APURBA POKHAREL
11627243

My approach and Additional Assumption:

OOP is something that I have a nice understanding of. So, my code is based on the ideas of classes and objects. I find it easy to work with classes and also, having a different class for each of the things like graph, customer, car, agent made writing the code easier and gave a structure to my code.
The classes that I have used and the main functionality of them are defined below:

1. Graph

- Generates the graph
- Computes the Astar path length as well as determines the shortest Astar path
- Can plot the graph if visualization is needed.
- The graph code is referred from Dr. Russel's tutorial 1 with slight changes as necessary.

2. Car

- Has methods to handle customer pickup requests, pickup customer, drop off customer.
- Stores the distance and trip for each car object.
- The information about capacity is stored here in the class.

3. Customer

- Has just the pickup and drop off node info (a randomly generated info) stored here.

4. Agent

- The brains of the entire operation.
- Has an array that stores all the cars and customer objects.
- An index (the actual array index) is used to refer to these cars and customers in the codebase.
  For Example: a car at index 0 in the car_array will be referred to by index 0 all over the code.
  Similarly, customer has the same rule.
- The Car object creation and customer object creation are done by the agent as specified by us in the main function.
- The request for picking up new customers, and selecting a car based on the shortest distance as well as the current capacity is handled here by the agent.
- The process of updating the wait queue based on the distance to the nearest customer is done here as well.
- Picking up and dropping off the customer is done by the agent.

The scheduling algorithm is the core of the code. Once that is figured out the rest of the code worked out on it's own.
Since, the scheduling algorithm works in a queue based manner. The data type that I used an array, had to have a queue based implementation for adding new customers and serving customers in the 0^{th} index of the array. So, this queue based approach can be seen in my code.

Some important consideration for my code:

1. As per Dr. Russel's tutorial 2 (clock tick 3), if two or more customers share the same pickup points then they need to be picked up together as long as the space is available.

2. The same goes for dropoff, if the currently being served customer shares dropoff with two or more customers (that are already picked up) then they need to be dropped off together.

Additional assumptions:

1. My program does not show the currently serving customer in the wait queue, I use something called a current_serving_customer to keep track of which customer needs to be picked up and dropped off.

2. Instead of having s1={(id1,p,8),(id1,d,9)} as service queue for tick 1, this program uses index of customer(starting from 0) like [0] for service queue in tick1.

   Additionaly for clock tick 3 instead of using notation like {(id1,d,9), (id3,p,4),(id3,d,7),(id5,p,1),(id6,p,1),(id5,d,7),(id6,d,9),(id4,p,2),(id4,d,4)}
   I use this [1,3,5,6,4]

   The index of customer can be used to get their pickup and drop off nodes. So, I don't store them.

3. The service queue is only updated and printed if there is request for customer.

4. Program counts customer and cars from 0 not 1.

5. Clock tick starts in 0 not 1.

The answers for each of the requirements are:

1. For R3

   The number of nodes = 100
   The number of cars = 30
   The connectivity = 3
   The reservation per hour = 600

a. the average distance traveled (over the fleet) per day when run on a road network of 100 nodes and average connectivity of 3?

   29.07

b. the average number of trips (over the fleet) per day when run on a road network of 100 nodes and average connectivity of 3?

   19.63

2. For R4

   The number of nodes = 100
   The number of cars = 60
   The connectivity = 3
   The reservation per hour = 600

   a. the average distance traveled (over the fleet) per day when run on a road network of 100 nodes and average connectivity of 3?

      16.29

   b. the average number of trips (over the fleet) per day when run on a road network of 100 nodes and average connectivity of 3?

      9.83

3. For R5

   The number of nodes = 100
   The number of cars = 60
   The connectivity = 4
   The reservation per hour = 600

   a. the average distance traveled (over the fleet) per day when run on a road network of 100 nodes and average connectivity of 3?

