CSC263 Project Part 2

Arun Jolly Marlier, Eric Hasegawa, Faustine Leung

August 9th 2020

1 Randomized Operation Implementations

Structure of Data

Our implementation will be built using the following three defined classes which will act as storage containers for relevant data.

```
class Car:
    int id # Unique id assigned to each car when first registering
    string driver_name # Full name of driver
    double avg_rating # Average star rating of driver from 0-5 stars
    string license_plate # Licence plate of driver's car
    double longitude #Longitude value of gps location
    double latitude #Latitude value of gps location
```

```
class Restaurant:
    double longitude # Longitude value of gps location
    double latitude # Latitude value of gps location
    string name # Name of restaurant
    double avg_ratings # Average customer review rating from 0-5 stars
```

```
class Weight_Key_Pair:
    int id # Unique car id of Car associated with present column
    int drive_time # Estimated time to drive from current column's car
        to current row's restaurant
```

For our operations to be implemented correctly, we will also assume that data is stored as described below.

```
List[List[Weight_Key_Pair]] adj_mat # This is the adjacency matrix for
    our graph. Each node stores a Weight_Key_Pair object that stores
    the unique id of the corresponding column, and the estimated
    driving time from the current column's car to the current row's
    restaurant

dict car_dict(<int>, <Car>) # This is the dictionary storing all active
    Car objects. The key is the car's unique id, and the value would be
    the matching Car object. Assume the implementation of the
    dictionary has best-case running time Θ(1) and worst-case running
    time Θ(1) for both the insert and delete operation.

List[Restaurant] restaurants # This is the list of active Restaurant
    objects in the graph. Each index in this list corresponds to the
    index of the target row in adj_mat.
```

We will also assume that an external function getDrivingTime(Restaurant r, Car c) exists where, given a Car and Restaurant object, it will return the estimated driving time for the Car to get to the Restaurant.

Assume indices start at 0.

Operations

Below are the implementations for each operation within our ADT. Assume valid input for each function.

This function is responsible for adding a new Car to the graph. It works by adding a new Weight_Key_Pair object to each row sublist within adj_mat. Each id in the node is the unique identifier given to the new Car, and each driving time in the node is given by the getDrivingTime() function. It then adds the Car object to car_dict under the key of it's matching unique id.

```
def addCar(Car c):
   for r_index in range(0, len(adj_mat)):
       curr_restaurant = restaurants[r_index]
       append Weight_Key_Pair(c.id,
           ceil(getDrivingTime(curr_restaurant, c))) to adj_mat[r_index]

       insert c into car_dict[c.id]
```

This function is responsible for removing a given Car from the graph. It works by first searching the first sublist in adj_mat for the matching Car, and then iterating through all rows and removing the item at the found column index. Finally it removes the matching Car object from car_dict.

```
def removeCar(Car c):
   bool found = False

   for c_index in range(0, len(adj_mat[0])):
       target_id = adj_mat[0][c_index].id

       if target_id == c.id:
           for r_index in range(0, len(adj_mat)):
               delete adj_mat[r_index][c_index]

           found = True
           break loop

   if found:
       delete car_dict[c.id]
```

```
    else:
        raise Exception
```

This function is responsible for adding a restaurant to the graph. It works by first appending a new row to adj_mat, then iterating through each column in the first sublist, and calculating new driving times for each Car in the graph. Finally we append the given Restaurant to the list of present restaurants. Assume the dictionary of cars is non-empty.

```
def addRestaurant(Restau!rant r):
    append empty list [] to adj_mat

    for i in range(0, len(adj_mat[0])):
        curr_car_id = adj_mat[0][i].id
        curr_car = car_dict[curr_car_id]

        append Weight_Key_Pair(curr_car.id, ceil(getDrivingTime(r,
            curr_car))) to adj_mat[len(adj_mat) - 1]

        append r to restaurants
```

This function is responsible for removing the given Restaurant from the graph. It works by searching through the storage list restaurants, and comparing it's contents to the given Restaurant object. When found, we remove the matching element from restaurants, and then remove the row corresponding to the matching index in adj_mat.

```
def removeRestaurant(Restaurant r):
    bool found = False

    for r_index in range(0, len(adj_mat)):
        if(restaurants[r_index] == r):
            delete adj_mat[r_index]
            delete restaurants[r_index]
            found = True
            break loop

    if not found:
        raise Exception
```

