Map routing documentation

Kryštof Hrubý

September 20, 2020

1 Introduction

The goal of this project is to create a web application that users could use to plan bike routes. In order to fluently plan routes, the web user interface provides a map editor that can be used browse map. When route endpoints (and midpoints if there are any) are specified, the shortest route is shown in the map editor.

This project serves as a baseline for bachelor project which will greatly improve its routing capabilities.

2 Developer documentation

2.1 Data source

OpenStreetMap map data are used for routing, map rendering and for geocoding. The data is usually downloaded in a standard xml format or a pbf format which is much more memory efficient. Since working with the data in this form would be extremely inefficient, it is preprocessed to suit the program that uses the data and loaded into a database. At the moment, only map data for Czech Republic are used in this project. However, it is possible to use map data of any part of the world for the project if all databases are updated with the new map data. Tile rendering, routing and geocoding parts all require data to be loaded to a database in a specific way so it is necessary to load the data to three different databases/ tables using different tools or different options.

2.2 Architecture overview

This project is a web application which means that it is split into two main parts - **client app** and **backend engine**.

Client application is written in **reactjs**. It consists of a standard map editor that can be used to browse the map or show found bike routes and of a primary panel that handles searching and routing functionality. This part uses a lot of javascript libraries e.g. react-leaflet that are included in the project and do not have to be installed (More about libraries and react implementation later.).

Backend engine uses **Apache** server and has three main purposes - **rendering and serving tiles to the client app**, **routing** and **location search based on user input**.

Rendering and serving tiles is done using **Mod_tile** which is a program for efficient serving and rendering tiles. The advantage of outsourcing **Mod_tile** is that it is very memory efficient and quite fast. It uses **Apache** server and **Mapnik** library.

Routing should be done fast and that is why its core is implemented in C++ and merely handling of http requests is implemented in python. C++ python bindings are provided using **Python.h** library. Python code uses **Flask** and **Flask-RESTPlus** for receiving and sending http requests.

First, it is necessary to parse the osm map data and create a graph of roads. Osm data are usually in xml and pbf formats which are supported by **Libosmium** library which is used for data parsing.

The road graph can then be used to find shortest routes. For performance reasons and because the size of data is enormous, the routing graph is saved in a database table as an list of edges. When request comes in to find the shortest path between two points, a local portion of the graph is loaded into memory and then used for routing purposes. In order to receive local graph in the shortest amount of time it is important to create a geospatial index on the graph. Index speeds up queries

like this since all edges of the local graph are close to each other in the map. Having geometries for each edge in the table also makes it possible to construct the geometry of the shortest route.

Location search based on user input is done using **Nominatim** library. **Nominatim** uses osm data to find locations on Earth by name and address (geocoding). It can also do the reverse, find an address for any location on the planet. It can be also set up with **Apache** server. Map data that Nominatim uses must be loaded to a database by program **osm2pgsql** with gazetteer output option. Unfortunately, this is not yet implemented and **Nominatim** public API is used instead of installing **Nominatim** locally. This will be rectified since **Nominatim** API user policies are not suitable for this project.

2.3 Client-Server interface

Rendering interface is handled entirely within **leaflet** library on the client side and **mod_tile** on the server side. **Leaflet** library only takes a url of the tile server as an argument. The url ends with three variables (x, y, zoom) that specify which tile is requested. The server then sends a tile (e.g. png picture) to the client.

Routing requests from the client side are sent to the server in form of http **post** requests. Each request contains a list of route point coordinates in json. When the shortest route is calculated, the route is sent to the client as a response to the **post** request. The route is a list of road geometries in geojson. The server side is implemented in python in ./api directory.

