrays
    rows, cols, weights = [], [], []
    for u, v, data in G.edges(data=True):
        rows.append(node_to_idx[u])
        cols.append(node_to_idx[v])
        weights.append(data['weight'])
```

```
    # Build the sparse matrix
    adjacency_matrix = csr_matrix((weights, (rows, cols)), shape=(len(node_l

    return adjacency_matrix, node_list
```

In [8]:
```
adjacency_matrix, node_list = graph_to_sparse_matrix(G)
```

Shortest Path Between Downtown San Francisco and Mission District

For this example, we are interested in the shortest path from Downtown San Francisco to the Mission District. By defining the coordinates of our start and end destination, we can use OSMnx's built-in methods to find the nearest node in our network.

In [6]:
```
# Coordinates for start and end points
start_coords = (37.7749, -122.4194)  # Example: SF downtown
end_coords = (37.7599, -122.4148)    # Example: Mission District

# Find the nearest nodes in the graph
start_node = ox.nearest_nodes(G, X=start_coords[1], Y=start_coords[0])
end_node = ox.nearest_nodes(G, X=end_coords[1], Y=end_coords[0])

print(f"Start node: {start_node}, End node: {end_node}")
```

```
Start node: 6338758297, End node: 5443111169
```

To find the shortest path in our network, we will create our own implementation of Dijkstra's algorithm.

In [ ]:
```
def dijkstra(matrix, start_idx, end_idx):
    """
    Simplified Dijkstra's algorithm to find the shortest path from the start

    Args:
    - matrix (np.array): Adjacency matrix (weight matrix) of the graph.
    - start_idx (int): The index of the starting node.

    Returns:
    - dist (list): List of shortest distances from the start node to all oth
    - path (list): List of predecessors for each node on the shortest path.
    """
    # Convert sparse matrix to dense if needed
    if isinstance(matrix, csr_matrix):
        matrix = matrix.toarray()

    n = matrix.shape[0]
    dist = [float('inf')] * n  # Initialize distances to infinity
    dist[start_idx] = 0  # Distance to start node is zero
    predecessors = [None] * n  # Predecessor list to reconstruct the path

    # Priority queue to select the node with the smallest distance
    pq = [(0, start_idx)]  # (distance, node)
```

```
    while pq:
        current_dist, current_node = heapq.heappop(pq)

        # If the current distance is already greater than the recorded one,
        if current_dist > dist[current_node]:
            continue

        # Early stopping: if we reach the target node, exit the loop
        if current_node == end_idx:
            break

        # Explore neighbors
        for neighbor_idx in range(n):
            # Check if there's an edge (no edge if weight is 0 or infinity)
            weight = matrix[current_node][neighbor_idx]
            if weight == 0 or weight == float('inf'):
                continue

            new_dist = current_dist + weight
            if new_dist < dist[neighbor_idx]:  # If a shorter path is found
                dist[neighbor_idx] = new_dist
                predecessors[neighbor_idx] = current_node
                heapq.heappush(pq, (new_dist, neighbor_idx))

        # Reconstruct the path from start_idx to end_idx
    path = []
    current = end_idx
    while current is not None:
        path.append(current)
        current = predecessors[current]
    path.reverse()

    # Return the shortest distance and the path
    return dist[end_idx], path
```

The steps are as follows:

1. Initialization

- Create a `dist` list to store the shortest distances from the start node to all other nodes, initialized to infinity (`float('inf')`), except for the start node which is set to `0` .
- Create a `predecessors` list to store the preceding node for each node on the shortest path, initialized to `None` .
- Use a priority queue (min-heap) to process nodes in order of their current shortest distance, starting with `(0, start_node)` .

2. Process Nodes

- While the priority queue is not empty:

1. Remove the node with the smallest distance (`current_node`) from the priority queue.
2. Skip the node if its current distance is greater than the recorded distance in `dist` (this ensures we don't reprocess outdated paths).
3. If the current node is the target node, stop early as the shortest path has been found.

3. Explore Neighbors

- For each neighbor of the `current_node` :
  1. Check if there is an edge (`matrix[current_node][neighbor] > 0` or not infinity).
  2. Calculate the new distance to the neighbor (`current_distance + weight`).
  3. If the new distance is shorter than the recorded distance in `dist` , update:
     - The distance in `dist` .
     - The predecessor of the neighbor in `predecessors` .
     - Add the neighbor to the priority queue with its updated distance.

4. Reconstruct Path

- To reconstruct the shortest path:
  1. Start from the `end_node` and follow the `predecessors` list backward to the `start_node` .
  2. Append each node to a `path` list.
  3. Reverse the `path` list to get the correct order from start to end.

5. Return Results

- Return the shortest distance from the start node to the target node (`dist[end_idx]`) and the reconstructed path (`path`).

Early Stopping Optimization

- If the algorithm reaches the target node during processing, it can terminate early, saving computation time.

In [10]:
```
# Function to find the shortest path between two nodes
def shortest_path_sparse(matrix, node_list, start_coords, end_coords):
    # Convert coordinates to node indices
    start_node = ox.nearest_nodes(G, X=start_coords[1], Y=start_coords[0])
    end_node = ox.nearest_nodes(G, X=end_coords[1], Y=end_coords[0])

    start_idx = node_list.index(start_node)
    end_idx = node_list.index(end_node)

