Best Neighbour Algorithm for Routing Traffic (BNART)

CP468 - Artificial Intelligence

Assignment 1

Dr. Dariush Ebrahimi

October 22nd, 2023

Shane Shamku - 210315940
Javier Rodrigues - 210631740
Sumanbir Kairon - 200598100
John Bannan - 200724500
Adam Camilleri - 200757020

Task 1 (2 points): To design the agent for the BNART problem (or the project problem of your choice), specify the environment type (i.e., specify whether the problem's environment is: fully observable or not, deterministic or stochastic, episodic or sequential, static or dynamic, discrete or continuous, single agent or multiagent), and elaborate on your answers.

The BNART problem can be classified as a multi-agent, sequential, fully observable, deterministic, sequential, and dynamic environment.

- Fully Observable: The Central Traffic Control Unit (CTCU), is responsible for routing vehicles and has access to real-time information on the state of the environment. When a new vehicle enters the road/environment, the CTCU has to be aware of it at all times, in order to predict and update other vehicles on traffic congestion.

- Deterministic: The outcome of the agent's actions is fully determined by the input data. The current position of vehicles, road structure, traffic conditions, and predefined rules and algorithms are all used to calculate optimal routes for the vehicles. There is no inherent randomness in how vehicle movements are simulated.

- Sequential: The outcome of one action affects the possibilities of the next action. The CTCU monitors and tracks the position of all vehicles at all times as well as new vehicles entering the network. The decisions made by the CTCU affect other decisions, when one vehicle is sent on a certain path, other vehicles may be sent on a different path in order to optimize the travel time. The CTCU adapts its routing decisions as new information is available.

- Dynamic: The environment changes over time as new information becomes available. The road network can change as new vehicles enter, traffic conditions change, and vehicle positions change. The CTCU network is continuously changing, making it a dynamic environment.

- Discrete: While the problem involves real-time decision-making and dynamic changes, the state space and action space are discrete in nature. The state space is discrete due to road segments, intersections, and locations of vehicles at specific points, each state is distinct. The action space is also discrete because the CTCU makes decisions to select routes for vehicles, these decisions are made from a finite set of choices.

- Multi-Agent: Each vehicle is considered an agent in the environment, and the CTCU acts as a central agent responsible for optimizing the entire system. The problem involves the collaboration of multiple vehicles (agents).

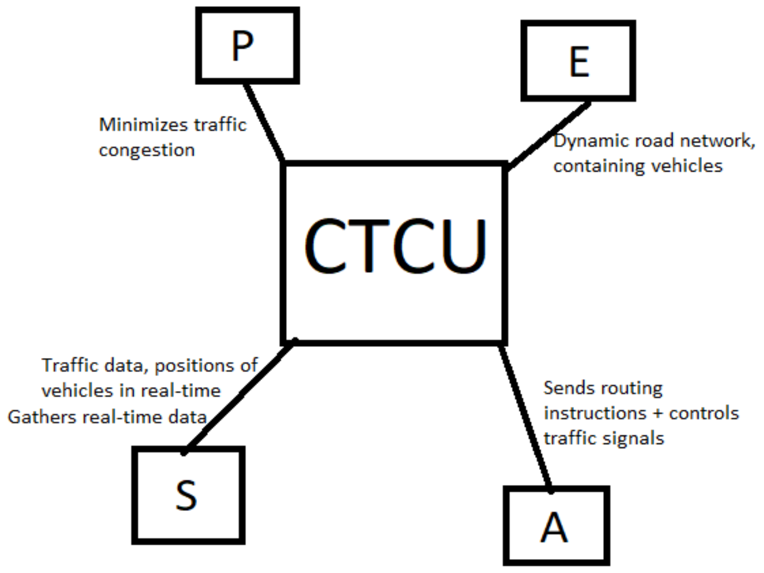
Task 2 (2 points): To design the agent for the BNART problem (or the project problem of your choice), specify the task environment (i.e., specifying PEAS). Then draw a figure showing the connection between the agent and the environment with sufficient explanation.

PEAS

Performance Measure: The performance of the CTCU will be made efficient by minimizing traffic congestion, optimizing travel time, and reducing fuel consumption.

Environment: A road network system containing vehicles. The environment is dynamic, meaning there are new vehicles constantly entering the network, as well as changing traffic conditions.