      14.346666666666671

   Explanation: The distance travelled decreases because there are more paths for the cars to take from one node to the other. Granted the decrease is not always the case since the values of the edges are generated in random. But for a graph with 100 nodes increasing connectivity from 3 to 4 decreases the distance travelled in most cases. Though in some cases the distance travelled was almost equal or even more than that of R4 but that is solely due to the randomness that is involved in assigning edges to the nodes.

Assignment1

September 30, 2023

[ ]:
```
import numpy as np
import random
import networkx as nx
import matplotlib.pyplot as plt
```

[ ]:
```
class Graph:
    def __init__(self, no_of_nodes, connectivity, increase, tutorial_weight =␣
 ↪[], seed = 1000):
        while(1):
            self.graph = nx.gnp_random_graph (no_of_nodes, connectivity, seed )
            if(not nx.is_connected(self.graph)):
                connectivity += increase
                print("running again as we don't have conncted graphs")
            else:
                break
        self.index = 0
        for u, v in self.graph.edges:
            if len(tutorial_weight) == 0:
                self.graph.add_edge(u, v, weight = random.randint(1,9)/10)
            else:
                self.graph.add_edge(u, v, weight = tutorial_weight[self.index])
                self.index += 1
        self.graph_edges = nx.get_edge_attributes(self.graph, "weight")
        self.no_of_nodes = self.graph.number_of_nodes()
        # print(self.graph_edges)

    def getEdgeWeight(self, search_key):
        for key in self.graph_edges:
            if key == search_key:
                return self.graph_edges[key]

    def getNumberOfNodes(self):
        return self.no_of_nodes

    def plotGraph(self):
        links = [(u, v) for (u, v, d) in self.graph.edges(data=True)]
        pos = nx.nx_agraph.graphviz_layout(self.graph)
```

```
        nx.draw_networkx_nodes(self.graph, pos, node_size=1200,␣
↪node_color='lightblue', linewidths=0.25)
        nx.draw_networkx_edges(self.graph, pos, edgelist=links, width=4)
        nx.draw_networkx_labels(self.graph, pos, font_size=20,␣
↪font_family="sans-serif")
        edge_labels = nx.get_edge_attributes(self.graph, "weight")
        nx.draw_networkx_edge_labels(self.graph, pos, edge_labels)
        plt.show()

    def computeAStarPathLength(self, start, finish):
        return nx.astar_path_length(self.graph, start, finish)

    def computeAStarPath(self, start, finish):
        return nx.astar_path(self.graph, start, finish)
```

[ ]:
```
class Car:
    #all cars are at node0 at the start of the day
    def __init__(self):
        self.capacity = 0
        self.max_capacity = 5
        self.current_node = 0
        self.nodes_traversed = [0]
        self.current_service_path = []
        self.customer_wait_queue = []
        self.customer_picked_up_queue = []
        self.distance_travelled = 0.0
        self.current_serving_customer = -1
        self.no_of_trips = 0

    def moveCar(self, new_node, distance):
        self.distance_travelled = self.distance_travelled + distance
        self.current_node = new_node
        self.nodes_traversed.append(new_node)

    def isFull(self):
        return self.capacity == self.max_capacity

    def pickUpCustomerRequest(self, customer_index):
        self.capacity += 1
        self.customer_wait_queue.append(customer_index)

    def pickUpCustomer(self, customer_index):
        self.customer_wait_queue.remove(customer_index)
        self.customer_picked_up_queue.append(customer_index)

    def dropOffCustomer(self, customer_index):
        self.capacity -= 1
```

```
        if self.current_serving_customer != customer_index:
            self.customer_picked_up_queue.remove(customer_index)
        self.no_of_trips += 1
        self.current_serving_customer = -1

    # call this after pickup done only and remove on dropoff
    def updateCurrentlyServingCustomer(self):
        next_to_be_served_index = self.customer_picked_up_queue[0]
        self.customer_picked_up_queue.remove(next_to_be_served_index)
        self.current_serving_customer = next_to_be_served_index

    def areAllJobsOver(self):
        is_wait_queue_empty = len(self.customer_wait_queue) == 0
        is_picked_up_queue_empty = len(self.customer_picked_up_queue) == 0
        is_serving_customer_empty = self.current_serving_customer == -1
        return is_wait_queue_empty and is_picked_up_queue_empty and␣
 ↪is_serving_customer_empty
```

