Shreya Wagley wagleyshreya@gmail.com

Complexity Analysis

The problem at hand focuses on addressing challenges related to electric vehicle (EV) charging in Sydney metropolitan. Despite the increasing popularity of EVs, the number of public charging stations remains limited compared to the growing number of EVs on the road. This assignment aims to provide solutions to several EV charging-related problems. The scenario involves a road network in Sydney metropolitan, consisting of 21 locations where some locations have charging stations (indicated by red dots) while others do not (black dots). The provided data includes information about distances between adjacent locations, location details, charging station availability, and charging prices. Factors such as origin, destination, remaining power, travel cost, amount of power to charge, and charging cost influence the EV's choice of where to charge. The objective is to find the nearest charging station when the vehicle is low on power or to identify the cheapest charging station within the vehicle's remaining travel range. Additionally, determining the number of charging stations along the shortest path between two locations is also required. By addressing these challenges, the assignment aims to contribute to the optimization of EV charging in Sydney metropolitan.

Data Structures:

Location.h :

The Location class represents a location with its properties such as index, name, charger installation status, and charging price. It also provides a function to print the location details and an overloaded less-than operator for comparison purposes. The following major data structures are used:

- int: used to store the index of a location.
- string: used to store the name of a location.
- bool: used to represent whether a charger is installed at a location (true or false).
- double: used to store the charging price at a location.

WeightedGraph.h :

The WeightedGraphType class represents a weighted graph. It provides functionalities to create the graph from an adjacency matrix file (weights.txt), print the adjacency matrix and list, and find the shortest path using Dijkstra's algorithm. The following major data structures are used:

- int: used to store the size of the graph (number of vertices).
- list<int>: used to store the adjacency list for each vertex in the graph.
- double**: this is a pointer to a two-dimensional array used to store the weights of edges in the graph.
- vector<double>: used to store the smallest weights for each vertex in the graph during the shortest path calculation.
- stack<int>: used to store the vertices in the shortest path from an origin to a destination.
- bool*: this is a pointer to a Boolean array used to keep track of visited vertices during the shortest path calculation.

EVCharging.h :

The EVCharging class represents a system for managing electric vehicle charging stations. It provides functionalities such as inputting and storing locations with their charging information, calculating and displaying various charging-related information, finding the cheapest charging stations based on different criteria, and determining the optimal charging path between two locations considering travel and charging costs. It interacts with the `WeightedGraphType` class for graph operations and path calculations. The following major data structures are used:

- map<int, Location> locations: this is a map that stores locations with their corresponding indices. The key is an integer representing the index, and the value is an object of the Location class.
- int numberOfLocations: used to hold the number of locations.
- WeightedGraphType* weightedGraph: this is a pointer to an object of WeightedGraphType, representing a weighted graph.
- vector<double> shortestPath1: this is a vector that stores the shortest path distances from the origin to each location in the graph.
- vector<double> shortestPath2: this is a vector that stores the shortest path distances from the destination to each location in the graph.
- stack<int> path1: this is a stack used to store the path from the origin to the lowestID in the cheapestChargingPath() and bestChargingPath() functions.
- stack<int> path2: this is a stack used to store the path from the lowestID to the destination in the cheapestChargingPath() and bestChargingPath() functions.
- stack<int> path3: this is a stack used to store the path from the freeCharging charging station to the destination in the bestChargingPath() function.

Algorithms:

Task 6:

This task determines the closest charging station to the user's location. It takes user input for the location, calculates the shortest distances from that location to all other stations using a graph algorithm, and then iterates through the distances to find the station with the smallest distance. The function considers only stations that have a charging station installed and outputs the name of the nearest charging station along with its distance from the user.

