Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Specification of Trajectories and Learning of User Preferences

Patrick Singer

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	M. Sc. Florian Barth
Commenced:	May 2, 2019
Completed:	October 31, 2019

Abstract

This bachelor thesis provides an implementation that solves the problem of learning individual route preferences of drivers, focusing on cycling routes and metrics for cyclists. Currently, many routing services consider only distance or travel time when calculating a route between some source and target, but there may be many other criteria that a user has in mind when planning a route, such as height ascent, route landscape, noise exposure along the route, suitability for cyclists and many more. Specifying each of these criteria is a non-trivial, unintuitive and time-consuming task for a user. The aim of this bachelor thesis is therefore to take the task of specifying such a preference out of the user's responsibility by automating the process. We present an algorithm that deduces a user preference from a given path and splits the path if no such preference can be found for the entire length. In addition, we provide an implementation utilizing this algorithm, which features an intuitive, state-of-the-art front end that allows a cyclist to specify his personal routes on a map, and a back end that calculates the preferences and handles user administration.

Contents

List of Figures

List of Tables

1 Introduction

Cyclists choose routes based on a wide variety of different personal preferences. Some may choose the quickest route possible, while others also try to avoid large gradients, busy roads, or roads at all. Conventional routing services like Google maps mostly try to optimize traveling distance or traveling time, but there is no way to add a user's personal criteria to the mix. Some services allow to set some values in a yes-or-no fashion, like Google or Bing maps' option to avoid ferries or tolled roads, but these on-and-off settings are not sufficient to capture the whole range of opinions on optimal paths that users might have. Personalized routing is one way of respecting the different criteria while still achieving relatively fast query times even for many criteria [FLS16].

The personalized route planning problem can be described as follows: Let $G(V, E)$ be a street network with a d-dimensional cost vector $c(e) \in \mathbb{R}^d$ for each edge $e \in E$. For example we could choose c_1 to be distance, c_2 suitability for cyclists, c_3 height ascent and so on. A user query now consists of a source and target $s, t \in V$ and a non-negative weight vector $\alpha = (\alpha_1, \alpha_2, ..., \alpha_d)^T$. The values of α express how important each metric in the cost vector c is for the user. The goal is to compute the path π from s to t in G, which minimizes the α-dependent path costs $\Sigma_{e \in \pi} \alpha^T c(e)$, i.e. the optimal path according to the parameters provided by the user.

Specifying this α, which will be called *preference* throughout this thesis, is not an easy task for a user. Expressing how much more important a fast arrival is compared to for example a route's scenicness is a non-trivial task, not to mention the countless compromises that have to be made when considering a wide range of different criteria. The application presented in this thesis intends to help users find these weights. Using the front end of the software, a user is able to precisely lay out a cycle route by placing waypoints on a map. The software will show the currently specified path to the user by calculating shortest paths connecting these waypoints using some initial α, which gives the user an idea of the route he is constructing. When the user has laid out his cycle route, he can prompt the software to calculate an individual preference of the given route. Our application will then greedily split the route into subpaths, finding preferences for which these subpaths become optimal. The resulting path split is displayed to the user by coloring his specified route in the front end, each colored subpath referring to one found preference. Should the user find a particular weight vector very interesting, he can save the preference in his profile and set it as the used α for calculating shortest paths between future waypoints.

Contribution

This bachelor thesis provides the following aspects and ideas for the problem of deducing user preferences:

- We present a full stack implementation which allows a user to input real-world data to deduce driving preferences
- If we can not find a single preference explaining the entire route, an algorithm is presented, which greedily splits the route in subpaths, each of which being explained by a preference

We test the applicability of our software for real-world usage with a case study, using routes that have been driven by a test person, whose cycling preferences are known to us.

This bachelor thesis is structured as follows. In Chapter 2, we will give an overview of the related work and introduce preliminaries as well as basic concepts that underlie the thesis. Chapter 3 presents the algorithm, which was developed and used in the implementation to deal with user paths, that we can not entirely explain. Chapter 4 explains the architecture and logic of the implementation for this bachelor thesis. The back end of the software will be discussed in section 4.2, the front end is introduced in section 4.3. In chapter 5, we present the case study for our application, in which we compare the results of our software with the actual decision process behind the routes. We continue with a small runtime evaluation of the algorithm in chapter 6 before we close this work with a conclusion and outlook in chapter 7.

2 Preliminaries

In this chapter, we aim to lay down the theoretical basis of this thesis, introducing the fundamental concepts on which this project is based. We begin with an introduction to the underlying paper where we present the problem which is the subject of this work and the algorithm proposed by the authors to solve this problem. Next, we present the graph representation used for the implementation of our software and also give a refresher on the concept of graph contraction hierarchies. We close the chapter with a section covering the third party libraries that we used to generate the graph files, which our software makes use of to construct the data structure.

