Fachpraktikum Algorithms on OpenStreetMap Data: eMaps

Tobias Mathony

May 11, 2020

Abstract

Electrically-powered vehicles, such as e-Bikes or e-Cars, play an important role in today's fight against climate change. Contrary to traditional vehicles, such as cars running on gasoline, electrical vehicles have unique characteristics like a limited cruising range, and long recharge times. To ensure that such vehicles never run out of power, adaptions to navigation and route planners are required. This project explains the implementation of a route planner that considers the limited cruising range of electric vehicles, as well as the availability of charging stations in the road network, based on OpenStreetMap data.

1 Introduction

Climate change and its consequences heavily impacted the engineering of alternative means of transport. As a result, vehicles powered by electric batteries emerged, such as electrically-powered cars, bikes or scooters. By using regenerative energy sources, electrically-powered vehicles have the potential to significantly reduce the dependency of fossil fuel reserves [AHLS10]. Furthermore, electric vehicles emit no emissions and are therefore more eco-friendly than vehicles running on gasoline. However, electric vehicles have characteristics that currently hinder its wide-spread adaption, i.e. (i) limited cruising range, and (ii) long recharge times [AHLS10]. Especially the limited cruising range requires an adaption of existing navigation and routing systems. We need to determine routes for electric vehicles considering the availability of charging stations and the range of electric vehicles to avoid running out of power.

Using OpenStreetMap[1] data, we demonstrate the implementation of a route planner that considers the range of an electric vehicle and the availability of charging stations.

2 OpenStreetMap

This project is implemented on OpenStreetMap data. OpenStreetMap (OSM) is a community-driven open-source project that provides user-generated map data [HW08]. Besides that, also further geographical information like charging stations, bars, clubs or restaurants are available. OpenStreetMap data is available in different formats with parsers for the most popular programming languages.

3 Implementation

This section outlines the architecture of the application, the used technologies and the main features of the implementation.

3.1 Architecture and Technologies

For the architecture of the application, the common **Model-View-Controller** paradigm was used for a strict separation of functionality and an increased re-usability of the components as depicted in Figure 1.

[1] `https://www.openstreetmap.org`

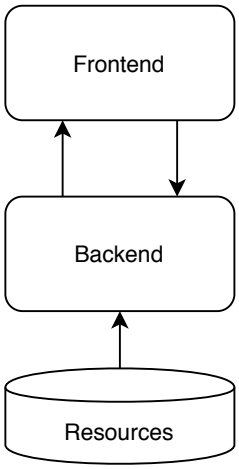

Figure 1: Conceptual overview of the application

The **Resources** represent the raw OpenStreetMap data which was provided in the *Protocolbuffer Binary Format (PBF)*[2].

The **Backend** provides the core functionality and implements algorithms on the OpenStreetMap data. It is implemented as Web Server in *Rust*[3]. The core functionality of the Backend includes (i) a parser to map the OpenStreetMap data to a domain model representation in Rust, (ii) a shortest path algorithm, and (iii) an *Application Programming Interface* (API) to expose the functionality for access from the Frontend.

The **Frontend** is built using the *React*[4] Web Framework and *Leaflet*[5]. Leaflet is an open-source JavaScript library for interactive maps based on OpenStreetMap. Leaflet is used to provide a user-friendly map and allows to e.g. set markers or draw routes. Figure 2 displays the Frontend in its initial state. The Frontend interacts with the Backend via *HTTP* requests.