    # Perform Dijkstra's algorithm
```

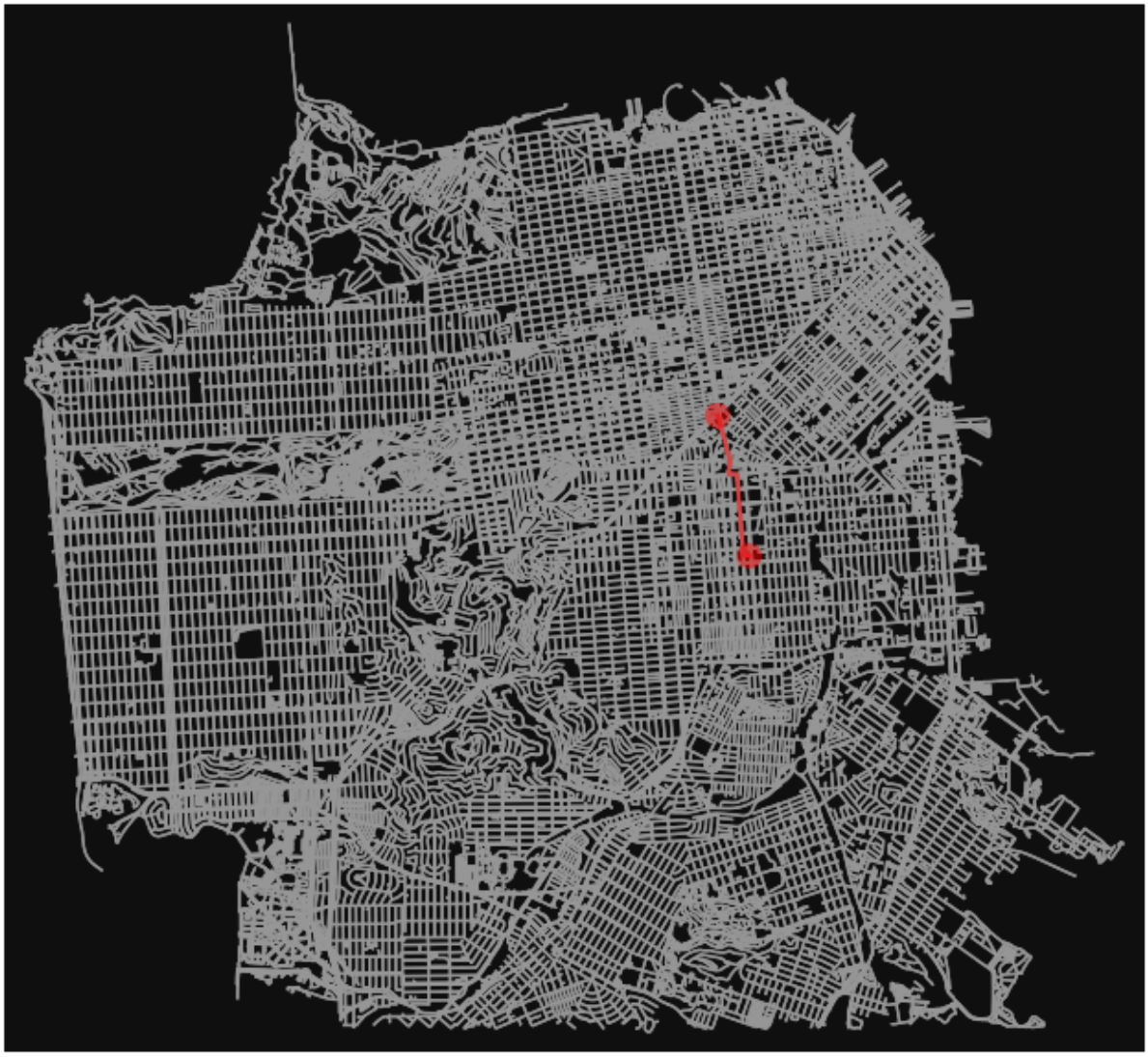
```
    dist_matrix, path = dijkstra(matrix, start_idx, end_idx)

    return [node_list[idx] for idx in path]
```

With this, we can find the shortest path between any two coordinates.

In [11]:
```
# Example coordinates
start_coords = (37.7749, -122.4194)  # San Francisco downtown
end_coords = (37.7599, -122.4148)    # Mission District

path = shortest_path_sparse(adjacency_matrix, node_list, start_coords, end_c
print(f"Shortest path: {path}")
```