[ ]:
```
class Customer:
    def __init__(self, pick_up_node, drop_off_node):
        self.pick_up_node = pick_up_node
        self.drop_off_node = drop_off_node
```

[ ]:
```
# Agent runs all the time
# Agent will have an instace of all cars and Customers generated

class Agent:
    def __init__(self, no_of_cars, no_of_nodes, connectivity, increase,␣
 ↪tutorial_edges = []):
        self.car_array = []
        # append no_of_cars objects to car_arrays
        for i in range(no_of_cars) :
            car_object = Car()
            self.car_array.append(car_object)
        self.graph = Graph(no_of_nodes, connectivity, increase, tutorial_edges)
        self.no_of_nodes = self.graph.no_of_nodes
        self.customer_array = []

    def createCustomerObject(self):
        customer_index = len(self.customer_array)
        pick_up_node = random.randrange(self.no_of_nodes)
        drop_off_node = -1
        while 1:
            drop_off_node = random.randrange(self.no_of_nodes)
            if drop_off_node != pick_up_node:
                break
        customer = Customer(pick_up_node, drop_off_node)
```

```
        self.customer_array.append(customer)
        return customer_index

    def getFirstEmptyCar(self, eq_distant_array):
        for i in eq_distant_array:
            if self.car_array[i].capacity == 0:
                return i
        return -1

    def getCarForCustomer(self, customer_index):
        # loop over all available car array
        # if equidistant cars then assign customer to the first non-empty car␣
↪from list of equidistant cars, else assign car to the lowest index car.
        # if no car equidistant then assign customer to car with smallest␣
↪distance
        # if all car have 5 passengers print wait message
        pick_up_node = self.customer_array[customer_index].pick_up_node
        smallest_distance = 10000000000
        eq_distant_array = []
        car_index = -1
        for i in range(len(self.car_array)):
            if self.car_array[i].isFull():
                print("Car ", i, "is full\n")
                continue
            distance = self.graph.computeAStarPathLength(pick_up_node, self.
↪car_array[i].current_node)
            if distance < smallest_distance:
                smallest_distance = distance
                eq_distant_array.clear()
                car_index = i
            if distance == smallest_distance:
                eq_distant_array.append(i)

        if len(eq_distant_array) != 0:
            first_non_empty_car_index = self.getFirstEmptyCar(eq_distant_array)
            if first_non_empty_car_index != -1:
                return first_non_empty_car_index
            else:
                return eq_distant_array[0]
        else:
            return car_index

    def updateWaitQueue(self, car_index):
        car_object = self.car_array[car_index]
        car_current_node = car_object.current_node
        customers_in_wait_queue = car_object.customer_wait_queue
        #sort
```

```
        for i in range(len(customers_in_wait_queue)):
            for j in range(i, len(customers_in_wait_queue)):

                customer_index_i = customers_in_wait_queue[i]
                distance_i = self.graph.
↪computeAStarPathLength(car_current_node, self.
↪customer_array[customer_index_i].pick_up_node)

                customer_index_j = customers_in_wait_queue[j]
                distance_j = self.graph.
↪computeAStarPathLength(car_current_node, self.
↪customer_array[customer_index_j].pick_up_node)

                if distance_j < distance_i:
                    temp = customers_in_wait_queue[j]
                    customers_in_wait_queue[j] = customers_in_wait_queue[i]
                    customers_in_wait_queue[i] = temp

        car_object.customer_wait_queue = customers_in_wait_queue
        print("\nthe service/wait queue is", customers_in_wait_queue)

    def moveCarObject(self, car_object, new_node):
        current_node = car_object.current_node
        search_key = ()
        if current_node < new_node:
            search_key = (current_node, new_node)
        else:
            search_key = (new_node, current_node)
        distance = self.graph.getEdgeWeight(search_key)
        if distance == None:
            distance = 0
        car_object.moveCar(new_node, distance)

    def checkPickUpOrDropOff(self, car_object):
        car_current_node = car_object.current_node

        current_servicing_customer_index = car_object.current_serving_customer
        current_servicing_customer_drop_off_node = -1
        if current_servicing_customer_index != -1:
            current_servicing_customer_drop_off_node = self.
↪customer_array[current_servicing_customer_index].drop_off_node