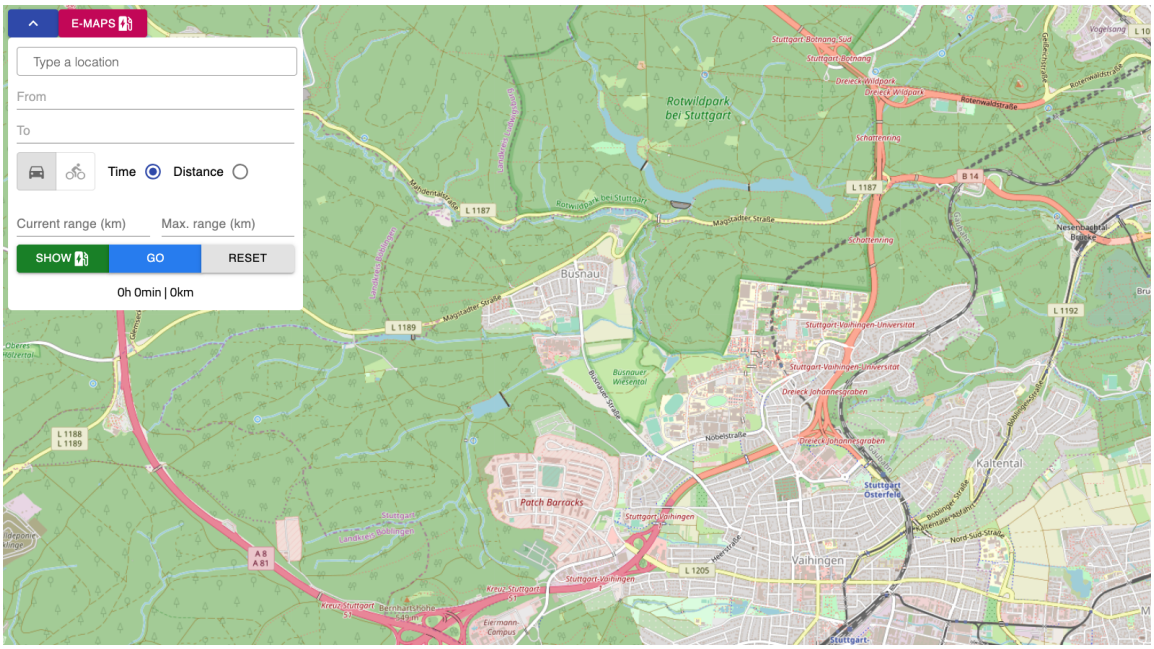

Figure 2: Screenshot of the Frontend

[2] https://wiki.openstreetmap.org/wiki/PBF_Format
[3] https://www.rust-lang.org
[4] https://reactjs.org
[5] https://leafletjs.com

3.2 Graph

The Backend parses the graph data from OpenStreetMap in the PBF format. Only necessary data is parsed from the PBF file for performance and efficiency reasons. We focus on *ways*, *nodes*, and *amenities* in OpenStreetMap. A node in OpenStreetMap is a single geographical data point with latitude and longitude. A way is a combination of multiple nodes building a sequence, e.g. a street. Amenities are arbitrary geographical POIs, e.g. a restaurant, a bar, or a charging station. Amenities are stored as Key-Value-Pairs in OpenStreetMap, e.g. $\{amenity : charging\_station\}$. The implementation of this route planner focuses on electric cars and electric bikes, hence we only parse ways where cars and bikes are allowed to drive on. Furthermore, we parse charging stations from the amenities to be able to consider those later on when calculating a route.

Our graph is structured as follows: $G = \{N, E, O, C, CN\}$ with (i) N being a set of nodes, (ii) E being a set of edges, (iii) O being a offset array, (iv) C being a cell hashmap and CN being a set of nodes that represent a charging station. The offset array is used to efficiently access the edges of a node. The cell hashmap is used to create a grid layer above the graph to speed up the search of the nearest node in the graph for given coordinates. While creating the graph, every node is hashed into a grid cell using the predecimal digits of latitude and longitude of its coordinates.

When the user requests a shortest path calculation from the Frontend, arbitrary start and goal locations may be passed. Since we only parsed ways that are valid for cars and bikes, the start and goal locations may not be present in our graph. Thus, we may need to locate the nearest node in our graph to the start and goal location as passed by the user.

Let us assume we want to locate the nearest node in our graph for a node called n. First, we locate n in the grid by using the predecimal digits of its coordinates. Then, we calculate the distance of all nodes in the cell to n and choose the node with the smallest distance to n. Since the nearest node to n may be in another cell, we also check neighboring cells for the nearest neighbor.

To extract charging stations from OpenStreetMap, we check for each node if a Key-Value-Pair $\{amenity : charging\_station\}$ exists, i.e. if the node is tagged as charging station in OpenStreetMap. Besides that, OpenStreetMap data may also provide information about the vehicles that can be charged at a charging station. However, since OpenStreetMap is community-driven and based on user-generated data, not all charging stations contain information about the vehicles that can be charged. Thus, for charging stations without further information, we just assume that both cars and bikes can be charged. Hence, a charging station in our graph model contains coordinates, an enum indicating the vehicles that can be charged, and an identifier.