Pseudocode:

```
function closestChargingStation():
   userInput = getLocationInput()
   if userInput is invalid:
      return

   shortestPath vector = shortest distance to each station form user’s location
   nearestDistance = infinity
   nearestStation = 0

   for each station in shortestPath:
      if (station is not the user's location and station has a charging station and
      shortestPath[station] is smaller than nearestDistance):
         nearestDistance = shortestPath[station]
         nearestStation = station

   print "The nearest charging station to you is", getLocationName(nearestStation), "which
   is", nearestDistance, "kilometers away."
```

Task 7:

This task takes user input for a location and generates a random charging amount. It then calculates the costs of charging at different charging stations (excluding the current charging station) based on the given location and charging amount. If a charging station is found and , it displays the name of the nearest and cheapest charging station along with the associated charging and travel costs. If no charging station is found, it displays a message indicating the absence of results.

Pseudocode:

```
function cheapestStationOther():
    userInput = getLocationInput()
    if userInput is invalid:
        return

    chargingAmount = random integer between 10 and 50
    print "You'll charge", chargingAmount, "kWh"

    chargingCost, travelCost = 0, 0
    cheapestStationId = cheapest charging station avoiding the current charging station

    if there is a charging station available:
        print "The nearest and cheapest charging station is" nameOf.cheapestStationId
        print chargingCost
        print travelCost
        print chargingCost + travelCost
    else:
        print "No charging stations found!"
```

Task 8:

This task calculates the cheapest charging path for an electric vehicle journey. It takes user inputs for the origin and destination locations, generates a random charging amount, and finds the charging station with the lowest cost along with the associated travel cost. It then displays the details of the lowest cost charging station, the travel path from the origin to the destination, and the total cost. If no suitable charging station is found, it notifies the user.

Pseudocode:

```
function cheapestChargingPath():
    origin = getLocationInput()
    if origin is invalid:
        return

    destination = getLocationInput()
    if destination is invalid:
        return

    chargingAmount = random integer between 10 and 50
    print "You'll charge", chargingAmount, "kWh"

    lowestId, chargingCost, travelCost = findCheapestChargingStation(origin, destination,
chargingAmount)

    if lowestId is not -1:
        print "The nearest charging station with the lowest cost is " +
        locations[lowestId].locationName
        print "Charging cost: $" + chargingCost
        print "Travel cost: $" + travelCost
        print "Total cost: $" + (chargingCost + travelCost)
    else:
        print "No results found!"

    print "Travel path:"

    if lowestId is not origin:
        stack path1 = calculateShortestPath(origin, lowestId)
        while path1 has more than one location:
            print locations[path1.top()].locationName + ", "
            path1.pop()

    stack path2 = calculateShortestPath(lowestId, destination)
    while path2 is not empty:
        print locations[path2.top()].locationName + ", "
        path2.pop()
```

Shreya Wagley wagleyshreya@gmail.com

Complexity Analysis:

Task 6:

The average complexity of the closestChargingStation() function can be estimated based on the complexity of the shortestPath() function (which uses Dijkstra's shortest path algorithm) and the subsequent loop to find the nearest charging station.

- Calculating the shortest paths using shortestPath():
  The time complexity of shortestPath() (Dijkstra's algorithm) is typically O((n + m)log(n)), where n is the number of vertices and m is the number of edges in the graph.
- Finding the nearest charging station:
  The loop iterates over shortestPath.size() elements, which is typically the number of vertices (n) in the graph.
  The operations inside the loop (locations[i].chargerInstalled and shortestPath[i] < nearest) are constant time operations.

Overall, the dominant factor in terms of time complexity is the calculation of the shortest path, which has an approximate complexity of O((n + m)log(n)), where n is the number of locations (locations) and m is the number of edges in the graph.Therefore, the overall average complexity can be approximated as O((n + m)log(n)).

Task 7:

The average complexity of the cheapestStationOther() function can be estimated based on the complexity of the cheapestChargingStation() function and other additional operations to find the other cheapest charging station.

- getLocationInput(): This operation retrieves input, which typically takes constant time, so its complexity is O(1).

- rand() % 41 + 10: Generating a random number within a range takes constant time, so its complexity is O(1).

- Assigning variables and printing messages: These operations are all constant-time operations and do not contribute significantly to the overall complexity.

- cheapestChargingStation(): This function is called with certain arguments. The complexity of the cheapestChargingStation() function is dominated by the computation of shortest paths using Dijkstra's algorithm, which is estimated above to be O((n + m)log(n)).