        if car_current_node == current_servicing_customer_drop_off_node:
            car_object.dropOffCustomer(current_servicing_customer_index)

        # need to check for same dropoff points iteratively
        for i in range(len(car_object.customer_picked_up_queue)):
```

```
            try:
                customer_index = car_object.customer_picked_up_queue[i]
            except:
                break
            pickup_customer_drop_off_point = self.
↪customer_array[customer_index].drop_off_node
            if car_current_node == pickup_customer_drop_off_point:
                car_object.dropOffCustomer(customer_index)


        # need to check for same pickup points iteratively
        capacity = car_object.capacity
        index = 0
        next_in_queue_customer_index_length = len(car_object.
↪customer_wait_queue)
        while next_in_queue_customer_index_length != 0:
            next_in_queue_customer_index = car_object.customer_wait_queue[index]
            next_in_queue_customer_pick_up_node = self.
↪customer_array[next_in_queue_customer_index].pick_up_node

            if car_current_node == next_in_queue_customer_pick_up_node and␣
↪capacity <=5:
                car_object.pickUpCustomer(next_in_queue_customer_index)
                capacity += 1
                next_in_queue_customer_index_length -= 1
            else:
                break

    def checkAndUpdateCurrentServicePath(self, car_object):
        # service path is the path taken by the car to
        # goto pickup a customer
        # or goto dropoff a picked customer
        # customer are picked based on the service queue

        car_current_node = car_object.current_node
        current_service_path = car_object.current_service_path

        if len(current_service_path) == 0:
            if len(car_object.customer_wait_queue) != 0:
                customer_index = car_object.customer_wait_queue[0]
            else:
                customer_index = car_object.customer_picked_up_queue[0]
            customer_pick_up_node = self.customer_array[customer_index].
↪pick_up_node
            new_service_path = self.graph.computeAStarPath(car_current_node,␣
↪customer_pick_up_node)
```

```
            if len(new_service_path) != 1:
                new_service_path.remove(car_current_node)
            car_object.current_service_path = new_service_path
            return new_service_path[0]
        else:
            car_object.current_service_path.remove(car_current_node)
            updated_service_path = car_object.current_service_path
            if len(updated_service_path) == 0:
                # either reached pick up or drop off point
                # update accordingly
                if len(car_object.customer_picked_up_queue) != 0:
                    # Just picked up or already picked customer need to drop␣
↪them off
                    first_queue_customer_index = car_object.
↪customer_picked_up_queue[0]
                    first_queue_customer_drop_off_node = self.
↪customer_array[first_queue_customer_index].drop_off_node
                    car_object.updateCurrentlyServingCustomer()
                    new_service_path = self.graph.
↪computeAStarPath(car_current_node, first_queue_customer_drop_off_node)
                    new_service_path.remove(car_current_node)
                    car_object.current_service_path = new_service_path
                    return new_service_path[0]
                else:
                    if len(car_object.customer_wait_queue) != 0:
                        # goto pickup first from wait queue/service queue if␣
↪present
                        first_wait_queue_customer_index = car_object.
↪customer_wait_queue[0]
                        fist_wait_queue_customer_pick_up_node = self.
↪customer_array[first_wait_queue_customer_index].pick_up_node
                        new_service_path = self.graph.
↪computeAStarPath(car_current_node, fist_wait_queue_customer_pick_up_node)
                        new_service_path.remove(car_current_node)
                        car_object.current_service_path = new_service_path
                        return new_service_path[0]
            else:
                # continue movement along the service path
                return updated_service_path[0]

    def processNewCustomerRequestSimulation(self, customer_objet,␣
↪customer_index):
        # get simulated customer object
        # compute ditance with the position of all cars, take capacity into␣
↪consderation, get the car index, else return wait 15 min message
```

```
        # assign customer to that car and update its service queue,
        # if no current service path find that else update current service path
        self.customer_array.append(customer_objet)