Actuators: The CTCU interacts with the environment through its actuators. The CTCU sends routing instructions to vehicles to influence the way they navigate the network of roads. In turn, the CTCU defines the optimal routes and communicates these routes to the vehicles, which lets them know which path is the correct one to take. The CTCU interacts with the environment in other ways as well, such as adjusting the timing of the traffic lights at intersections to optimize the flow of traffic, or communicating with individual vehicles and providing them updates or guidance to ensure they follow the optimal routes.


P
E
Minimizes traffic congestion
Dynamic road network, containing vehicles
CTCU
Traffic data, positions of vehicles in real-time
Gathers real-time data
Sends routing instructions + controls traffic signals
S
A

Sensors: Similar to actuators, the CTCU perceives its environment through sensors. This includes data on the positions of vehicles and traffic conditions, in real-time. These sensors support the CTCU in making decisions based on the information it receives.

Drawn Figure:

←(Not final)

>Task 3 (2 points): Model the BNART problem (or the project problem of your choice) as a Search Problem (i.e., define all the required functions for the search problem, such as Initial-State, Actions, Transition-Model, Goal-Test, Path-Cost, etc.).

Initial State: The initial state is the starting configuration of the road network, which includes the positions of all vehicles and the initial traffic conditions.

Actions: Actions are the decisions made by the Central Traffic Control Unit (CTCU) to influence the traffic flow. These actions include:

- Routing instructions to individual vehicles, specifying the path they should take.
- Adjusting the timing of traffic lights at intersections.
- Communicating with vehicles to provide updates and guidance for following optimal routes.
- Managing the flow of new vehicles entering the network.

Transition Model: The transition model defines how the state of the road network changes when the CTCU takes actions. It incorporates the deterministic nature of the environment, where the outcome of actions is fully determined by the input data. For example, if the CTCU sends a routing instruction to a vehicle, the new state reflects the vehicle following that route.

Goal Test: The goal test checks if the road network is in an optimal state. This is defined as:

- Minimizing traffic congestion.
- Optimizing travel time for all vehicles.
- Reducing fuel consumption.
- Achieving a specific target for vehicle throughput and efficiency.

Path Cost: The path cost represents a measure of efficiency or cost associated with each action taken by the CTCU. It is based on factors like time, fuel usage, and overall system efficiency.

Task 4 (4 points): Solve the BNART problem (or the project problem of your choice) using one or more methods learned in the class for optimization problems. For that, you need to implement the method/methods using any programming language of your choice (for example, Python, Jupyter Notebook, etc.). Create a simple input (or inputs) for testing your implementation. P.S.: Note that for your project you will compare the performance of your Reinforcement Learning algorithm with the methods used in this assignment.

The following python program is an implementation of the BNART algorithm in a simplified limited capacity.

Methodology:

In this code, there are 3 cars travelling on a series of randomly generated roads between 20 distinct locations. Each car has a starting point and an ending point set out at the beginning of the program, and their objective is to reach their destination in the most time efficient manor. Every time a car drives along a road, they take one "turn" which can be denoted by 'Time step x' in the output console, where x denotes the turn number. Each route has randomly generated traffic delay values which increase when a car utilizes a route. This means, that when two or more cars are near eachother, routes that would put them closer to another car result in increased traffic time. To account for this, the cars utilize reasoning based on a BNART algorithm to find the 'best neighbour' road to take. This also means that a car can take a route that is turn inefficient (drives along more roads) while still taking the most time efficient route (because of traffic consideration). At each destination (after driving along a road), the three cars run the BNART decision making algorithm to determine their next choice of road. Each route that a car takes (series of roads) is highlighted by a unique colour. We used a network graph with matlio imports to visualize the BNART algorithm in this hypothetical scenario. The inputs for this code are the randomly generated nodes (destinations) and the roads that connect them.

Code:

