EV Charging Station Route Optimization Application

Adam Tam - 100868600

March 26, 2024

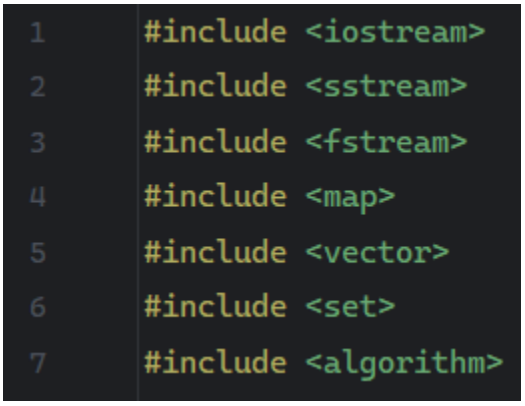
Introduction

This project aims to find, given a node in a graph, the shortest distance and route to the nearest EV charging station.

Program Outline

Libraries

First let's take a look at the **libraries** used to create the program:



```
#include <iostream>
#include <sstream>
#include <fstream>
#include <map>
#include <vector>
#include <set>
#include <algorithm>
```

`iostream`: Used to print information about the graph and algorithms to the console

`sstream`: Used to split each extracted .csv line into tokens separated by commas to generate the graph

`fstream`: Used to read the .csv file and iterate over each line in the file

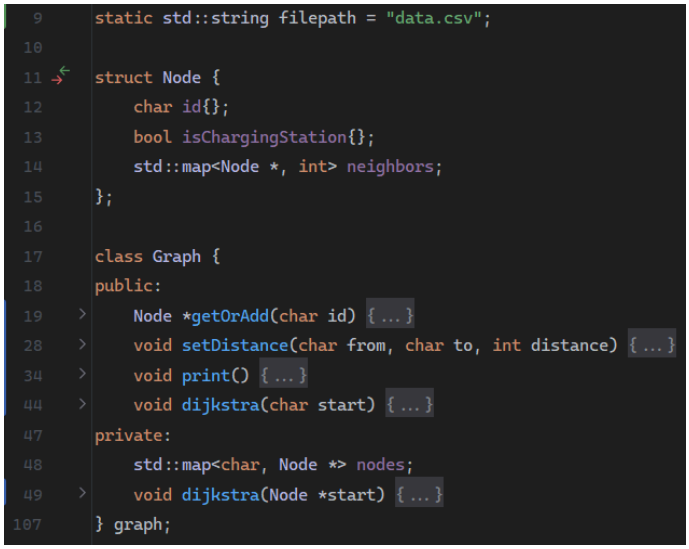
`map`: Used to create hashmaps/dictionaries for quick retrieval of nodes in the graph

`vector`: Used to create lists that don't need quick retrieval and instead simple iteration tasks such as showing the user the distance and path to every charging station

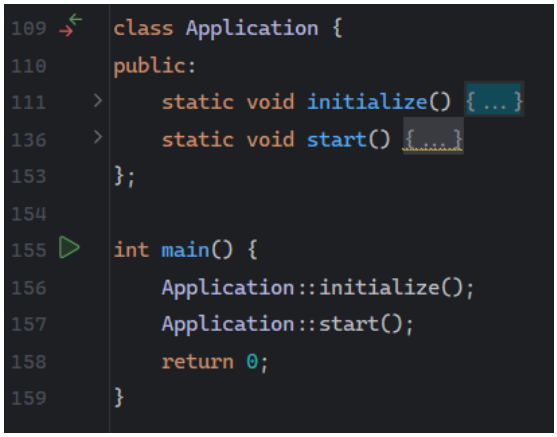
`set`: Used to create automatically sorted lists for the priority queue

`algorithm`: Used to sort the paths by shortest distance at the end of the pathfinding

Fields and Functions



```
9   static std::string filepath = "data.csv";
10
11  struct Node {
12      char id{};
13      bool isChargingStation{};
14      std::map<Node *, int> neighbors;
15  };
16
17  class Graph {
18  public:
19      Node *getOrAdd(char id) { ... }
28      void setDistance(char from, char to, int distance) { ... }
34      void print() { ... }
44      void dijkstra(char start) { ... }
47  private:
48      std::map<char, Node *> nodes;
49      void dijkstra(Node *start) { ... }
107 } graph;
```



```
109 class Application {
110 public:
111     static void initialize() { ... }
136     static void start() { ... }
153 };
154
155 int main() {
156     Application::initialize();
157     Application::start();
158     return 0;
159 }
```

Globals

`(string)filepath`: A global identifier for where the data/file is stored. Although this variable was only referenced once, it's very clear where the file should be stored.