This function is responsible for changing the estimated driving time from a

given Car, to a given Restaurant. This would occur when a Car's location would be updated within the active system. It works by searching for the matching row, and then the matching column within that list, to get the exact target location. We then change the target Weight_Key_Pair's drive_time attribute to the new given time.

```
def setEdge(Car c, Restaurant r, int new_time):
    bool found = False
    for r_index in range(0, len(adj_mat)):
        if r == restaurants[r_index]:

            for c_index in range(0, len(adj_mat[r_index])):
                curr_car_id = adj_mat[r_index][c_index].id

                if car_dict[curr_car_id] == c:
                    adj_mat[r_index][c_index].drive_time = new_time
                    found = True
                    break loop

            break loop

    if not found:
        raise Exception
```

This function is responsible for returning the driving time from the given Car to the given Restaurant. It works by searching for the matching row, and then the matching column within that list, to get the exact target location. It then return's that target's drive_time value.

```
def getEdge(Car c, Restaurant r):
    bool found = False
    for r_index in range(0, len(adj_mat)):
        if r == restaurants[r_index]:

            for c_index in range(0, len(adj_mat[0])):
                curr_car_id = adj_mat[r_index][c_index].id

                if car_dict[curr_car_id] == c:
                    return adj_mat[r_index][c_index].drive_time

    if not found:
        raise Exception
```

These functions return the storage structures used within the graph to log

active Restaurant objects, and Car objects.

```
def getCars():
    return list of values of car_dict


def getRestaurants():
    return restaurants
```

This function is responsible for finding the three best cars to deliver food, given a Restaurant and estimated cooking time. It works by first iterating through every element in restaurants, and checking to see if any match the given Restaurant. Once we find the matching index i, we navigate to the row adj_mat[i] and then use an augmented merge sort, where the pivot point is selected randomly, on it. We want to sort the sublist by abs((Driving Time) - (Cooking Time)), from smallest to largest. This is to maximize optimization and minimize the amount of waiting a Car or Restaurant will have to do. After the sort, we then take the first 3 elements of the new list and return them as a sorted list. We provide 3 candidates, to account for if the first choice car declines the delivery.

```
def delivery(Restaurant r, int cook_time):
    bool found = False
    for r_index in range(0, len(adj_mat)):
        if restaurants[r_index] == r:
            adj_mat[r_index] = sort(adj_mat[r_index],
                len(adj_mat[r_index]), cook_time)
            found = True

            closest_cars = []
            for i in range(0, 3):
                id = adj_mat[r_index][i].id
                append car_dict[id] to closest_cars

            return closest_cars

    if not found:
        raise Exception


#Helper function
#Reference: https://www.geeksforgeeks.org/merge-sort/
def sort(List[Weight_Key_Pair] arr, int cook_time) : #returns
    List[Weight_Key_Pair]
    if len(arr) >1:
        pivot = randint(1, n-1) # Finding random pivot in array
```

```
L = arr[:pivot] # Dividing the array elements
R = arr[pivot:] # into 2 parts

mergeSort(L) # Sorting the first part
mergeSort(R) # Sorting the second part

i = j = k = 0

# Copy data to temp arrays L[] and R[]
while i < len(L) and j < len(R):
    if abs(L[i].drive_time - cook_time) < abs(R[j].drive_time -
        cook_time):
        arr[k] = L[i]
        i+= 1
    else:
        arr[k] = R[j]
        j+= 1
    k+= 1

# Checking if any element was left
while i < len(L):
    arr[k] = L[i]
    i+= 1
    k+= 1

while j < len(R):
    arr[k] = R[j]
    j+= 1
    k+= 1
```

2 Amortized Analysis Implementations

Structure of Data

Let's define classes that we will need to use.

```
class Car_Node:
    int id #Unique id assigned to each car when first registering
    int drive_time #Estimated time for the car to drive to a restaurant
        in minutes
    Car_Node next #Pointer to the next car in the linked list; next_car
        is None if there are no more cars in the linked list
```

```
class Car:
    int id #Unique id assigned to each car when first registering
    string driver_name #Full name of driver
    double avg_rating #Average star rating of driver from 0-5 stars
    string licence_plate #License plate of car
    (double,double) location #Location of car in coordinates
```

```
class Restaurant:
    Car_Node 1st_head #First car in restaurant's linked list of cars
    string name #Name of restaurant
    double avg_rating #Average customer review rating from 0-5 stars
    (double,double) location #Location of restaurant in coordinates
    Restaurant next #Pointer to the next restaurant in the linked list;
        next is None if there are no more restaurants in the linked list
```

