CSCE 5210- Fundamental of AI (Project 1-Part B)

RIDE SHARING APPLICATION

Submitted By: Salina Khadka (11695006)

Project link : https://github.com/salina9843/FAIassign

This project simulates a ride sharing application using a network graph.

- Libraries:
  - NetworkX: Used for graph operations like generating random graph and finding out shortest distance between two nodes and it's length.
  - Random: Used for random number generation.

APPROACH USED:

Four classes are used to fulfil the functionalities. They are explained below.

- **Class Road:**
  This class is basically responsible for creating the network graph. Two functions getDistanceEdge and getTotalNodes retrieve the weight of a specific edge and calculate total number of nodes in the graph respectively.

- **Class Van**
  This class represents the Vans used in the ride sharing app. relocateVan, hasReachedLimit, pickupPassengerReq, pickUpPassenger and dropOffPassenger, serveNextPassenger and isServiceComplete is provided for moving the van, checking its customer's capacity,picking up and dropping off passengers, serving the next customer and checking if the service is complete respectively.

- **Class Passenger**
  This class represents passenger or customers in the environment. Each passenger has a pickup and drop off location.

- **Class Algorithm**
  This is the core class where all the logic or algorithm for simulating the entire environment is written. It manages generation of passenger requests, allocation of vans to the customers, updating service queues, moving van from one node to another, deciding whether to pickup or drop off passengers and updating the routes.

  Computation of average distance travelled and average number of trips made for requirement 3-5 is done in this class.

Some of the approaches taken for main functionalities are described below:

- **Passenger Requests:**
  - Requests are generated at each clock tick. The clock tick starts from 0. The requests are hardcoded for R2 and are randomly generated for R3, R4 and R5.

- **Van Allocation:**
  - Vans are allocated based on van's current node and capacity.
  - Vans with available capacity and minimum distance to a customer's pickup location are assigned to that passenger. **A* algorithm** is used to find out shortest distance path and calculate the distance between two nodes.
  - If van is not available, the pickup node is added to **pass_pickup_list.**
  - The first empty van with least ID is given priority to pickup customers.

- **Van Movement:**
  - The movement of van (pickup/dropoff/moving to next location) is decided at each clock tick starting from 0.
  - Van's service queue determines its next node. If a van is at pickup or drop-off node it performs the corresponding action.

ASSUMPTIONS:

The assumptions made in this project are listed below:

- Minimum distance is set to 10000 in order to compare with the shortest distance between two nodes.
- The VanID, customerID and the clock ticks begins with 0.
- The vans are not allocated initially.

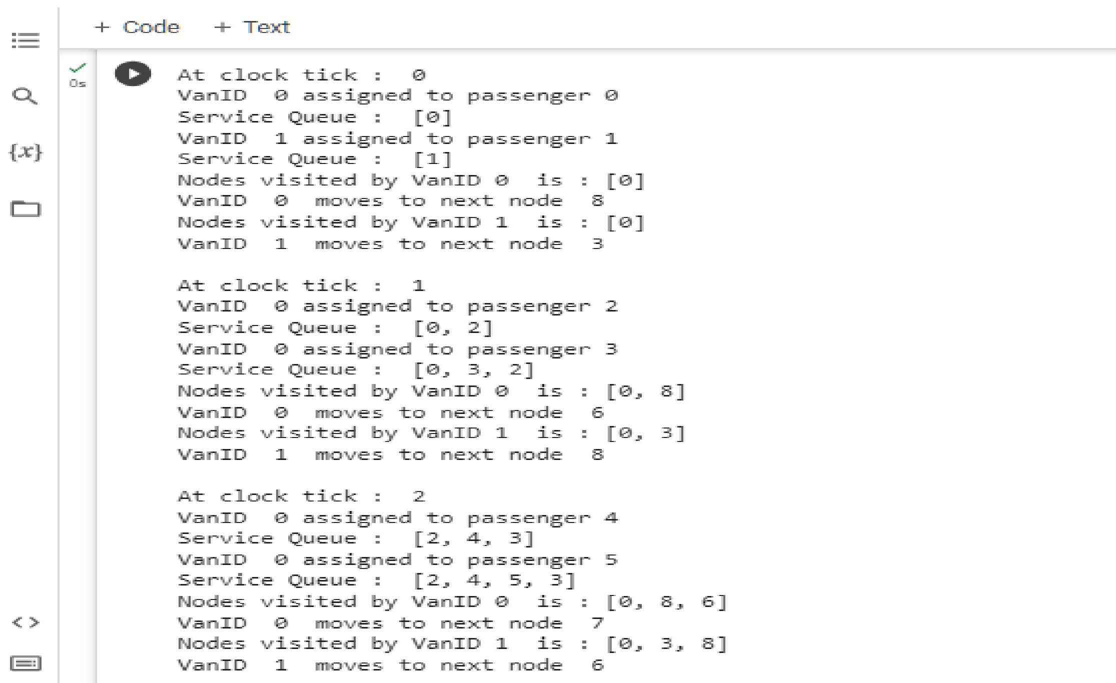
IMPORTANT NOTES:

- The 15 minutes customer wait rule is not implemented in my project.
- Graph is not displayed but for R2 the nodes and edge weights are hardcoded as given in the project description.
- For R2, the service queue only displays passengerID.

REQUIREMENT 2:

Below is the screenshot of output for R2 for clock tick 0,1 and 2. The service queue and route i.e. the node visited is displayed.

For requirement 2, total number of vans is set as 2, total nodes are 10, total requests are 6, the connectivity is 3 and the service is completed in 20 clock ticks.



```
+ Code   + Text

At clock tick :  0
VanID  0 assigned to passenger 0
Service Queue :  [0]
VanID  1 assigned to passenger 1
Service Queue :  [1]
Nodes visited by VanID 0  is : [0]
VanID  0  moves to next node  8
Nodes visited by VanID 1  is : [0]
VanID  1  moves to next node  3

At clock tick :  1
VanID  0 assigned to passenger 2
Service Queue :  [0, 2]
VanID  0 assigned to passenger 3
Service Queue :  [0, 3, 2]
Nodes visited by VanID 0  is : [0, 8]
VanID  0  moves to next node  6
Nodes visited by VanID 1  is : [0, 3]
VanID  1  moves to next node  8

At clock tick :  2
VanID  0 assigned to passenger 4
Service Queue :  [2, 4, 3]
VanID  0 assigned to passenger 5
Service Queue :  [2, 4, 5, 3]
Nodes visited by VanID 0  is : [0, 8, 6]
VanID  0  moves to next node  7
Nodes visited by VanID 1  is : [0, 3, 8]
VanID  1  moves to next node  6
```

REQUIREMENT 3

For requirement 3, total number of vans is set as 30, total nodes are 100, total requests are 600, and connectivity is 3.

(a) The average distance travelled over the fleet is 27.650000000000027
(b) The average no. of trips made over the fleet is 19.8

REQUIREMENT 4

For requirement 3, total number of vans is set as 60, total nodes are 100, total requests are 600, and connectivity is 3.

-> The average distance travelled is 13.035000000000014
-> The average no. of trips is 9.833333333333334

REQUIREMENT 5

For requirement 3, total number of vans is set as 60, total nodes are 100, total requests are 600, and connectivity is 4.

-> The average distance travelled is 14.963333333333338

Reason: We can see that the average distance travelled has decreased when we increase connectivity from 3 to 4. The probability of having shortest path between two nodes increases as there are more edges or paths between two nodes. The A* algorithm stores the shortest distance, hence there is a slight difference in average distance travelled.

