```
z'''
Optimizing Delivery Routes for A Logistics Company

Problem :
 Say you want to create a way to find the shortest routes to deliver raw materials
and  you want to consider the following constraints: Traffic Conditions, Multiple Warehouse services.

Things to note:
- Choose the best traffic conditions
- edges should be weighted and should be represented with its current traffic conditions
- Since we have multiples warehouses services, choose which warehouse can provide the simplest route to deliver the items with regards to th

Approach:
1.) Create a Graph to represent the networks between the warehouse(depot) and the delivery
2.) Represent the warehouse as source and delivery as target destination
3.) Using Graph Theory and DFS calculate the shortest route
'''
class Node(object):
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

class Graph(object):
    def __init__(self):
        self.adj = {}  # Adjacency List

    def addEdge(self, src, dest, w):  # w represents the weight of the edge in terms of traffic conditions
        if src not in self.adj:
            self.adj[src] = []
        self.adj[src].append((dest, w))

    def findNode(self, node):
        for exsNode in self.adj.keys():
            if exsNode == node:
                return node.getName()
        raise ValueError("Node Does Not Exist!")

    def addNode(self, node):
        if node in self.adj:
            raise ValueError("Node Already Exist!")
        self.adj[node] = []

    def shortestPath(self, src, dest):
        # Initialize distances and previous nodes
        distances = {node: float('inf') for node in self.adj}
        distances[src] = 0
        previous = {node: None for node in self.adj}

        # Priority queue implemented using a list
        pq = [(0, src)]

        while pq:
            # Pop node with the smallest distance from the priority queue
            current_distance, current_node = min(pq)
            pq.remove((current_distance, current_node))

            # Check if the current node is the destination
            if current_node == dest:
                break

            # Explore neighbors of the current node
            for neighbor, weight in self.adj[current_node]:
                distance = current_distance + weight
                # Update distance and previous node if a shorter path is found
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    previous[neighbor] = current_node
                    pq.append((distance, neighbor))

        # Reconstruct the shortest path from src to dest
        shortest_path = []
        current = dest
        while current is not None:
            shortest_path.insert(0, current.getName())  # Append node name instead of node object
            current = previous[current]
```

```
        return shortest_path, distances[dest]

    def display(self):
        for node, neighbors in self.adj.items():
            print(f"{node.getName()} ---->")
            for neighbor, weight in neighbors:
                print(f"    {neighbor.getName()} (Weight: {weight})")


# Create Graph Instance
graph = Graph()

# Create Nodes
warehouse1 = Node("Warehouse1")
warehouse2 = Node("Warehouse2")
store1 = Node("Store1")
store2 = Node("Store2")
store3 = Node("Store3")
store4 = Node("Store4")
store5 = Node("Store5")

# Add nodes to the Graph
graph.addNode(warehouse1)
graph.addNode(warehouse2)
graph.addNode(store1)
graph.addNode(store2)
graph.addNode(store3)
graph.addNode(store4)
graph.addNode(store5)

# Add edges representing routes between warehouses and stores
graph.addEdge(warehouse1, store1, 10)
graph.addEdge(warehouse1, store2, 15)
graph.addEdge(warehouse1, store3, 12)
graph.addEdge(warehouse2, store3, 8)
graph.addEdge(warehouse2, store4, 20)
graph.addEdge(warehouse2, store5, 18)

# Add edges between stores
graph.addEdge(store1, store2, 5)
graph.addEdge(store2, store3, 7)
graph.addEdge(store3, store4, 6)
graph.addEdge(store4, store5, 4)

# Display the graph
graph.display()

# Example usage of shortestPath method
shortest_path_warehouse1_to_store3, shortest_distance_warehouse1_to_store3 = graph.shortestPath(warehouse1, store3)
shortest_path_warehouse2_to_store5, shortest_distance_warehouse2_to_store5 = graph.shortestPath(warehouse2, store5)

print("\nShortest Path from Warehouse1 to Store3:", shortest_path_warehouse1_to_store3)
print("Shortest Distance from Warehouse1 to Store3:", shortest_distance_warehouse1_to_store3)

