FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Kryštof Hrubý

Map routing

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Filip Zavoral, Ph.D.
Study programme: Computer Science
Study branch: Software and Data Engineering

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor RNDr. Filip Zavoral, Ph.D. for his patience, feedback and advice. I am grateful to my family for their tremendous support and encouragement throughout my work on the thesis.

Title: Map routing

Author: Kryštof Hrubý

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: Web route planning applications have become common tools for cyclists. Although there is a relatively large number of such applications, they do not give a user enough options to choose a route according to his or her own preferences. Therefore, the aim of this thesis is to find a way to use known routing algorithms to create more individualized routes. To maintain the demands on the speed of route search, it is advisable to use heuristic algorithms that preprocess a routing graph. Furthermore, it is necessary to preprocess the routing graph for each combination of preferences in order to quickly find an optimal route with respect to entered preferences. As a proof of concept, a web application was created that uses one of the suitable heuristic algorithms (Contraction Hierarchies) to find optimal routes. The application also provides an interactive map editor and map search. The evaluation of the application showed that the proposed concept is functional, i.e., it is possible to find the required route according to the specified preferences in an acceptable time frame.

Keywords: map routing, preferences, large graphs

Contents

1. Introduction

Nowadays, it is quite common to plan bike trips in advance. Bikers are used to download planned trips to a mobile device and view them during biking. There are many applications (e.g. Mapy.cz [1], Google Maps [2], Bikemap [3], Komoot [4], Ride with GPS [5]) that provide route planning for bikers. These applications provide excellent user interfaces with interactive editors and auxiliary features such as an altitude profile. However, they lack support for finding bike routes that would meet biker's specific preferences. They generally find a bike route based on vague predefined routing profiles often called "mountain bike" or "road bike". It is not completely clear what these profiles exactly mean and whether they fulfill user's specific requirements. A user often has to painstakingly adjust routes so that they perfectly fit the user's particular preferences.

Therefore, the goal of this thesis is to build on current routing algorithms and find a way to perform routing based on a custom routing profile. The idea is that a user can select his/her preferences in more areas of interest (e.g. prefer green area, altitude, types of paths, ...) which would define his/her custom profile. As a proof of concept, the thesis also includes a web application with an interactive map editor, options for specification of preferences as well as routing functionality capable of finding routes based on selected preferences. Furthermore, it should possible to export routes so that users could view them on their favourite mobile map applications.

1.1 Text structure

In Chapter 2, we discuss how current routing algorithms can be used to perform routing based on custom profiles. Chapter 3 provides a specification of the application. Implementation alternatives are discussed in Chapter 4. The implementation itself is described in Chapter 5. In Chapter 6, the user interface of the application is illustrated. The application is evaluated in Chapter 7. Conclusion is in Chapter 8.

2. Analysis

2.1 Routing in road graphs

Graphs representing road networks of countries or continents contain enormous number of edges and vertices. While routing in such large graphs can be, in theory, done using standard shortest path algorithms such as Dijkstra's algorithm, routing times for queries where route endpoints are farther from each other are noticeably slow. Such slowdown can be observed for queries whose source and target are merely tens of kilometres apart.

Therefore, heuristic routing algorithms have been developed specifically for large graphs to significantly speed up query time at the cost of having a chance of not finding an optimal path and requiring some sort of preprocessing to be done on a graph in advance. Preprocessing of such algorithms computes some auxiliary data such as additional edges or new properties for edges or vertices. These algorithms typically use the auxiliary data to prune the search space of Dijkstra's algorithm.

2.2 Routing based on preferences

Shortest path algorithms generally assume that each edge contains a length property which denotes its distance. If the meaning of length property is changed, we can use shortest path algorithms to find an optimal path based on the new meaning of length. Then, each preferences only needs to contain an index of values with a value for each graph edge. A weighted collection of such preferences is called (routing) **profile** in the context of the thesis. A profile provides a length value for each edge.

Using Dijkstra's algorithm with a profile is straightforward and it always finds an optimal path based on the profile. However, the situation is not as straightforward when it comes to heuristic algorithms with preprocessing.

Consider that a preprocessing algorithm is run on a graph whose lengths are defined by a profile and some auxiliary data are created. When the corresponding query algorithm is run on the graph with the profile, it provides a standard results: an optimal path based on the profile.

However, if a different profile with the same graph is used with the query algorithm, the algorithm does not have to work properly and instead provide a suboptimal result since the auxiliary data assume different edge lengths. If so, how much a result is suboptimal depends on how the used routing algorithm exactly works and on the difference between the auxiliary data provided by preprocessing with the profile actually used in preprocessing and the profile that defined the lengths in the query.

This leads to two options of how to use profiles (i.e. preferences) with algorithms with preprocessing:

1. Run preprocessing on a road graph for each possible profile. When routing is requested with a given profile, the corresponding auxiliary data must be used in the query. The disadvantage is that preprocessing can be quite a

time expensive operation and all the auxiliary data and road graphs must be stored somewhere. We call this approach **Static Profile mode**.

2. Run preprocessing on only one road graph for one profile. When routing is requested with a given profile, the query algorithm is always run with the same auxiliary data but with the given profile. This approach prefers reducing preprocessing time and memory/disk space consumption at the cost of the possibility of not yielding optimal results. We call this approach **Dynamic Profile mode**.

2.3 Heuristic routing algorithms

A routing algorithm should be fast and give somewhat good results if used in Dynamic Profile mode. We considered the following algorithms:

1. **Contraction Hierarchies** [6] algorithm is based on a concept of vertex contraction. A hierarchy is created iteratively by always contracting the least important vertex. A contraction of a vertex means replacing shortest paths going through the vertex by edges called shortcuts. The query algorithm is modified Bidirectional Dijkstra. The forward search uses only edges leading to more important vertices while the backward uses the edges coming from the more important vertices.

2. **Reach-based** [7] algorithm is based on a concept of reach. Reach of a vertex encodes the lengths of shortest paths which contain the vertex. If a vertex has a high reach, then it is a part of at least one shortest path that is long in both directions from the vertex. The query algorithm uses a modification of Dijkstra's algorithm in which before a vertex is added to a priority queue, it may or may not be inserted to the priority queue. It is not inserted if its reach is smaller than the distance of the shortest path from the origin to the vertex and smaller than a straight-line distance from the vertex to the destination. The preprocessing of the algorithm computes a reach (or rather its upper bound to shorten preprocessing time) of each vertex.

3. **Transit** [8] algorithm is based on a observation that each long shortest path passes through at least one node from a relatively small set of nodes called transit nodes. The set is meant to be as small as possible. The preprocessing of the algorithm finds the set of transit nodes and computes the shortest path between each pair. It also calculates the shortest paths to the closest transit nodes for each node. The query algorithm is not based on Dijkstra if source and target vertices are not close to each other. Instead it is based on a table lookup. It fetches the shortest paths from the source to transit nodes around the source, the shortest paths between transit nodes and the shortest paths from transit nodes around the target to the target. The shortest path is the triplet of the paths with the smallest distance. If the source and the target are close to each other, a different algorithm must be employed.

Since Transit algorithm precomputes shortest paths and optimal path are found by choosing the best combination of the shortest paths, its query algorithm completely ignores if a profile is changed for a query. Therefore, it is unsuitable for our purposes.

As for the other algorithms, it is difficult to estimate how well an algorithm which uses a heuristic to prune Dijkstra's algorithm performs when its query algorithm runs on a graph with a different routing profile than the one used in preprocessing. Therefore, the project aims to test both routing algorithms and their performance.

3. Application specification

The main function of the web application should be to provide routing between route points based on specified preferences (i.e. routing profile).

Based on the analysis (see Chapter 2) the app should satisfy the following requirements:

1. The app should support the concept of preferences and routing profiles. Profiles provide edge lengths for road graphs which are used by routing algorithms. It also should be possible to add custom preferences.

2. The app should implement heuristic routing algorithms (see Section 2.3) and it should be possible to add additional routing algorithms.

3. The app should implement the concepts of Static Profile and Dynamic Profile modes (see Section 2.2).

The application should also include an interactive map editor, an interface for specifying route points and an option for exporting a route. It should be possible to specify route points by clicking on a place in the map editor or by typing a description of a place which should trigger fetching address suggestions based on what was typed.

4. Implementation analysis

Based on the specification, the web application should contain three main functions: **Map editor**, **Geocoding** and **Preference based routing**.

First, we will introduce these functions and their requirements concerning map frameworks. Afterwards, we will take a look at available map frameworks and compare how suitable each framework is for the project. Lastly, we will go in depth through the three main functions and see how they are assembled to create one web application.

4.1 Main functions

4.1.1 Map editor

An interactive map editor should display a map and support common functions such as zooming and dragging. Additionally, the editor needs to show any route that is found. It requires two external inputs. The former are map pictures (i.e. tiles) which make up the map in the map editor. The other is a route that needs to be shown in the map editor.

A map editor is a common functionality; hence, there are many public API servers that provide map tiles which can in turn be used by client-side map libraries to create a map editor. Alternatively, self-hosted tile server can be created instead. How a route can be added to the map depends on where the map tiles are generated. If a map tile is rendered by a self-hosted server, the route can be incorporated in the tile directly. The other option is drawing the route on top of a tile which is a compatible solution with both a public API server or a self-hosted server.

4.1.2 Geocoding

Geocoding means translating a description of a place to a location on Earth and vice versa which is very useful for specifying route points in user interface. It can be used for providing address suggestions when a user types an address or for searching the address of a location selected in the map editor. The functionality is standard in the world of map applications; therefore, there are many public API servers for geocoding. Alternatively, it is possible to create a self-hosted geocoding server.

4.1.3 Preference based routing

Preference based routing aims to find optimal routes based on preferences (see Chapter 2). Routing requires that a road graph can be built from map data. Routing functionality needs to be implemented on the server side since sending even a portion of a road graph to a client to do routing there is far more time expensive than even running slow Dijkstra's algorithm on the server instead and returning an optimal route. Therefore, map data and data structures created from them (e.g. routing graph) must be stored on a server.

As for preferences, each preference contains an index of values for each graph edge so the server must also store the index and possibly the data from which the index was created. Creating such indices can be quite costly so it should be done ideally only once. It is also important to have a direct access to map data in order to create a custom preference index.

4.2 Map framework

The main criteria for choosing a map framework were to satisfy requirements for the three main functions mentioned in Section 4.1, a comprehensive documentation and tutorials, free libraries and a free access to data. The following frameworks were taken into account:

1. **Google Maps Platform** framework [9] seems to be particularly oriented towards providing public APIs and contains a comprehensive documentation and tutorials for working with them. It provides both a library and a public API that can be easily used together to create a fully interactive map editor. Geocoding API is also supported. However, its pricing page [10] states that each API request is priced. While it actually provides a monthly credit for each developer, it is not an open source solution and the usage conditions might change in future. Significantly more burning issue is that it is not clear whether users can download the map data and use them to create entirely new apps. That is a necessary condition for preference routing.

2. **ArcGIS Developer** framework [11] is quite similar to Google Maps Platform in that it is primarily an enterprise level framework and focuses mainly on providing public API. A small number of requests is free each month then it is priced. It provides API for map tiles and geocoding. However, it is again not readily clear how to access the underlying data to create a new functionality or how to extend e.g. a routing library. The documentation only describes how to use the already implemented tools.

3. **OpenStreetMap** framework [12] is a community collaborative project providing map data and number of open source tools and libraries built around it. Its documentation is easily navigable to what the data look like and which tools can be used together for what purpose. Public API of tile servers within the framework are in general for personal use only and should not be used in a product without agreement with tile server providers. However, it is possible to build a self-hosted tile server for free. The situation is similar for geocoding. The public APIs are either not free to use (e.g. **LocationIQ** [13], **OpenCageData** [14]) or explicitly for personal and debug use (e.g. **Nominatim** [15]). Nevertheless, it is possible to build a self-hosted geocoding server by using an open source geocoding tool. The main distinction from the other frameworks is that its documentation [12] describes the format of the data and which tools can be used to work either with OSM data directly or which tools can load the data into a database.

OpenStreetMap framework was selected as it is the only framework which readily supports working directly with its map data which is necessary for preference based routing. It also has a comprehensive documentation. While **Google Maps Platform** or **ArcGIS Developer** frameworks could be used for requesting map tiles or geocoding services through their API, using multiple map frameworks seems to only increase the overall complexity of the whole project. Also, these frameworks are not open source and user conditions for pricing may change in future.

4.3 Map editor

As mentioned before, the map editor should display a map and provide common functions such as zooming and dragging. In addition, it is required that a found optimal route (or rather its geometry) can be shown in the map. There are many JavaScript libraries that create a fully interactive map editor with the only input being map tiles. We considered **Leaflet** [16] and **OpenLayers** [17] libraries. Both libraries support drawing geometries on top of a map and both can be used with standard OSM API for providing map tiles. Both libraries seemed perfectly capable of doing what we wanted from a map editor. We chose Leaflet since it seemed to be simpler to use.

It was discussed in Section 4.2 that there are no public tile servers that provide map tiles for free. Therefore, it is necessary to build a tile server. A tile server requires a map rendering library to generate a map tile which it serves. **Mapnik** [18] is the most popular server-side rendering library. There are a number of available tile server implementations which use Mapnik library. Alternatively, it is possible to implement a simple tile server on top of Mapnik. There are available map stylesheets for Mapnik which determine what each map tile looks like (e.g. what is included, colors, ...) so the server would only need to give Mapnik parameters about the area to render and serve rendered tiles to client.

Available tile servers generally serve tiles based on **Mapnik** styled stylesheets and they are not very customizable without delving into the stylesheets. The map editor must be able to display route geometries. Either a geometry can be drawn on top of a map using Leaflet library or a tile server could provide map tiles which already contain a route geometry. Rendering route geometries on tiles has several disadvantages. First, tiles with a route geometry cannot be cached on server since any route geometry is useful only temporarily and only for one client. Second, more routes can be found on the same location and tiles could be reused if they did not contain route geometries. Lastly, rendering tiles is a somewhat time expensive operation so it should not be done needlessly. Therefore, we chose to use **Leaflet** to draw route geometries on top of a map and there was no need to complicate tile server behaviour with non-standard requirements.

Besides the idea of implementing our own tile server, the following tile server implementations were considered: **TileSweep** [19], **Mod_tile** [20] Apache module, **OSM Scout Server** [21]. OSM Scout Server's documentation is vague about how a tile server is installed and configured. Documentations of both TileSweep and mod_tile include an installation and a configuration overview. Mod_tile [20] is designed for high performance tile serving and is used to serve tiles for the main OSM server. While implementing a simple tile server wrapper on top of

Mapnik is possible, it is a significantly slower solution than using mod_tile since mod_tile also implements a memory efficient tile caching mechanism.