2.1 Underlying Work

This bachelor thesis builds on the ideas presented in [FLS16]. In this paper, the authors have proposed an algorithm that deduces a preference based on a set of routes previously driven by the user. They achieve this by setting up a linear program (LP) formulation of the problem that makes use of a Dijkstra-based separation oracle. The authors tested their concept by running experiments, in which they generated random preferences to simulate user behavior and then compared the results of the algorithm with said preferences. Their experiments showed that the learned preferences reflected the users definition of an optimal route very well.

2.1.1 Notation

The road network is assumed to be a directed graph $G(V, E)$ with $|V| = n$ nodes and $|E| = m$ edges. Since we take multiple metrics into account, edge costs are no longer scalar but vectors $c : E \longrightarrow \mathbb{R}^n_+$. A path from some source s to some target t in G (which does not have to be optimal with respect to any criterion) is denoted as $p(s, t)$. For a given preference α, we refer to the graph G with now scalar edge costs $\alpha^T c(e)$ as G_α. The optimal path for a source s and target t for a given α in G_α is referred to as $\pi(s, t, \alpha)$. Given a path p in G_α, we define the total path costs as $c(p, \alpha) := \Sigma_{e\in p}\alpha^T c(e)$. The unweighted path costs in each dimension $i = 1, ..., d$ are referred to as $c_i(p) := \Sigma_{e\in p} c_i(e)$.

As an example let us consider a scenario where $d = 2$. Given a path p, the value $c_1(p) = 2$ might be the total distance in kilometers, while $c_2(p) = 6$ could be the total travel time in minutes. If we set the preference $\alpha = (0.5, 0.5)$ we get $c(p, (0.5, 0.5)) = 1 + 3 = 4$ as the total path cost.

It can be assumed that all a_i are in the range of $[0, 1]$ and that $\Sigma^d_{i=1}\alpha_i = 1$. If this should be not the case for any preference vector, we simply substitute α with α', which normalizes the values in α like so: $\alpha'_i = \alpha_i/\Sigma^d_{j=0}\alpha_j$. This substitution does not change the ratio between any of the weights and therefore $\pi(s, t, \alpha) = \pi(s, t, \alpha')$ for any choice of s and t.

2.1.2 Problem Definition

The authors of [FLS16] formalize the problem of learning user preferences based on previously driven routes as follows: Given a set of paths P, compute a preference α for which P is preferentially feasible. A path p from s to t is called preferentially feasible if there exists a preference α, such that $p = \pi(s, t, \alpha)$. A set of paths P is called preferentially feasible if there exists a preference α such that $\forall p_i \in P : p = \pi(s_i, t_i, \alpha)$, i.e. all paths in P are preferentially feasible for the same α.

2.1.3 LP-Formulation

To determine an α which makes a path preferentially feasible, the authors present an LP-Formulation with variables $\alpha_1, ..., \alpha_d$. Our model suggests that all α_i are non-negative and in sum equal 1. These properties lead to the following initial constraints for our LP.

$$\begin{gathered}\alpha_1 \geq 0 \\ \alpha_2 \geq 0 \\ ... \\ \alpha_d \geq 0 \\ \alpha_1 + \alpha_2 + ... + \alpha_n = 1\end{gathered}$$

The procedure now is to solve the LP to optimality and then check if the returned α makes P preferentially feasible. If yes, we are done and can return α, otherwise we identify violated constraints and add them to the LP, which removes this α from the set of possible solutions, then solve the LP again, check for preferential feasibility with the new α, and so on. In our case, a constraint is violated if there exists a path $p \in P$, which is not optimal for the current α. So $p(s, t) \neq \pi(s, t, \alpha)$ or $c(p, \alpha) > c(\pi, \alpha)$. This can quickly be checked with a Dijkstra run from s to t with preference α. If the returned path has a lower cost than p, we know that p is not the optimal path for the current α, and therefore have to exclude it from the possible LP solutions. We accomplish this by adding the following constraint to the LP:

$$\Sigma_{i=1}^{d}(c_i(p) - c_i(\pi)) \cdot \alpha_i \leq 0$$

2.1.4 Algorithm

The algorithm that the authors of [FLS16] propose to deduce a preference from a set of user routes is shown in Figure 2.1. The procedure starts by setting up the initial LP, and then solving it to receive the first α candidate. They then proceed to loop over all user paths, checking for each $p \in P$ if the current α explains p, and add the respective constraint to the LP should this not be the case. If all paths in P are explained by the current α, the algorithm terminates and returns α, otherwise the updated LP is solved again and we continue with line 7.

```
Algorithm 1: Preference Estimator
input  : path set $P$, network $G(V, E)$, cost vectors $c : E \to \mathbb{R}^d_+$
output: feasible preference $\alpha$
1  begin
       /* initialize LP                                              */
2      LP.add_constraint($\sum_{i=1}^{d} \alpha_i = 1$);
3      LP.add_constraint($\alpha_1 \geq 0$);
4      ...
5      LP.add_constraint($\alpha_d \geq 0$);
       /* get feasible $\alpha \in \mathbb{R}^d$                      */
6      $\alpha$ = LP.solve();
       /* check and refine                                           */
7      while true do
8          all_explained = true ;
9          for $p_k \in P$ do
10             $\pi$ =Dijkstra($G, s_k, t_k, \alpha$);
11             if $c(p_k, \alpha) > c(\pi, \alpha)$ then
                   /* path $p_k$ not explained by current $\alpha$      */
12                 all_explained = false;
                   /* add constraint to make current $\alpha$ infeasible */
13                 LP.add_constraint($\sum_{i=1}^{d}(c_i(p_k) - c_i(\pi)) \cdot \alpha_i \leq 0$);
14         if all_explained then
15             break;
16         else
17             $\alpha$ = LP.solve();
18     return $\alpha$;
```

Figure 2.1: The Algorithm presented in [FLS16]

2.2 Graph Representation

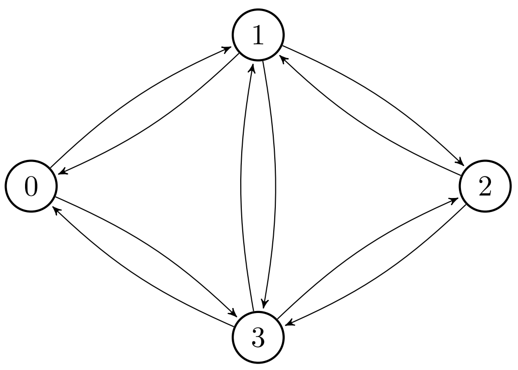
The graph representation we use for this project can be found in chapter 8.2 of [MS08]. Given a graph $G(V, E)$ with $|V| = n$ and $|E| = m$, the general idea is to store the edges leaving any node in a separate array. If we have a static graph that does not change, which is the case here, we can concatenate these arrays into a single edge array F. An additional array G stores the starting positions of the edges for each node, i.e. for every node $v \in V$, $G[v]$ holds the index in F of the first outgoing edge of node v. All outgoing edges of node v are now easily accessible as $F[G[v]], ..., F[G[v + 1] - 1]$. Adding a dummy entry $G[n] = m$ is recommended and ensures that the properties of this data structure also hold for node $n - 1$.

Used Metrics

In the course of this bachelor thesis, we work with the following four different edge metrics

Distance Distance is of course one of the most obvious metrics. It denotes the edge distance in meters. The data for this metric stems from the OpenStreetMap (OSM) project, which provides open-source geodata for countless regions throughout the entire world [Gmb18].

Unit The unit metric is a constant which has the value 1 for every edge.



F edge array	
0	1
1	3
2	0
3	2
4	3
5	1
6	3
7	0
8	1
9	2

G offset array	
0	0
1	2
2	5
3	7
4	10

Figure 2.2: Bidirected Graph and its representation as adjacency array. Edge array F holds the target nodes of the respective edges

Height The height of an edge is its positive ascent in meters. Data for this metric is extracted from the files of the NASA Shuttle Radar Topography Mission [FRC+07]. Downloading this data is free of charge, but requires a user account on the website. Since only positive altitude differences are taken into account, the value of this metric is high for routes that run up and down a lot.

Unsuitability Weighted Distance The last metric, which in the following will be abbreviated with *UnsuitDist*, combines two different metrics. The unsuitability of an edge is a constant, which is dependent on the type of road that corresponds to the edge. Road types are also extracted from the OSM data. Values for this metric are defined hierarchically, Table 2.1 shows some of the values, which are used throughout this thesis. Roads that have very high traffic load like primary roads are penalized by this metric, while bike paths are assigned the lowest value and are therefore preferable. The reason behind weighting the distance of an edge with its unsuitability is OSM's placement of nodes in a path. For paths that are mostly straight, we have much less intermediate waypoints and therefore less edges that make up the path compared to a curvy way. Figure 2.3 shows an example, where the path that we normally would consider to be not ideal for cyclists would be preferred over the one that we would expect to be better suited. In practice we encounter this problem when we are dealing with cycle paths that run parallel to a fairly straight stretch of a car road.

primary	5.0
secondary	4.0
tertiary	3.0
residential	2.0
living street	1.0
track	1.0
cycleway	0.5