https://trinket.io/python3/5446d13aa8

```
import math
import random
import matplotlib.pyplot as plt

# Parameters
num_nodes = 20
```

```
P = {chr(ord('A') + i): (random.randint(0, 10), random.randint(0, 10)) for i in range(num_nodes)}
# Set of nodes with (x, y) coordinates
L = [(random.choice(list(P.keys())), random.choice(list(P.keys()))) for _ in range(num_nodes *
2)]  # Set of links
V = ['Vehicle' + str(i) for i in range(1, 4)]  # Set of vehicles

# Ensure that starting and destination nodes for each vehicle are different and connected to other
nodes
Sv = {}
Dv = {}
for vehicle in V:
    while True:
        start_node = random.choice(list(P.keys()))
        destination_nodes = [node for node in list(P.keys()) if node != start_node and (start_node,
node) in L]
        if destination_nodes:
            destination_node = random.choice(destination_nodes)
            Sv[vehicle] = start_node
            Dv[vehicle] = destination_node
            break

B = 10000  # Large constant bigger than end of system time
Tvs = {vehicle: 0 for vehicle in V}  # Time when vehicle v enters the network
TRoad = 1  # Time to traverse any link with maximum road speed
TDelay = 0.1  # Delay added to the system time
INode = list(P.keys())  # Set of intersection nodes
CPercent = 10  # Percentage of added time delay based on the number of vehicles sharing the
same link

# Decision Variables
Rv = {}  # Indicate whether link <i, j> is set on the route of vehicle v
Xv = {}  # System time when vehicle v traversed link <i, j>
TxD = {}  # System time when vehicle v arrives to destination
Nv = {}  # Number of vehicles sharing link <i, j> with vehicle v
Avv = {}  # Indicate vehicle v' is sharing link <i, j> with vehicle v
Alvv = {}  # Indicate if vehicle v' is passing through link <i, j> [To linearize the multi-vehicle
constraint]
Agvv = {}  # Indicate if vehicle v' is arriving to link <i, j> [Require to linearize the multi-vehicle
constraint]
```

```
# Initialize the decision variables
for vehicle in V:
    for link in L:
        Rv[(vehicle, link)] = 0
        Xv[(vehicle, link)] = B
        Nv[(vehicle, link)] = 0
        Avv[(vehicle, link)] = 0
        Alvv[(vehicle, link)] = 0
        Agvv[(vehicle, link)] = 0
```

```
# Initialize the decision variables
for vehicle in V:
    for link in L:
        Rv[(vehicle, link)] = 0
        Xv[(vehicle, link)] = B
        Nv[(vehicle, link)] = 0
        Avv[(vehicle, link)] = 0
        Alvv[(vehicle, link)] = 0
        Agvv[(vehicle, link)] = 0
```

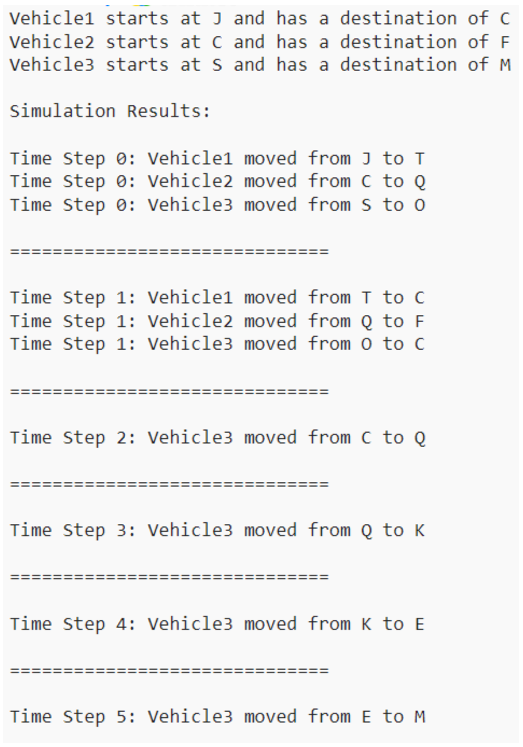
```
def FindNeighbors(node, graph):
    """
    This function finds the neighboring nodes of the given node in the graph.

    Args:
    - node: The current node.
    - graph: The graph represented as an adjacency list.

    Returns:
    - neighbors: The neighboring nodes of the given node.
    """
    return graph[node]


def GetTrafficTime(node1, node2, traffic_data):
    """
    This function calculates the traffic time between two nodes based on the traffic data.

    Args:
```

```
    - node1: The first node.
    - node2: The second node.
    - traffic_data: The traffic data represented as a dictionary.

    Returns:
    - traffic_time: The traffic time between the two nodes.
    """
    # Example traffic data format: {('A', 'B'): 5, ('B', 'D'): 10, ...}
    return traffic_data.get((node1, node2), TRoad)  # Return default traffic time if link not present


def GetDistance(node1, node2, coordinates):
    """
    This function calculates the Euclidean distance between two nodes.

    Args:
    - node1: The first node.
    - node2: The second node.
    - coordinates: The coordinates of the nodes represented as a dictionary.

    Returns:
    - distance: The Euclidean distance between the two nodes.
    """
    x1, y1 = coordinates[node1]
    x2, y2 = coordinates[node2]
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)


def Normalized(value, scale=1.0):
    """
    This function normalizes the given value based on the scale.

    Args:
    - value: The value to be normalized.
    - scale: The scale for normalization.

    Returns:
    - normalized_value: The normalized value.
    """
    return value / scale
```