Node

`(char)id`: Identifies the node/locations by name. This was used to quickly retrieve a Node by char lookup.

`(bool)isChargingStation`: A boolean identifying if a node/location is a charging station

`(map<Node,int>)neighbors`: Hashmap of neighbors with the signature **<Neighbor node, distance>**

Graph

`(Node* func)getOrAdd(char id)`: A helpful function used to retrieve a node by id, or create one if it doesn't exist.

`(void func)setDistance(char from, char to, int distance)`: Given two ids of nodes, set the distance between them. This sets both nodes as neighbors of each other to respect the undirected graph structure

`(void func)print()`: Prints each node along with its neighbors and distances and if it's a charging station

`(void func)dijkstra(char/Node* start)`: Given a char or a Node, compute the shortest distance to each charging station. This will also print the distances and paths necessary.

`(map<char,Node*>)nodes`: List of all nodes with signature **<id, Node>**

`(Graph)graph`: Global instance of a Graph

Application

`(void func)initialize()`: Reads the .csv file, parses the data, and loads the data into the graph

`(void func)start()`: The interface for the user to use the program. Has inputs “exit” to exit, “print” to print the graph, and (char) to find the distance to the closest charging station by id. Also has protection measures in case the input is invalid, or the id isn’t a node/location

Program

Let's go through every step of the program.

```
int main() {
    Application::initialize();
    Application::start();
    return 0;
}
```

When the program starts, the program runs the Application.Initialize function

```
static void initialize() {
    std::ifstream file( s: filepath);
    std::string line;

    if (!file.is_open()) {
        std::cerr << "File not found\n";
        std::ofstream newFile( s: "PUT data.csv HERE.txt");
        exit(1);
    }
```

The first step is to open the file and create a variable to store a line and parse it. If the file is not found, we'll exit early and the program will end. However, the program will create a new file to tell the user where to place the data.csv file.

```
while (std::getline( &: file,  &: line)) {
    std::istringstream iss( str: line);
    std::vector<std::string> tokens;
    std::string token;
    while (std::getline( &: iss,  &: token,  delim: ',')) {
        tokens.emplace_back(token);
    }
    Node *from = graph.getOrAdd( id: tokens[0][0]);
    from->isChargingStation = tokens[1][0] == 'Y';
    for (int i = 2; i < tokens.size(); i++) {
        char to = tokens[i][0];
        int distance = std::stoi( str: tokens[i].substr( pos: 1));
        graph.setDistance( from: from->id, to, distance);
    }
}

file.close();
```

Now, we'll iterate over every line and convert every line into graph and node data. Before we continue, here's how each row of the .csv is written:

`ID(char),isChargingStation(Y/N),params NeighborIDDistance`

That may look a little confusing, so I'll work with an example. The first line of the .csv is:

`A,N,B16,F5`

The first entry, **A**, is the id. **N** means that the node is not a charging station, and **B16** says that from A to B, the distance is 16. With **F5**, it means from A to F, the distance is 5.

```
while (std::getline( &: file,  &: line)) {
```

Going back to the code, let's go through it step by step. we'll start with a while loop iterating over every line in the file

```
std::istringstream iss( str: line);
std::vector<std::string> tokens;
std::string token;
while (std::getline( &: iss,  &: token,  delim: ',')) {
    tokens.emplace_back(token);
}
```

Now, we'll create a **`stringstream`** which we use with the **`getline`** function in the while loop to extract every token in the .csv into a list of tokens which we store in a **`vector`**. The reason I used **`emplace_back`** instead of **`push_back`** is only because CLion (the IDE I use), said it's safer than **`push_back`**, though it would work the same regardless.

```
Node *from = graph.getOrAdd( id: tokens[0][0]);
```

Now we'll take the first token, which in the first row of the .csv, **`A,N,B16,F5`**, is A, and pass it into the graph's getOrAdd method. Note that two indexers were used (**`[0][0]`**). This is because the first one gets the first token as a string, and the second one turns it into a char,which is what the getOrAdd method takes.

```
Node *getOrAdd(char id) {
    if (nodes.find( x: id) == nodes.end()) {
        Node *node = new Node();
        node->id = id;
        nodes[id] = node;
        return node;
    }
    return nodes[id];
}
```