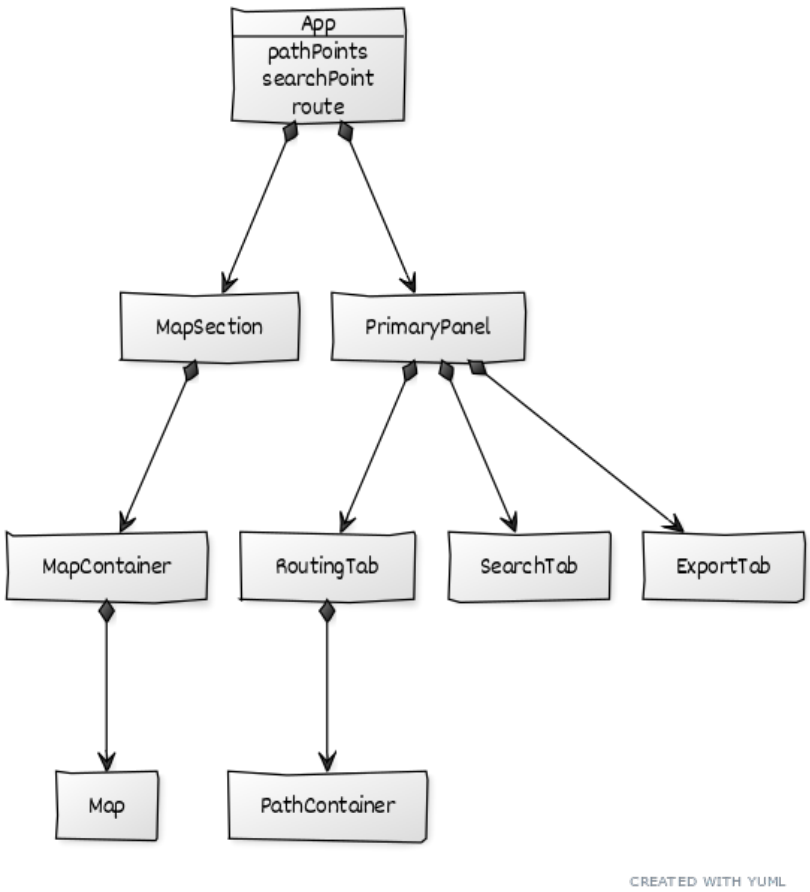

Figure 1: Client app structure

2.4 React app

It is assumed that the reader already knows a bit how the user interface works in this section. If not then please see section **User documentation** or have a look at the web application.

The client app is written in react which means that the code is split into components that contains other components which together creates a tree structure. It also has only one page. Figure 1 shows the structure of the client app. Component App represents the whole application. The main states are also defined here since they are used in both MapSection and PrimaryPanel:

- State pathPoints represents the addresses/names and locations of places selected or not yet selected (empty text boxes) by user.

- State searchPoint represents the address/name and location of place in SearchTab.
- State route represents geojson geometry of found shortest route.

Component Map is a component from react-leaflet library and provides functionality for browsing the map. It only needs an address of a tile server. Component MapContainer creates map Markers for path points or search point based on the values of the aforementioned states and forwards them to component Map which renders them. MapContainer also computes all necessary information about area and zoom level that should be applied to e.g. show all path points in the map editor. Moreover, it forwards route geojson geometry to the component Map.

Component PathContainer consists of a list of Point components and AddPoint components. Component Point represents one route point = one text box with corresponding buttons that can be seen in the routing tab. It knows its position within the pathPoints state and is able to fill in information about itself to the state when user selects a particular place by either typing the name of the place or clicking in the map. That triggers re-rendering of the map.

Component SearchContainer works in a similar way as PathContainer but contains only one point.

Finding addresses and location of places based on user input as well as finding addresses based on specified location in the map is done via **Nominatim API**. The API returns list of potential candidates.

Javascript libraries used in the apps are:

- react-leaflet and leaflet libraries are used to implement map editor functionality such as showing tiles, zooming, etc...
- react-bootstrap library is used for styling components such as buttons.
- From react-select library is used the Select component which provides a select with drop-down menu with options that can be dynamically updated. It is used for all (path) points and the options are potential candidates to select from.

2.5 Tile rendering

As mentioned before, tiles are rendered and served by Apache module **Mod_tile** which uses **Mapnik** library for rendering. Map data must be placed in a mapnik-style database which can be created using tools such as **osm2pgsql**. What map elements should look like is defined in a xml stylesheet (./backend/stylesheet_osm_bright directory). This project also contains a C++ module with python bindings that actually implements rendering tiles using Mapnik. Since it is much slower than Mod_tile its main use is to check if Mapnik is correctly installed and if a xml stylesheet is correctly configured.

2.6 Routing graph generation

In order to perform routing it is necessary to have a road graph. Executable C++ target **graph_builder** is capable of parsing osm data in a pbf file and create a sql script that loads road graph to a table in a database in a form of an edge list. It is a command line tool which takes four arguments:

- osm data file path.
- table name - the generate sql script creates this table and loads the graph into this table.

- sql script output path - the sql script creates a table, loads edge data into the table, adds a column length, calculates lengths for all edges and creates a geospatial index on the table.
- edge data output path - there are two possible writers in the program that generate sql and insert data to a table in a different way. The former is $InsertWriter$ which generates edge data directly to sql script in a form of INSERTs. The latter is $CopyWriter$ which generates COPY command to sql script and writes edge data to this file in csv format. $CopyWriter$ is the default one used since it is much faster!