```
Shortest path: [6338758297, 8023685477, 9992295989, 11230365828, 65315830, 2
60118044, 11230365832, 3386704351, 260118094, 413748886, 4548596610, 3005149
27, 300514894, 4547291172, 65374289, 300515687, 276546182, 276546183, 817061
4555, 65299205, 6441824074, 6441824070, 6441824063, 12131158485, 1213115848
7, 12131158470, 12131158471, 5438253806, 1677788984, 12127069996, 855419625
1, 65365910, 8554191671, 3188130246, 65345337, 2649313724, 2649313729, 65317
378, 2819486339, 4624714830, 8982516108, 4624714828, 8982550521, 8982550564,
4763953378, 5443111173, 8982550579, 5443111166, 5443111168, 5443111169]
```

In [12]:
```
fig, ax = ox.plot_graph_route(G, route=path, route_linewidth=2, node_size=0)
```



Shortest Path Between Golden Gate Park and Mission Dolores Park

Out of curiousity, we'll take a look at another example and see if we can find the shortest path between Golden Gate Park and Mission Dolores Park.

In [13]:
```
# Initialize geolocator
geolocator = Nominatim(user_agent="shortest-path-app")

# Function to get coordinates from an address
def get_coordinates(address):
    location = geolocator.geocode(address)
    if location:
        return (location.latitude, location.longitude)
    else:
        raise ValueError(f"Address '{address}' could not be geocoded.")

# Example usage
start_address = "Golden Gate Park, San Francisco, CA"
end_address = "Mission Dolores Park, San Francisco, CA"
```

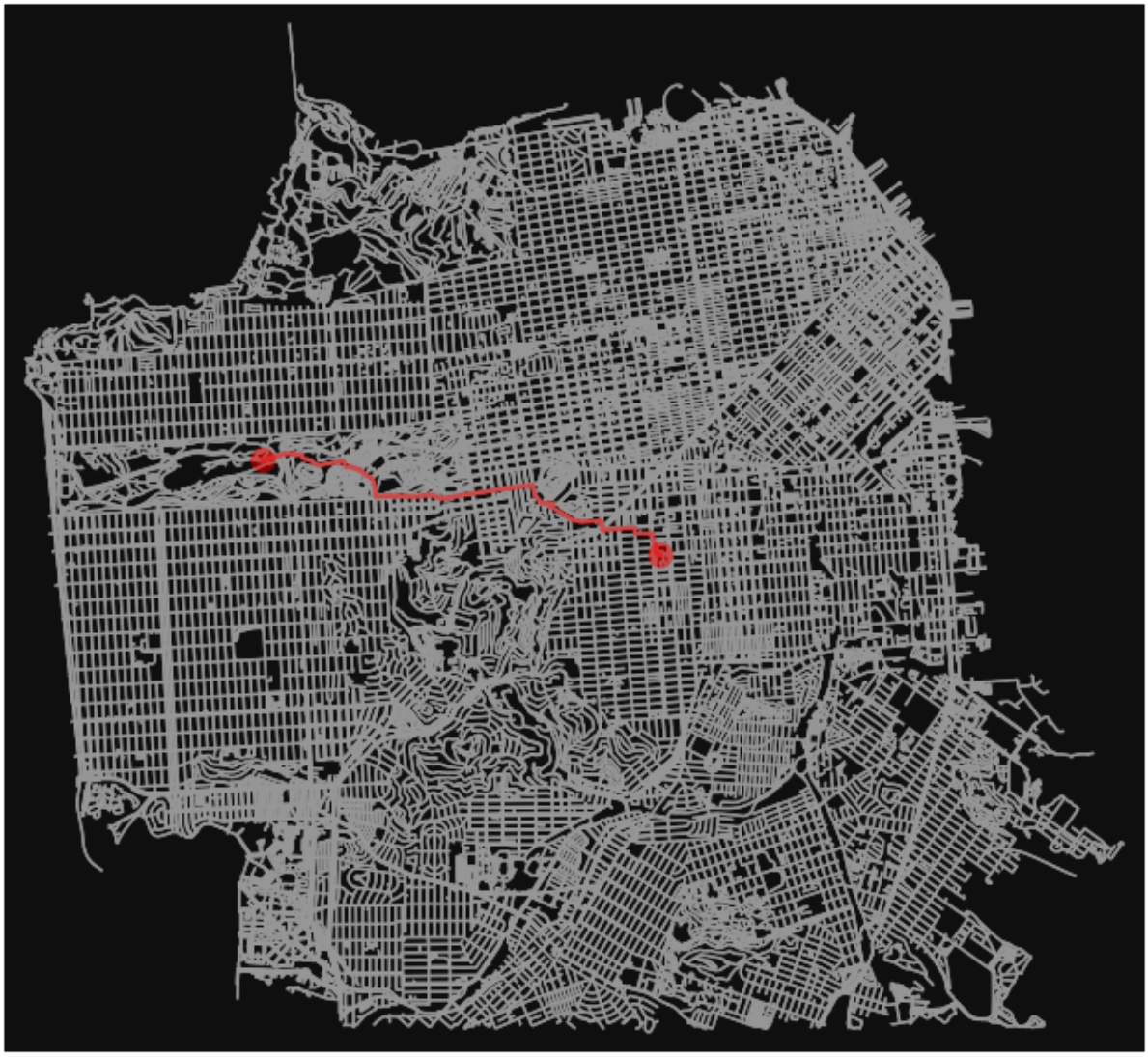
```
start_coords = get_coordinates(start_address)
end_coords = get_coordinates(end_address)