print("\nShortest Path from Warehouse2 to Store5:", shortest_path_warehouse2_to_store5)
print("Shortest Distance from Warehouse2 to Store5:", shortest_distance_warehouse2_to_store5)

shortest_path_warehouse1_to_store5, shortest_distance_warehouse1_to_store5 = graph.shortestPath(warehouse1, store5)

print("\nShortest Path from Warehouse1 to Store5:", shortest_path_warehouse1_to_store5)
print("Shortest Distance from Warehouse1 to Store5:", shortest_distance_warehouse1_to_store5)
```

```
Warehouse1 ---->
    Store1 (Weight: 10)
    Store2 (Weight: 15)
    Store3 (Weight: 12)
Warehouse2 ---->
    Store3 (Weight: 8)
    Store4 (Weight: 20)
    Store5 (Weight: 18)
Store1 ---->
    Store2 (Weight: 5)
```

```
Store2 ---->
    Store3 (Weight: 7)
Store3 ---->
    Store4 (Weight: 6)
Store4 ---->
    Store5 (Weight: 4)
Store5 ---->

Shortest Path from Warehouse1 to Store3: ['Warehouse1', 'Store3']
Shortest Distance from Warehouse1 to Store3: 12

Shortest Path from Warehouse2 to Store5: ['Warehouse2', 'Store5']
Shortest Distance from Warehouse2 to Store5: 18

Shortest Path from Warehouse1 to Store5: ['Warehouse1', 'Store3', 'Store4', 'Store5']
Shortest Distance from Warehouse1 to Store5: 22
```

```
class Node(object):
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

class Graph(object):
    def __init__(self):
        self.adj = {}  # Adjacency List

    def addEdge(self, src, dest, w):  # w represents the weight of the edge in terms of traffic conditions
        if src not in self.adj:
            self.adj[src] = []
        self.adj[src].append((dest, w))

    def findNode(self, node):
        for exsNode in self.adj.keys():
            if exsNode == node:
                return node.getName()
        raise ValueError("Node Does Not Exist!")

    def addNode(self, node):
        if node in self.adj:
            raise ValueError("Node Already Exist!")
        self.adj[node] = []

    def shortestPath(self, src, dest):
        # Initialize distances and previous nodes
        distances = {node: float('inf') for node in self.adj}
        distances[src] = 0
        previous = {node: None for node in self.adj}

        # Priority queue implemented using a list
        pq = [(0, src)]

        while pq:
            # Pop node with the smallest distance from the priority queue
            current_distance, current_node = min(pq)
            pq.remove((current_distance, current_node))

            # Check if the current node is the destination
            if current_node == dest:
                break

            # Explore neighbors of the current node
            for neighbor, weight in self.adj[current_node]:
                distance = current_distance + weight
                # Update distance and previous node if a shorter path is found
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    previous[neighbor] = current_node
                    pq.append((distance, neighbor))

        # Reconstruct the shortest path from src to dest
        shortest_path = []
        current = dest
        while current is not None:
            shortest_path.insert(0, current.getName())  # Append node name instead of node object
            current = previous[current]
```

```
        return shortest_path, distances[dest]

    def display(self):
        for node, neighbors in self.adj.items():
            print(f"{node.getName()} ---->")
            for neighbor, weight in neighbors:
                print(f"    {neighbor.getName()} (Weight: {weight})")


# Create Graph Instance
graph = Graph()

# Create Nodes
warehouse1 = Node("Warehouse1")
warehouse2 = Node("Warehouse2")
store1 = Node("Store1")
store2 = Node("Store2")
store3 = Node("Store3")
store4 = Node("Store4")
store5 = Node("Store5")

# Add nodes to the Graph
graph.addNode(warehouse1)
graph.addNode(warehouse2)
graph.addNode(store1)
graph.addNode(store2)
graph.addNode(store3)
graph.addNode(store4)
graph.addNode(store5)

# Add edges representing routes between warehouses and stores
graph.addEdge(warehouse1, store1, 10)
graph.addEdge(warehouse1, store2, 15)
graph.addEdge(warehouse1, store3, 12)
graph.addEdge(warehouse2, store3, 8)
graph.addEdge(warehouse2, store4, 20)
graph.addEdge(warehouse2, store5, 18)

# Add edges between stores
graph.addEdge(store1, store2, 5)
graph.addEdge(store2, store3, 7)
graph.addEdge(store3, store4, 6)
```