        min_distance_car_index = self.getCarForCustomer(customer_index)
        if min_distance_car_index == -1:
            print("All vans are full, please try again in 15 minutes")
        else:
            print("\nCar ", min_distance_car_index, "allocated to customer",␣
↪customer_index)
            self.car_array[min_distance_car_index].
↪pickUpCustomerRequest(customer_index)
            self.updateWaitQueue(min_distance_car_index)

    def processNewCustomerRequest(self):
        # create a new customer object and get it's index
        # compute ditance with the position of all cars, take capacity into␣
↪consderation, get the car index, else return wait 15 min message
        # assign customer to that car and update its service queue,
        # if no current service path find that else update current service path
        customer_index = self.createCustomerObject()
        min_distance_car_index = self.getCarForCustomer(customer_index)
        if min_distance_car_index == -1:
            # no car to take in customer
            print("All vans are full, please try again in 15 minutes")
        else:
            # print("Car ", min_distance_car_index, "allocated to customer",␣
↪customer_index)
            self.car_array[min_distance_car_index].
↪pickUpCustomerRequest(customer_index)
            self.updateWaitQueue(min_distance_car_index)

    def moveAllCars(self):
        # check if either pickup or dropoff available
        # check and update current service path (need to do this to get next␣
↪node to move to)
        # take the current service path and update the path as well as move the␣
↪car
        car_array_objects = self.car_array
        for i in range(len(car_array_objects)):
            print("\nTraversed history for car", i, " is :", self.car_array[i].
↪nodes_traversed)
            if len(car_array_objects[i].customer_wait_queue) ==0 and␣
↪len(car_array_objects[i].customer_picked_up_queue) == 0 and␣
↪car_array_objects[i].current_serving_customer == -1:
                # this car has no customer so dont move
```

```
                # print("Car ", i, "has no customer so stays parked in␣
↪location", car_array_objects[i].current_node)
                continue
            else:
                self.checkPickUpOrDropOff(car_array_objects[i])
                next_node_to_move_to = self.
↪checkAndUpdateCurrentServicePath(car_array_objects[i])
                print("\nCar ", i, " moves to new node ", next_node_to_move_to)
                if next_node_to_move_to != None:
                    self.moveCarObject(car_array_objects[i],␣
↪next_node_to_move_to)

    def moveSpecificCar(self, i):
        car_array_objects = self.car_array
        if len(car_array_objects[i].customer_wait_queue) ==0 and␣
↪len(car_array_objects[i].customer_picked_up_queue) == 0 and␣
↪car_array_objects[i].current_serving_customer == -1:
            # this car has no customer so dont move
            print("\nCar ", i, "has no customer so stays parked in location",␣
↪car_array_objects[i].current_node)
        else:
            self.checkPickUpOrDropOff(car_array_objects[i])
            next_node_to_move_to = self.
↪checkAndUpdateCurrentServicePath(car_array_objects[i])
            print("\nCar ", i, " moves to new node ", next_node_to_move_to)
            if next_node_to_move_to != None:
                self.moveCarObject(car_array_objects[i], next_node_to_move_to)

    def areAllServicesComplete(self):
        remaining_car_index = []
        for i in range(len(self.car_array)):
            car_object = self.car_array[i]
            is_all_jobs_over = car_object.areAllJobsOver()
            if is_all_jobs_over != True:
                remaining_car_index.append(i)
        return remaining_car_index

    def areSpecificServicesComplete(self, service_array):
        remaining_car_index = []
        for i in range(len(service_array)):
            car_index = service_array[i]
            car_object = self.car_array[car_index]
            is_all_jobs_over = car_object.areAllJobsOver()
            if is_all_jobs_over != True:
                remaining_car_index.append(i)
        return remaining_car_index
```

```
    def calculateAverageDistanceTravelled(self):
        total_distance = 0
        for i in range(len(self.car_array)):
            car_object = self.car_array[i]
            total_distance += car_object.distance_travelled
        return total_distance/len(self.car_array)

    def calculateAverageNoOfTrips(self):
        no_of_trips = 0
        for i in range(len(self.car_array)):
            car_object = self.car_array[i]
            no_of_trips += car_object.no_of_trips
        return no_of_trips/len(self.car_array)
```