print(f"Start coordinates: {start_coords}")
print(f"End coordinates: {end_coords}")
```

```
Start coordinates: (37.769368099999994, -122.48218371117709)
End coordinates: (37.7597203, -122.4271322952394)
```

In [14]:
```
path = shortest_path_sparse(adjacency_matrix, node_list, start_coords, end_c
print(f"Shortest path: {path}")
```

```
Shortest path: [65289999, 3066652079, 3713966077, 274468610, 11397317496, 49
00937596, 274468611, 4804650403, 2179309516, 65360030, 1350854689, 217930950
5, 2179309511, 2179309491, 696347147, 274470157, 665618178, 4804647795, 6656
18173, 8878461496, 7838689758, 3066641472, 3066641471, 4417157200, 441715688
6, 65368552, 4417156884, 343531215, 2179216800, 1211324459, 343525997, 83951
59589, 2179216742, 2179292625, 12346842000, 12346841982, 4417156820, 4417156
821, 65304270, 65302755, 65287605, 9595885997, 4034027164, 9595885999, 95958
86000, 9595886006, 9595886007, 9595886009, 1511324892, 295505887, 258912607,
1511324967, 1511339725, 1511339712, 1511339715, 1511339732, 7148948992, 6529
8380, 7148948989, 8263295655, 8263295659, 8263295652, 8263295645, 65298376,
8263295640, 8358989416, 65298373, 8358989412, 4828619090, 65298370, 48286190
91, 65298367, 4828619094, 65298359, 4828619095, 8395224513, 8276472272, 8276
472264, 4017343144, 8369872030, 8276472252, 8276472253, 4828619097, 82764722
38, 8276472237, 4018688214, 8276472234, 8221193911, 65376340, 65329677, 8221
198718, 8322587349, 65292412, 2329540044, 65303314, 2329540039, 5271301478,
6265008478, 6728777355, 8365948788, 65298023, 319357702, 5438557807, 8433575
298, 9856643379, 9856643372, 9856643377, 1996007292, 65296324, 260494513, 62
50503246, 1723587834, 8248612213, 1723580562, 6250503262, 6250503267, 625050
3266, 6355595075, 6355595334, 6355595076, 6355595080, 6355595333, 65307883,
5443111255, 1563088443, 65307880, 1563088440, 6336997285, 310317986, 1848703
910, 310317974, 5352317712, 850751601, 9756178581, 9756178572, 9756178576, 9
756340483, 9756340489]
```

In [15]:
```
fig, ax = ox.plot_graph_route(G, route=path, route_linewidth=2, node_size=0)
```



Future Use Cases

There are many potential use cases of Dijkstra's algorithm in logistics and transportation planning.

1. Delivery Services

- Delivery companies like UPS, FedEx, or Amazon need to plan efficient routes for their drivers.
- Use Dijkstra's algorithm to calculate the shortest path between warehouses and delivery destinations.
- Factor in real-time traffic conditions, distances, or toll costs as edge weights.

2. Navigation Systems

- Real-time navigation apps like Google Maps or Waze guide drivers to their destinations.

- Calculate the fastest routes based on current traffic data or road closures.
- Dynamically update routes when conditions change (e.g., accidents or congestion).

3. **Public Transportation Network Design**

- Urban planners design or improve public transportation systems (e.g., buses, trains).
- Identify optimal bus or train routes between key locations.
- Plan transfer points and schedules for minimal travel time.

4. **Emergency Response Routing**

- Emergency services (e.g., ambulances, fire trucks) need the fastest route to a destination.
- Use Dijkstra's algorithm to find the quickest path to emergencies based on road networks and current conditions.
- Factor in additional weights like one-way streets or traffic signal delays.

5. **Traffic Management**

- City planners aim to manage traffic flow and reduce congestion in urban areas.
- Use Dijkstra's algorithm to recommend alternate routes during peak hours.
- Incorporate IoT devices for real-time traffic monitoring and path optimization.