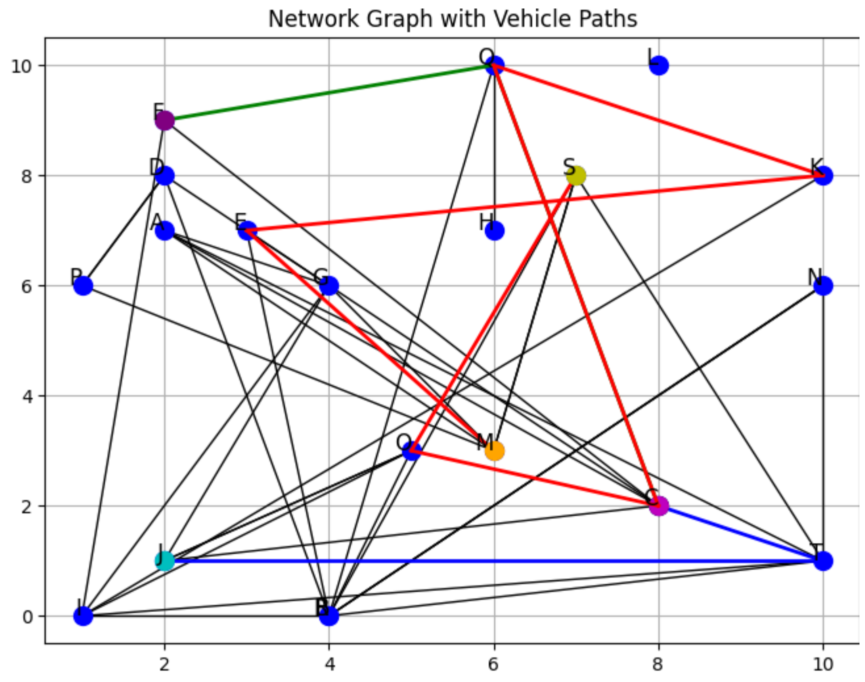
```
# BNART algorithm to find the best neighbor node for vehicle v
def BNART(graph, dead_end_nodes, current_vehicle_locations, destination, traversed_nodes,
vehicle, traffic_data):
    node = traversed_nodes[-1]  # Current intersection node
    neighbors = FindNeighbors(node, graph)
    best_value = float('inf')
    best_neighbor = None

    for neighbor in neighbors:
        if neighbor not in traversed_nodes and neighbor not in dead_end_nodes:
            traffic_time = GetTrafficTime(neighbor, node, traffic_data)
            distance = GetDistance(neighbor, destination, P)
            norm_value = Normalized(traffic_time + distance)

            if norm_value < best_value:
                best_value = norm_value
                best_neighbor = neighbor

                # Update the decision variables
                Rv[(vehicle, (node, neighbor))] = 1
                Xv[(vehicle, (node, neighbor))] = Tvs[vehicle]
                if (vehicle, (node, neighbor)) not in Nv:
                    Nv[(vehicle, (node, neighbor))] = 0
                Nv[(vehicle, (node, neighbor))] += 1
                Avv[(vehicle, (node, neighbor))] = 1
                Alvv[(vehicle, (node, neighbor))] = 1
                Agvv[(vehicle, (node, neighbor))] = 1

    if best_neighbor is not None:
        current_vehicle_locations.append((node, best_neighbor))
        traversed_nodes.append(best_neighbor)
        Tvs[vehicle] += TRoad + TDelay * Nv[(vehicle, (node, best_neighbor))]

    return best_neighbor


def plot_graph(P, L):
    plt.figure(figsize=(8, 6))
    for link in L:
        x = [P[link[0]][0], P[link[1]][0]]
        y = [P[link[0]][1], P[link[1]][1]]
        plt.plot(x, y, 'k-', lw=1)
    for node, coords in P.items():
        plt.plot(coords[0], coords[1], 'bo', markersize=10)
        plt.text(coords[0], coords[1], node, fontsize=12, ha='right')


def plot_network(ax, P, L, traversed_nodes, vehicle, Sv, Dv, path_color, start_color, end_color):
    prev_node = Sv[vehicle]
    for node in traversed_nodes[vehicle][1:]:
        x = [P[prev_node][0], P[node][0]]
        y = [P[prev_node][1], P[node][1]]
        ax.plot(x, y, color=path_color, lw=2)
        prev_node = node
    start = Sv[vehicle]
    end = Dv[vehicle]
    ax.plot(P[start][0], P[start][1], 'o', markersize=10, color=start_color)
    ax.plot(P[end][0], P[end][1], 'o', markersize=10, color=end_color)

def plot_paths(P, L, traversed_nodes, Sv, Dv, path_colors, start_colors, end_colors):
    fig, ax = plt.subplots(figsize=(8, 6))
    for link in L:
        x = [P[link[0]][0], P[link[1]][0]]
        y = [P[link[0]][1], P[link[1]][1]]
        ax.plot(x, y, 'k-', lw=1)
    for node, coords in P.items():
        ax.plot(coords[0], coords[1], 'bo', markersize=10)
        ax.text(coords[0], coords[1], node, fontsize=12, ha='right')

    for i, vehicle in enumerate(V):
        plot_network(ax, P, L, traversed_nodes, vehicle, Sv, Dv, path_colors[i], start_colors[i],
end_colors[i])

    ax.set_title("Network Graph with Vehicle Paths")
    ax.grid(True)
    plt.show()

# Print starting and destination points for each vehicle
```

```
for vehicle in V:
    print(f"{vehicle} starts at {Sv[vehicle]} and has a destination of {Dv[vehicle]}")

# Initialize simulation
graph = {node: [random.choice(list(P.keys())) for _ in range(random.randint(1, 5))] for node in
P.keys()}
dead_end_nodes = []
current_vehicle_locations = []
traversed_nodes = {vehicle: [Sv[vehicle]] for vehicle in V}

# Example traffic data (time in minutes)
traffic_data = {(link[0], link[1]): random.randint(1, 15) for link in L}

# Colors for paths, start points, and end points
path_colors = ['b', 'g', 'r']
start_colors = ['c', 'm', 'y']
end_colors = ['k', 'purple', 'orange']

# Start simulation
print("\nSimulation Results:\n")
for time_step in range(10):
    for vehicle in V:
        if traversed_nodes[vehicle][-1] != Dv[vehicle]:
            best_neighbor = BNART(graph, dead_end_nodes, current_vehicle_locations,
Dv[vehicle], traversed_nodes[vehicle], vehicle, traffic_data)
            print("Time Step {}: {} moved from {} to {}".format(time_step, vehicle,
traversed_nodes[vehicle][-2], traversed_nodes[vehicle][-1]))
    print("\n" + "=" * 30 + "\n")

# Plot paths of all vehicles on a single graph
plot_paths(P, L, traversed_nodes, Sv, Dv, path_colors, start_colors, end_colors)
```