What the map in the map editor looks like can be defined using a Mapnik compatible XML stylesheet. Customizing a stylesheet was out of the scope of this project so only the default stylesheets of **OSM Bright** [22] and **OpenStreetMap Carto** [23] were tested. OSM Carto default stylesheet differentiates between green areas (e.g. between meadows, heathlands, forests, farmland) and path types (e.g. track, bridleway, cycleway, footway) and gives each such map entity a different style which is quite useful for a map for cyclicts. OSM Bright default stylesheet on the other hand does not differentiate between various areas much and instead offers a simple and uncluttered map style. We used OSM Carto.

Note that OSM Carto and OSM bright stylesheets and Mapnik library require that map data are loaded in PostgreSQL [24] database with PostGIS [25] geospatial extension.

4.4 Geocoding

As it was discussed in Section 4.2, there are no suitable public geocoding APIs to use for free without a strict limit. They are either priced or strictly for personal use. Therefore, it was necessary to build a geocoding server. There are several open source geocoders within the OSM framework: **Nominatim** [15], **Photon** [26] and **Osmnames** [27]. Photon and Osmanmes provide a standalone geocoding servers while Nominatim only provides PHP scripts that can serve geocoding requests. Osmnames seems to be having difficulties with searching for a specific house. It does not seem to be taking account of land registry house numbers. On the other hand, both Nominatim and Photon seem to consider house numbers and typos. Nominatim requires database PostgreSQL with PostGIS extension while Photon requires Elastisearch. Since PostgreSQL with PostGIS are already needed by the tile server, we chose Nominatim over Photon.

4.5 Preference based routing

The goal of the routing part of the project is to find an optimal route between two locations on Earth based on a custom routing profile (see Chapter 2). Any routing functionality must be implemented on the server (see Subsection 4.1.3).

The routing should be fast; therefore, any routing algorithm is implemented in C++.

This section first describes the workflow of the routing server. Afterwards, it is mentioned how a road network can be represented and how can a road graph be stored and created. Lastly, important design alternatives are discussed.

4.5.1 Server workflow

The main responsibility of the server is to receive requests with route endpoints specified by their latitude and longitude coordinates and a route profile and return the optimal route based on the route profile between the endpoints. The route should be represented as a geometry since internal identifiers mean nothing outside of the context of the server.

When a request is received, the server should retrieve a road graph and integrate an input route profile into it. Input endpoints are defined only by their latitude and longitude and are not represented in the road graph; therefore, they are connected to the road graph in a geographically correct place (e.g. to a geographically nearest vertex or edge). Routing algorithm is then applied on the graph to find the optimal route. Lastly, the server returns the geometry of the optimal route.

Routing between two vertices usually only touches a small part of a large road graph; therefore, a part of a road graph around the vertices can be loaded for each routing request and used for routing or parts of the road graph can be loaded over time as a routing algorithm crawls through the graph. However, loading even a small portion of a road graph to memory takes significantly more time than routing using Dijkstra's algorithm.

Hence, a road graph must be already loaded in memory before the server starts receiving requests. A road graph of a country or continent can be quite enormous easily taking tens of gigabytes of memory if not designed to be memory efficient; therefore, one of the main criteria considered when designing the architecture of the routing server is memory consumption. There even may be more road graphs loaded in memory when Static Profile mode is used to add preferences to routing (see Section 2.2).

4.5.2 Road network representation

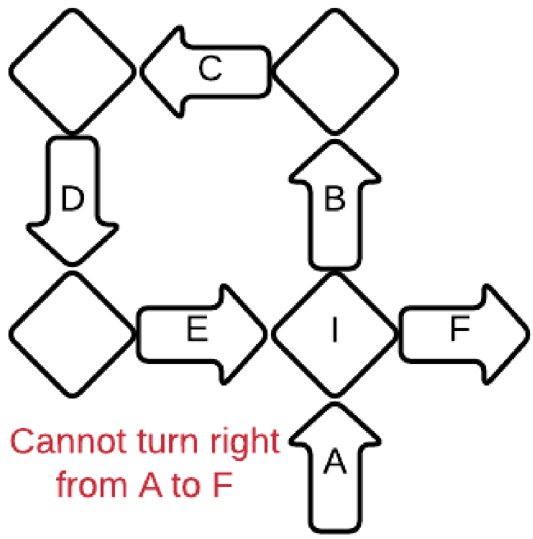

Figure 4.1: Turn restriction situation

An intuitive way to represent a road network with a graph is to have each vertex represent a road intersection or a road end and each edge be a road segment between two vertices. The other considered representation is based on turns. A vertex is a road segment and an edge represents whether one can turn from one road to another. The advantage of the latter option is apparent: it is easy to take account of traffic turn restrictions.

Consider a situation shown on figure 4.1. It is forbidden to turn right from road **A** to road **F** on intersection **I**. Since the intuitive representation does not implicitly contain a turn restriction information, it must be kept in road edge **A** or vertex intersection **I**. Storing additional information does not solve the following issue though. If any shortest path contains a sequence of road edges [**A, B, C, D, E, F**], the path goes through vertex **I** twice. Such path cannot be found by a Dijkstra-like algorithm. In contrast, the turn representation suffers from no such problem.

If no extra information is kept, the turn representation has generally higher memory consumption:

$$|V|_{turn} = |E|_{intuitive}$$

$$|E|_{turn} > |E|_{intuitive}$$

While the turn representation would be quite useful for motor vehicle routing, cyclists might not appreciate the feature. Cyclists usually prefer to avoid traffic and having to do loops on a possibly high traffic roads can be needless not to mention dangerous when cyclists can walk the bike on a nearby sidewalk instead. Moreover, majority of cyclists might prefer to cycle on paths or cycleways which have no turn restrictions. Furthermore, it is really important to reduce memory consumption as much as possible since a whole road graph is in memory at any moment (see Subsection 4.5.1).

Therefore, we chose to use the intuitive graph representation.

4.5.3 Networking libraries

The routing server only needs to serve routing requests. Implementing a high performance server in somewhat complex networking libraries such as Boost.Beast [28] or Boost.Asio [29] seems quite unnecessary when a server does not serve many requests per second. Instead we chose to use **Crow** [30] library which aims to be easy to use. It provides an interface to define handlers for urls which is similar to Flask [31] library. It also includes built-in support for JSON.

4.5.4 Road graph storage

As mentioned in Subsection 4.5.2, a vertex represents an intersection or a road end and edges are road segments between vertices. Apart from a road graph being used by routing algorithms, it is used in two other situations. As mentioned in Subsection 4.5.1, route endpoints from a request are not represented in a road graph and must be added to it in a geographically correct place which typically means finding the nearest neighbours. The other situation is constructing a route geometry from a route found by a routing algorithm.

While vertex geometries can be used to determine the closest vertex to an endpoint, the constructed route geometry should not lead only from an intersection to an intersection but rather between the geographical points of a road graph closest to the route endpoints. Hence, a road graph must contain edge geometries.

A whole road graph is loaded in memory at all times. If a geometry is stored in an edge, it increases the size of the edge multiples times. Only a small number of geometries need to be accessed when serving a request; therefore, they should be stored in a database to reduce memory consumption. The database must provide a way to quickly find edges that are closest to an endpoint and retrieve them. It also must be possible to retrieve a geometry of any edge quickly.

Both the geocoding server and the tile server use **PostgreSQL** [24] database with geospatial extension **PostGIS** [25]. Both PostgreSQL and PostGIS have comprehensive documentations. PostGIS also includes a guide about using a spatial database with tips for optimizing spatial queries such as a nearest geospatial neighbour search. PostGIS provides exactly the functionality we needed to work with geometries efficiently. It also reduces complexity of the whole application by not having to depend on multiple database systems. Therefore, the routing server uses PostgreSQL to store road graphs and data of profile preferences.

4.5.5 Road graph creation

This section discusses how a road graph can be created from OSM data. As mentioned in section 4.5.2, a vertex represents an intersection or a road end and edges are road segments between vertices.

Evaluating the alternative ways to create a road graph requires to know what elements there are in OSM data. OSM wiki [32] states that OSM has three basic element types: nodes, ways and relations. A **node** represents a specific point on Earth (e.g. a road intersection). It consists of an id, a latitude and a longitude. A **way** is an ordered list of nodes that define a polyline. Roads are ways with non-null tag highway. A **relation** documents relationship between two or more elements.

There are several ways to build a road graph:

1. A tool from a routing library can be used to parse OSM data and create a road graph to load into a PostgreSQL database (e.g. **PgRouting** [33]). However, it gives us a little control over what roads the road graph consists of and it is impossible to add a custom properties to an edge without extending the tool.

2. **Mapnik** library used by the tile server requires that map data for rendering are loaded in the **Pgsql** [34] layout in a PostgreSQL database. One of the tables in the layout contains all roads that are rendered. The tables are optimized for rendering so each road contains its raw geometry instead of a list of nodes. Therefore, intersections have to be identified by checking where road geometries intersect. Each road that crosses an intersection has to also be split to smaller segments to create edges. The issue with this approach is that there are tunnels and bridges and when two road geometries intersect in a point, it does not always have to be a road intersection.

3. Raw OSM data can be parsed from scratch to create a road graph. OSM data can be downloaded in multiple formats. While one of them is XML

and the data can be parsed by a general-purpose XML parser, there are specialised C++ libraries for parsing OSM data. We considered **Libosmscout**, **OsmPbf** and **Libosmium**. The only library with a good documentation (or guide) that explains how use it (e.g. manipulate different types of elements, construct geometries, ...) is **Libosmium** [35]. Libosmium library provides support for each OSM element type and is capable of creating geometries that can be converted to GIS formats. Roads are way elements so they comprise of an ordered list of nodes. Roads must be also split to road segments since any node in the lists can be an intersection. An intersection is much easier to identify since each node contains an id and we work with a list instead of with a geometry.

4. The last option is to use a general-purpose OSM data processing tools that can read OSM data and load them into a PostgreSQL database. A SQL code can then create a road graph from the loaded data. The criteria for the tools are that each road must have a column with an ordered list of nodes and that any tag can be loaded. The criteria are satisfied by tool **Osm2pgsql** [36]. The tool can take a Lua script as an input to define the style of a table (e.g. columns, their types, ...). **Osmosis** [37] tool can also be used to load data into PostgreSQL but it does not seem that there is a way to get a list of nodes to a column in a road table. Other tools either do not support loading data to PostgreSQL (e.g. **Osmium** [38]) or create a special schema not fit for our purposes (For example, tool from **Nominatim** [15] geocoder which creates a schema for geocoding.). The SQL script has then have the same responsibility as the C++ code using Libosmium library in the previous alternative.

While using a tool from a routing library to create a road graph is the easiest from the options, it is not very extensible for adding custom properties to edges and vertices. The solution using the Pgsql layout, which is used by the tile server, does not necessarily produce an accurate graph representation of real roads. The last two options both require more work than the others but are extensible. We chose to parse OSM data with Libosmium since splitting roads to road segment seems to be much more easier to be done in C++ than in SQL.

4.5.6 Road Graph representation

The most common operation that routing algorithms require from a graph is to get edges adjacent to a certain vertex and to get vertex neighbours. Therefore, some sort of an adjacency list graph representation is necessary. While a graph could be defined by a container with edges and vertices, it is used by many components and it is much better to provide a standard interface by wrapping the container in a class. Either an explicit interface should be created for the other classes to work on top of or they can accept a graph implementation as a template argument.

Vertex and edge classes in graph

Algorithms and other components using graphs require vertices and edges to have certain different properties; hence, a graph should be built on top interfaces of general-purpose vertex and edge classes. A graph can receive either abstract classes or template classes.

Vertex or edge instances of different types can be stored in a graph with the abstract option. Vertices and edges loaded from a database generally have the same type. However, there are also route endpoint vertices and their edges that must be added to a graph before a routing algorithm is run. Since they are created on the fly, they might require a different implementation for one of the common vertex and edge properties. This actually happens for a property denoting a length of an edge which can be defined by a profile which would have no knowledge of a new edge being created so one of the possible solutions is for the edge is to be of different type and store its length itself.

The disadvantage of using abstract classes is that a container in a class representing graph can only store pointers to an abstract class and the actual instances must be stored elsewhere. Each type of class can be stored in one container (e.g `std::vector`) which cannot be in a graph class so that the generality of the graph class is preserved. Therefore, the containers would have to be maintained by the user of the graph which seems to be a unwieldy interface. The other option is allocating memory for each instance directly which depending on the allocator implementation can lead to an additional memory overhead.

The other option is that graph class gets vertex and edge types as template arguments. One graph can contain only one type of an edge and of a vertex. Because of that a graph can actually store vertices or edges in a single container. The other advantage is a compile time check that suitable types implementing implicit template interfaces are used everywhere.

Since a road graph is present in memory at all times and all or almost all edges or vertices are usually of one type in a graph, we chose to use a templated graph implementation to be memory efficient.

How to store edges

In an adjacency list representation, a vertex should provide a way to access its outgoing edges. Although a vertex can directly contain its edges, if each vertex contains a small container (e.g. `std::vector`), the memory overhead (the unused preallocated space) of each small container stacks up.

The other option is that a vertex contains only two pointers (or iterators) to where its edges are stored. The idea is that if all edges are stored in one container, it is much easier to trim its capacity.

The disadvantage of this approach is that adding edges to a vertex can require moving all its edges to the end of the container and updating the pointer properties of the corresponding vertex (and if the container has no more space, it is reallocated and all vertices must be updated). Therefore, we used the memory efficient alternative for immutable graphs and the former one for mutable graphs.

4.5.7 Temporary endpoint vertices

When a request is received, input route endpoints are defined by their latitude and longitude and they are obviously not represented in a road graph. Any routing algorithm must route between two vertices; therefore, vertices representing endpoints must be added to a road graph. Endpoint vertices are generally only useful for the duration of the request so they should be removed from a road graph once they are no longer needed. Although such an approach works in a single-threaded situation, the graph is shared among any threads handling requests and these operations are mutable. Moreover, a road graph cannot in this case be implemented by the memory efficient immutable graph (see Section 4.5.6).

Therefore, no endpoint vertices or any edges adjacent to endpoint vertices are added to a road graph. Instead another graph class built on top of the road graph can be created to simulate the behaviour as if the vertices were added to the underlying graph. Since no vertices or edges are added to the underlying road graph, no removal action has to be done. Furthermore, the road graph can be implemented using the immutable graph implementation (see Section 4.5.6).

4.5.8 Algorithm integration

The server should contain multiple routing algorithms and should be extensible with respect to adding more routing algorithms. Routing algorithms typically require different functionality, especially in their preprocessing algorithm. That is why we chose not to include any abstractions (e.g. abstract classes, interfaces) in the implementation of routing algorithms and rather focus to split their implementations into classes with respect to single responsibility principle.

Exceptions are any graphs used which as it was discussed in Subsection 4.5.6 are general-purpose implementations with template arguments for vertices and edges. Each routing algorithm should also provide the same interface for its query algorithm so that any algorithm can be easily used in Static Profile mode or Dynamic profile mode mentioned in Section 2.2.

