APURBA POKHAREL

11627243

The github link to my project repo is [here](https://github.com/apurbapokharel/CSCE-5210/tree/main/Assignment%201).  
(https://github.com/apurbapokharel/CSCE-5210/tree/main/Assignment%201)

**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.

1. 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.

1. Customer

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

1. 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 0th 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

* 1. 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
  2. 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

1. For R4

The number of nodes = 100

The number of cars = 60

The connectivity = 3  
The reservation per hour = 600

* 1. 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
  2. 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

1. For R5

The number of nodes = 100

The number of cars = 60

The connectivity = 4  
 The reservation per hour = 600

* 1. 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.

The code

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:

# print("car drops off current serving customer")

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:

# print("-----------------------Pickedup queue empty breaking...--------------------------------")

break

pickup\_customer\_drop\_off\_point = self.customer\_array[customer\_index].drop\_off\_node

if car\_current\_node == pickup\_customer\_drop\_off\_point:

# print("car drops off pickedup customer")

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:

# print("car picks up a waiting customer index ", next\_in\_queue\_customer\_index)

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 car\_current\_node != new\_service\_path[0]:

# new\_service\_path.remove(car\_current\_node)

if len(new\_service\_path) != 1:

new\_service\_path.remove(car\_current\_node)

car\_object.current\_service\_path = new\_service\_path

# print(" The newly assigned service path is", 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:

# print("Service path ended need a new path")

# either reached pick up or drop off point

# update accordingly

if len(car\_object.customer\_picked\_up\_queue) != 0:

# serve the 1st from picked up queue

# print("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

# print(" The newly assigned service path is 2", 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

# print("need to go and pick up from wait queue")

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

# print(" The newly assigned service path is 3", new\_service\_path)

return new\_service\_path[0]

else:

# continue movement along the service path

# print("Continue on the same 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:

# no car to take in customer

# let this tick continue without picking up customer

print("ALL CARS FULL")

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

# let this tick continue without picking up customer

print("ALL CARS FULL")

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)

if \_\_name\_\_ == "\_\_main\_\_":

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")

"""

# 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

"""

# R3 #R4 #R5

# change the values as needed

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

# For R3

# Average distance covered = 29.070000000000025

# Average no of trips = 19.633333333333334

# For R4 Best case result

# Average distance covered = 16.289999999999996

# Average no of trips = 9.83

# R5 Best case result

# Average distance covered = 13.725000000000021

# Average no of trips = 9.883333333333334