(From Graph)

This is where the importance of the getOrAdd method comes into play. If you take a look at the third token, **`B16`**, B hasn't been initialized yet, and if we were to just create a new one, we'd end up creating two nodes of B. As the name suggests, a new node is only created if the node doesn't exist already in the graph.

```
from->isChargingStation = tokens[1][0] == 'Y';
```

Now, we'll assess if the char value of the second token is Y. If it is, we know it's a charging station. Otherwise, it's not.

```
for (int i = 2; i < tokens.size(); i++) {
    char to = tokens[i][0];
    int distance = std::stoi( str: tokens[i].substr( pos: 1));
    graph.setDistance( from: from->id, to, distance);
}
```

Since there may be more than one neighbor for a given node, we'll iterate over the 3rd index (**`[2]`**) to the end of the tokens to ensure we parse every neighbor.
We already know that the current node is the first token, so we grab the id of the second node, which is the first character of the token (remember, the structure looks like: **`B16`**).

We'll split the string by the first character, which in this case, removes the **B** from **B16**, and turn the remainder into an int by using **`stoi`**.
Finally, we'll set the distance of both the nodes.

```
void setDistance(char from, char to, int distance) {
    Node *fromNode = getOrAdd( id: from);
    Node *toNode = getOrAdd( id: to);
    fromNode->neighbors[toNode] = distance;
    toNode->neighbors[fromNode] = distance;
}
```

(From Graph)

This function is very simple. It uses the getOrAdd method to get or add the from and to nodes, and make them each other's neighbor with the distance. If you remember, this is to respect the undirected graph structure. This way, we don't have to write the neighbors twice in the .csv, which is prone to human error.

```
file.close();
```

To finish up the Application's initialization phase, we'll simply close the file.

```
static void start() {
    std::cout << "Welcome to the charging station finder!\n"
                 "Please enter the node you are at (A-W)\n"
                 "Type 'exit' to quit, or 'print' to view the map:\n";
    std::string input;
    while (std::getline( &: std::cin,  &: input)) {
        if (input == "exit") {
            break;
        } else if (input == "print") {
            graph.print();
        } else if (input.size() == 1) {
            graph.dijkstra( start: toupper( c: input[0]));
        } else {
            std::cerr << "Invalid input\n";
        }
    }
}
```

Now' let's take a look at the start function, which is pretty simple.
We start off telling the user about how to use the program, and we create a string to hold the user input.
We then use **`getline`** to grab the user's input for processing. We don't use cin simply because **`cin`** deals with blank characters (' ') in ways that can make the program behave weirdly in my testing.
If the user enters "exit", the program quits
If the user presses print,the graph will print:

```
void print() {
    for (auto &pair : pair<...> & : nodes) {
        std::cout << "Node " << pair.first << " is "
                  << (pair.second->isChargingStation ? "a charging station" : "not a charging station")
                  << '\n';
        for (auto &pair2 : pair<Node *const, int> & : pair.second->neighbors) {
            std::cout << "  -> " << pair2.first->id << " with distance " << pair2.second << '\n';
        }
    }
}
```

(From Graph)

We'll go through every node in the graph (which in this case is a **`map<char,Node*>`**), and print whether it's a charging station, then we'll go each node's neighbors (which is a **`map<Node*, int>`**, I know, the order is reversed but the readability is exactly the same so I won't change it 😅), and print each one along with the distance to each neighbor.

```
        } else if (input.size() == 1) {
            graph.dijkstra( start: toupper( C: input[0]));
        } else {
            std::cerr << "Invalid input\n";
        }
    }
}
```

Going back to the original code, if the user presses a valid char,which is assessed by seeing if the input string has a length of 1. But what if the user types an id such as 'Z', but it doesn't exist? Great question. That's handled inside the dijkstra function, which the char is passed into, and which we'll cover in the next section of this report.
Lastly, if the input string doesn't match any condition, the program simply errors.

In all cases except for "exit", the start() method will repeat over and over again.