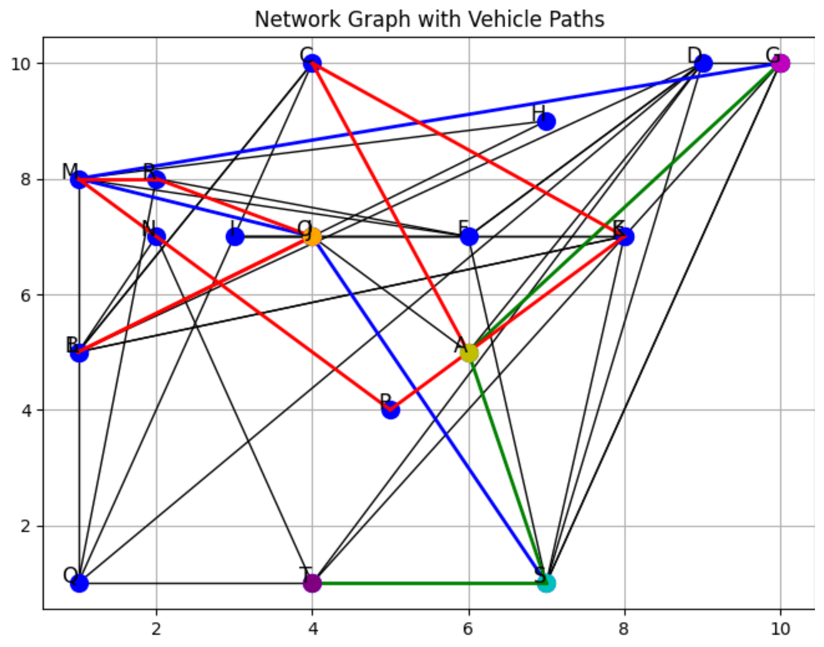
Sample Outputs:



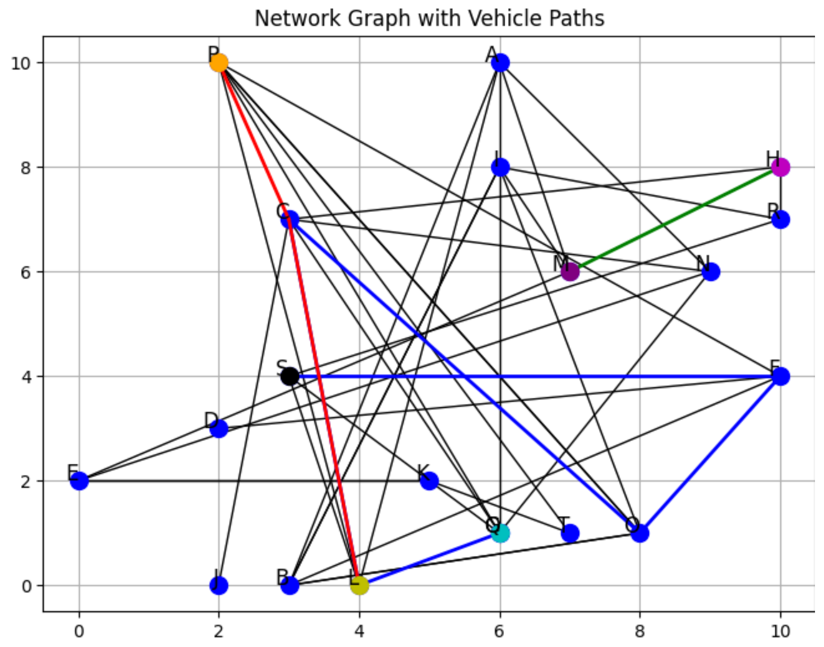
```
Vehicle1 starts at J and has a destination of C
Vehicle2 starts at C and has a destination of F
Vehicle3 starts at S and has a destination of M

Simulation Results:

Time Step 0: Vehicle1 moved from J to T
Time Step 0: Vehicle2 moved from C to Q
Time Step 0: Vehicle3 moved from S to O

==============================

Time Step 1: Vehicle1 moved from T to C
Time Step 1: Vehicle2 moved from Q to F
Time Step 1: Vehicle3 moved from O to C

==============================

Time Step 2: Vehicle3 moved from C to Q

==============================

Time Step 3: Vehicle3 moved from Q to K

==============================

Time Step 4: Vehicle3 moved from K to E

==============================

Time Step 5: Vehicle3 moved from E to M
```



```
Vehicle1 starts at S and has a destination of G
Vehicle2 starts at G and has a destination of T
Vehicle3 starts at A and has a destination of J

Simulation Results:

Time Step 0: Vehicle1 moved from S to O
Time Step 0: Vehicle2 moved from G to A
Time Step 0: Vehicle3 moved from A to C

==============================

Time Step 1: Vehicle1 moved from O to M
Time Step 1: Vehicle2 moved from A to S
Time Step 1: Vehicle3 moved from C to K

==============================

Time Step 2: Vehicle1 moved from M to G
Time Step 2: Vehicle2 moved from S to T
Time Step 2: Vehicle3 moved from K to P

==============================

Time Step 3: Vehicle3 moved from P to M

==============================

Time Step 4: Vehicle3 moved from M to R

==============================
```



```
Vehicle1 starts at Q and has a destination of S
Vehicle2 starts at H and has a destination of M
Vehicle3 starts at L and has a destination of P

Simulation Results:

Time Step 0: Vehicle1 moved from Q to L
Time Step 0: Vehicle2 moved from H to M
Time Step 0: Vehicle3 moved from L to C

==============================

Time Step 1: Vehicle1 moved from L to C
Time Step 1: Vehicle3 moved from C to P

==============================

Time Step 2: Vehicle1 moved from C to O

==============================

Time Step 3: Vehicle1 moved from O to F

==============================

Time Step 4: Vehicle1 moved from F to S

==============================
```


Network Graph with Vehicle Paths
P
A
I
H
C
R
M
N
S
F
D
E
K
Q
T
O
J
B
L