Routing algorithms generally require special properties in vertices and edges. When endpoint vertices (and their edges) are added to a road graph, they cannot be added the same way for each routing algorithm. For example, if a query algorithm is Dijkstra, it requires that edges adjacent to a target vertex are added to the vertices they originate in so that the target vertex is reachable. On the other hand, Bidirectional Dijkstra used in CH applies backward search from a target vertex; therefore, its adjacent ingoing edges must be accessible from it. Hence, it is necessary to implement the function of adding endpoint vertices to a road graph for each algorithm. This functionality in combination with an algorithm that finds to which vertices an endpoint vertex should be connected (and creates such edges) creates a general handler for adding endpoints to a road graph. The functionality should be implemented in policy or strategy classes in order to avoid code duplication — There are more also more ways to find which vertices an endpoint should be connected to.

4.6 Server

The tile server uses Apache HTTP Server [39] with a custom module mod_tile to serve map tiles (see Section 4.3). **The geocoding server** uses tool Nominatim which contains PHP scripts to provide geocoding services; therefore, it needs a HTTP server implementation which is capable of serving PHP scripts. **The routing server** is implemented using Crow library running on a custom port.

It requires a HTTP server capable of redirecting requests to the routing server. Lastly, **the client app** needs to be served statically to users.

The tile server already uses Apache and Apache can serve PHP scripts and static files. It can also redirect requests to a backend server. While Apache could be also run on a different port than the HTTP port, there is no reason to overcomplicate the whole setup and consider using another server such as Nginx [40]. Therefore, we used Apache HTTP Server.

5. Implementation

The project is a web application so it consists of a client app and a server. The application overview is shown in Figure 5.1. The client app is a single page application written in JavaScript. It contains a map editor and and interface for specifing route endpoints and preferences. The entire client app is served to a client all at once and the communication with the server is done using only AJAX requests.

Apache HTTP Server [39] is used to receive any request and it either responds directly or forwards requests to backend servers. The server has four chief functions:

1. Serve map tiles. This is done by a tile server which is implemented entirely by third-party Apache module mod_tile [20].

2. Serve geocoding (see Subsection 4.1.2 or Section 5.2 for an explanation of what geocoding is) requests. Any geocoding request is served by third-party tool Nominatim [15].

3. Serve client app. The app files are served statically by the Apache server.

4. Serve routing requests. The requests are forwarded to the routing server implemented by us.

Tile server, geocoding server and even routing server require map data to operate. Any map data used in the project come from **OpenStreetMap** project.

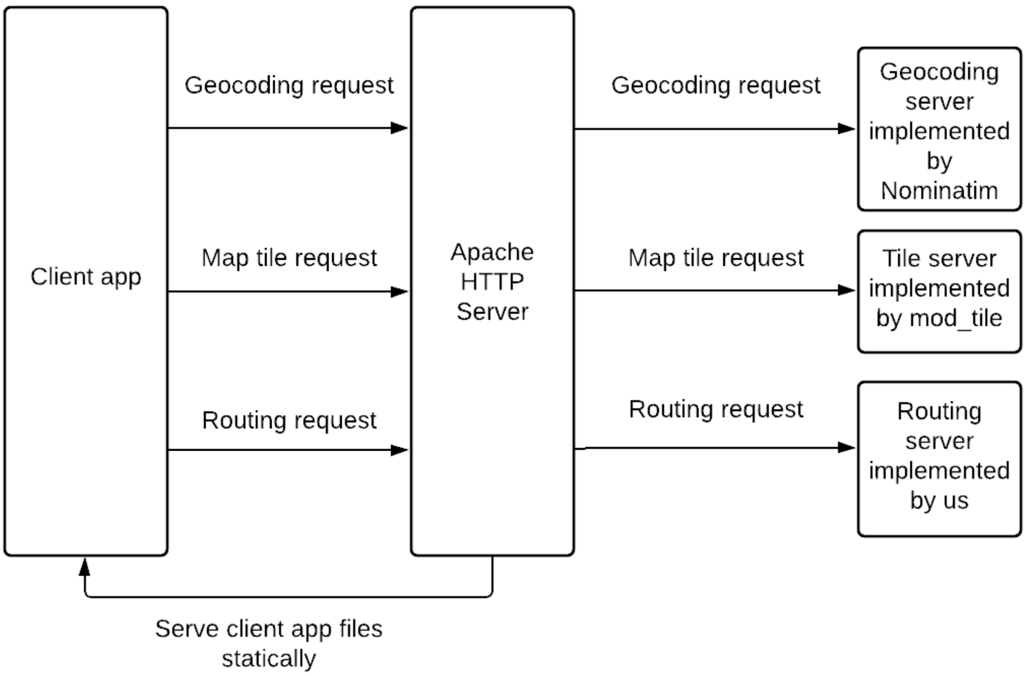

Figure 5.1: Application overview

5.1 Tile server

Tile server is responsible for serving tiles. That is done entirely by third-party Apache module **mod_tile** [20] which provides dynamic combination of an efficient tile caching and on the fly rendering. Tile caching strategy focuses on keeping only a small fraction of overall tiles on disk while allowing high performance serving. Its documentation states that it can support several thousands requests per second.

When a tile is requested, the module checks if the tile is available in cache or if it has changed since. If not, it requests the tile from rendering backend **Renderd** [20] which renders the tile using rendering library **Mapnik** to the file system.

Renderd [20] needs to be supplied with a Mapnik XML stylesheet which determines what map elements are displayed in a tile and their style. The stylesheet contains references to the map data sources which in this case are tables in a PostgreSQL database with PostGIS extension enabled. We use the default **OpenStreetMap Carto** [23] stylesheet.

The stylesheet requires that the map data are in the **Pgsql** [34] database output. It is a layout designed for rendering OSM data with Mapnik. The layout is shown in Table 5.1. The layout is the default output of program **Osm2pgsql** [36]. Osm2pgsql is a tool for importing OSM data into a PostgreSQL/PostGIS database.

Table	**Description**
planet_osm_point	Point geometries created from nodes.
planet_osm_line	Line geometries created from ways and relations tagged type=route.
planet_osm_roads	Contains some of the same data as the line table but with attributes selected for low-zoom rendering. It does not only contain roads!
planet_osm_polygon	Polygon geometries created from closed ways and relations tagged type=multipolygon or type=boundary.

Table 5.1: Pgsql database layout

5.2 Geocoding server

Geocoding server is responsible for transforming a description of a place on Earth to a location (a pair of coordinates) and vice versa. The server is implemented entirely by third-party tool **Nominatim** [15]. It requires a PostgreSQL/PostGIS database where it stores data for geocoding. It internally uses **Osm2pgsql** to load OSM data into the database. Nominatim contains PHP scripts that can be served by Apache PHP module to provide any geocoding service.

5.3 Routing server

The routing server is implemented in C++ and targets C++20. Google C++style guide [41] is used for any naming conventions. The build structure is maintained by **CMake**.

The routing server receives routing requests with route endpoints specified by their latitude and longitude coordinates and a routing profile (i.e. a set of preferences) and returns the geometry of the optimal route based on the profile between the endpoints.

Before any routing can be done, a road graph needs to be created. Executable **graph_builder** focuses on building a road graph from OSM data. The output of the program is a SQL script and a CSV file. The SQL script loads a road graph into a PostgreSQL database.

The road graph can be then loaded from database and used in routing. There are two implemented routing algorithms: Dijkstra and Contraction Hierarchies. Contraction Hierarchies [6] is an algorithm specialised for routing in large graphs. It requires preprocessing to be done on a road graph before it can be used for routing (see Section 2.3 for more information about Contraction Hierarchies).

Routing algorithms are implemented in library **routing**. The library also implements the concepts of Static Profile and Dynamic profile modes described in Section 2.2. Moreover, it also implements several preferences which can be assembled in a routing profile and used in routing. A route preference is basically an index of values that map to edges of a road graph.

Executable **routing_preprocessing** heavily uses the **routing** library and serves as a front for running any algorithm preprocessing or creating route preferences.

Executable **routing_server** is the application that runs the actual server. It implements handlers for any communication over network and uses library **routing** for any routing functionality.

5.3.1 Executable graph_builder

Tool **graph_builder** serves for building a road graph from **OSM** data. The tool outputs a SQL script and a CSV file containing a graph edge list. Running the SQL results in loading the road graph into a table in a **PostgreSQL** database. The tool heavily uses library **Osmium** [35] for parsing raw OSM data and accessing parsed OSM entities.

Osmium library

Osmium [35] library is a header-only library for reading from and writing to OSM files in XML and PBF formats. It supports all types of OSM elements: nodes, ways, relations. Osmium can create WKT, WKB, OGR, GEOS and GeoJSON geometries which can be used in many GIS formats. It can also create polygons and multipolygons from ways and relations.

We use the library primarily for reading OSM files to create a road graph and creating geometries for road segments so let us focus on that. Osmium library provides Handler classes for processing nodes and ways. A custom handler must be created to define a specific operation for node and way elements. Creating a custom handler is done by deriving a class from `osmium::handler::Handler` and implementing and overriding the method for the object type (resp. OSM element) to process. The library reads data and feeds it object by object into handlers by calling the overridden methods. When defining multiple handlers, they can be chained and the library will feed objects into the handlers one by one. The library provides a class for each OSM element through which the element's tags can be accessed.

Another relevant feature for the tool is creating geometries. The library provides a factory for any aforementioned geometry format which can create geometries of any element. It is also possible to create a geometry of a list of nodes resulting in a line geometry. That was especially useful since a way usually consists of multiple road segments which therefore form different edges with different geometries.

Architecture overview

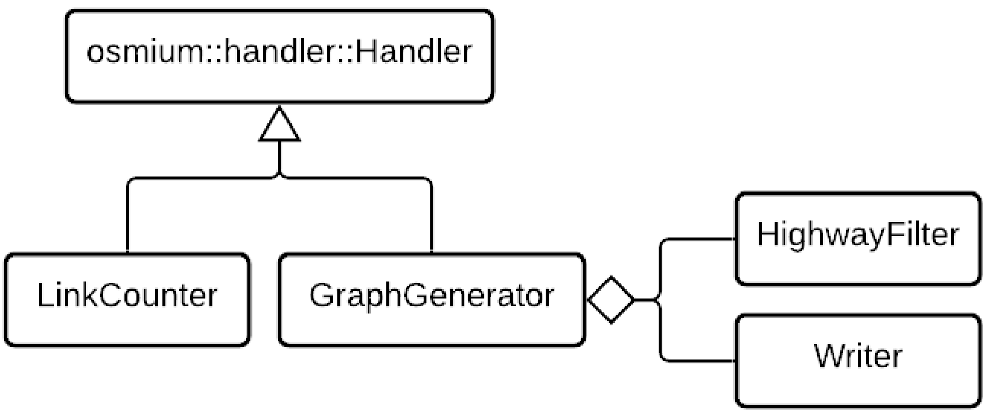

Figure 5.2: Architecture of **graph_builder**

Figure 5.2 illustrates the architecture of the **graph_builder**. The tool aims to produce a graph suitable for routing which means that each edge should be a road segment that ends at intersections or road ends and cannot cross any intersection. Since an OSM way representing a road usually spans over intersections, they must be split into road segments. A way contains a list of nodes which denote its shape in a map (see Subsection 4.5.5 for a quick introduction into OSM). Nodes do not contain information if they are intersection; therefore, intersection nodes must be first identified. That is done by `LinkCounter` class. `LinkCounter` is derived from `osmium::handler::Handler`. An intersection is a node which is referenced from at least two road ways.

`HighwayFilter` is an interface which defines what type of roads are filtered out. For example, filter for bicycles filters out motorways and any roads that a biker is forbidden to cycle on.

`Writer` is an interface for creating the output SQL script file and for adding edges to the output CSV edge list file.

`GraphGenerator` handler class splits road ways into road segments by the intersection nodes identified by `LinkCounter` and creates their geometries. The road segments are saved using a class implementing `Writer` interface to the output edge list file. Specific road types can be ignored by passing a class implementing `HighwayFilter` interface to GraphGenerator's constructor.

It would make more sense for `GraphGenerator` to save the edges in a vector data structure so that not every logic is done in one place. However, the number of edges is enormous and especially the geometries are quite large (e.g. There are millions of edges of a road graph of even Czech Republic and storing all of them in memory takes a few GB of memory.); therefore, edges are saved immediately to the output file to decrease memory consumption.

In conclusion, the tool reads the input data and uses the `LinkCounter` handler to identify all road intersections. A writer implementing `Writer` interface creates an output SQL file and an output CSV file. Lastly, the input data are then read again using the `GraphGenerator` handler to create road graph edges and save them to the output CSV file.

SQL script

The script's main purpose is to load a road graph in a form of an edge list to a database table. In addition, it also does the following:

1. Each edge contains a source vertex id and a target vertex id. These ids are the original OSM node ids which can be any 64-bit integer. In many graph representations, it is convenient when vertex ids start from zero or one and continue from that number with no gaps so that they can be stored in a array-like data structure efficiently and indexed with ease. Since there is no guarantee that the ids start from zero, the script remaps the vertex ids to start from one.

2. It adds a length column to the table and writes edge lengths calculated from edge geometries to the table.

3. It creates a geospatial index on top of edge geometries.

Graph table

Table 5.2 describes an edge row of a graph table. The table has two indices built on top of it. One for column **uid** since it is a primary key. The other is a geospatial index for **geog** column.

Column name	**Type**	**Description**
osm_id	64-bit int	Id of an OSM way from which the corresponding edge is created.
uid	64-bit int	Id of an edge generated from **graph_builder**. It ranges from a small number to the number of edges.
geog	geography	Geography of an edge. Geography is a PostGIS geometry type for a geographic spatial reference system (SRID 4326).
from_node	64-bit int	An id of a vertex from which an edge originates.
to_node	64-bit int	An id of a vertex to which an edge points.
undirected	boolean	Whether an edge is undirected or oriented.
highway	text	Type of the road. It contains values of OSM highway tag.
length	64-bit float	Length of an edge.