CODE:

```
# Importing necessary libraries
import networkx as nx
import random

# creating a class named Road
class Road:
  # Constructor initializes a Road object when an instance of the class is
created
    def __init__(self, total_nodes, no_of_connection, add_connection,
specify_weight = [], seed = 1000):
        while(1):
            self.road = nx.gnp_random_graph (total_nodes, no_of_connection, seed
)
            # checking if all the nodes are connected
            if(not nx.is_connected(self.road)):
                print("Graph is not connected")
                print("--- Increasing connectivity----")
              # increase the connectivity if any node is not connected
                no_of_connection += add_connection
                print("Graph is connected now")
            else:
                break
        # determining an index for weights
        self.point = 0
        # Assigning random edge weights
        #  u and v are two nodes
```

```
        for u, v in self.road.edges:
          # when there is no weight
            if len(specify_weight) == 0:
              # random weights are assigned
                self.road.add_edge(u, v, weight = random.randint(1,9)/10)
          #  when there is weight
            else:
              # assign weights from specify_weight list
                self.road.add_edge(u, v, weight = specify_weight[self.point])
                # increment the index to assign another weight from the list
                self.point += 1
        # storing assigned weighths
        self.road_edges = nx.get_edge_attributes(self.road, "weight")
        # storing number of nodes
        self.total_nodes = self.road.number_of_nodes()

    # gettig the weight of edge in graph
    def getDistanceEdge(self, find_dis):
      # looping through road_edges to get all edge weights
        for dis in self.road_edges:
            if dis == find_dis:
              # return when matching weight is found
                return self.road_edges[dis]

    # getting total number of nodes
    def getTotalNodes(self):
        return self.total_nodes

    def findAStarDistance(self, begin, end):
        return nx.astar_path_length(self.road, begin, end)

    def findAStarPath(self, begin, end):
        return nx.astar_path(self.road, begin, end)

# creating van class
class Van:
    def __init__(self):
        self.limit = 0
        self.max_limit = 5
        self.visited_nodes = [0]
        self.current_location = 0
        self.pass_wait_list = []
        self.curr_service_route = []
        self.dis_travelled = 0.0
        self.pass_pickup_list = []
```

```
        self.active_pass = -1
        self.total_trips = 0

    # function for moving the van
    def relocateVan(self, nxt_node, distance):
        self.dis_travelled = self.dis_travelled + distance
        self.current_location = nxt_node
        self.visited_nodes.append(nxt_node)

    # checking if the van has exceeded it's limit
    def hasReachedLimit(self):
        return self.limit == self.max_limit

    # function for passenger's pickup request
    def pickUpPassengerReq(self, passenger_position):
        self.limit += 1
        self.pass_wait_list.append(passenger_position)

    # function for picking up passenger
    def pickUpPassenger(self, passenger_position):
        self.pass_wait_list.remove(passenger_position)
        self.pass_pickup_list.append(passenger_position)

    # function for dropping off passengers
    def dropOffPassenger(self, passenger_position):
        self.limit -= 1
        if self.active_pass != passenger_position:
            self.pass_pickup_list.remove(passenger_position)
        self.total_trips += 1
        self.active_pass = -1

    # function for removing pending passenger from list and updating it as active
passenger
    def serveNextPassenger(self):
        pending_pass_position = self.pass_pickup_list[0]
        self.pass_pickup_list.remove(pending_pass_position)
        self.active_pass = pending_pass_position

    # function for checking if all the lists are empty which indicates the
servicing os complete
    def isServiceComplete(self):
        has_pending_passenger = len(self.pass_wait_list) == 0
        has_pickup_pass = len(self.pass_pickup_list) == 0
        is_pass_being_served = self.active_pass == -1
        return has_pending_passenger and has_pickup_pass and is_pass_being_served
```

```
# creating class named passenger
class Passenger:
    def __init__(self, pickup_location, dropoff_location):
        self.pickup_location = pickup_location
        self.dropoff_location = dropoff_location

# creating a class which contains all the algorithms fot the environment
class Algorithm:
    def __init__(self, total_vans, total_nodes, no_of_connection, add_connection,
specify_edges = []):
        self.van_array = []
        # loop to create "total_vans" number of van objects
        for i in range(total_vans) :
            # create new van object
            van_obj = Van()
            self.van_array.append(van_obj)
        self.road = Road(total_nodes, no_of_connection, add_connection,
specify_edges)
        self.total_nodes = self.road.total_nodes
        self.pass_array = []

    # creating new passenger object
    def generatePassenger(self):
        # determine current position in passenger array
        passenger_position = len(self.pass_array)
        # generate a random pickup location within the total_nodes range
        pickup_location = random.randrange(self.total_nodes)
        dropoff_location = -1
        # looping to generate random dropoff location untill it's different than
pickup location
        while 1:
            dropoff_location = random.randrange(self.total_nodes)
            if dropoff_location != pickup_location:
                break
        # creating passenger object
        passenger = Passenger(pickup_location, dropoff_location)
        self.pass_array.append(passenger)
        return passenger_position

    # creating function to find first empty van
    def findFirstVacantVan(self, equidistance_list):
        # looping through the list of vans at equidistant to passenger
        for i in equidistance_list:
            if self.van_array[i].limit == 0:
```

```
                return i
        # if no van is found
        return -1

    #  creating function to allocate van to passengers
    def allocateVanToPass(self, passenger_position):
        pickup_location = self.pass_array[passenger_position].pickup_location
        minimum_dis = 10000
        equidistance_list = []
        # no van is allocated initially
        van_position = -1
        for i in range(len(self.van_array)):
            # check if the current van has reached it's capacity
            if self.van_array[i].hasReachedLimit():
                print("Van ", i, "is not available \n")
                continue
            # calculate distance from van's current location to passenger's
pickup location
            # comparing the distance to minimum distance
            # if the distance is smaller, update minimum_dis with the new
distance
            distance = self.road.findAStarDistance(pickup_location,
self.van_array[i].current_location)
            if distance < minimum_dis:
                minimum_dis = distance
                equidistance_list.clear()
                van_position = i
            # if the distance is equal to "minimum_dis" add the current van's
index to equidistance_list
            if distance == minimum_dis:
                equidistance_list.append(i)
        # check if there are any vans in equidistance_list
        if len(equidistance_list) != 0:
            # find the index of first non-vacant van
            # return its index
            first_non_vacant_van_position =
self.findFirstVacantVan(equidistance_list)
            if first_non_vacant_van_position != -1:
                return first_non_vacant_van_position
            else:
                return equidistance_list[0]
        else:
            # if no equidistant vans return the van with minimum distance
            return van_position
```

```
    # funstion for updating the service queue
    def changeWaitList(self, van_position):
        van_obj = self.van_array[van_position]
        # get the current location of van
        van_current_location = van_obj.current_location
        pass_in_wait_list = van_obj.pass_wait_list
        #sort the passengers in the wait list based on distance between their
pickup location and van's current location
        for i in range(len(pass_in_wait_list)):
            for j in range(i, len(pass_in_wait_list)):
                # calulate the distance for passenger i
                passenger_position_i = pass_in_wait_list[i]
                distance_i = self.road.findAStarDistance(van_current_location,
self.pass_array[passenger_position_i].pickup_location)
                # calulate the distance for passenger j
                passenger_position_j = pass_in_wait_list[j]
                distance_j = self.road.findAStarDistance(van_current_location,
self.pass_array[passenger_position_j].pickup_location)
                # compare the distances between passenger i and j
                if distance_j < distance_i:
                    # swap their position
                    temp = pass_in_wait_list[j]
                    pass_in_wait_list[j] = pass_in_wait_list[i]
                    pass_in_wait_list[i] = temp