We will be using a linked list of linked lists and a hash table to store the data.

```
Restaurant rest_1st_head #This is the head of the linked list containing
    nodes that represent restaurants. Each Restaurant contains its
    name, its average rating, its location, a pointer to the next
    restaurant in the linked list, and a pointer to a linked list
    containing nodes that represent cars. The cars in this linked list
    are differentiated by their unique id and their driving time to the
    restaurant that points to this linked list. This will be None if
    there are no elements in the linked list. If there are no cars, the
    first node of each restaurant's linked list of cars is None.

dict cars(<int>, <Car>) #This is the dictionary that contains
    information about each car: its unique id, the driver's name, the
    driver's average rating, its license plate, and its location.
```

```
Assume the implementation of the dictionary has best-case running
time Θ(1) and worst-case running time Θ(1) for both the insert and
delete operation.
```

We will also assume that an external function getDrivingTime(Restaurant r, Car c) exists where, given a Car and Restaurant object, it will return the estimated driving time from the Car to the Restaurant.

Assume indices start at 0.

Operations

Below are the implementations for each operation within our ADT. Assume valid input for each function.

This function is responsible for adding the given car with its respective estimated driving time to each restaurant. It traverses through the linked list of restaurants. For each of the restaurant's linked lists of cars, it adds a Car Node with the same id as Car c and its driving time to the restaurant. It also adds the car's information to the dictionary of cars.

```
def addCar(Car c):
    curr_rest = rest_1st_head

    while curr_rest is not None:
        curr_car = curr_rest.1st_head

        if curr_car is None:  #if the linked list of cars is empty
            curr_rest.1st_head = Car_Node(c.id,
                ceil(getDrivingTime(curr_rest, curr_car)))
        else:
            #iterate until curr_car is the last car in the list
            while curr_car.next is not None:
                curr_car = curr_car.next

            curr_car.next = Car_Node(c.id, ceil(getDrivingTime(curr_rest,
                curr_car)))

        curr_rest = curr_rest.next #iterate to next restaurant

    insert c into cars[c.id]
```

This function is responsible for removing the given car from each restaurant's linked list of cars. It iterates through each restaurant's linked list of cars and changes the pointer of the Car node in front of the Car node with the same id

as Car c. It also deletes the car from the dictionary. If the car is not found, nothing happens.

```
def removeCar(Car c):
    curr_rest = rest_1st_head

    #iterate through entire linked list of restaurants
    while curr_rest is not None:
        #if first car is given car
        if cars[rest_1st_head.id] == c.id:
            rest_1st_head = rest_1st_head.next
        else:
            prev_car = rest_1st_head
            curr_car = rest_1st_head.next

            #iterate through entire linked list of cars
            while curr_car is not None:
                if cars[curr_car.id] == c.id:
                    prev_car.next = curr_car.next

                    delete cars[c.id]

                    break loop

                #iterate curr_car to next car in list
                curr_car = curr_car.next
                #iterate prev_car to next car in list
                prev_car = prev_car.next

        curr_rest = curr_rest.next #iterate to next restaurant
```

This function is responsible for adding the given restaurant to the linked list of restaurants. It inserts Restaurant r to the front of the linked list of restaurants. It also iterates through each car in the dictionary and creates a linked list of cars for Restaurant r to point to. Assume the dictionary of cars is non-empty.

```
def addRestaurant(Restaurant r):
    #insert r to front of linked list of restaurants
    temp = rest_1st_head
    rest_1st_head = r
    r.next = temp

    i = 0
    for car in cars:
        if i == 0: #if car is first car in dictionary
            r.1st_head = Car_Node(car.id, ceil(getDrivingTime(r, car)))
```

```
            prev = r.1st_head
        else:
            temp = Car_Node(car.id, ceil(getDrivingTime(r, car)))
            prev.next = temp
            prev = temp
        i += 1
```

