

Fachpraktikum Algorithms on OpenStreetMap Data: eMaps

Tobias Mathony

March 16, 2020

Abstract

Electric vehicles, such as eBikes or eCars, play an important role in today's

1 Introduction

Climate change and its consequences heavily impacted the engineering of 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, hence, being 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 require 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 OpenStreetMap¹ 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 [?]. 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.

2.1 Dijkstra

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.

The **Resources** represent the raw OpenStreetMap data which was provided in the *Protocol-buffer Binary Format (PBF)*².

¹<https://www.openstreetmap.org>

²https://wiki.openstreetmap.org/wiki/PBF_Format

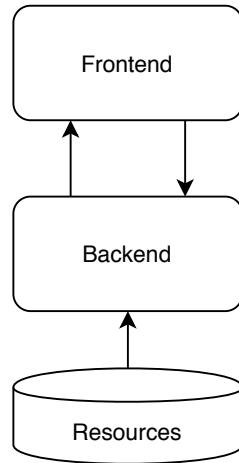


Figure 1: Conceptual overview of the application

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 Object Model* (DOM) 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 shows the basic look of the Frontend. Furthermore, the *Nominatim*⁶ API is used to allow the user to search for places, cities or Point-Of-Interests (POIs). The Frontend interacts with the Backend via *HTTP* requests.

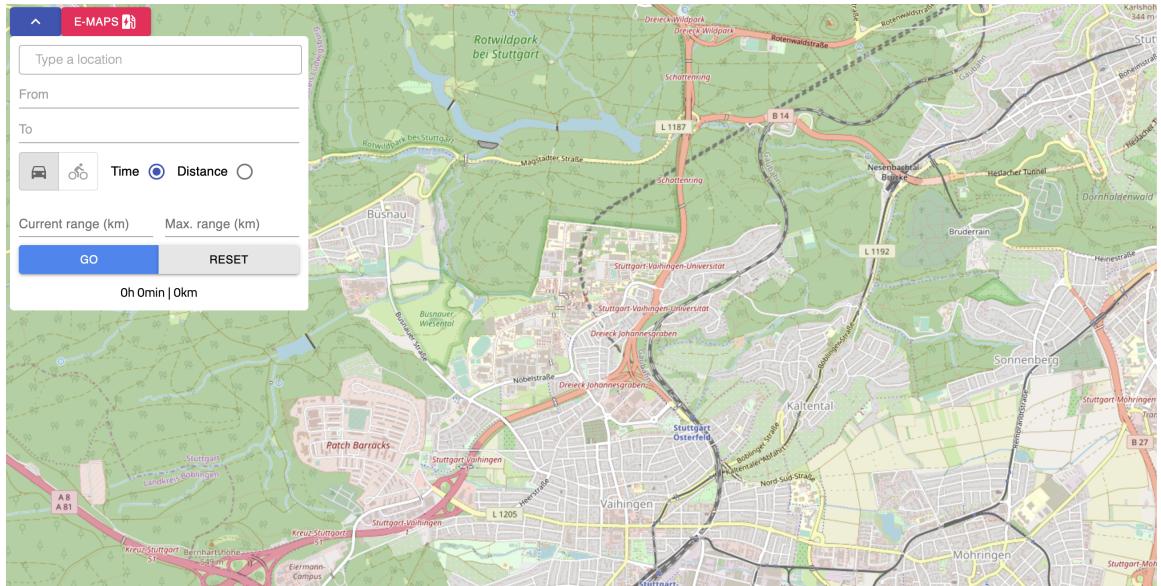


Figure 2: Screenshot of the Frontend

³<https://www.rust-lang.org>

⁴<https://reactjs.org>

⁵<https://leafletjs.com>

⁶<https://nominatim.openstreetmap.org>

3.2 Graph

The Backend parses the graph data from OpenStreetMap in the PBF format. This allows to map the graph data from a PBF file, e.g. the road network graph of Germany, to a DOM representation in Rust. However, 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) W 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 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 target locations may be passed, however since we only parsed ways that are valid for cars and bikes, we may need to locate the nearest node in our graph to the start and target 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 also provides 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 [?]. The user is able to select between routing for shortest time, or shortest distance, as visible in Figure 3. TODO: explain distance/time

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 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

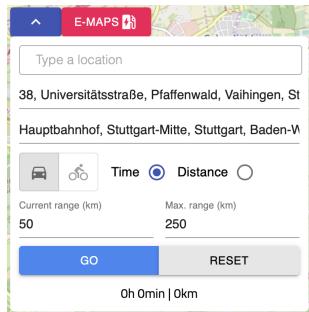


Figure 3: Modal for the user to e.g. enter current and maximum range

maximum range of the electric vehicle used for travelling. Once a start and target location is chosen, the user can submit the request for route planning including the parameters of current and maximum range to the Backend. Initially, in the Backend, a shortest path calculation is started from start to target. The pseudo code depicted in TODO simplifies the processing of the Backend once receiving such a request.

```

Data: start coordinates, target coordinates, current range, maximum range
Result: route
route  $\leftarrow$  shortest path(start coordinates, target coordinates);
required range  $\leftarrow$  route.distance;
while required range  $>$  current range do
    route  $\rightarrow$  null;
    charging station coordinates  $\leftarrow$  find optimal charging station coordinates;
    route to charging  $\leftarrow$  shortest path(start coordinates, charging station coordinates);
    route  $\leftarrow$  route + route to charging;
    current range  $\leftarrow$  maximum range;
    start coordinates  $\leftarrow$  charging station coordinates;
    route to target  $\leftarrow$  shortest path(start coordinates, target coordinates);
    required range  $\leftarrow$  route to target.distance;
    if route to target.distance  $<$  current range then
        | route  $\leftarrow$  full route + route to target;
    end
end
return route

```

Algorithm 1: Simplified route calculation

4 Conclusion and Future Work

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.