        # update the van's service queue with sorted passengers
        van_obj.pass_wait_list = pass_in_wait_list
        print("Service Queue : ", pass_in_wait_list)

    # creating a function to move the van from current node to next node
    # retrieving the current node of the van object
    def relocateVanObject(self, van_obj, nxt_node):
        current_location = van_obj.current_location
        find_dis = ()
        if current_location < nxt_node:
            find_dis = (current_location, nxt_node)
        else:
            find_dis = (nxt_node, current_location)
        distance = self.road.getDistanceEdge(find_dis)
        if distance == None:
            distance = 0
        van_obj.relocateVan(nxt_node, distance)

    # function to determine if the van should pick or drop the passenger
    def decideNextAction(self, van_obj):
        van_current_location = van_obj.current_location
```

```
        # get position of the passenger currently being served by van
        active_service_passenger_position = van_obj.active_pass
        active_service_passenger_dropoff_location = -1
        if active_service_passenger_position != -1:
            active_service_passenger_dropoff_location =
self.pass_array[active_service_passenger_position].dropoff_location

        if van_current_location == active_service_passenger_dropoff_location:
            van_obj.dropOffPassenger(active_service_passenger_position)

        # checking for same drop off node
        for i in range(len(van_obj.pass_pickup_list)):
            try:
                passenger_position = van_obj.pass_pickup_list[i]
            except:
                break
            pickup_passenger_dropoff_loc =
self.pass_array[passenger_position].dropoff_location
            if van_current_location == pickup_passenger_dropoff_loc:
                van_obj.dropOffPassenger(passenger_position)


        # checking for same pickup node
        limit = van_obj.limit
        point = 0
        # calculating no of passenger in waiting list
        nxt_in_line_passenger_position_length = len(van_obj.pass_wait_list)
        # loop continues as long as there are passenger in waiting list
        while nxt_in_line_passenger_position_length != 0:
            nxt_in_line_passenger_position = van_obj.pass_wait_list[point]
            nxt_in_line_pass_pickup_location =
self.pass_array[nxt_in_line_passenger_position].pickup_location

            if van_current_location == nxt_in_line_pass_pickup_location and limit
<=5:
                van_obj.pickUpPassenger(nxt_in_line_passenger_position)
                limit += 1
                nxt_in_line_passenger_position_length -= 1
            else:
                break

    # funcyion for updating current service route
    # handling van reaching it's destination
    # needs to drop off or pickup passengers
    def validateAndChangeCurrentServiceRoute(self, van_obj):
```

```
        van_current_location = van_obj.current_location
        curr_service_route = van_obj.curr_service_route
        # if service route is empty
        if len(curr_service_route) == 0:
            if len(van_obj.pass_wait_list) != 0:
                passenger_position = van_obj.pass_wait_list[0]
            # if service route has locations
            else:
                passenger_position = van_obj.pass_pickup_list[0]
            pass_pickup_location =
self.pass_array[passenger_position].pickup_location
            changed_service_route = self.road.findAStarPath(van_current_location,
pass_pickup_location)

            if len(changed_service_route) != 1:
                changed_service_route.remove(van_current_location)
            van_obj.curr_service_route = changed_service_route
            return changed_service_route[0]
        else:
            van_obj.curr_service_route.remove(van_current_location)
            updated_service_path = van_obj.curr_service_route
            if len(updated_service_path) == 0:
                if len(van_obj.pass_pickup_list) != 0:
                    first_queue_passenger_position = van_obj.pass_pickup_list[0]
                    first_queue_customer_dropoff_location =
self.pass_array[first_queue_passenger_position].dropoff_location
                    van_obj.serveNextPassenger()
                    changed_service_route =
self.road.findAStarPath(van_current_location,
first_queue_customer_dropoff_location)
                    changed_service_route.remove(van_current_location)
                    van_obj.curr_service_route = changed_service_route
                    return changed_service_route[0]
                else:
                    if len(van_obj.pass_wait_list) != 0:
                        first_wait_queue_passenger_position =