This function is responsible for removing the given restaurant from the linked list of restaurants. It traverses through the linked list of restaurants and removes Restaurant r by changing the restaurant in front's pointer. If Restaurant r is not found, nothing happens.

```
def removeRestaurant(Restaurant r):
    curr = rest_1st_head

    if curr is not None:  #if linked list of restaurants not empty
        #iterate until curr is last restaurant in list or curr is r
        while curr.next is not None and curr is not r:
            prev = curr
            curr = curr.next

        if curr is r:
            prev.next = curr.next
```

This function is responsible for setting Car c's drive time to Restaurant r to a new time. It traverses through the restaurant's linked list of cars and finds the Car node with the same id as the given car. Once the given car is found, it sets the drive time to the new drive time. If it does not find the given car, nothing happens. Note: Restaurant r is a pointer to a node in the linked list of restaurants.

```
def setEdge(Car c, Restaurant r, int new_time):
    curr = r.1st_head

    if curr is not None:  #if linked list of cars is not empty
        #iterate until curr is last car in linked or curr is given car
        while curr.next is not None and c.id != curr.id:
            curr = curr.next

        if c.id == curr.id:
            curr.drive_time = new_time
```

This function is responsible for returning the given car's driving time to a given restaurant. It traverses through the restaurant's linked list of cars and finds the Car node with the same id as the given car. If it finds Car c, it returns the estimated driving time. Note: Restaurant r is a pointer to a node in the linked list of restaurants.

```
def getEdge(Car c, Restaurant r):
    curr = r.1st_head

    #iterate until curr is last car in list or curr is given car
    while curr.next is not None and c.id != curr.id:
        curr = curr.next

    if c.id == curr.id:
        return curr.drive_time
```

These functions return all the cars and restaurants associated with the graph. getCars() returns a list of all Car objects containing all the cars' information. getRestaurants() traverses through the linked list of restaurants and appends each restaurant to a list. The function returns a list of restaurants.

```
def getCars():
    return list of all cars in dict cars


def getRestaurants():
    curr = rest_1st_head
    restaurants = []

    while curr is not None: #iterate through linked list of restaurants
        append curr to restaurants
        curr = curr.next

    return restaurants
```

This function is responsible for returning the three cars with the smallest absolute difference of the cooking time and the car's driving time to Restaurant r. It iterates through the restaurant's linked list of cars and creates a list of the value stated above for each car. It uses a helper function to insert all the values in a min priority queue using a linked list. It then pops off the first three nodes, which are the three smallest absolute values. It then returns the three cars with the corresponding smallest values in the dictionary of cars. Note: Restaurant r is a pointer to a node in the linked list of restaurants; Assume class Node is

defined with attributes - value and next (referring to the next node in the linked list) - and the next attribute will point to None if there are no more Nodes in the linked list.

```
def delivery(Restaurant r, int cook_time):
    curr = r.1st_head

    #values and car_ids are indexed the same - first value in values
        corresponds to the first car in car_ids
    values = []
    car_ids = []

    #store stated absolute value in list for all cars
    while curr is not None:
        val = abs(curr.drive_time - cook_time)
        append val to values
        append curr.id to car_ids

        curr = curr.next

    #insert all elements in values in linked list of increasing order
    head = Node(values[0])
    for i in range(1,len(values)):
        head = push(head,values[i])

    #find three smallest values
    three_smallest_values = []
    for i in range(0,3):
        node, head = pop(head)
        append node.value to three_smallest_values

    #find ids of the three cars with the smallest absolute value
    three_smallest_car_ids = []
    for i in range(0,len(values)):
        for j in range(0,len(three_smallest_values)):
            if values[i] == three_smallest_values[j]:
                append i to three_smallest_car_ids

    #find corresponding cars to return
    three_smallest_cars = []
    for id in three_smallest_car_ids:
        append cars[id] to three_smallest_cars

    return three_smallest_cars


#Helper function - PUSH
#Reference:
    https://www.geeksforgeeks.org/priority-queue-using-linked-list/
def push(Node head, int new_val): #returns linked list with increasing
```

```
    priority
    new = Node(new_val)

    if new.value < head.value:
        temp = head
        head = new
        new.next = temp
    else:
        curr = head
        while curr.next is not None and curr.next.value < new_val:
            curr = curr.next
        new.next = curr.next
        curr.next = new

    return head


#Helper function - POP
#Reference:
    https://www.geeksforgeeks.org/priority-queue-using-linked-list/
#returns Node with lowest priority in linked list and updated linked list
def pop(Node head):
    temp = head
    head = head.next
    return temp, head
```