Dijkstra's Algorithm

Dijkstra's Algorithm was the algorithm chosen for this project. Let's take a look at every step of the algorithm

```
void dijkstra(char start) {
    if (nodes.find( x: start) == nodes.end()) {
        std::cerr << "Node " << start << " does not exist\n";
        return;
    }
    dijkstra( start: nodes[start]);
}
```

We start off with a protection measure to ensure that the node actually does exist in the graph. We make use of the map's quick lookup feature to assess this quickly. If it passes, we run the same method, but with a **`Node*`** signature instead of a **`char`**.

```
void dijkstra(Node *start) {
    if (nodes[start->id]->neighbors.empty()) {
        std::cerr << "Node " << start->id << " has no neighbors\n";
        return;
    }
```

But why are there two versions of the function? Simply, it's safer. Way safer. If you look back into the **Program Outline**, the **`Node*`** version is private, which means the only way to interact with the Graph publicly is with an id. This is so that you don't pass your own Node into the function, which would create very unexpected outcomes.
We do a second check before we continue - if the node has no neighbors, we know there's no way we're ever finding a path, so we exit early there, too.

```
std::cout << "Finding closest charging station from node " << start->id << " ...\n";

const int INF = 2147483647;
std::map<Node *, int> distances;
std::map<Node *, Node *> previous;
std::set<std::pair<int, Node *>> priorityQueue;
```

Next, we do a simple print to confirm with the user that their inquiry is being processed.
We then create a few variables:
`INF` stores the max signed int value, the equivalent for infinity, which we default to for every distance.
`distances` is a **`map<Node*,int>`** that stores the distance to every other node. This gets set to infinity at the beginning, but gets updated to the lowest distance possible once the algorithm starts searching
`previous` is a **`map<Node*,Node*>`** which stores the previous node for each node. This is needed so we can find the most optimal path needed to find our way back to the "home" node

`priorityQueue` is a **`set<pair<int,Node*>>`** which stores the shortest distance from every node. We need a priority queue because we want to keep track and process the node with the smallest distance that from the home node that has not been processed yet.

```
for (auto &pair : pair<...> & : nodes) {
    Node *node = pair.second;
    distances[node] = (node == start ? 0 : INF);
    previous[node] = nullptr;
    priorityQueue.insert( x: { &: distances[node], &: node});
}
```

Now, we'll iterate over every node in nodes (**`map<char,Node*>`**) and do the following:

1. Get a reference to the node (which is the second type in the map)
2. Set all distances of every other node to infinity, but the home node is set to 0 because, well, it's the home node.
3. Set the previous node of every node as null, as we haven't computed a path yet.
4. Insert the distance of each node along with the node itself in the priority queue

Remember, the priority queue automatically sorts, which is why you won't see any sorting of the priority queue in the code.

```
while (!priorityQueue.empty()) {
    Node *u = priorityQueue.begin()->second;
    priorityQueue.erase( position: priorityQueue.begin());
```

We'll now loop over the priority queue until it's empty. In this loop, we'll get a reference to the first item in the priority queue, which is also the smallest index in the queue. We'll call this the current node. We use **`.second`** as **`.first`** refers to the pair's first type, which is an **`int`**. We will then remove the node from the priority queue. But remember, we kept a reference, so we can continue to process it. Because we removed the current node, that means we have marked it as visited.

```
    for (auto &pair : pair<Node *const, int> & : u->neighbors) {
        Node *v = pair.first;
        int alt = distances[u] + pair.second;
        if (alt < distances[v]) {
            priorityQueue.erase( x: { &: distances[v], &: v});
            distances[v] = alt;
            previous[v] = u;
            priorityQueue.insert( x: { &: distances[v], &: v});
        }
    }
```

Next, we iterate over every neighbor of the current node and do the following:

1. Store the neighbor (**`v`**)
2. Store a new distance (**`alt`**) as the sum of the current distance (in the first iteration, will be 0, as the home node has the lowest distance), and the distance to **`v`**

3. If **`alt`** is lower than the current lowest distance from the current node to **`v`** (in the first iteration, will be infinity), update **`v`** in the priority queue with the new distance (**`alt`**), and set the previous node for **`v`** as the current node

```
std::vector<std::pair<int, Node *>> chargingStations;
for (auto &pair : pair<...> & : nodes) {
    Node *node = pair.second;
    if (node->isChargingStation) {
        chargingStations.emplace_back( a: distances[node], b: node);
    }
}
```

When this line of code is computed, the priority queue has been emptied. We'll now create a list of charging stations (**`vector<pair<int,Node*>>`**), which as the name suggests, stores all the charging station, along with the distance to the station. To do this, we loop over all the nodes, and do the following:

1. Store a reference to the node.
2. If the node is a charging station, we'll store it in the list along with the distance. By this point, the smallest distance to the station has already been computed.

```
std::sort( first: chargingStations.begin(), last: chargingStations.end());
```

Finally, we sort it so the program tells the closest station to the user first. We use **`<algorithms>`** to sort it for us.

```
for (auto &pair : pair<int, Node *> & : chargingStations) {
    Node *node = pair.second;
    std::cout << "Distance to charging station " << node->id << ": " << pair.first << '\n';
    std::cout << "Path: ";
    std::string path;
    for (Node *v = node; v != nullptr; v = previous[v]) {
        path += v->id;
    }
    std::reverse( first: path.begin(), last: path.end());
    std::cout << path << '\n';
    std::cout << '\n';
}
```

To finish off the function, we'll loop over every charging station and do the following with every station:

1. Tell the user the name of the station
2. Create a **`string`** that stores the path. We get this path by looping through every node's previous node. Remember that home's previous is **`nullptr`**, so the for loop will eventually end. We use the **`string +`** operator to add on the id to the path

3. We reverse the string using **`<algorithms>`**. We do this because we start off from the charging station and make our way to home. In order to get from home to the charging station, we simply reverse the string
4. Print the path

And we're complete!

Not quite though. Keep scrolling.

Efficiency

Dijkstra's algorithm does indeed find the shortest path to each path, but not in the best way computationally-wise. If you remember, the program had to loop over every node to find the shortest path. Because the algorithm has no sense of how close it is while it's searching, it has to search every possible path.

However, with only 23 nodes, each with a maximum of 2 neighbors, we don't do that many computations compared to if we had, say 300 nodes with 10 neighbors each.

Let's take a look at the complexity:

- When we used priorityQueue.begin(), that's an O(1) operation, which is a great start.
- We then used a priority queue, which means that removal becomes an O(log n) operation, where n represents the number of nodes in the priority queue. This is because when we did a remove operation, the set had to readjust. Sets in C++ are a binary tree.
- For each neighbor of the current node,we had to update the map again, if the distance was small enough, which is another O(log n) operation.

This brings our time complexity to O((V+E) log V), where V is the number of nodes and E is the number of edges in the graph.

What's the better choice, though? Pretty simple: A*. Any implementation would work. Some will also guarantee the shortest path, and some will be more greedy at the cost of speed (cutting off the search once a solution has been found, and relying on the heuristic more). This comes down to how much speed and accuracy the program needs.

A* implementation would be an obvious choice, as it's the better version of Dijkstra's in every way. This is because A* is an **informed version** of Dijkstra's, meaning it has the same capabilities, but it knows which direction is better to search in, which cuts down a tremendous amount of searches, increasing exponentially with the size of the graph

The only downside would be finding a way to implement a heuristic function, as the "classic" A* uses grids, which have a very simple way to determine which direction to search in.

Real-World Applicability

This specific situation - finding the closest charging station - is a very relevant task in the real world. With the rise of electric vehicles, it's becoming pretty common for people to need to find the closest location to charge their vehicles. It's not even specific for EVs.

Modern GPS applications, such as Google Maps, Apple Maps, or Waze, all try to get you to the closest, whatever. That could be a restaurant, gas/EV station, workplace, anywhere. On top of that, they also have to calculate different routes if you're taking the bus, driving, walking, or biking. AND they have to give you the estimated times for everything AS WELL AS keep traffic in mind. This sort of route optimization is very complex and is continuing to get better as technology improves, but this field is one that is very applicable and active today.

Insights & Conclusion

This task helped me grasp a lot about pathfinding using Dijkstra's, as well as learning to modify it to fit my needs. Normally, search algorithms require you to find just one path, but I had to find multiple paths as a result of searching for different charging stations.

Aside from the specifics of the search algorithm, I learned more and more about clean code, and how to structure beautiful, safe, and performant code, which happens as I do more and more projects. In this project specifically, I tried to make the .csv a little foolproof so small human errors are corrected, though it will still error if major issues are fed through.

Additionally, you may have read earlier that I hid the more complex code for Dijkstra's behind a public interface from which you are only allowed to pass in a char. This helped me learn how to write safer code as well.

I also made sure to focus on following the SOLID principles for this project, as I've struggled with it on my past projects, and I think I made a very good improvement from this project.

Overall, this helped me with my programming habits, as well as how to adapt already known algorithms with my needs injected into them.