To understand how to parse osm data it is necessary to know the main types of osm elements - nodes, ways, relations. Nodes represent specific points on the planet defined by longitude and latitude. (e.g. intersections). Ways are list of nodes that define a polyline (e.g. roads). Relations are multi-purpose data structures that document relationships between two or more data elements (nodes, ways, and/or other relations).

Since the size of the map data is enormous, it would be impossible to parse osm elements such as ways using a DOM xml parser (the DOM tree would not fit in memory). It would be necessary to use a SAX parser but fortunately there exists library **Libosmium** which is specialised for reading osm data. It not only supports basic osm xml files but also pbf files which are much smaller. Moreover, it can be selected what type of osm elements should be parsed and what to do with every osm type. This functionality is provided in libosmium by handlers. It is necessary to define a derived custom class of libosmium handler. If this class then defines a function with the name of the osm type, this function is called for each parsed element of the type. Libosmium also provides a way to create geometries that can later be used with PostGIS.

The program needs to read ways, which represent roads, and convert them to graph edges. Ways however can contain road intersections arbitrarily so intersections need to be identified. If all intersections are known it is easy to split them to segments/edges whose middle points are not intersections. Ways that represent roads have tag "highway" not set to null. Class $LinkCounter$, which is derived from libosmium handler class, identifies all nodes that are intersection and saves them to an index. Class $GraphGenerator$, also derived from libosmium handler class, splits ways to segments/edges and by using one of aforementioned Writer classes saves the segment data. It takes as an argument the index that $LinkCounter$ generates. Furthermore, it is capable of creating geometries for the segments.

The program does the following:

- Read osm data once using $LinkCounter$ to identify intersections and save them to index.
- Use $Writer$ class to generate the first part of sql script.
- Read osm data a second time and apply libosmium $LocationHandler$ as well as $GraphGenerator$ handler. $LocationHandler$ makes it possible to get longitude and latitude coordinates through node references in way's list of nodes.
- Use $Writer$ class to generate last part of sql script if there is any.

Afterwards, it is only necessary to execute the output sql script in a database with PostGIS extension enabled.

2.7 Routing itself

Routing is done within the C++ library target **routing**. Its public API is only one function $CalculateShortestPath(graph\_table, from, to)$ which calculates the shortest path from node from

to node to. This function can be also called from python. The bindings of the module are defined in routing_module.cpp.

The library expects that graph edge list is in database called gis in the table whose name is specified in arguments of the main function. Each row in the table must have at least following properties:

- uid - unique id for all edges.
- from_node - osm id of the node where the edge originates.
- to_node - osm ud of the node which is the edge's destionation.
- geog - postgis geography of the edge saved as LINESTRING and with SRID=4326.

The routing algorithm is divided into multiple steps:

1. Load local graph based on locations of route endpoints.
2. Find the closest edge to each endpoint (start, end) and split it into multiple edges with the origin being the point of the edge closest to the endpoint and destination being one of the intersections (=origin/destination) of the closest edge. Moreover, create geometries for these edges and save them. This improves accuracy of routing since the cost is measured directly from the endpoint and not for example from the closest intersection. It also helps render the final route in a nice way.
3. Add edges created in the previous step to the graph.
4. Run routing algorithm on the graph.
5. Construct a list of geometries that represent the found route and return them.

Class $DatabaseHelper$ handles all communication with the database which internally uses library **pqxx** to connect to the database. This prevents other classes to know the detail interface of a library that is currently used to connect to the database. For simplicity, the class also provides function with hard-coded sql queries for all common needs such as retrieving graph from database.

Class $Graph < Vertex,\ Edge >$ represents any routing graph. Since it has two template parameters that define the types of vertices and edges, it is possible to use it with any routing algorithm. It also helps to cope with possible changes of internal graph structure by providing public functions that are the only way to retrieve or add graph data. Retrieving edges from the database and adding them to a graph instance is done through $DatabaseHelper$ class. At the moment, the area of a local graph is computed beforehand and no continuous loading is supported which results in unnecessarily big graphs that consume a lot of memory.

Class $EndpointHandler < EndpointHandlerImplementation >$ provides functionality for creating these new endpoint edges mentioned in step 2. It also stores the corresponding geometries that can be retrieved later. Since different routing algorithms can require different types of nodes, policy pattern is employed to make algorithm changes have smaller impact in the code. It is again necessary to access the graph data in database and compute new edges' information such as lengths and geometries directly in the database. $DatabaseHelper$ provides such functionality.