Table 5.2: Graph table

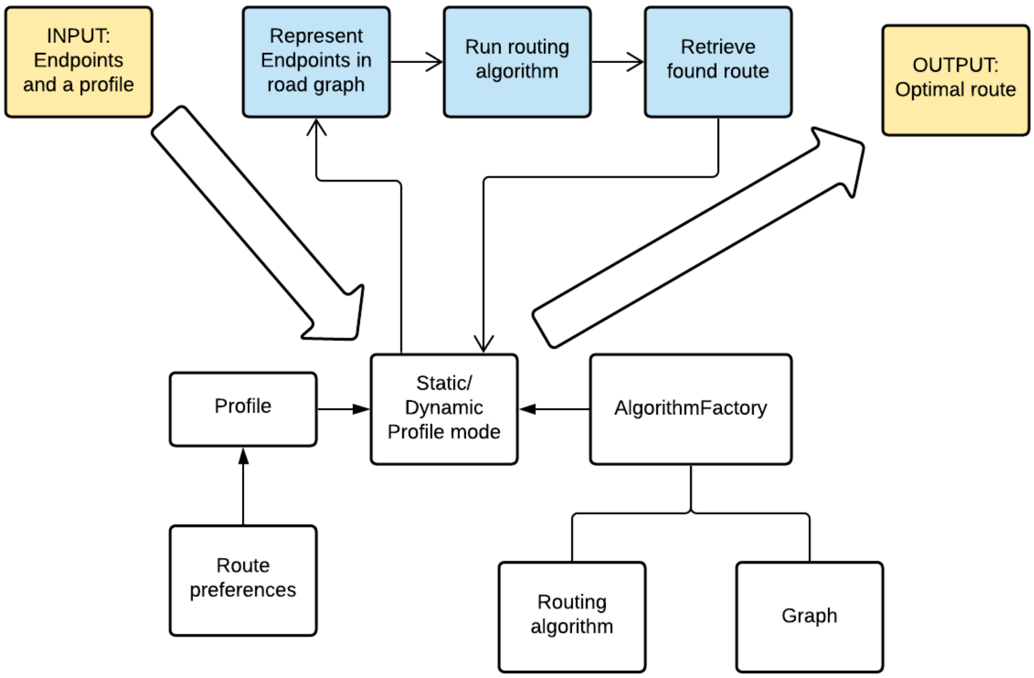

Figure 5.3: Routing library overview

5.3.2 Routing library

The main feature of **routing** library is to provide preference based routing between two arbitrary Earth locations. Figure 5.3 shows a simplified overview of some of the main concepts of the library. Static and Dynamic Profile modes were introduced in Section 2.2. They are capable of adding an input custom profile to a routing algorithm so that the optimal route based on the profile between input endpoints can be found. The modes can be used with any implemented routing algorithm so they are provided with an algorithm factory which e.g. creates a class with implemented query algorithm or defines the type of a road graph.

The yellow rectangles show the expected input and the expected output. Blue rectangles represent what needs to be done so that the optimal route can be found.

Endpoints are characterised by a pair of coordinates and they have no counterpart in a routing graph. Hence, a vertex is created for each endpoint and connected to a road graph. Each routing algorithm defines how an endpoint should be added to a road graph (This was discussed in Subsection 4.5.8 and is the responsibility of class `EndpointsCreator` which is introduced later).

Once the endpoints are represented, the routing algorithm finds the optimal route which is retrieved afterwards.

The modes typically only integrate the input profile with a road graph and return a class which is capable of the blue actions.

There are two implemented routing algorithms: **Dijkstra** and **Contraction Hierarchies**. Dijkstra algorithm is quite slow when it is used in a large graph such as our road graph. One advantage it has is that it can be used without any sort of preprocessing. Therefore, it is viable for debugging or fast setup. Contraction Hierarchies is on the other hand an algorithm specifically made for a large graph (see Section 2.3).

Dependencies

Library **libpqxx** [42] is used to connect to a PostgreSQL/PostGIS database. Routing graphs and preference indices are stored in a database. The library does not work directly with OSM files; therefore, it is necessary to use tools such as aforementioned **graph_builder** to load data into a database.

The library uses fast hash map implementation **tsl-robin-map** [43] because standard `std::unordered_map` has an inefficient memory strategy and tsl-robin-map is significantly more memory efficient and faster than `std::unordered_map`.

Lastly, the library uses **TOML11** [44] library to parse TOML files. Any file input is a file in TOML format.

Routing preferences

Figure 5.4 illustrates the architecture of routing preferences. `Profile` class serves for adding preferences to routing. It provides length (resp. routing cost) values for edges. `Profile` contains an array of classes implementing `PreferenceIndex` with a weight denoting how important a given preference is within the profile. `PreferenceIndex` is an interface implemented by any classes representing a preference that specifies that a preference value must be accessible by an edge id and that an index can be loaded from a database.

`Profile` can either set edge lengths directly in a graph or can serve as a base of edge lengths. The latter option provides an option of using one graph for routing multiple times with different profiles. Nevertheless, it takes significantly more memory since all preference indices must be stored in memory in order not to decrease performance.

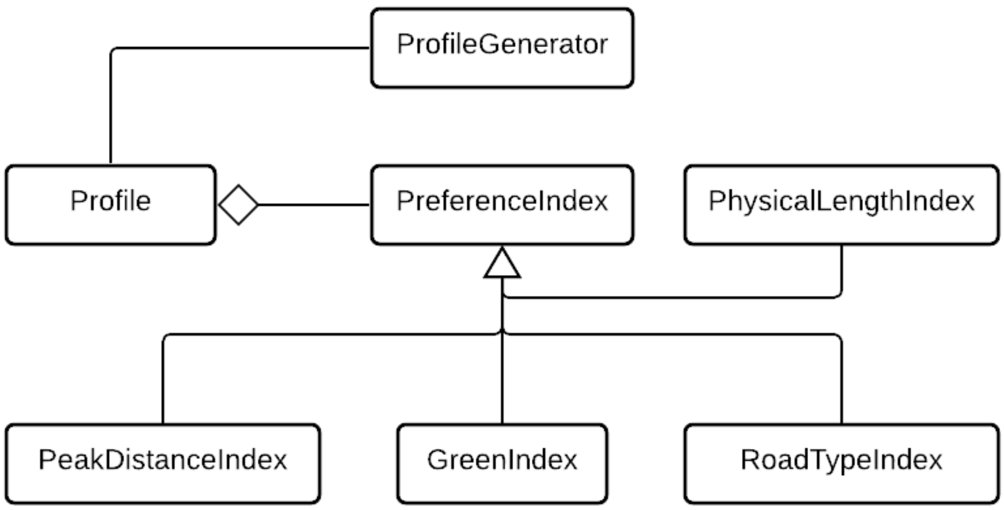

Figure 5.4: Routing preferences architecture

`GreenIndex` class represents a preference that denotes how much green area is around each road edge. `PhysicalLengthIndex` contains the real lengths computed from edge geometries which is a default length for edges in any routing graph. `PeakDistanceIndex` represents how close is each edge to hill or mountain peaks. `RoadTypeIndex` provides values based on types of roads each edge is created from. For example, it can be set up to prefer small roads such as paths and trails over large roads.

`ProfileGenerator` generates routing profiles. It is possible to register a `PreferenceIndex` class with an array of weights. `ProfileGenerator` creates a `Profile` class with all preferences for each combination of weights.

Graphs

Figure 5.5 shows graph classes. The only general-purpose graph implementation is `AdjacencyListGraph`. As the name suggests, it is an adjacency list representation of a graph. Since some routing algorithms require to traverse graph in a direction reverse with respect to the orientation of edges, `BidirectionalGraph` can be built on top of any general-purpose graph implementation. However, it might double the number of edges of a graph by providing edges with the reverse orientation.

`RoutingGraph` can be built on top of any graph whose type is specified as template argument `Graph`. It provides an option of adding vertices and edges that are not added directly to the underlying graph but rather `RoutingGraph` simulates the behaviour of the underlying graph as if they were added to the underlying graph. This is useful for adding temporary edges and vertices to a graph because they do not have to be deleted afterwards.

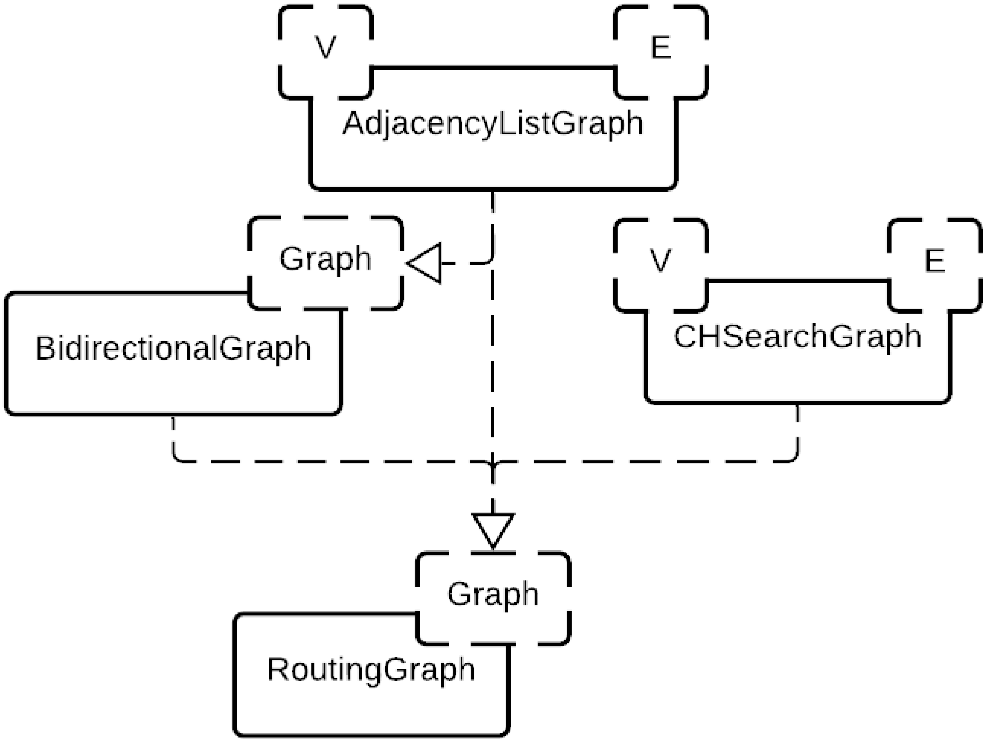

Figure 5.5: Graph classes

Lastly, while the aforementioned graphs can be used with routing algorithms, they may not be the most memory efficient implementation for a particular part of an algorithm. For example, Contraction Hierarchies generally use Bidirectional Dijkstra as the query algorithm which requires a bidirectional graph. While `BidirectionalGraph` could be used, the query algorithm takes into account exactly a half of all edges. Forward search only uses edges leading to more important vertices while backward search uses edges with reverse orientation leading to more important vertices. Therefore, edges leading to less important vertices are ignored. There is a special graph implementation `CHSearchGraph` which contains only relevant edges.

Figure 5.6 shows which vertices and edges can be supplied to the the aforementioned graph classes. `BasicEdge` is a predecessor of all edge classes. Among other properties such as a source and a target vertex of an edge it contains template class `LS` which provides a length for an edge. `ProfileLengthSource` implements the implicit interface of `LS` by providing lengths from a `Profile` class. The advantage this approach is that the profile in `ProfileLengthSource` can be swapped dynamically; therefore, edge lengths can be changed with little overhead. On the other hand, the profile must be loaded in memory along with preference indices; hence, it consumes more memory than using `NumberLengthSource` which contains a length value directly which it returns when queried.

`BasicVertex` serves as a base vertex class for all vertices. `BasicVertex` has a template argument `EdgeRange` which provides from and to iterators to access adjacent edges. There are currently two classes implementing `EdgeRange`:

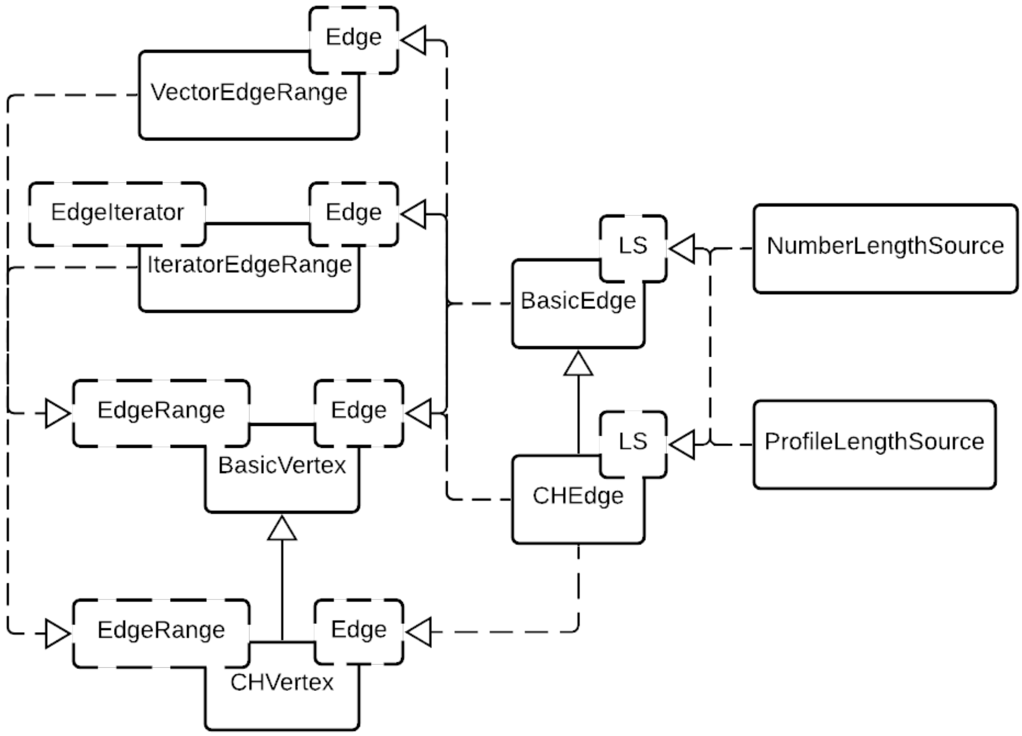

Figure 5.6: Vertex and Edge class overview

`VectorEdgeRange` and `IteratorEdgeRange`. `VectorEdgeRange` class contains `std::vector` which stores every edge. The advantage of that approach is that edges can be added anytime. In contrast, `IteratorEdgeRange` only contains a pair of iterators of the type of its template argument `EdgeIterator`. By using `IteratorEdgeRange` and storing edges efficiently elsewhere e.g. in a trimmed array, memory consumption can be significantly reduced. However, no edges can be added to `IteratorEdgeRange`. Therefore, it is mostly used in an immutable graph.

Routing

Figure 5.7 illustrates a simplified class diagram of the routing part of the library. Routing from one location on Earth to another requires that both source and target are somehow represented in a road graph. In general, vertices represent only junctions so it is necessary to create a vertex for each endpoint and add it to a graph. That is the responsibility of `EndpointsCreator`. It is mainly a general-purpose wrapper around its template classes: `EndpointEdgesCreator`, `EndpointAlgorithmPolicy`. An endpoint vertex must be connected to a road graph in a geographically correct place. Template class `EndpointEdgesCreator` for a given location and an endpoint returns an array of edges connected to the endpoint along with their geometries.