van_obj.pass_wait_list[0]
                        fist_wait_queue_pass_pickup_location =
self.pass_array[first_wait_queue_passenger_position].pickup_location
                        changed_service_route =
self.road.findAStarPath(van_current_location,
fist_wait_queue_pass_pickup_location)
                        changed_service_route.remove(van_current_location)
                        van_obj.curr_service_route = changed_service_route
                        # returns first location in changed service route
```

```
                        return changed_service_route[0]
            else:
                # continue moving with current service path
                return updated_service_path[0]

    # handling a new passenger request
    def simulateNewPassengerReq(self, pass_obj, passenger_position):
        self.pass_array.append(pass_obj)
        min_distance_van_position = self.allocateVanToPass(passenger_position)
        if min_distance_van_position == -1:
            print("No car is available, try again in 15 minutes")
        else:
            print("VanID ", min_distance_van_position, "assigned to passenger",
passenger_position)

self.van_array[min_distance_van_position].pickUpPassengerReq(passenger_position)
            self.changeWaitList(min_distance_van_position)

    # generating and a new passenger request
    def newPassengerReqProcess(self):
        # creating a passenger request
        passenger_position = self.generatePassenger()
        min_distance_van_position = self.allocateVanToPass(passenger_position)
        if min_distance_van_position == -1:
            print("No car is available, try again in 15 minutes")
        else:

self.van_array[min_distance_van_position].pickUpPassengerReq(passenger_position)
            self.changeWaitList(min_distance_van_position)

    # this function interates through all the vans
    # checks current state
    # if there is a valid next node move van
    # decide pickup of drop off
    # supdate service route
    # simlulate relocation
    def relocateAllVans(self):
        # get a reference to the van array object
        van_array_obj = self.van_array
        for i in range(len(van_array_obj)):
            print("Nodes visited by VanID", i, " is :",
self.van_array[i].visited_nodes)
            if len(van_array_obj[i].pass_wait_list) ==0 and
len(van_array_obj[i].pass_pickup_list) == 0 and van_array_obj[i].active_pass == -
1:
```

```
                continue
            else:
                self.decideNextAction(van_array_obj[i])
                nxt_location_move =
self.validateAndChangeCurrentServiceRoute(van_array_obj[i])
                print("VanID ", i, " moves to next node ", nxt_location_move)
                if nxt_location_move != None:
                    self.relocateVanObject(van_array_obj[i], nxt_location_move)

    # function for relocating certain Van
    # checks current state
    # if there is a valid next node move van
    # decide pickup of drop off
    # supdate service route
    # simlulate relocation
    def relocateSpecificVan(self, i):
        van_array_obj = self.van_array
        # check if the van has no passengers waiting
        if len(van_array_obj[i].pass_wait_list) ==0 and
len(van_array_obj[i].pass_pickup_list) == 0 and van_array_obj[i].active_pass == -
1:
            print("\nVanID ", i, "has no passengers left so it is at Node",
van_array_obj[i].current_location)
        else:
            self.decideNextAction(van_array_obj[i])
            nxt_location_move =
self.validateAndChangeCurrentServiceRoute(van_array_obj[i])
            print("\nVanID ", i, " moves to node", nxt_location_move)
            if nxt_location_move != None:
                self.relocateVanObject(van_array_obj[i], nxt_location_move)

    # function to determine which van has service to be completed
    def checkAllServiceCompletion(self):
        remaining_van_position = []
        for i in range(len(self.van_array)):
            van_obj = self.van_array[i]
            has_all_service_done = van_obj.isServiceComplete()
            # check if the van has completed all the service assigned
            if has_all_service_done != True:
                # add van position to remianing position list
                remaining_van_position.append(i)
        return remaining_van_position

    # function to determine which van has service to be completed
    # check if specific service in serve_arr is completed by van
```

```
    def checkSpecificServiceCompletion(self, serve_arr):
        remaining_van_position = []
        for i in range(len(serve_arr)):
            van_position = serve_arr[i]
            van_obj = self.van_array[van_position]