[ ]:
```
print("-------------------------DISCLOSURE---------------------------------------\n")
print("")
print("MY PROGRAM DOES NOT SHOW THE CURRENTLY SERVING CUSTOMER IN THE WAIT␣
 ↪QUEUE\n")
print("INSTEAD OF HAVING S1={(id1,p,8),(id1,d,9)} AS SERVICE QUEUE FOR TICK␣
 ↪1\n")
print("THIS PROGRAM USES INDEX OF CUSTOMER(STARTING FROM 0) LIKE [0] FOR␣
 ↪SERVICE QUEUE IN TICK1\n")
print("THE SERVICE QUEUE IS ONLY UPDATED AND PRINTED AS LONG AS THERE IS␣
 ↪REQUEST FOR CUSTOMER\n")
print("BUT C1 IS ALREADY BEING SERVED SO IT IS NOT IN WAIT QUEUE SO MY WAIT␣
 ↪QUEUE IS [C2,C4,C5,C3](PROGRAM COUNTS CUSTOMER AND CAR FROM 0 NOT 1)\n")
print("CLOCK TICK STARTS IN 0 NOT 1\n")
print("-----------------------------------------------------------------------------\n")
```

[ ]:
```
# Run this for R2

if __name__ == "__main__":

    # FOR R2
    no_of_cars = 2
    no_of_nodes = 10
    connectivity = 0.3
    increase = 0.1
    # these are the edges of the nodes, since nodes are generated randomly we␣
 ↪need to generate node with these value to match tutorial 2
    tutorial_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]
    agent = Agent(no_of_cars, no_of_nodes, connectivity, increase,␣
 ↪tutorial_edges)
    # agent.graph.plotGraph()
```

```
    # Takes 20 clock ticks so
    c1 = Customer(8,9)
    c2 = Customer(3,6)
    c3 = Customer(4,7)
    c4 = Customer(2,4)
    c5 = Customer(1,7)
    c6 = Customer(1,9)
    index = 0
    for i in range(20):
        print("CLOCK TICK ", i, "\n")
        if i == 0:
            # use first customer request
            agent.processNewCustomerRequestSimulation(c1, index)
            index += 1
            agent.processNewCustomerRequestSimulation(c2, index)
            index += 1
            agent.moveAllCars()
        elif i == 1:
            # use second customer request
            agent.processNewCustomerRequestSimulation(c3, index)
            index += 1
            agent.processNewCustomerRequestSimulation(c4, index)
            index += 1
            agent.moveAllCars()
        elif i == 2:
            # use second customer request
            agent.processNewCustomerRequestSimulation(c5, index)
            index += 1
            agent.processNewCustomerRequestSimulation(c6, index)
            index += 1
            agent.moveAllCars()
        #just move cars
        else:
            agent.moveAllCars()
        print("\nCLOCK TICK ENDS", i, "\n")
        print("----------------------------------\n")


    # check if all service queue empty else do until empty
    # get the arrays of cars who's pickup queue, wait queue or current serving␣
↪is not emty
    # run an infinite loop over these cars until they are empty
    remaining_car_index = agent.areAllServicesComplete()
    index = 1
    if len(remaining_car_index) !=0:
        while(len(remaining_car_index) != 0):
```

```
            print("Additional clock tick", index)

            for i in range(len(remaining_car_index)):
                car_index = remaining_car_index[i]
                agent.moveSpecificCar(car_index)

            index += 1
            remaining_car_index = agent.
 ↪areSpecificServicesComplete(remaining_car_index)
            print("Additional tick ends", index, "\n")
        print("The job took an additional of", index - 1, " ticks to complete")

    del agent
```