Routing algorithms themselves determine how to add endpoint vertices and their adjacent edges to a graph in order to have their corresponding query algorithms work smoothly. For example, Contraction Hierarchies uses modified Bidi-

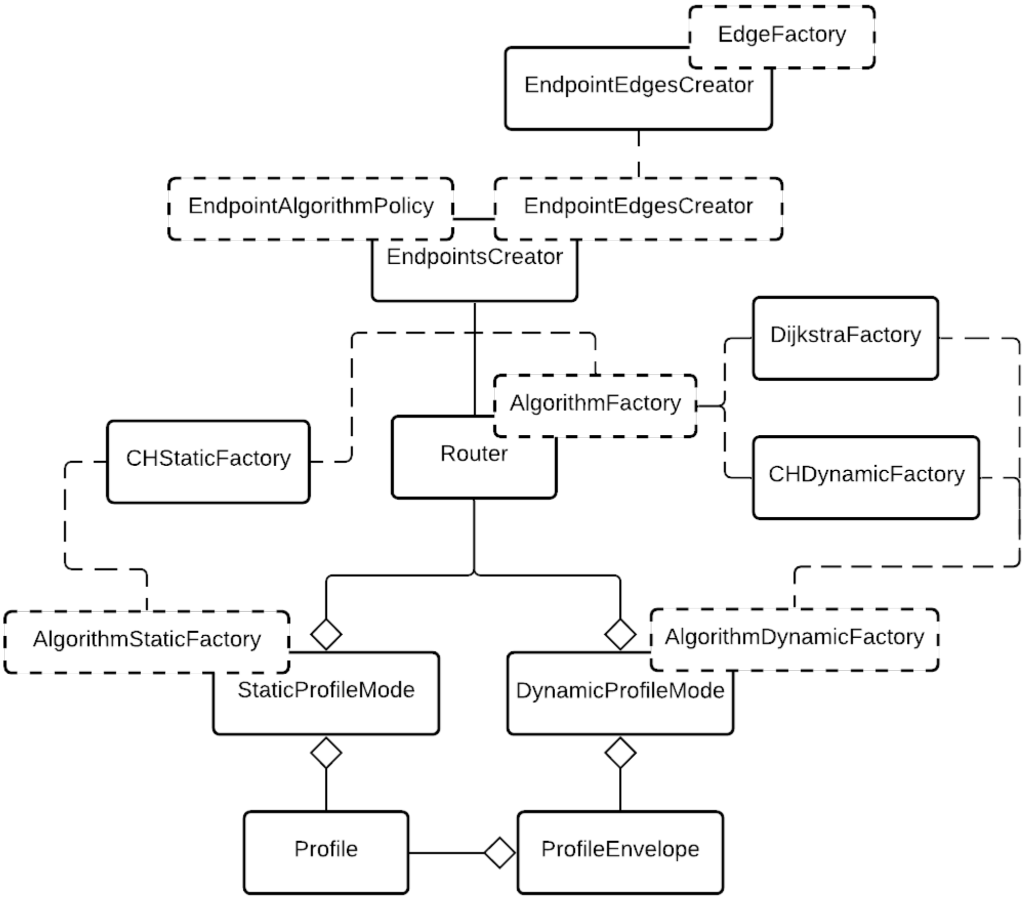

Figure 5.7: Routing architecture overview

rectional Dijkstra algorithm for queries which demands that all edges of a target vertex must be backward (resp. with a reverse orientation) so that the backward search can access vertices adjacent to the target vertex. Whereas Dijkstra algorithm requires adding edges connected to a target vertex to all adjacent vertices so that the target vertex can be reached from a source unidirectionally. Therefore, that is the purpose of the other template class `EndpointAlgorithmPolicy`. For simplicity, `EndpointsCreator` also stores all geometries of endpoints' edges which can be retrieved later.

There is currently only one class satisfying implicit interface of template `EndpointEdgesCreator` which is class `EndpointsEdgesCreator` with template argument `EdgeFactory` which creates edges. The class finds the geographically closest edge to a route endpoint location. The edge is split into two segments in the point of the closest edge where it is closest to the endpoint location. The class then returns edges created from the segments and the segments' geometries. This is done for each endpoint before any routing action; therefore, most of this functionality is implemented using PostGIS in SQL to maximize performance.

`Router` class encapsulates the location to location routing functionality which is essential for the routing server. It holds a road graph as its member. On passing it a source and a target location, it uses `EndpointsCreator` to append the source and target vertices to the graph. A routing algorithm is then used to find a route between the source and target vertices. Once the optimal route is found, it is returned along with its geometry. Note that the source and target vertices are no longer useful. Therefore, when they are created, they are added to `RoutingGraph` instance built on top of the road graph so that the road graph does not change.

Template argument `AlgorithmFactory` of `Router` provides it with template types for `EndpointsCreator`, a class with the query algorithm (e.g. Bidirectional Dijkstra, ...) and a type of the immutable graph. Currently, there are three classes implementing such implicit interface: `CHStaticFactory`, `CHDynamicFactory`, `DijkstraFactory`.

`StaticProfileMode` and `DynamicProfileMode` modes are built on top of `Router` and implement the concepts of Static Profile mode and Dynamic profile mode mentioned in Section 2.2. They both incorporate preferences in the form of `Profile` classes into routing. `DynamicProfileMode` contains a `Router` router and a `Profile` profile. It uses the profile as a base for graph edge lengths which are used by the router. `ProfileEnvelope` shown in the figure serves as a container for the profile. It is referenced instead of the profile so that the profile can be swapped dynamically.

On the other hand, `StaticProfileMode` contains several routers each implicitly using a different profile. A router can be added by providing a graph table name and a profile whose values are already set to edge lengths in the graph table. Storing a lot of routers takes a lot of memory since each contains a graph. However, routing algorithm with a preprocessing are guaranteed to produce optimal results if the preprocessing was run with the same length values as the routing algorithm in a router. In contrast, `DynamicProfileMode` prioritizes memory consumption but routing algorithms with a preprocessing might not be able to find optimal result routes.

The common use case of both modes is the same. On receiving a profile, they return a router with its graph using the given profile values.

Let us take a closer look at the algorithm factories. As shown in Figure 5.7 they are passed to `Router`, `StaticProfileMode` and `DynamicProfileMode` as template arguments. They have the following functions:

1. Specify the type of vertices, edges and a graph. Besides selecting types implementing necessary functionality for a given routing algorithm, a few options can be selected to suit the needs of a user. For instance, it can be selected where edges are stored by specifying vertex `EdgeRange` template argument along with a suitable graph type. Moreover, factories used by `DynamicProfileMode` must specify a graph with edges whose lengths are determined by `ProfileLengthSource` whereas `StaticProfileMode` generally uses `NumberLengthSource`.

2. Load graph from a database and return it in the specified type.

3. Determine the type of the query algorithm of a routing algorithm.

4. Create and return template arguments for `EndpointsCreator`.

Routing algorithm preprocessing

There is only one routing algorithm with a preprocessing which is **Contraction Hierarchies**. Before looking at how its preprocessing algorithm is implemented, let us talk in more detail about the proprocessing itself.

Vertex hierarchy is created by always contracting the least important vertex. Contracting a vertex means replacing shortest paths between its neighbours with shortcuts. The importance of a vertex depends on a several measures. The most important is the edge difference:

$$ED = |E|_{after\ contraction} - |E|_{before\ contraction}$$

In general, it is better to contract uniformly everywhere rather than in a small region. For example, the number of deleted neighbours can be computed. The importance of a vertex is a weighted sum of the measures ([6]).

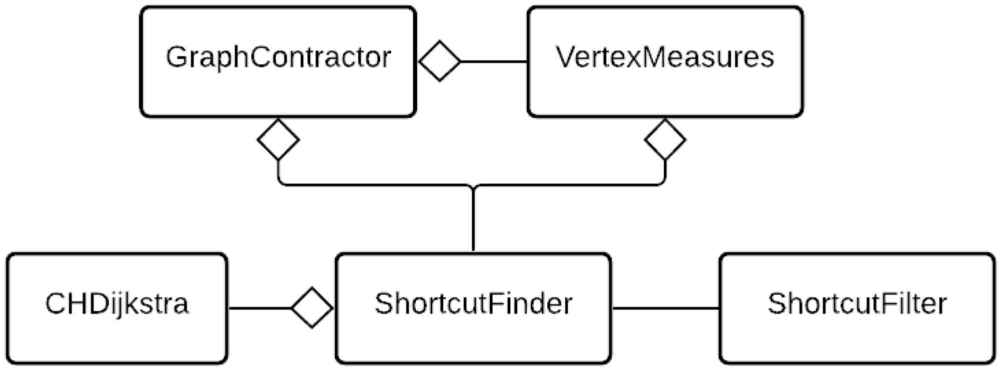

Figure 5.8: Contraction Hierarchies preprocessing

Figure 5.8 shows how the preprocessing of Contraction Hierarchies is implemented. `GraphContractor` iteratively contracts all vertices of an input graph. It uses `VertexMeasures` class to compute importance measures such as edge difference. Both contraction and edge difference computation require calculating the replacement shortcuts to be added if a vertex were to be contracted. That is the chief responsibility of `ShortcutFinder` class.

A shortcut between two neighbours made from the edges connecting the neighbours to the contracted vertex must be added to the input graph if the shortcut is shorter than the shortest path between the neighbours which does not contain the contracted vertex.

This presents an issue if the graph contains multiple edges because then it is possible that two shortcuts with the same source and target are added to the graph. One of the shortcut must have smaller length than or the same length as the other shortcut; hence, it is useless. Moreover, it leads to an extreme growth of the average vertex degree towards the end of the preprocessing when the majority of vertices have already been contracted. That causes major performance issues. Therefore, such multiple shortcuts are filtered by `ShortcutFilter` which speeds up the preprocessing by at least tens of times. Modified Dijkstra algorithm is used as the shortest path between neighbours search algorithm. It is implemented in `CHDijkstra` class.

Database

It was mentioned that the library uses a database to load and save graphs, preference indices and there are SQL algorithms that work with geometries. Classes associated with a database are shown in figure 5.9.

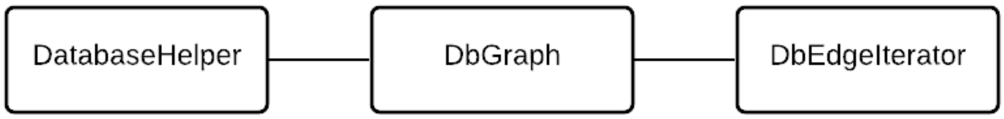

Figure 5.9: Routing architecture overview

`DatabaseHelper` handles any communication with a PostgreSQL database. It internally uses library **pqxx** to connect to a database. The idea is that other classes should not know communication details. For simplicity, it also provides functions with hard-coded SQL queries for common needs such as loading a graph from a database.

Since graphs are stored in tables that can have different columns, there is a class for each graph type which provides methods to work with each graph tables using an unified interface. There is abstract class `DbGraph` which defines such interface. Another abstract class `DbEdgeIterator` defines an iterator interface for iterating through edge table rows. `DbGraph` descendants are usually linked with descendants of `DbEdgeIterator`.

5.3.3 Executable routing_server

Executable **routing_server** as the name suggests is a server which handles routing requests. It uses **crow** [30] micro web framework to create a simple server application and **routing** library for routing.

Server requests

The server serves two requests: **routing request**, **route profile properties request**. Route profile properties request's purpose is to tell the client app which preferences are available and their supported weight values. Routing requests contain source location coordinates, target location coordinates, a route profile which translates to a set of arguments whose names are the preference index names and their values are weights within the profile. The response is a route geometry between the source and the target in GeoJSON.

Server configuration

The server is dependent on a configuration file whose path is passed to it as an argument. The configuration file determines the following:

1. The mode (`DynamicProfileMode`, `StaticProfileMode`) from **routing** library to be used.

2. The algorithm to perform routing. The algorithm must be compatible with the selected mode.

3. The preference indices available and their weight options.

4. Database connection information.

5. The table name of a base graph. Based on the chosen algorithm and mode different graph tables are used to load road graphs but these names always depend on a base road graph which was e.g. used in preprocessing.

Server workflow

The server first reads the configuration and initializes the selected mode from **routing** library which stays in memory until the server is stopped. The mode contains one or more routers which in turn contain road graphs. It is important that the graphs are loaded to memory before any requests come since loading a graph to memory or even its portion takes significantly more time than running even the slowest routing algorithm. The goal is to reuse the graphs for each request; therefore, the graphs are immutable and routing algorithms must take account of that.

When a routing request comes, a router instance is returned from the mode instance based on the profile argument in the request. The router then finds a route between the route endpoints and returns the route whose geometry is returned as a server response.

5.3.4 Executable routing_preprocessor

Executable **routing_preprocessor** is responsible for creating preference indices and running routing algorithm preprocessing. It heavily makes use of **routing** library.

Create preference indices

Preference indices satisfying `PreferenceIndex` interface contain a method to create and store themselves in a table. Creation algorithms are mainly implemented in SQL using PostGIS. This application receives a file as an argument which determines which indices should be created. The application merely parses the file and subsequently calls creation methods of the specified indices.

Routing algorithm preprocessing

Algorithm preprocessing part of the tool receives the same configuration file as an argument as **routing_server**. The idea is that the tool runs any algorithm preprocessing that is required to run **routing_server**. Besides the contents of the configuration described in the **routing_server** section, it also contains parameters for preprocessing algorithms. Preprocessing algorithms take two arguments: a road graph and a route profile to use as a base for edge lengths of the road graph. The output of preprocessing algorithms are new graphs which are saved in a database so that **routing_server** can access them.

The mode specified in the configuration file is what determines how many times the preprocessing algorithm is run. `StaticProfileMode` contains a router with a graph for each possible route profile; therefore, the preprocessing algorithm is run as many times as many profiles can be created. While `DynamicProfileMode` contains only one router; hence, the algorithm is run only once.

There is currently only one routing algorithm which requires a preprocessing and that is Contraction Hierarchies.

5.4 Client application

The client application is a single page application written in JavaScript in **React** [45] framework. It communicates with the server via AJAX requests. It provides an interactive map editor and a panel for specifying route points, a route profile and an route export option.