Class $Algorithm < Implementation >$ provides routing functionality. It also contains a class with a implementation of a concrete routing algorithm. At the moment, only Dijsktra's algorithm is supported but more routing algorithms will crop up in future and it is good to be prepared beforehand for such changes.

Finally, *DatabaseHelper* is used to search for geometries of the found shortest route. Start/end edge geometries are retrieved from *EndpointHandler*. Geometries are then returned in GeoJSON format encapsulated in string.

3 Server installation

This section is a guide how to install all libraries and their dependencies, databases and their extensions and server deployment. Since installing such things differs on most operating systems, this guide is more or less a list of what needs to be installed. The project is developed on Ubuntu.

3.1 Database setup

All parts of the project work with osm map data. Therefore, it is crucial to set up database server along with PostGIS and HStore extensions and data loading tools.

Install postgresql and postgis extension. The project does work seamlessly with Postgresql 10.14 and PostGIS 3.0.2.

```
https://wiki.openstreetmap.org/wiki/PostGIS/Installation
```

Create a database called gis:

```
sudo -u postgres createuser gisuser
sudo -u postgres createdb --encoding=UTF8 --owner=gisuser gis
```

Add necessary extensions:

```
psql --username=postgres --dbname=gis -c "CREATE EXTENSION postgis;"
psql --username=postgres --dbname=gis -c "CREATE EXTENSION postgis\_topology;"
psql --username=postgres --dbname=gis -c "CREATE EXTENSION hstore;"
```

3.2 Load data to database

Map data can be downloaded from:

```
http://download.geofabrik.de/
```

Install osm2pgsql:

```
sudo apt-get install osm2pgsql
```

This tool is used to load data to databases for rendering and geocoding.

Load data to database for rendering:

```
osm2pgsql -c --slim --hstore -G --username postgres --database gis data.osm.pbf
```

Load data to database for geocoding. Use osm2pgsql with gazetteer option.

Data for routing are loaded to database by C++ program graph_builder. It is therefore necessary to build this project first. The project has CMakeLists.txt to build the project. When the project is built, just run:

```
./graph\_builder input.osm.pbf table\_name output.sql output.csv
Then run the output sql in the database where you want to have the routing graph.
```

3.3 C++

Installing mapnik library can be quite troublesome:

`https://github.com/mapnik/mapnik/wiki/Mapnik-Installation`

Install libpqxx:

```
sudo apt-get install -y libpqxx-4.0
```

Install python.h for python bindings:

```
sudo apt-get install python-dev   # for python2.x installs
sudo apt-get install python3-dev  # for python3.x installs
```

Install libosmium library:

`Install from source: https://github.com/osmcode/libosmium`

The project can be build via cmake and make. There is CMakeLists.txt file in the root directory of the project.

3.4 Build tile server

This guide is absolutely great for building a tile server:

`https://switch2osm.org/serving-tiles/manually-building-a-tile-server-20-04-lts/`

The only things that did not work correctly from the guide were installation of mapnik and generating a xml stylesheet. Osm-bright was used to generate xml stylesheet instead of openstreetmap-carto. Have a look at their github page and their guide of how to generate xml stylesheets:

`https://github.com/mapbox/osm-bright`

Or if you plan to use only data for Czech republic, use **carto** to generate a xml stylesheet from directory:

`./backend/stylesheet\_osm\_bright`

3.5 Python

The project has a virtual environment defined so it is not necessary to install any python libraries. Flask and Flask-RESTPlus were used to handle http requests. To run a flask development server:

```
source ./venv/bin/activate # activate virtual environment
yarn start-api # run the development server
```

3.6 Javascript

Used javascript libraries are part of the project. However, it is necessary to install yarn and node.js.

```
sudo apt install nodejs
npm install -g yarn
```

To run a react development server:

```
yarn start
```

4 User documentation

Project provides no public server API so the only thing a user can access is the web html user interface. Figure 2 shows the features that will be supported in the upcoming bachelor project. Features such as adding/ deleting constraints, importing/exporting paths and showing altitude profile are not yet supported.

The user interface is split into two parts - **Map editor** on the left and **Primary panel** on the right (See Figure 3.).

Map editor provides standard map editor functionality. Is is possible to move around the map by pressing left mouse button and moving the cursor. Moving the mouse wheel or pressing the button in the top left corner does map zooming.

Primary panel consists of three tabs - Search, Routing, Export tab(This one is not supported yet.).