- Conditional statements and printing results: These operations are constant-time operations and do not significantly affect the overall complexity.

Therefore, considering that the dominant factor is the cheapestChargingStation() function, the average complexity of the cheapestStationOther() function can be estimated as O((n + m)log(n)) based on the complexity of cheapestChargingStation().

Task 8:

The average complexity of the cheapestChargingPath() unction can be estimated based on the complexity of the cheapestChargingStation(), shortestPath() and other additional operations to find the cheapest charging path form origin to destination.

- getLocationInput(): This operation retrieves input, which typically takes constant time, so its complexity is O(1).
- Conditional statements: Checking if the input is -1 is a constant-time operation and does not significantly affect the overall complexity.
- rand() % 41 + 10: Generating a random number within a range takes constant time, so its complexity is O(1).
- Assigning variables and printing messages: These operations are all constant-time operations and do not contribute significantly to the overall complexity.
- cheapestChargingStation(): This function is called with certain arguments. Based on the analysis performed above, its complexity is dominated by the computation of shortest paths using Dijkstra's algorithm, which is estimated to be O((n + m)log(n)).
- Conditional statements and printing results: These operations are constant-time operations and do not significantly affect the overall complexity.
- weightedGraph->shortestPath(): This function is called twice, once for `path1` and once for `path2`. Since its use Dijkstra's algorithm we know its complexity is typically O((n + m)log(n)).
- Printing the travel path: This operation involves iterating over the paths and printing the locations, which takes linear time based on the number of locations in the paths.

Therefore, considering the dominant factors affecting the average complexity of the cheapestChargingPath() function are the cheapestChargingStation() function and the weightedGraph->shortestPath() function, the average complexity can be estimated as O((n + m)log(n)) based on the complexities of these functions.

In conclusion, this complexity analysis report focuses on addressing the challenges related to electric vehicle (EV) charging in Sydney metropolitan. It proposes solutions to various EV charging-related problems, such as finding the nearest charging station, identifying the cheapest charging station within the travel range, and determining the number of charging stations along the shortest path between two locations. The report introduces key data structures like Location, WeightedGraphType, and EVCharging classes, along with their respective functionalities and interactions. The report also presents pseudocode for three tasks: determining the closest charging station, finding the nearest and cheapest charging station based on a charging amount, and calculating the cheapest charging path for a journey. Additionally, the complexity analysis provides an estimate of the average complexities for these tasks, considering factors such as graph operations and shortest path calculations. By addressing these challenges and utilizing the proposed algorithms, the assignment aims to contribute to the optimization of EV charging in Sydney metropolitan, ultimately enhancing the accessibility and efficiency of electric vehicle infrastructure in the region.

Shreya Wagley wagleyshreya@gmail.com

Appendix

cheapestChargingStation function :

Overview:

The `cheapestChargingStation` function aims to find the most cost-effective charging station for an electric vehicle trip between an origin and destination. It calculates the shortest paths from the origin and destination to all locations, considering the charging amount required. Then, it iterates over the locations, excluding the origin location specified, and calculates the cost for each location based on the combined travel distance, charging amount, and charging price. The function selects the location with the lowest cost, considering certain conditions such as the presence of a charging station and pricing thresholds. It returns the ID of the chosen charging station and provides the travel cost and charging cost as output parameters.

Pseudocode:

```
function cheapestChargingStation(origin, destination, avoid, chargingAmount, travelCost,
chargingCost):
    shortestPath1 = calculateShortestPathFromOriginToAllLocations(origin)
    shortestPath2 = calculateShortestPathFromDestinationToAllLocations(destination)
    lowestCost = infinity
    lowestID = -1

    for i = 0 to the number of locations:
        if i is equal to avoid:
            continue

        cost = calculateTotalCost(shortestPath1[i], shortestPath2[i], chargingAmount,
locations[i].chargingPrice)

        if locations[i] has a charging station and cost < lowestCost:
            if locations[i].chargingPrice > 0 or chargingAmount > 25:
                lowestCost = cost
                lowestID = i

    travelCost = calculateTravelCost(shortestPath1[lowestID], shortestPath2[lowestID])
    chargingCost = calculateChargingCost(chargingAmount,
locations[lowestID].chargingPrice)

    return lowestID
```