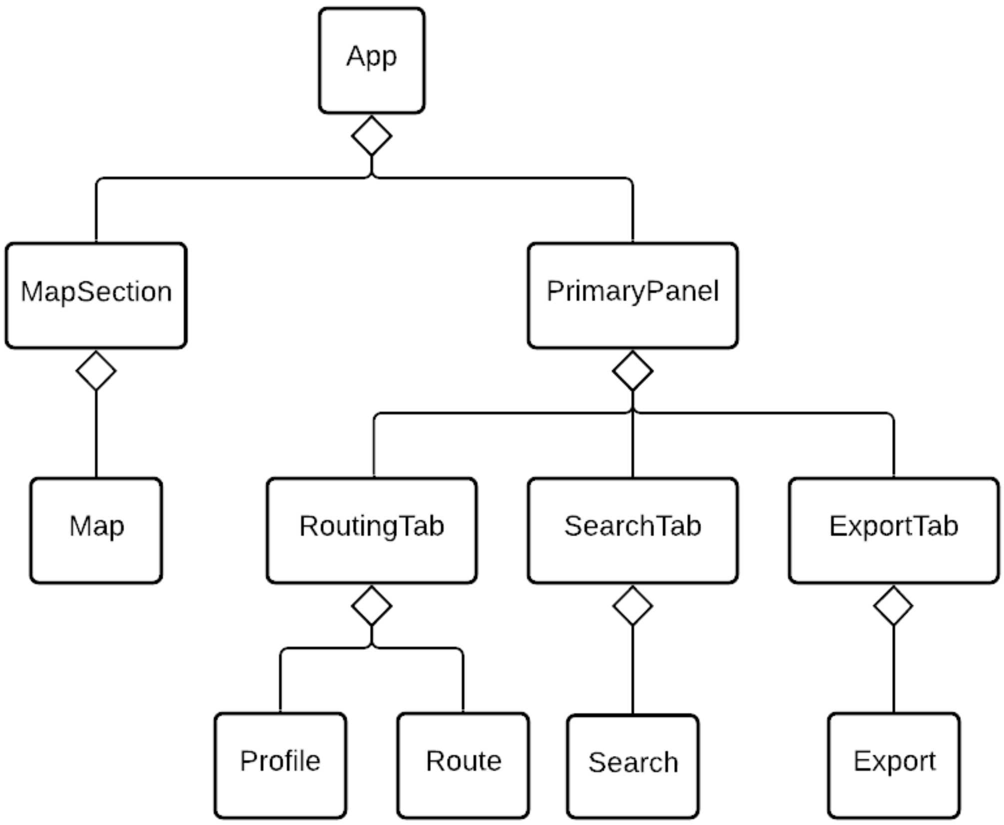

Figure 5.10: Client app components

Figure 5.10 shows the architecture of the client app. All of the entities in the figure are React components. A React component is a piece of a code that returns a piece of html code. Components can store data as states. When a state is changed, the component containting the state is rerendered. If components in two branches need to access same piece of data, the data must be stored in a component where the branches meet. `App` component represents the whole app. It contains two components: `MapSection` and `PrimaryPanel`. It stores route points which are referenced by both `MapSection` and `PrimaryPanel`.

`MapSection` uses **Leaflet** [16] and **React-leaflet** [46] libraries to create an interactive map editor. The libraries require a url to the tile server. `MapSection` itself adds a route geometry and route markers to the map editor. Moving a route marker triggers a change of a route point location; therefore, a request is send to the geocoding server to look for an address of the new location and a request is sent to the route server to find a new route.

`PrimaryPanel` contains three tab components: `RoutingTab`, `SearchTab` and `ExportTab`. Only one tab is visible at one time. `RoutingTab` contains `Profile` and `Route`. `Profile` lets users specify a route profile. Available preferences with their weight options are provided by the routing server. `Route` contains an array of `PathPoint` components. `PathPoint` provides a html input for users to select a route point. It sends requests to the geocoding server so that it can give location suggestions to users as they type. A routing request is sent to the routing server each time there are at least two defined route points and any point or profile change happens. The route is immediately updated in the map. `SearchTab` basically contains `PathPoint` which provides a search in map by description. Export tab lets users export the route shown in the map editor.

6. User documentation

The main part of the client user interface is shown on Figure 6.1. The map editor on the left shows a map. It is possible to zoom and drag the map to other areas. The map editor also displays any found routes. A route point specified by a user is represented by a red marker. The primary panel on the right provides three tabs: Search, Route and Export.

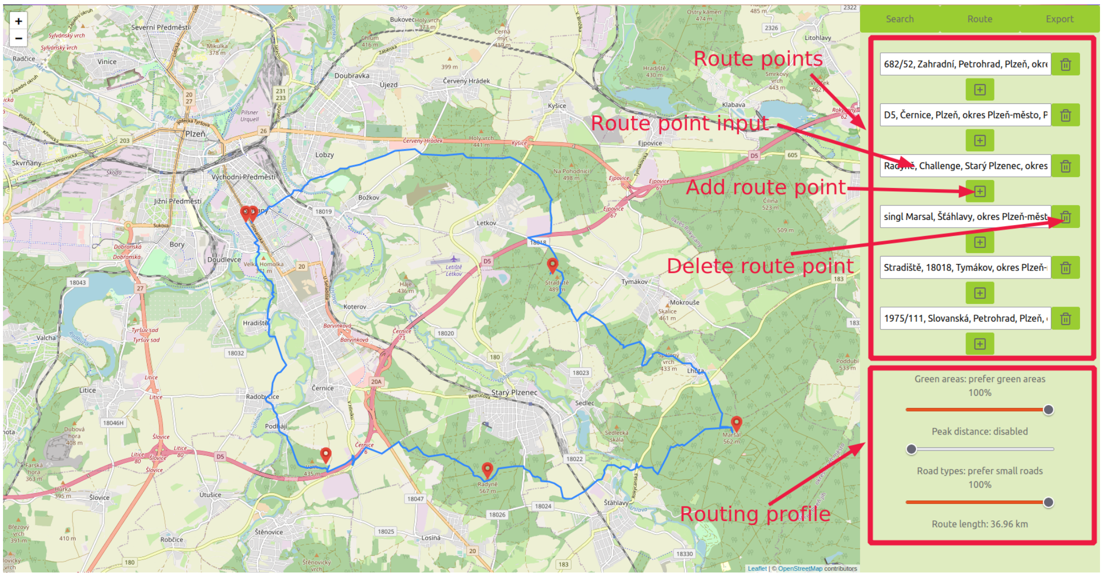

Figure 6.1: Client user interface

Search tab provides a text field for searching in the map. Place suggestions are displayed below the text field as a user types in a description (i.e. address) of a place. A place can be selected by clicking on a suggestion. Once a place is selected, the map editor zooms in on the area around the place.

Route tab serves for defining route points and routing profile. It is the tab that is shown in the figure. A route point corresponds to a text field. Route points can be specified in two ways:

1. A route point can be specified by typing a place description (i.e. address) into its corresponding text field. Place suggestions are displayed below the text area as a user types. The route point is specified once a suggestion is clicked on.

2. A route point can also be selected in the map editor by clicking on the route point's text field and then clicking on a place in the map which the point should represent.

Once at least two route points are defined, the optimal route between the route points is found and displayed in the map editor. The topmost point represents the start point of the route and the bottommost the last. As long as at least two points are defined, any change triggers route recalculation and update.

Additional points can be added by clicking on plus buttons. A new point (with a new plus button below it) is always inserted just below the plus button which it was clicked on. Any route point can be deleted by clicking on a corresponding trash button.

How routes are found is based on a selected routing profile. There is a list of preferences below the route points where a user can specify whether and how much a preference should be considered in routing. Currently, there are three available preferences: green preference, peak distance preference and road type preference.

1. Green preference determines if the route to be found should be rather in green areas or not.

2. Peak distance preference determines whether the route should be near hills and peaks if possible.

3. Road type preference determines what types of roads the route should be primarily on. For example, paths and trails can be preferred over large roads.

Export tab contains a button which exports a found route in GPX format.

7. Evaluation

In this chapter, we evaluate the performance of routing algorithms and the quality of routes based on preferences.

7.1 Routing performance

The performance was measured on i5-8265U CPU clocked at 1.60 GHz with 8 GB main memory. The program was compiled by GNU C++ compiler 9.3.0 using optimization level 2.

Routing was done on a road graph of the Czech Republic containing 4 828 140 directed edges and 1 894 004 vertices.

7.1.1 Routing algorithms' performance

Table 7.1 shows how implemented routing algorithms perform. Each algorithm was run to find the optimal route between two points a predetermined distance away (10 km, 50 km, 100 km, 200 km and 300 km) multiple times (for different points) for each distance. The times shown in the figure are averages of measured times. Only raw query times are measured. Graph initialization and route retrieval are not included.

It can be seen that Dijkstra's algorithm gets much slower when two endpoints are farther away from each other. In contrast, Contraction Hierarchies seem to take only slightly more time as the distance increases. It is clear that Contraction Hierarchies are as expected perfect fit for routing in large graphs.

Note that cities and towns have higher density of roads than countryside. This fact does not influence the routing times of Contraction Hierarchies. However, Dijkstra's algorithm can e.g. for 25 km distance output a route in around 20 ms or 230 ms based on road density.

When planning a trip, a user usually uses more points than two in order to define where the planned route should approximately be. If the majority of casual bike trips are meant to be around 50 km, then two subsequent route points are somewhat close to each other and it might make some sense to use Dijkstra (or optionally implement classic A* algorithm) since it requires no preprocessing.

Algorithm	**10 km**	**25 km**	**50 km**	**100 km**	**200 km**	**300 km**
Dijkstra	41 ms	103 ms	261 ms	387 ms	916 ms	1051 ms
Contraction Hierarchies	22 ms	19 ms	30 ms	43 ms	37 ms	73 ms

Table 7.1: Routing algorithms' performance

7.1.2 Additional routing actions' performance

Table 7.2 shows time measurements for other actions done when handling a routing request. A routing request takes route endpoints as a pair of coordinates.

They are added to a road graph and connected to geographically close vertices. Moreover, the found route must be assembled from the output of the used routing algorithm and its geometry created.

Operation	10 km	25 km	50 km	100 km	200 km	300 km
Add end-points to a road graph	151 ms	206 ms	198 ms	185 ms	170 ms	162 ms
Retrieve route geometry	6 ms	23 ms	35 ms	58 ms	205 ms	206 ms

Table 7.2: Performance of additional actions done when serving routing requests

Again, we measured the executions times for different distances between points. Adding endpoints to a road graph is not dependent on a distance between the endpoints while retrieving (i.e. creating) route geometry definitely is. It can be seen that execution time for retrieving route geometry progressively increases as the distance between endpoints increases.

It is apparent that these actions can take even more time than running a routing algorithm. Both require to work with edge geometries which are not loaded in main memory and instead stored in database where geospatial queries are used to handle any interaction with geometries. That probably causes such high execution times. Geometries take a lot of memory so this is a speed versus memory tradeoff. Since the execution times are so high, it may be better to have geometries and any auxiliary data structures such as spatial trees in memory.

7.2 Quality of routes based on preferences

There are currently three available configurations in the project. Contraction Hierarchies can be used in Static Profile mode as well as Dynamic Profile mode. Dijkstra's algorithm is used with Dynamic Profile mode since it requires no preprocessing. Detailed explanation of these algorithms and modes can be found in Chapter 2.

7.2.1 Static profile routing

In Static Profile mode, routing algorithm preprocessing and query algorithms are run with the same profile denoting edge lengths; therefore, the routes found by the query algorithm are optimal based on the profile. Dijkstra's algorithm in Dynamic Profile mode also produces optimal paths since it requires no preprocessing.

Figures 7.1, 7.2, 7.3, 7.4 show how preferences can alter routing for the exactly same input route points. The arrows on the figures compare the routes found based on preferences to the shortest route on Figure 7.1.

It can be seen that enabling green preference shifts the route to go through green areas more (see Figure 7.2). This is especially the case when the shortest route with disabled preferences leads through a town or village area with houses, important roads and few green areas such a as park which occurs between route points 1-2 and 3-4. This preference does not take into account how large a road is; therefore, it can even prefer primary or secondary roads which are surrounded by greenery. This can be seen on the route between points 5-6 where a route segment leads over a secondary road (yellow in the map).

That is why a road type preference was created. We mainly use the road type preference to prefer trails and cycleways over large roads such as primary and secondary roads. It can be seen to a lesser extend in Figure 7.3 or to a greater extent in Figure 7.5.

Three mini figures in Figure 7.5 also illustrate how beneficial road type preference is. In the top figure, all preferences are disabled. The shortest route is on a yellow secondary road. In the middle figure, green preference is set to 100%. Since the secondary road is in a relatively green area, almost a half of the route leads on it. Small road type preference is set to 100% in the bottom figure where the route is entirely off the secondary road. Therefore, road preference is quite useful for planning routes in countryside.

Lastly, both road type preference and green preference are set to 100% for the route in Figure 7.4. The route is for the most part on paths and trails in green areas. The main difference to the route in the previous figures are the changes between point 4-5 and 5-6.

The route displayed in Figure 7.4 seems to satisfy what a user looking for a trip in green areas away from large roads wants. If somebody instead desires the shortest route, he/she can disable all preferences to get the same route as in Figure 7.1.

Therefore, Static Profile mode seems to be working perfectly with custom routing profiles as far as result routes are concerned. However, the big disadvantage is that road graphs for all profiles must be kept in memory so that the performance measured in Subsection 7.1.1 does not drop.