3.3 Routing

To route between two nodes in our graph, we use Dijkstra's Shortest Path Algorithm [Dij59]. The user is able to select between routing for shortest time or shortest distance, as visible in Figure 3.

3.4 Routing for Electric Vehicles

The main functionality of this project is a route planner that considers the current and maximum range of electric vehicles, as well as the availability of charging stations to determine a route where an electric vehicle never runs out of battery. Figure 3 depicts the modal where the user can enter the current and maximum range of the electric vehicle used for travelling. Once a start and goal location is chosen, the user can submit the request for route planning including the parameters of current and maximum range to the Backend. The pseudo code depicted in Algorithm 1 simplifies the processing of the Backend once receiving such a request.

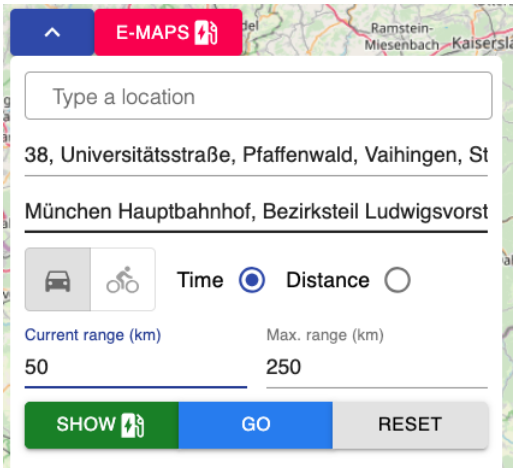

Figure 3: Modal for the user to e.g. enter current and maximum range

Algorithm 1: Simplified route calculation

Data: start coordinates, goal coordinates, current range, maximum range
Result: route

```
1  naive route ← shortest path(start coordinates, goal coordinates);
2  required range ← route.distance;
3  while required range > current range do
4  |   charging station coordinates ← find optimal charging station coordinates;
5  |   route to charging ← shortest path(start coordinates, charging station coordinates);
6  |   route ← route + route to charging;
7  |   current range ← maximum range;
8  |   start coordinates ← charging station coordinates;
9  |   route to goal ← shortest path(start coordinates, goal coordinates);
10 |   required range ← route to goal.distance;
11 |   if required range < current range then
12 |   |   route ← route + route to goal;
13 |   |   return route
14 |   end
15 end
16 return naive route
```

At first, we calculate the shortest path based on the start and goal parameters. In line 3, we check if the distance of the shortest path surpasses our current range, i.e. if we need to charge our electric vehicle to travel from start to goal. If the current range is sufficient for the calculated shortest path, we jump to line 16 and return the route to the Frontend. If the current range is not sufficient, we need to visit at least one charging station. Hence, we need to determine a charging station to visit, based on the start, goal, the current and maximum range, as well as our means of transportation (car or bike).

This calculation is depicted in Algorithm 2. After initializing variables we need later on, we start iterating over all charging stations of our graph in line 5. In line 6, we check if the current charging station supports the means of transportation passed by the user. If this is the case, we need to determine whether the current charging station is within our current range, i.e. if the range of our electric vehicle is sufficient to reach the charging station.

Algorithm 2: Simplified charging station coordinates calculation

Data: start coordinates, goal coordinates, current range, transportation mode
Result: chosen charging station

```
1  global min distance sum ← max;
2  chosen charging station ← null;
3  range utilization ← 0.5;
4  haversine distance multiplier ← 1.5;
5  for charging station in charging stations do
6      if charging station.supported contains transportation mode then
7          temp distance from start ← distance(start coordinates, charging station);
8          if temp distance from start × haversine distance multiplier =< current range then
9              if temp distance from start >= current range × range utilization treshold then
10                 temp distance to goal ← distance (goal coordinates, charging station);
11                 temp dist sum = temp distance from start + temp distance to goal;
12                 if temp dist sum < global min distance sum then
13                     global min distance sum ← temp dist sum;
14                     chosen charging station ← charging station;
15                 end
16             end
17         end
18     end
19 end
20 return chosen charging station
```

To do this, we need to calculate the distance from the start to the current charging station. Due to performance and efficiency, we do not calculate this distance via our shortest path algorithm, but with the haversine distance formula [Rob57], which determines the great-circle distance between two points on a sphere. Since the distance for a shortest path between two points on a road network is bigger than the haversine distance, we need to make sure that the current charging station is actually reachable with our current range considering the actual distance of a shortest path on a road network. To ensure this, we use a multiplier of 1.5 as a heuristic for the haversine

