EV Charging Station Route Optimization Application

Sharubaa Kirubakaran

Aysha Bilal

Caroline Nkan

Ontario Tech University

Data Structures and Algorithms

Md Al Maruf

August 4th, 2024

Project Report:

→ Include introduction, methodology, application details, and conclusion.

To reduce global carbon emissions it is important to have more electric vehicles being used over any other types of cars. For that to happen it is crucial for us to make a sustainable algorithmic solution of route planning for EV users. This led to the main goal of the project to create an efficient application that is able to compute the shortest path to each charging station while also showing the best route to take based on the most important factor, distance. Nodes A through W represent the network, whereas nodes H, K, Q, and T are home to the four charging stations.

Dijkstra's algorithm was then applied into the code to find the shortest path from a given starting point to the nearest charging station, Heuristic definition which are estimates of a distance from a node to the goal (charging station), helping prioritize the nodes closer to the goal, graphing nodes that represent intersections or points, edges that represent the distances and charging station on specific nodes.

To begin the code, the “heapq” module was first imported that enabled the use of the priority queue which makes sure that the necessary nodes are being used to manage and determine the shortest path used. A class called “graph” was then assigned to serve as an empty dictionary that represents the adjacency list of the graph. To continue, methods such as add_edge and full_network_data are added under class graph. The add_edge method adds a directed edge from and to the nodes with a specified weight to it. On the other hand, the full_network_data method adds a defined set of nodes and weighted edges. This creates connections and weights between every node. In addition, the dijkstra method is where the algorithm comes to play. This finds the shortest path from the start node to all the other nodes. In addition, heuristic definitions, which are estimates of the distance from node to the goal were incorporated to help prioritize nodes closer to the goal. The recommend command takes the result and returns the answer which does both recommend efficient route and station. At the end of the code there is graph, charging_station, start_node and the graph.recommend_efficient_route which are all used for the code to run and test.

Then we have the code output:

```
Charging station Q: Distance = 19, Path = ['W', 'R', 'Q']
Charging station T: Distance = 21, Path = ['W', 'V', 'U', 'T']
Charging station H: Distance = 48, Path = ['W', 'R', 'N', 'M', 'I', 'H']
Charging station K: Distance = 38, Path = ['W', 'V', 'U', 'P', 'L', 'K']

Recommended Charging Station:
Charging station Q: Distance = 19, Path = ['W', 'R', 'Q']
```

Out[1]: ('Q', ['W', 'R', 'Q'])

After analyzing the path the result will tally up how far each station is and say which station is closest to the designated area including the path that it has taken for that particular path.

This application is highly applicable in many route sharing scenarios. Mainly in navigation systems, logistics and any system that requires a path and a destination while keeping track of distance. Having an efficient route algorithm This application demonstrates the effectiveness of Dijkstra's algorithm in optimizing route planning for EV users by finding the shortest paths in a graph representing a road network with charging stations. This approach ensures that EV users can reach the nearest charging station in the most efficient manner.

Within a network of 23 nodes, this project successfully constructs an application that determines the quickest routes to each of the four charging stations from a specified starting point. Using efficient algorithms, the application creates a graph, converts network data into a structured format, finds the shortest paths, and suggests the most efficient routes while taking distance into account. The output makes it easy for users to choose their charging strategies by clearly displaying the suggested routes and the distances between them.

Code Implementation:

→ Submit a GitHub repository with the complete source code.

https://github.com/ayshabilai/final-algorithms-project/blob/main/final

Presentation:

→ Create a 2-3 minute video presentation.

https://www.canva.com/design/DAGM6N3ZpTo/IpiVnbIiXmY9iwXpe_aEEw/edit?utm_content=DAGM6N3ZpTo&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Efficiency Analysis:

→ Analyze the algorithm's performance.

A well-known technique for determining the shortest pathways between nodes in a graph—which could, for instance, represent a road network—is Dijkstra's Algorithm. Below is an examination of its effectiveness in terms of both space and time complexity and contrast it with other routing techniques.

Dijkstra’s algorithm

- This algorithm uses a priority queue (min-heap) to extract the node with the smallest distance.

Time complexity:

- Creating the priority queue: O(1)
- Initializing the visited and path dictionaries: O(V), where V is the number of vertices
- Extracting the minimum element from the priority queue: O(logV) per operation, meaning a total of O(VlogV)

→ the overall time complexity of Dijkstra’s algorithm is O((V+E)logV)

Space complexity:

- Adjacency list: O(V+E), E is the number of edges
- Priority queue: O(V), hold all vertices
- Visited dictionary: O(V), keeps track of visited nodes
- Paths dictionary: O(V), stores the shortest path to each node

→ the overall space complexity is O(V+E)

Comparison with other algorithms:

- A search algorithm: this algorithm can be more efficient than Dijkstra's algorithm if a good heuristic is there. Its time complexity will be similar to the Dijkstras but will be faster in practice with the correct heuristic.