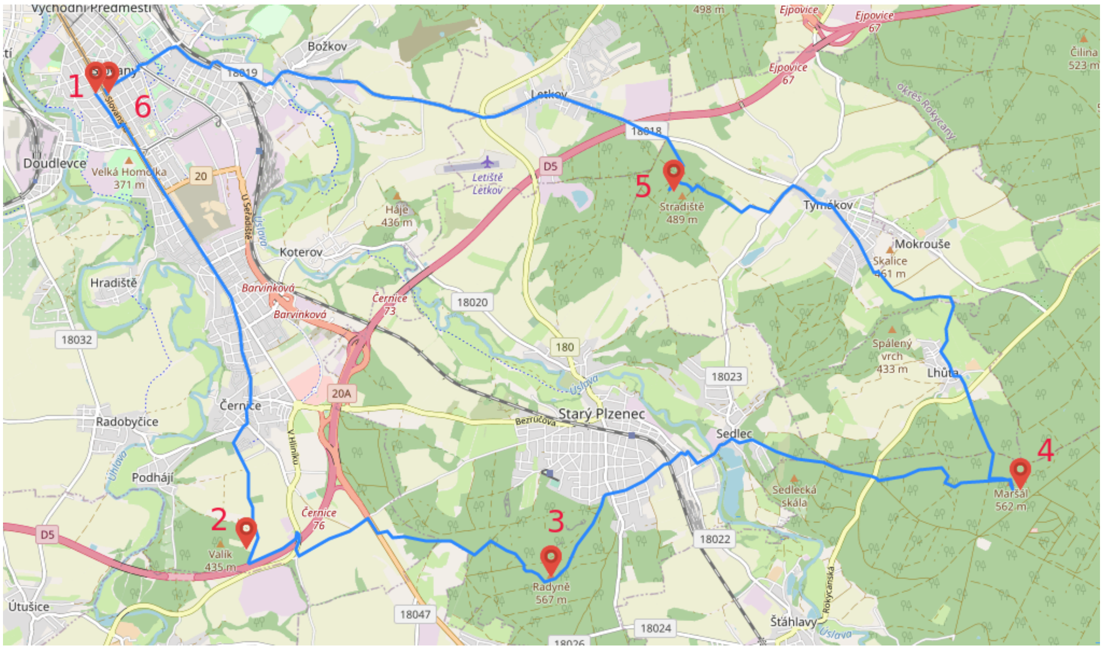

Figure 7.1: The optimal route with disabled preferences

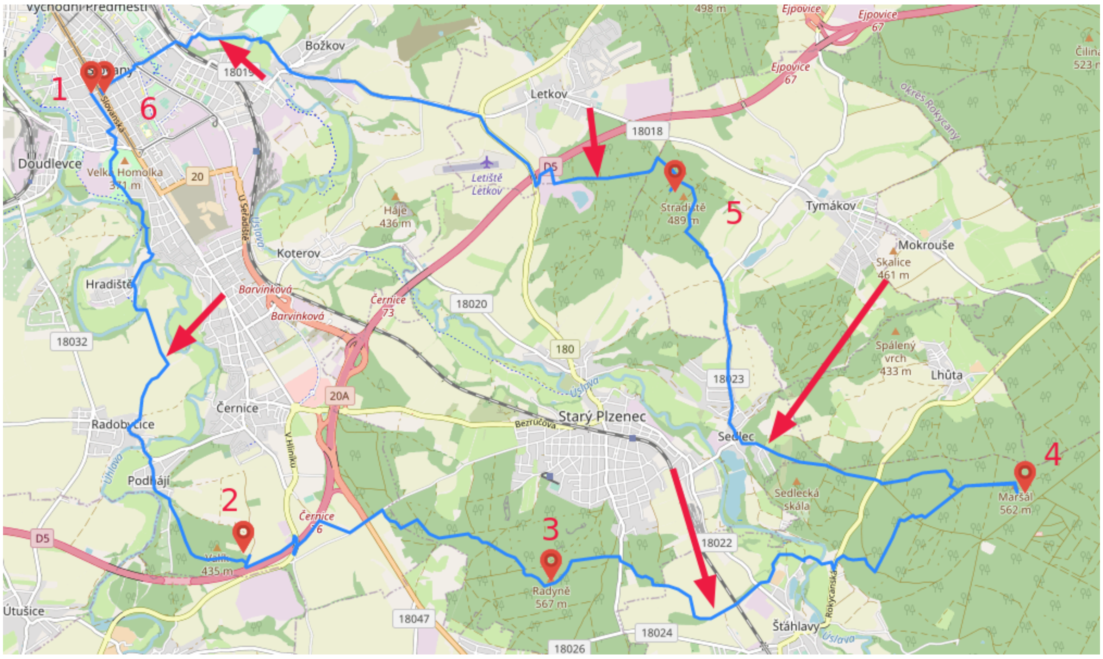

Figure 7.2: The optimal route based on green preference set to 100%

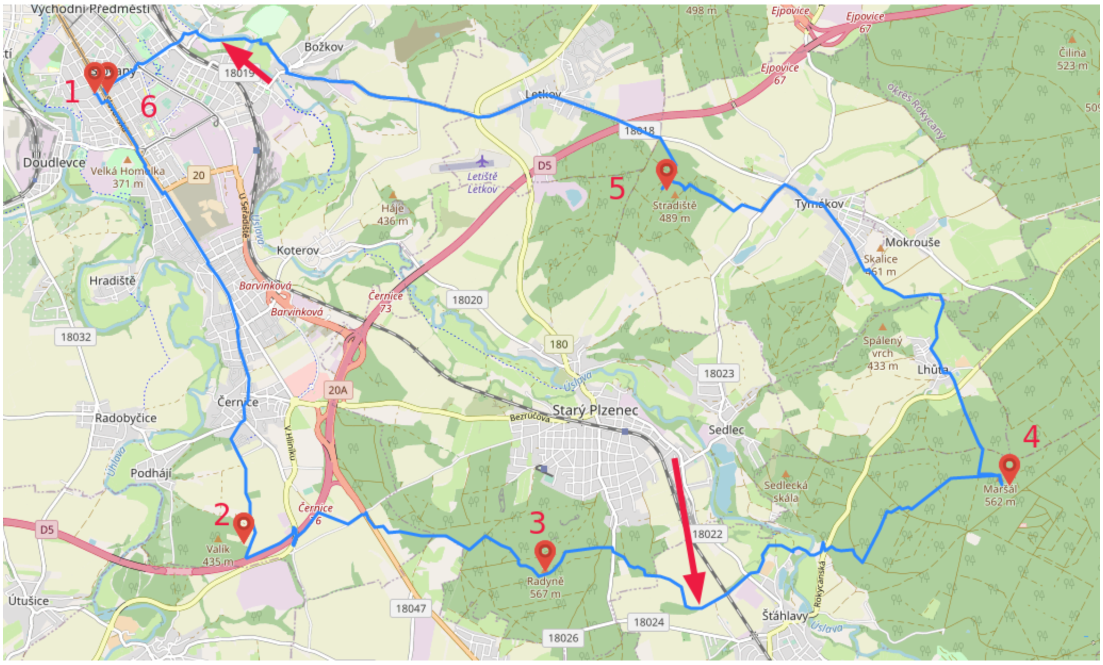

Figure 7.3: The optimal route based on road type preference preferring small roads set to 100%

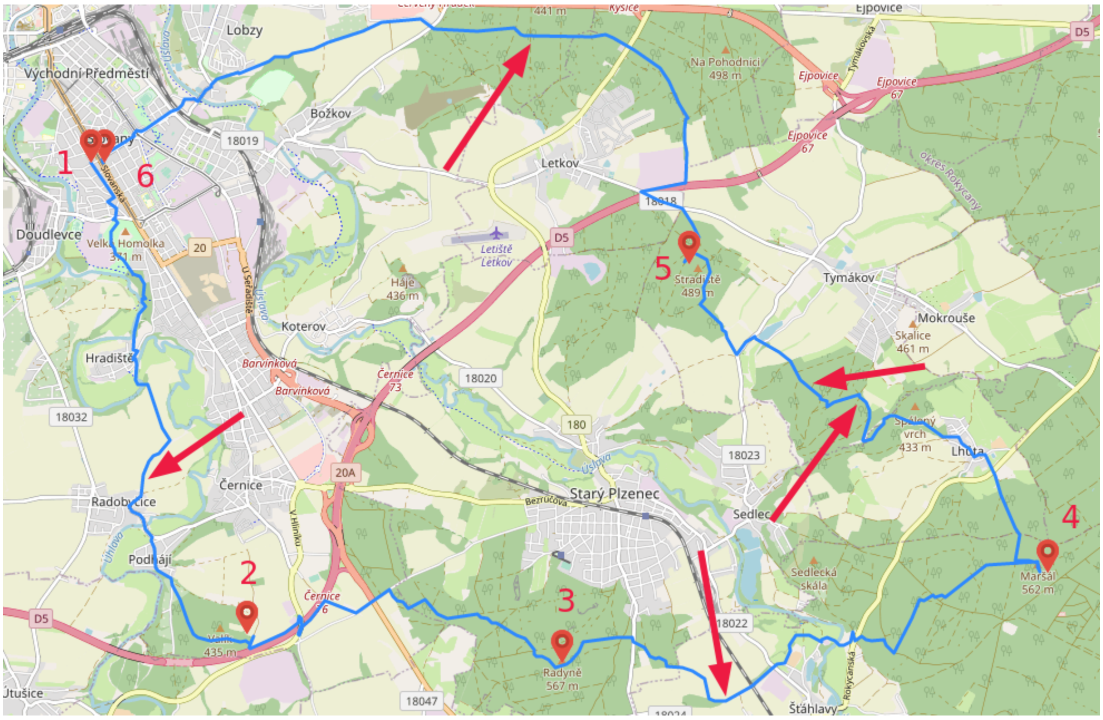

Figure 7.4: The optimal route based on small road type preference set to 100% and green set to 100%

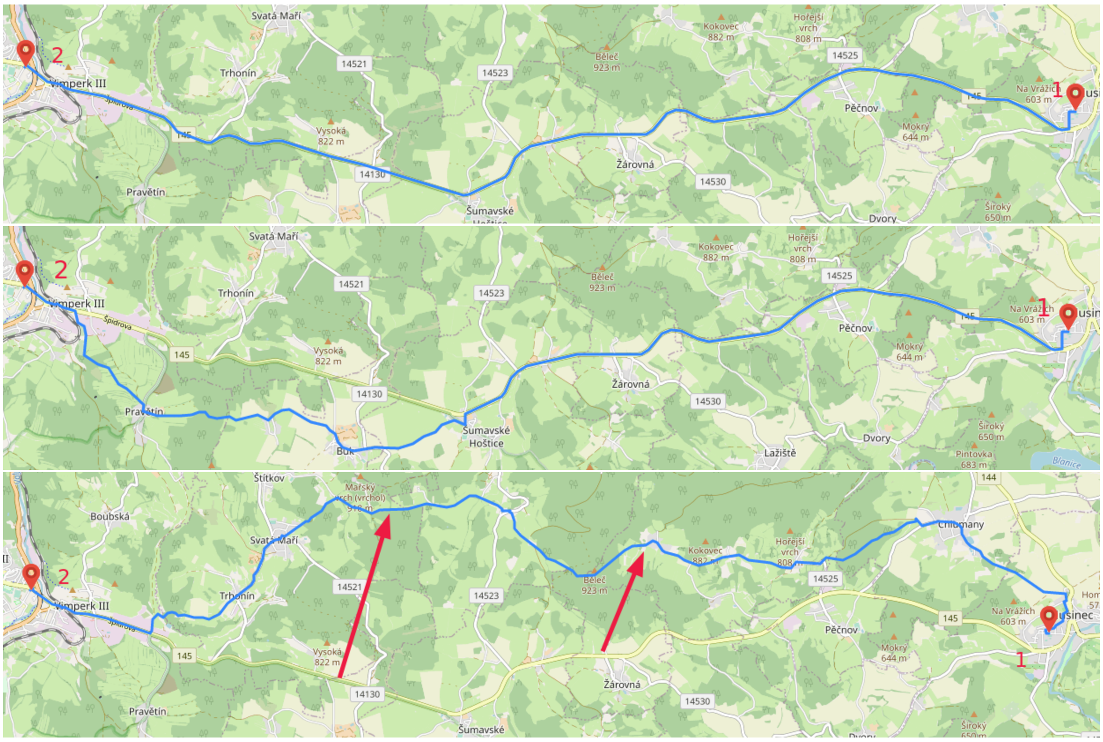

Figure 7.5: Comparison of found routes based on different preferences. The top figure has preferences disabled, the middle figure has green preference 100%, the bottom figure has small road type preference 100%.

7.2.2 Dynamic Profile routing

In Dynamic Profile mode, algorithm's preprocessing runs once with one profile and the query algorithms typically finds routes based on different profiles. The route in Figure 7.6 was found using CH in Dynamic Profile mode. CH preprocessing was done on a profile with preferences disabled (i.e. real distance denotes edge lengths.). The query algorithm ran based on a profile with green and road type preferences. There is only a small difference between the route and the shortest path (see Figure 7.1). The route in Figure 7.4 with the same preference setting looks to be in more green areas and on smaller roads.

Routes found by CH in Dynamic Profile mode when different routing profiles were set were equally underwhelming. It seems that CH algorithm does not perform well with the dynamic mode.

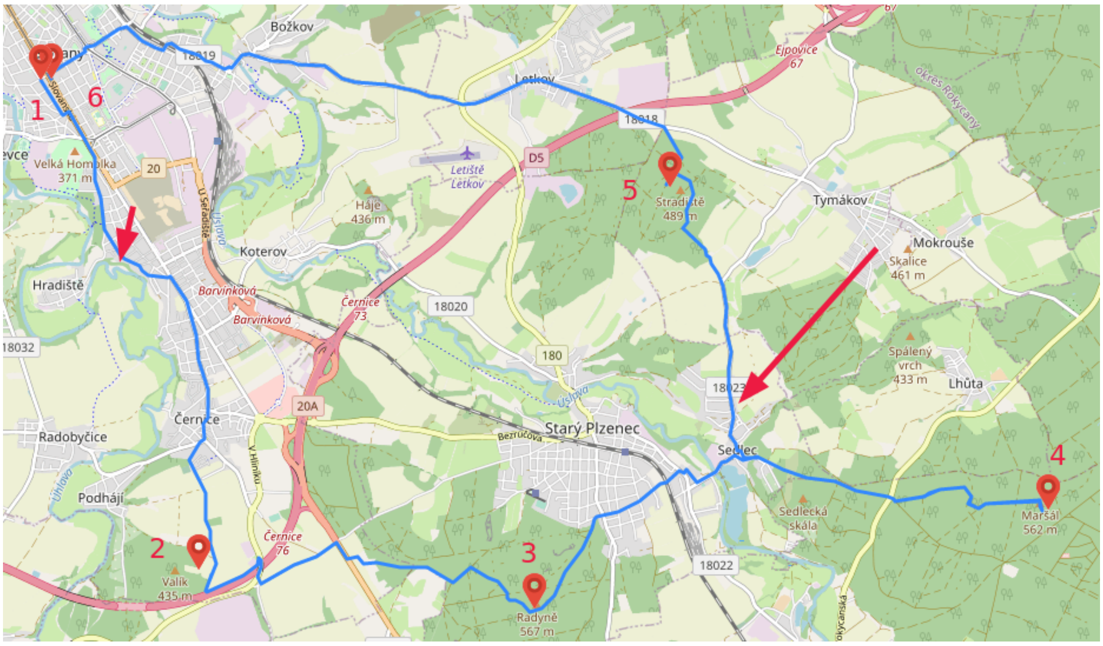

Figure 7.6: The found route based on road type preference preferring small roads set to 100% and green preference set to 100% found by CH in Dynamic Profile mode

8. Conclusion

The goal of the thesis was to use routing algorithms to find a way to perform routing based on a custom routing profile and implement a web application capable of finding routes based on a custom routing profile as a proof of concept.

We first analyzed that heuristic routing algorithms, which require among other things preprocessing to be done on a road graph beforehand, are needed for routing in road graphs for performance reasons. Two approaches for adding preferences into routing were proposed: Static Profile mode and Dynamic Profile mode. In Static Profile mode, preprocessing is run on a road graph for each possible routing profile so that optimal paths based on any profile can be found in query. In Dynamic Profile mode, preprocessing is run only once for one profile and and the query algorithm uses the output of the preprocessing in query for any input profile.

We implemented both modes in the web application along with Contraction Hierarchies heuristic routing algorithm. We also created several preferences which can be selected by users and added to routing.

The evaluation of the application showed that Contraction Hierarchies algorithm is as expected quite fast for finding both short and long routes in large graphs. The efficiency of preference based routing was evaluated by comparing routes found based on different profiles. Static Profile mode with CH responded well to each profile by routing a route through areas preferred by the profile. However, due to the decision that a road graph for each profile is in memory at all times, it consumes a lot of memory. On the other hand, Dynamic Profile mode with CH did not alter routes for different profiles much from the route that was found for the profile that was also used in preprocessing.

References

[1] Mapy.cz. [2021-07-13]. URL https://en.mapy.cz.

[2] Google Maps. [2021-07-13]. URL https://www.google.com/maps.

[3] Bikemap route planner. [2021-07-13]. URL https://www.bikemap.net/en/routeplanner.

[4] Komoot. [2021-07-13]. URL https://www.komoot.com/plan.