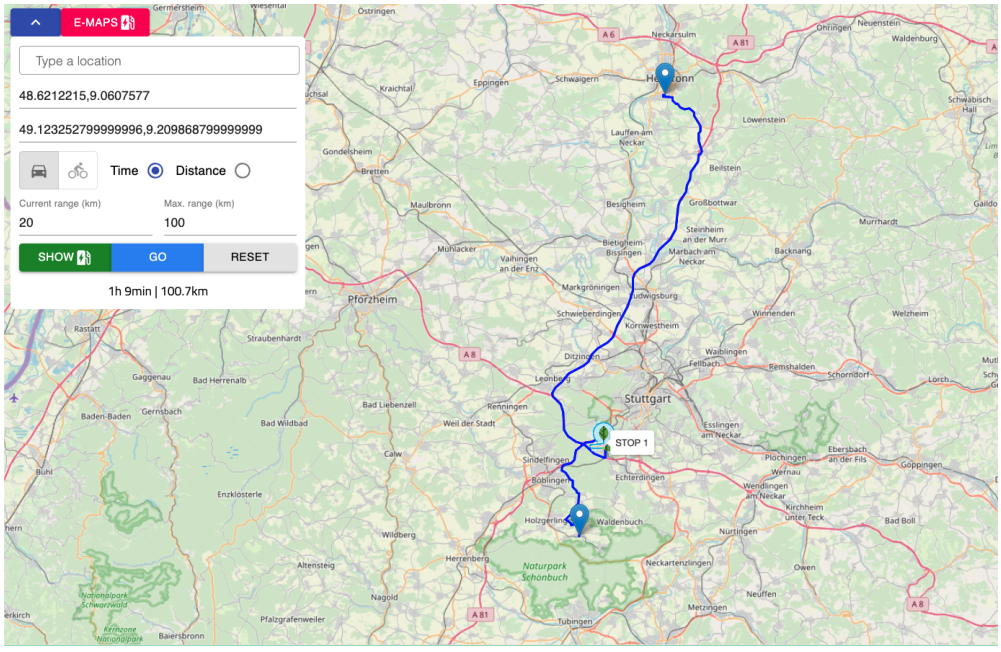

Figure 4: Successfully calculation of route with charging stations

distance between the start and the charging station when comparing it to our current range as seen in line 8. By using this value, we assume that the actual shortest path distance between