4.1 Routing tab

Routing tab lets user choose route endpoint and midpoints. Each point is represented by a text box as can be seen in Figure 3. There are two ways how to specify a route point:

- The former option is to write its partial name/ address to the text box and click on the search button. Drop-down menu appears with options of full addresses of potential places to choose from. When an option is selected the place is selected as a route point and route is updated.
- The latter option is to click on the text box and then click on a place in map. If the place on map (latitude, longitude coordinates) has an address/ name, it is written into the text box which the user had previously clicked on. Then route is updated.

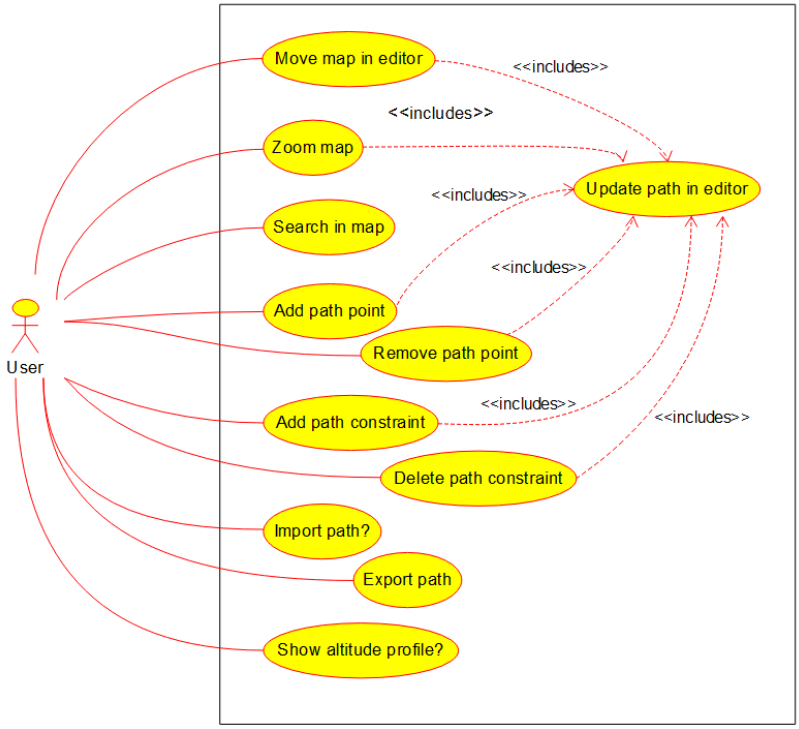

Figure 2: Use case diagram

Additional route point can be added via pressing the plus button below each text box. A text box appears at the position of the plus button - the corresponding route point is in the middle of the adjacent points or is the end of the route if the most down plus button was clicked. When a route point is added, it is possible to just click somewhere in map to select what location it represents (no need to click on the text box).

Route points can be deleted by clicking on corresponding trash container buttons. However, there must be at least two route points and they cannot be deleted.

If route has only one specified point map editor zooms to an area that has the specified point as a center point. If route has two or more than two point each route update triggers shortest route recalculation and map editor zooms to an area that covers all route points.

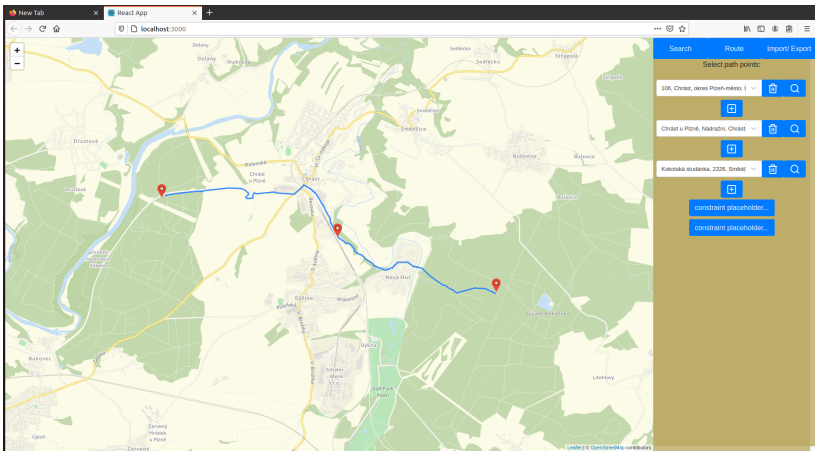

Figure 3: Web user interface

4.2 Search Tab

Search tab provides a way for users to search in the map by writing a partial place's address/ name in a text box. When users click on a search button, which is adjacent to the text box, a drop-down menu appears with list of potential places for user to select from. When a potential place is selected map editor zooms to an area with the place in center.

5 References