[5] Ride with GPS. [2021-07-13]. URL https://ridewithgps.com.

[6] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA*, pages 319–333, 2008.

[7] R. Gutman. Reach-based routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proc. 6th International Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004.

[8] H. Bast, S. Funke, D. Matijevic, C. Demetrescu, A. Goldberg, and D. Johnson. TRANSIT: Ultrafast Shortest-Path Queries with Linear-Time Preprocessing. In *9th DIMACS Implementation Challenge — Shortest Path, DIMACS (2006)*, 2006.

[9] Google Maps Platform - Google Developers. [2021-06-12]. URL https://developers.google.com/maps.

[10] Google maps platform pricing. [2021-07-06]. URL https://cloud.google.com/maps-platform/pricing.

[11] ArcGIS Developer. [2021-07-07]. URL https://developers.arcgis.com.

[12] OpenStreetMap Wiki. [2021-06-12]. URL https://wiki.openstreetmap.org/wiki/Main_Page.

[13] LocationIQ web page. [2021-07-07]. URL https://locationiq.com.

[14] OpenCageData. [2021-07-07]. URL https://opencagedata.com.

[15] Nominatim web page. [2021-07-07]. URL https://nominatim.org.

[16] Leaflet - a JavaScript library for interactive maps. [2021-07-07]. URL https://leafletjs.com.

[17] OpenLayers library. URL https://openlayers.org.

[18] Mapnik library. [2021-07-07]. URL https://mapnik.org.

[19] TileSweep tile server. [2021-07-07]. URL https://github.com/seemk/TileSweep.

[20] Apache module mod_tile and renderd repository. [2021-07-07]. URL https://github.com/openstreetmap/mod_tile.

[21] OSM Scout server. [2021-07-07]. URL https://github.com/rinigus/osmscout-server.

[22] OSM Bright. [2021-07-07]. URL https://github.com/mapbox/osm-bright.

[23] OpenStreetMap Carto. [2021-07-07]. URL https://github.com/gravitystorm/openstreetmap-carto.

[24] PostgreSQL. [2021-07-10]. URL https://www.postgresql.org.

[25] PostGIS. [2021-07-10]. URL https://postgis.net.

[26] Photon. [2021-07-07]. URL https://github.com/komoot/photon.

[27] Osmnames. [2021-07-07]. URL https://github.com/klokantech/osmnames-sphinxsearch.

[28] Boost.Beast networking library. [2021-07-11]. URL https://www.boost.org/doc/libs/1_76_0/libs/beast/doc/html/index.html.

[29] Boost.Asio networking library. [2021-07-11]. URL https://www.boost.org/doc/libs/1_75_0/doc/html/boost_asio.html.

[30] Crow networking library. [2021-07-11]. URL https://crowcpp.org.

[31] Flask. [2021-07-09]. URL https://flask.palletsprojects.com/en/2.0.x.

[32] OpenStreetMap wiki - elements page. [2021-07-09]. URL https://wiki.openstreetmap.org/wiki/Elements.

[33] Pgrouting. [2021-07-09]. URL https://pgrouting.org.

[34] Pgsql database layout. [2021-07-10]. URL https://osm2pgsql.org/doc/manual.html#the-pgsql-output.

[35] Libosmium. [2021-07-10]. URL https://osmcode.org/libosmium.

[36] Osm2pgsql. [2021-07-10]. URL https://osm2pgsql.org.

[37] Osmosis. [2021-07-10]. URL https://github.com/openstreetmap/osmosis.

[38] Osmium. [2021-07-10]. URL https://osmcode.org/osmium-tool.

[39] Apache HTTP server project. [2021-07-12]. URL https://httpd.apache.org.

[40] Nginx. [2021-07-12]. URL https://www.nginx.com.

[41] Google C++ Style Guide. [2021-07-12]. URL https://google.github.io/styleguide/cppguide.html.

[42] Libpqxx. [2021-07-12]. URL http://pqxx.org/development/libpqxx.

[43] Tessil robin-map: Fast hash map implementation. [2021-07-12]. URL https://github.com/Tessil/robin-map.

[44] Toml11. [2021-07-12]. URL https://github.com/ToruNiina/toml11.

[45] React - A JavaScript library for building user interfaces. [2021-07-12]. URL https://reactjs.org.

[46] React Leaflet. [2021-07-12]. URL https://react-leaflet.js.org.

[47] Googletest. [2021-07-22]. URL https://github.com/google/googletest.

[48] Yarn. [2021-07-22]. URL https://yarnpkg.com.

List of Figures

List of Tables

List of Abbreviations

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CH	Contraction Hierarchies
CPU	Central Processing Unit
CSV	Comma-Separated Values
GB	Gigabyte
GHz	Gigahertz
GIS	Geographic Information System
GNU	GNU's Not Unix
GPS	Global Positioning System
GPX	GPS Exchange Format
HTTP	Hypertext Transfer Protocol
OSM	OpenStreetMap
PBF	Protocolbuffer Binary Format
PHP	PHP Hypertext Preprocessor
SQL	Structured Query Language
SRID	Spatial Reference Identifier
TOML	Tom's Obvious, Minimal Language
XML	Extensible Markup Language

A. Attachments

This thesis includes source code of the application as a digital attachment. Its structure is described in Appendix B.

B. Source code structure

Our source codes include a client application and a routing server. Geocoding server and Tile server are entirely created by third-party tools. The source code of this project is available at `https://github.com/georgeus19/maps`. The code structure can be seen below. Routing server contains a lot of source files; therefore, mainly directories are shown.

- `/`
  - `backend` .... Routing server
    - `include` .... Header files
      - `graph_builder` .... Classes of graph_builder target (5.3.1)
      - `routing`
        - `database` .... Database related functionality (5.3.2)
        - `edge_ranges` .... Edge range classes stored in vertices (5.3.2)
        - `edges` .... Graph edge classes (5.3.2)
        - `preprocessing` .... Classes only for preprocessing (5.3.2)
        - `profile` .... Profile and preference related classes (5.3.2)
        - `query` .... Classes for routing queries (5.3.2)
        - `utility`
        - `vertices` .... Graph vertex classes (5.3.2)
      - `tests` .... Unit tests
    - `src` .... Implementation files
      - `graph_builder` .... Classes of graph_builder target (5.3.1)
      - `routing`
        - `database` .... Database related functionality (5.3.2)
        - `preprocessor` .... Files exclusively for preprocessor target (5.3.4)
        - `profile` .... Profile and preference related classes (5.3.2)
        - `server` .... Files exclusively for server target (5.3.3)
        - `utility`
    - `lib`
      - `tsl` .... Header only fast hash map library
    - `inputs` .... Input files for preprocessing and server executables
      - `ch_dynamic_config.toml`
      - `ch_static_config.toml`
      - `dijkstra_config.toml`
      - `indices.toml`
    - `doxyfile` .... Run doxygen doxyfile to create reference documentation
    - `CMakeLists.txt`
  - `frontend` .... Client app (5.4)
    - `src`
      - `App.js` .... Top level app component
      - `mapSection.js` .... Map editor components
      - `primaryPanel.js` .... Components for primary panel

C. Installation manual

The application contains a client app and a server. As mentioned in Chapter 5, HTTP Apache Server [39] is used to serve any requests. There are three sub-servers that require separate installation. Routing server was implemented by us; therefore, we go through its installation and configuration process in detail. Geocoding server and tile server are entirely handled by third-party tools. We mainly show how to incorporate them into the whole server. We describe installation on Ubuntu.

Apache server can be installed from:

```
httpd.apache.org/download.cgi
```

Copy 000-default.conf file in /etc/apache2/sites-available to create a configuration for the server and name it maps.conf. The Apache config file can be activated by running:

```
sudo a2ensite /etc/apache2/sites-available/maps.conf
service apache2 reload
```

C.1 Routing server

C.1.1 Dependencies

The server has twe following dependencies:

1. GoogleTest [47] library for unit tests
2. Libosmium [35] library to parse OSM files
3. Crow [30] library to create backend server
4. PostgreSQL [24] database wit PostGIS [25] extension
5. Pqxx [42] library to connect to PostgreSQL database

GoogleTest library is downloaded from the repository below and configured when running Cmake.

```
github.com/google/googletest.git
```

Libosmium library is used to parse OSM files. It requires Expat, Zlib, Bz2lib, Protozero (1.6.3 and later) and Boost (1.55 and later) libraries. Installation guide of the library can be found at:

```
osmcode.org/libosmium/manual.html#building-libosmium
```

Crow library requires Boost (1.70 and later) and Zlib libraries. We use the library as a single header file which can be downloaded from:

```
github.com/crowcpp/crow.git
```

The following page contains details about downloading and installing crow:

```
crowcpp.org/getting_started/setup/
```

PostgreSQL (10.15 and later) and PostGIS (3.1 and later) can be downloaded and installed from:

```
www.postgresql.org/download/
postgis.net/install/
```

Libpqxx can be either built from source or installed from package manager.

```
github.com/jtv/libpqxx
```

C.1.2 Configuration

The source code of the routing server is available in backend directory at `https://github.com/georgeus19/maps`. All targets can be built by running the following script from backend directory:

```
mkdir release
cd release
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

It creates targets graph_builder 5.3.1, routing 5.3.2, routing_preprocessor 5.3.4 and routing_server 5.3.3.

Any data that the routing server uses while running is saved in PostgreSQL/PostGIS database. Ideally, the routing server should use the same database as the tile server. The tile server stores data for rendering in Pgsql layout [34] which is also used for creating preference indices (e.g. green index, peak distance index).

If that is infeasible, then create a new PostgreSQL database and enable PostGIS and Hstore extensions. Data should be loaded to be in Pgsql layout. OSM data should be in PBF format which are available at `http://download.geofabrik.de/`. We recommend downloading data of small countries to first check that the servers work as expected.

```
Add extension in database (execute as SQL query):
CREATE EXTENSION postgis;
CREATE EXTENSION hstore;

Load data from terminal:
osm2pgsql -d {dbname} --create --slim  -G --hstore {data.pbf}
```

Create road graph

Run executable graph_builder 5.3.1 to create a SQL script and a CSV file which together load a road graph to database. It has the following interface:

```
./graph_builder {input_file.pbf} {table_name} {output_sql_path.
   sql} {output_data_path.csv}

{input_file.pbf} - pbf OSM data e.g. from http://download.
   geofabrik.de/
{table_name} - table where the road graph is saved
{output_sql_path.sql}
{output_data_path.csv} - absolute path is compulsory
```

Run the script to load the road graph to database (its column structure is portrayed in Table 5.2).

Create preference indices

Run routing_preprocessor 5.3.4 to create preference indices:

```
./algorithm_preprocessing --create-index {indices_inputs.toml}

{indices_inputs.toml} - File specifying what preference indices
   to create
```

A file with all available preferences is available at backend/inputs/indices.toml. The input file has the following structure:

```
[database]
name = "dbname"
user = "username"
password = "password"
host = "127.0.0.1"
port = "5432"

# Array of indices to create
[[indices]]
name = "green"
edges_table = "czedges" # graph table
polygon_table = "planet_osm_polygon"  # Table from Pgsql layout
   with polygons
index_table = "czedges_green_index" # new table with index values
# [[indices]]
# ...
```

Configuration file

Both routing_preprocessor 5.3.4 and routing_server 5.3.3 use the same config file structure. The idea is that routing_preprocessor runs any necessary algorithm preprocessing specified in a config file so that routing_server can start server based on the config file. Configuration file structure is illustrated below:

```
[database]
name = "dbname"
user = "username"
password = "password"
host = "127.0.0.1"
port = "5432"

[algorithm]
name = "ch" # another option is "dijkstra"
base_graph_table = "czedges"
mode = "static_profile" # the other option is "dynamic_profile"
[algorithm.parameters] # parameters for Contraction Hierarchies
   preprocessing
hop_count = 5
edge_difference = 190
deleted_neighbours = 120
space_size = 0

[preferences] # base index specification - the length values of
   other indices in a profile are based on this one
base_index = "length"
base_index_table = "czedges_length_index" # table with index
   values
```

```
[[preferences.indices]]
name = "green" # also possible "peak", "road_type"
table_name = "czedges_green_index" # table with index values
importance = [ 0.0, -0.5, -1.0 ] # possible weights of
   preferences in a profile
display_name = "Green areas"
display_importance = [ "disabled", "prefer green areas 50%", "
   prefer green areas 100%" ] # What is shown in client to use
```

The most important options to configure are routing algorithms, profile modes (see Section 2.2) and what preferences to use. In Static Profile mode, profiles are created for every combination of weights among specified preferences. Preprocessing is then run for one road graph with each profile providing edge lengths. In Dynamic Profile mode, preprocessing runs only for one profile (it consists of all preferences with weights on index 0).

If preference has importance (i.e. weight) options negative, it means that its meaning is inverted. For example, if an index prefers green areas, then inverted means that green areas are avoided.

Run preprocessing of routing algorithms

Run routing_preprocessor 5.3.4 to run preprocessing of routing algorithms:

```
./algorithm_preprocessing --algorithm {config_path.toml}
```

Run routing server

Run routing_server 5.3.3 to start backend server:

```
./algorithm_preprocessing {config_path.toml}
```

The routing server runs on port 18080. Any routing requests must be forwarded from Apache server to the backend routing server. The following should be added to the configuration file of Apache in element VirtualHost:

```
/etc/apache2/sites-available/maps.conf:

ProxyPreserveHost On
ProxyPass /route http://127.0.0.1:18080/route
ProxyPassReverse /route http://127.0.0.1:18080/route

ProxyPass /profile_preferences http://127.0.0.1:18080/
   profile_preferences
ProxyPassReverse /profile_preferences http://127.0.0.1:18080/
   profile_preferences
```

C.2 Tile server

A detailed tutorial how to set up the whole tile server on Ubuntu can be found below. Apart from specific package managers for Ubuntu a lot is applicable in general:

```
switch2osm.org/serving-tiles/manually-building-a-tile-server
   -20-04-lts (delete any spaces around - characters)
```

Additional information about installing Apache module mod_tile [20] and OSM Carto [23] can be found at:

```
github.com/openstreetmap/mod_tile
github.com/gravitystorm/openstreetmap-carto/blob/master/INSTALL.
    md
```

Note that the stylesheet produced by OSM Carto must be adjusted by swapping y_0=0 on line 3 for y_0=0.0.

C.3 Geocoding server

Geocoding server is entirely handled by Nominatim [15]. See its installation page for full guide on how to build the server:

```
nominatim.org/release-docs/latest/admin/Installation/
```

C.4 Client Application

Client application source files are in directory frontend. We use Yarn [48] package manager. Client application can be set up by running:

```
yarn // in frontend directory
yarn build // creates optimized build
```

Files are statically served by Apache; therefore, add the following to the Apache configuration file between element VirtualHost:

```
/etc/apache2/sites-available/maps.conf:

DocumentRoot {path/to/build/folder}

<Directory {path/to/build/folder}>
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>
```