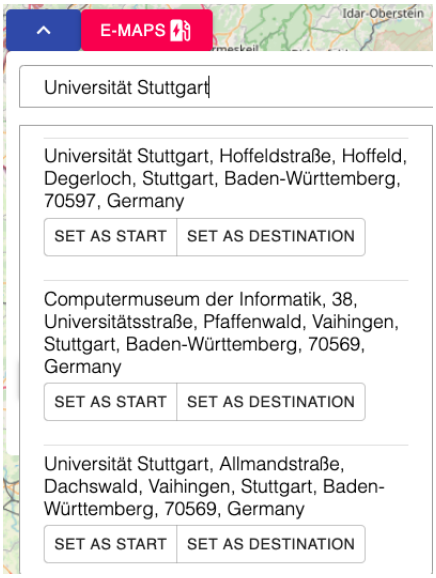

Figure 5: Search via Nominatim

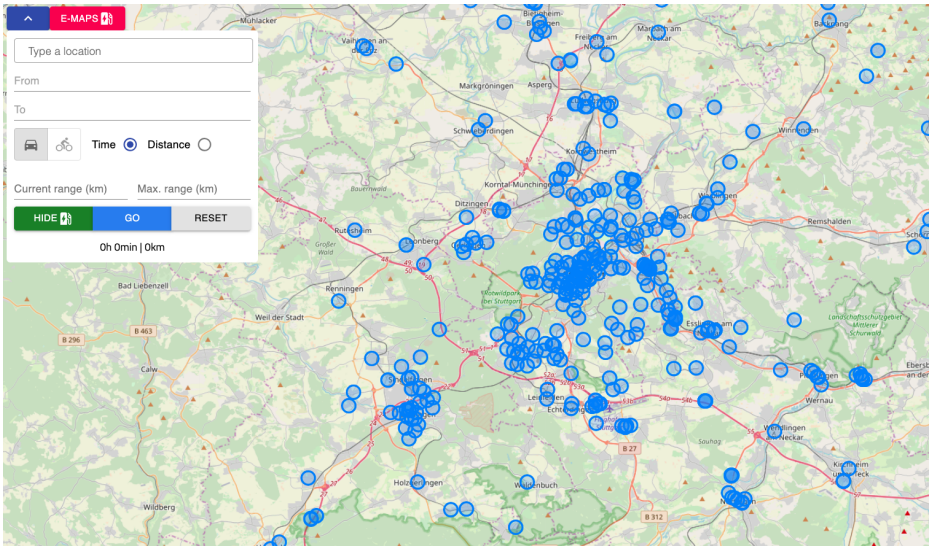

Figure 6: Display all charging stations

two points on our road network is less or equal than 1.5 times the haversine distance. If the distance to the charging station is within our current range, we check in line 9 if at least 50% of our current range is utilized to avoid wasting our range. If at least 50% of our range is utilized, we calculate the haversine distance from the current charging station to the goal. In line 11, we sum up the haversine distances from the start to the current charging station, and from the charging station to the goal, as we want to determine the charging station with the smallest total distance between start, charging station and goal. In line 12, we check if the sum is smaller than the current minimal total distance and, if this is the case, update the current global minimal distance sum and the chosen charging station accordingly.

Once we determined a charging station with Algorithm 2, we jump back to Algorithm 1 and calculate the shortest path from the start location to the chosen charging station location. Then, we set our current range as maximum range, since we assume that we fully charged our electric vehicle, and replace the original start with the charging station as depicted in line 8 and 9. Following, we calculate the shortest path from our new start to the goal location, and update the required range accordingly. In line 11, we check if the required range is less or equal our current range, i.e. if the remaining distance does not require further charging, and if that is the case, we concatenate the

calculated routes to one single route and return it in line 12 and 13. However, if the remaining distance is still bigger than our current range, we jump back to the start of the while loop in line 4 and repeat the procedure until the remaining distance is within our current range.

Figure 4 shows the Frontend with a successfully calculated route with charging stations. The visited charging stations are highlighted for the user.

3.5 Further Features

Another feature of the application is the use of the *Nominatim*[6] API to allow the user to search for places, cities or Point-Of-Interests (POIs) as seen in Figure 5. Furthermore, it is possible to show (or hide) all charging stations that exist, as seen in Figure 6. These charging stations are, upon request, fetched from the backend via a HTTP GET request and rendered.

4 Limitations and Future Work

Since this project was within the scope of the lecture *Fachpraktikum Algorithms on OpenStreetMap Data* the time spent on the concept and implementation was limited. Hence, the route calculation with charging stations may not be optimal, especially determining the charging stations on the route. Furthermore, considering the elevation profile of the road network might help to determine more energy efficient routes for electric vehicles, i.e. routes that go downhill or have a low elevation in general.

For example, Artmeier et al. [AHLS10] proposed an extension to general shortest-path algorithms that address the problem of energy-optimal routing by extending a graph with a weight function representing the energy consumption [AHLS10]. To achieve a better routing for electric vehicles in terms of energy efficiency, a similar approach could be implemented on top of this project.

Worley et al. [WKS12] presented a model that addresses the problem of locating charging stations and calculate routes for electric vehicles based on discrete integer programming optimization and the traditional Vehicle Routing Problem [WKS12] that could be used to optimize the current implementation.

5 Conclusion

This project demonstrated the use and suitability of OpenStreetMap road network and amenity data to perform route planning for electric vehicles. The implemented application is a route planner that considers the limited cruising range of electric vehicles, as well as the availability of charging stations along the route with the primary goal to avoid running out of power.

References

[AHLS10] Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The shortest path problem revisited: Optimal routing for electric vehicles. In Rüdiger Dillmann, Jürgen Beyerer, Uwe D. Hanebeck, and Tanja Schultz, editors, *KI 2010: Advances in Artificial Intelligence*, pages 309–316, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[HW08] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, Oct 2008.

[Rob57] C. C. Robusto. The cosine-haversine formula. *The American Mathematical Monthly*, 64(1):38–40, 1957.

[6] `https://nominatim.openstreetmap.org`

[WKS12] O. Worley, D. Klabjan, and T. M. Sweda. Simultaneous vehicle routing and charging station siting for commercial electric vehicles. In *2012 IEEE International Electric Vehicle Conference*, pages 1–3, March 2012.