            has_all_service_done = van_obj.isServiceComplete()
            if has_all_service_done != True:
                # add van position to remianing position list
                remaining_van_position.append(i)
        return remaining_van_position

    def computeAvgDisTravelled(self):
        total_distance = 0
        for i in range(len(self.van_array)):
            van_obj = self.van_array[i]
            total_distance += van_obj.dis_travelled
        return total_distance/len(self.van_array)

    def computeTotalTrips(self):
        total_trips = 0
        for i in range(len(self.van_array)):
            van_obj = self.van_array[i]
            total_trips += van_obj.total_trips
        return total_trips/len(self.van_array)

"""REQUIREMENT 2"""

if __name__ == "__main__":

    print("Author : Salina Khadka (11695006)\n")
    print("Fundamentals of AI - Project 1 :Part B")

    total_vans = 2
    total_nodes = 10
    no_of_connection = 0.3
    add_connection = 0.1
    # determining weights as provided in the question
    specify_edges = [0.1, 0.8, 0.6, 1.0, 1.0, 0.7, 0.8, 0.5, 0.5, 0.4, 1.0, 0.8,
0.9, 0.7, 0.4]
    algorithm = Algorithm(total_vans, total_nodes, no_of_connection,
add_connection, specify_edges)

    # defining the pickup and drop off point of each passenger
    pass1 = Passenger(8,9)
    pass2 = Passenger(3,6)
```

```
    pass3 = Passenger(4,7)
    pass4 = Passenger(2,4)
    pass5 = Passenger(1,7)
    pass6 = Passenger(1,9)
    point = 0
    for i in range(20):
        print("\nAt clock tick : ", i)
        if i == 0:
            # for passenger 1 request
            algorithm.simulateNewPassengerReq(pass1, point)
            point += 1
            algorithm.simulateNewPassengerReq(pass2, point)
            point += 1
            algorithm.relocateAllVans()
        elif i == 1:
            # for passenger 2 request
            algorithm.simulateNewPassengerReq(pass3, point)
            point += 1
            algorithm.simulateNewPassengerReq(pass4, point)
            point += 1
            algorithm.relocateAllVans()
        elif i == 2:
            # for passenger 3 request
            algorithm.simulateNewPassengerReq(pass5, point)
            point += 1
            algorithm.simulateNewPassengerReq(pass6, point)
            point += 1
            algorithm.relocateAllVans()
        else:
            algorithm.relocateAllVans()

    del algorithm

"""REQUIREMENT 4"""

# for R4
    total_vans = 60
    total_nodes = 100
    no_of_connection = 0.03
    add_connection = 0.01
    algorithm = Algorithm(total_vans, total_nodes, no_of_connection,
add_connection)
    for i in range(200):
        print("CLOCK TICK ", i)
    #    generating 600 passenger requets per hour
```

```
        for j in range(3):
            algorithm.newPassengerReqProcess()
            algorithm.relocateAllVans()
    remaining_van_position = algorithm.checkAllServiceCompletion()
    point = 1
    if len(remaining_van_position) !=0:
        while(len(remaining_van_position) != 0):
            print("At Clock Tick: ", point)

            for i in range(len(remaining_van_position)):
                van_position = remaining_van_position[i]
                algorithm.relocateSpecificVan(van_position)

            point += 1
            remaining_van_position =
algorithm.checkSpecificServiceCompletion(remaining_van_position)
        print("All Vans have completed their service")

    print("Average distance travelled is ", algorithm.computeAvgDisTravelled())
    print("Average no. of trips is ", algorithm.computeTotalTrips())
    del algorithm

"""REQUIREMENT 5"""

# for R5
    total_vans = 60
    total_nodes = 100
    no_of_connection = 0.04
    add_connection = 0.01
    algorithm = Algorithm(total_vans, total_nodes, no_of_connection,
add_connection)
    for i in range(200):
        print("CLOCK TICK ", i)
    #    generating 600 passenger requets per hour
        for j in range(3):
            algorithm.newPassengerReqProcess()
            algorithm.relocateAllVans()
    remaining_van_position = algorithm.checkAllServiceCompletion()
    point = 1
    if len(remaining_van_position) !=0:
        while(len(remaining_van_position) != 0):
            print("At Clock Tick: ", point)

            for i in range(len(remaining_van_position)):
                van_position = remaining_van_position[i]
```

```
                algorithm.relocateSpecificVan(van_position)

            point += 1
            remaining_van_position =
algorithm.checkSpecificServiceCompletion(remaining_van_position)
        print("All Vans have completed their service")

    print("Average distance travelled is", algorithm.computeAvgDisTravelled())
    del algorithm
```