Complexity analysis report

Introduction

The program is implemented to address the problem of limited availability of the EV charging stations by allowing the user to utilise the functionality of the program to perform various tasks based on user's requirements. As the EV charging stations are limited, the program can locate the nearest charging station from user's current location as well as cheapest route to any charging station. The implementation of finding the cheapest path between the two locations uses Dijkstra's algorithm to determine the shortest path or most efficient charging station along the path. The program is broken down into multiple functions for user to select.

Data Structures:

STL map: is used to store the data of the locations. The key are the indexes of all locations, where the index increments as each location being inserted into the map from the file. The data of each key will be an object of class Location which contains the information of the location e.g., name, price, and charger status.

STL list:

- Used for the graph to store adjacent locations of the location as linked list of integers in a dynamic array.
- Storing adjacent location list in EVCharging class.

Dynamic array of pointer to double typed value for 2-dimensional arrays to store the data of weights for each vertex (Location) to its adjacent locations or distances between locations.

Priority queue: used throughout the program comparing different class objects of Location, costs, distances through the operator overloading of each class.

- Class Location: the priority is the lowest charging price.
- Class Costs: the priority is the lowest total cost.
- Class Distances: the priority is the lowest distant.

STL stack: is used to store the indexes of shortest path between the two locations.

STL queue: is used to store the visited indexes for task 9.

The stack and queue are only used to store the travelling path between the two locations.

Algorithms:

Task 7:

- Get user input location.
- Get the return of the distances of current location to all other locations from shortestPath function in Class weightedGraph and store in distances double pointer variable.
- Declare/initialise priority queue called sortedCost.
- Iterate: For (int i = 0; i < weightedGraph->getSize(); i++)
    - If index i location has charging station and not the current location
        - Declare class Cost as c.
        - Calculate the total cost from current location to other locations.
        - Push index, distance, and cost of this "c" into sortedCost.
- Print cheapest station details from the top of sortedCost priority queue.

Task 8:

- Get current and destination location.
- Declare an integer variable called charge_station to store the index of charging station.
- Call CheapestMidPointStation function and store the return of index in charge_station.
    - CheapestMidPointStation function
        - Call shortestPath function in class weightGraph to the distances for the current location.
        - Call shortestPath function in class weightGraph to the distances for the destination location.
        - Store them in separate variables.
        - For (int i = 0; i < weightedGraph->getSize(); i++)
            - If charge amount < 25 and the location has charging station
                - Calculate the total cost from current and destination location to this index.
                - Push into sortedCost.
            - If charge amount > 25 and location price > 0 (not free)
                - Calculate the total cost from current and destination location to this index.
                - Push into sortedCost.

        Cost is passed by reference, updated with the top of sortedCost pq.

        Return sortedCost.top().index.

- Print details of charge amount, locations, charge location, and path.
- For the path,
    - If charge_location is not current location (otherwise, continue as current = charge location)
        - Store the return from overridden shortestPath(current, charge location) function in a stack.
    - After that, do the same to get the path from charge location to destination.
    - Print path, then total cost.

Task 8 and 9, uses overridden shortedPath function in class weightedGraph:

Algorithm for shortestPath(vertex, dest):

- Declare stack<int> called path.
- Declare vector<int> checkedPath with gSize.
- Declare bool *weightFound for gSize.
- Initialise with iteration of j with gSize, for (int j = 0; j < gSize; j++)
    - smallestWeight[j] = weights[vertex][j];
    - weightFound[j] = false;
    - checkedPath[j] = INT_MAX;
- Set weightFound[vertex] to true and smallestWeight[vertex] to 0, as this is the starting index for path.
- For (int j = 0; j < gSize; j++)
    - Declare double minWeight = DBL_MAX;
    - Declare int v = INT_MAX;
    - For first inner loop, as smallestWeight for adj locations of vertex has been assigned previously. Adj indexes of vertex are not DBL_MAX, but actual weight.
        - If weight not found in 'j' and smallestWeight < minWeight.
            - v = j; (v is the index shortest distant)
            - update minWeight = smallestWeight[j];
        - Check if v == INT_MAX or v == dest
            - Break;
              (This checks if v has gone through all indexes (found all) or reached the require destination.)
        - weightFound[v] = true; (Set v index to true as shortest path. (wont be checked again))
    - For second inner loop,
        - If index is not found and minWeight + weight[v][j] < index's current smallestweight.
            - Update smallestweight
            - Assign checkedPath[index] = v (Mark v is the path)

      (this updates the smallestWeight of adj indexes of v so they are no longer DBL_MAX and assign v to checkedPath[index] for the purpose of getting the path indexes.)
- if the path exists
    - declare currentVertex = dest.
    - While (currentVertex is not vertex and currentVertex is not INT_MAX(meaning its not part of the path))
        - If currentVertex is not dest (don't want dest to be included in the stack)
            - Push the index into path stack
            - currentVertex = checkedPath[currentVertex];
    - Then push the vertex last.
- Return path (the stack top is the vertex which is the current location)

Complexity analysis:

M = size of graph

N = number of locations

Task 7:

ShortestPath = O(m^2)

First loop = O(n)

While loop and priority queue operation is mostly at constant time as only getting the top element = O(1), and O(log n) for insertion.

The dominant factor in the time complexity is calling the shortest path.

The overall worse-case complexity would be O(m^2) + O(n), we can consider it as O(n^2) as n^2 is at higher order than n.

Task 8:

cheapestMidPointStation function = O(m^2 + m) = O(n^2)

Calling shortestPath = O(m^2)

The overall worse-case complexity would be O(m^2 + n^2) = O(n^2)

Overridden shortestPath(int vertex, int dest):

The dominant factors are the two nested loops:

- the outer loop iterating gSize - 1 times
- the inner loop iterating gSize times.

the while loop is O(n), where n is the size of checkedPath.

Therefore, the worse-case complexity of the function = O(m^2), where m represents the size of the graph. O(n^2)

Conclusion:

The program is implemented for tasks up to 9. Many data structures were used to optimise their functionalities to organising and accessing the data. Priority queues were used to compare the class objects to determine the lowest cost, the lowest travel distance, and the lowest charging price each location. Task 9 is the most complex and challenging where the main condition to determine the operations is the charge amount. The charge amount less than 25 will have the operations similarly to Task 8. When the charge amount is more than 25, another set of operations will be followed, where the cheapest charging station most likely will be the first to charge 25kWh, charge the rest at different location. This condition is checked using while loop until the charge amount is less than 0, then find the path to the destination.