Shreya Wagley wagleyshreya@gmail.com

vector shortestpath(index) :

Overview:

The `shortestPath` function uses Dijkstra's shortest path algorithm to efficiently compute the shortest paths in a graph, it calculates the shortest path distances from a given vertex to all other vertices in a weighted graph. It initializes a `smallestWeight` vector with the weights of the edges connecting the given vertex to other vertices. The function iteratively selects the vertex with the smallest weight, marks it as visited, and updates the `smallestWeight` vector by considering the neighboring vertices. This process continues until all vertices have been visited. Finally, the function returns the `smallestWeight` vector, which represents the shortest path distances from the given vertex to all other vertices in the graph. The function uses Dijkstra's shortest path algorithm to efficiently compute the shortest paths in a graph.

Pseudocode:

```
function shortestPath(index):
   smallestWeight <- create a vector of size gSize

   for each vertex j in the graph:
      set smallestWeight[j] as the weight of the edge between index and j

   weightFound <- create a boolean array of size gSize

   for each vertex j in the graph:
      set weightFound[j] as false

   set weightFound[index] as true
   set smallestWeight[index] as 0

   repeat gSize - 1 times:
      minWeight <- infinity
      v <- 0

      for each vertex j in the graph:
         if weightFound[j] is false:
            if smallestWeight[j] is less than minWeight:
               set v as j
               set minWeight as smallestWeight[v]

      set weightFound[v] as true

      for each vertex j in the graph:
         if weightFound[j] is false:
            if minWeight + weight of the edge between v and j is less than smallestWeight[j]:
               set smallestWeight[j] as minWeight + weight of the edge between v and j

   return smallestWeight
```

stack shortestPath(origin, destination) : Pseudocode:

```
function shortestPath(origin, destination):
   smallestWeight <- create an empty vector of size gSize

   for each vertex j in the graph:
      set smallestWeight[j] as the weight of the edge between origin and j
   weightFound <- create a boolean array of size gSize

   for each vertex j in the graph:
      set weightFound[j] as false

   set weightFound[origin] as true
   set smallestWeight[origin] as 0

   repeat gSize - 1 times:
      minWeight <- infinity
      v <- 0

      for each vertex j in the graph:
         if weightFound[j] is false:
            if smallestWeight[j] is less than minWeight:
               set v as j
               set minWeight as smallestWeight[v]

      set weightFound[v] as true

      for each vertex j in the graph:
         if weightFound[j] is false:
            if minWeight + weight of the edge between v and j is less than smallestWeight[j]:
               set smallestWeight[j] as minWeight + weight of the edge between v and j

   pathStack <- create an empty stack
   current <- destination
   pathStack.push(current)

   while current is not equal to origin:
      pathFound <- false

      for each vertex j in the graph:
         if weight of the edge between j and current is less than infinity and
smallestWeight[current] is equal to smallestWeight[j] + weight of the edge between j and
current:
            set current as j
            push current onto pathStack
            set pathFound as true
            break

      if pathFound is false:
         break
  return pathStack
```

Shreya Wagley wagleyshreya@gmail.com

Overview:

The `shortestPath` function calculates the shortest path from a given `origin` vertex to a `destination` vertex in a weighted graph. It uses Dijkstra's algorithm to find the shortest path by iteratively updating the smallest weights between vertices. The function initializes the `smallestWeight` vector with the weights of the edges connecting the `origin` vertex to other vertices. It then determines the vertex with the smallest weight, marks it as visited, and updates the `smallestWeight` vector accordingly. This process continues until all vertices have been visited. Next, the function builds the shortest path by backtracking from the `destination` vertex to the `origin` vertex based on the computed smallest weights. Finally, it returns a stack (`pathStack`) containing the vertices of the shortest path from `origin` to `destination`.