[ ]:
```
# Run this for R3

if __name__ == "__main__":

    no_of_cars = 30
    no_of_nodes = 100
    connectivity = 0.03
    increase = 0.01
    agent = Agent(no_of_cars, no_of_nodes, connectivity, increase)
    # agent.graph.plotGraph()
    for i in range(200):
        print("CLOCK TICK ", i)
        # generating 10 reservation per minute i.e 600 request per hour
        for j in range(3):
            agent.processNewCustomerRequest()
            agent.moveAllCars()

    remaining_car_index = agent.areAllServicesComplete()
    index = 1
    if len(remaining_car_index) !=0:
        while(len(remaining_car_index) != 0):
            print("Additional clock tick", index)

            for i in range(len(remaining_car_index)):
                car_index = remaining_car_index[i]
                agent.moveSpecificCar(car_index)

            index += 1
            remaining_car_index = agent.
 ↪areSpecificServicesComplete(remaining_car_index)
        print("The job took an additional of", index - 1, " ticks to complete")
```

```
    print("Average distance covered = ", agent.
↪calculateAverageDistanceTravelled())
    print("Average no of trips = ", agent.calculateAverageNoOfTrips())
    del agent
```

[ ]:
```
# Run this for R4

if __name__ == "__main__":

    no_of_cars = 60
    no_of_nodes = 100
    connectivity = 0.03
    increase = 0.01
    agent = Agent(no_of_cars, no_of_nodes, connectivity, increase)
    # agent.graph.plotGraph()
    for i in range(200):
        print("CLOCK TICK ", i)
        # generating 10 reservation per minute i.e 600 request per hour
        for j in range(3):
            agent.processNewCustomerRequest()
            agent.moveAllCars()

    remaining_car_index = agent.areAllServicesComplete()
    index = 1
    if len(remaining_car_index) !=0:
        while(len(remaining_car_index) != 0):
            print("Additional clock tick", index)

            for i in range(len(remaining_car_index)):
                car_index = remaining_car_index[i]
                agent.moveSpecificCar(car_index)

            index += 1
            remaining_car_index = agent.
↪areSpecificServicesComplete(remaining_car_index)
        print("The job took an additional of", index - 1, " ticks to complete")

    print("Average distance covered = ", agent.
↪calculateAverageDistanceTravelled())
    print("Average no of trips = ", agent.calculateAverageNoOfTrips())
    del agent
```

[ ]:
```
# Run this for R5

if __name__ == "__main__":

    no_of_cars = 60
```

```
    no_of_nodes = 100
    connectivity = 0.04
    increase = 0.01
    agent = Agent(no_of_cars, no_of_nodes, connectivity, increase)
    # agent.graph.plotGraph()
    for i in range(200):
        print("CLOCK TICK ", i)
        # generating 10 reservation per minute i.e 600 request per hour
        for j in range(3):
            agent.processNewCustomerRequest()
            agent.moveAllCars()

    remaining_car_index = agent.areAllServicesComplete()
    index = 1
    if len(remaining_car_index) !=0:
        while(len(remaining_car_index) != 0):
            print("Additional clock tick", index)

            for i in range(len(remaining_car_index)):
                car_index = remaining_car_index[i]
                agent.moveSpecificCar(car_index)

            index += 1
            remaining_car_index = agent.
 ↪areSpecificServicesComplete(remaining_car_index)
        print("The job took an additional of", index - 1, " ticks to complete")

    print("Average distance covered = ", agent.
 ↪calculateAverageDistanceTravelled())
    print("Average no of trips = ", agent.calculateAverageNoOfTrips())
    del agent
```

[ ]:
```
!jupyter nbconvert --to pdf /content/Assignmnet1.ipynb
```

```
[NbConvertApp] Converting notebook /content/Assignmnet1.ipynb to pdf
[NbConvertApp] Writing 90439 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 70028 bytes to /content/Assignmnet1.pdf
```