Table 2.1: Unsuitability Values for some example roads

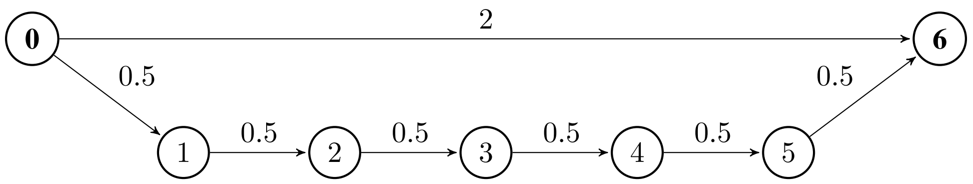


Figure 2.3: Problem with solely using a fixed number for any edge length. We travel from 0 to 6. Taking the upper path would result in an unsuitability cost of 2, while the lower path results in a cost of 3, which is suboptimal.

2.3 Contraction Hierarchies

Contraction Hierarchies are a means of contracting a graph $G(V, E)$ by augmenting the existing set of edges E by a set of shortcuts S to obtain an augmented graph $G'(V, E')$. Performance of shortest path queries benefits significantly from this pre-processing step. The shortcut graph is created by first assigning each node in the original graph a value, called *priority*, and then contracting the nodes in ascending priority. In the contraction step of a node $v \in V$, we have a look at all neighboring nodes of v, and add a shortcut edge between two neighbors $u, w \in V$ if and only if uvw is a shortest path, which can be checked with a Dijkstra run from u to w. If this is the case, we obtain the cost of the shortcut edge by summing the costs of the two edges connecting u and w: $c(u, v) + c(v, w)$ [GSSD08].

When calculating a shortest path in G' with source s and target t, $s, t \in V$, we run a bidirectional Dijkstra search. In the forward search originating from s, we only relax edges whose target nodes have a *higher* priority than the source nodes. Correspondingly, in the backward search originating from t, we only relax edges whose target nodes have a *lower* priority than the source nodes.

Multi-Criteria CH Multi-Criteria Contraction Hierarchies consider cost vectors with a dimension greater than 1. This changes the condition, under which we add a shortcut between u and w when contracting v. Now, a shortcut between the two nodes is added if and only if there *exists a preference* α, which makes the shortcut an optimal path between u and w [FLS17]. In Figure 2.4, we illustrate how these changed conditions may lead to multiple shortcuts being added between two nodes. In

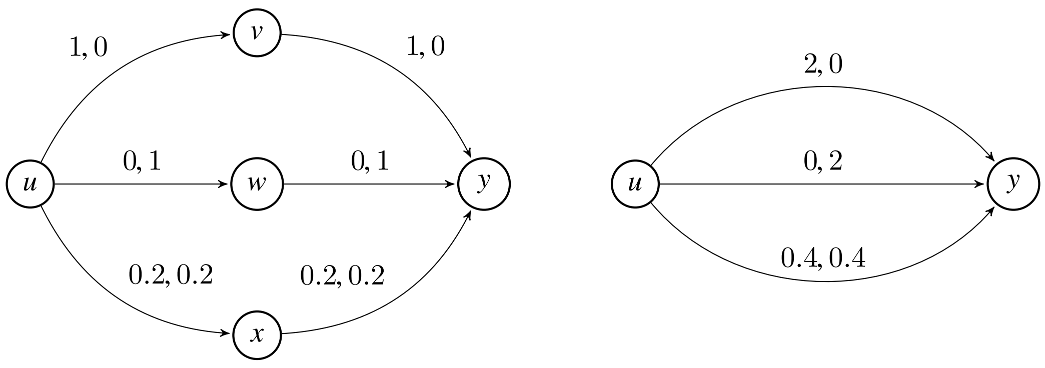


Figure 2.4: A multi-criteria contraction step with $d = 2$. Original graph is on the left, contracted graph on the right. Processed nodes are v, w and x.

this case we add three shortcuts between the nodes u and y because we can find an α that encodes an optimal path from u to y for each contracted node. We could choose the α vectors like so: $(0, 1)$ for v, $(1, 0)$ for w and $(0.5, 0.5)$ for x. The single criteria contraction step on the other hand would have only added one shortcut to the given graph, as the shortest path from u to y would have to be unique.

2.4 Used Software

In this last section of the preliminaries for this thesis, we give an introduction to the third party applications that were used to generate a graph representation for our implementation. Both pieces of software are publicly available on GitHub. They operate on a graph representation, which specifies all nodes of the graph, one node per line, followed by all edges of the graph, one edge per line. Nodes are represented with the following pattern:

```
<internal id> <osmid> <lat> <lng> <height> <ch level>
e.g. 0 125799 53.074 8.786 3.4728 0
```

Edges follow a different pattern:

```
<source id> <target id> <cost 1> ... <cost d> <replaced edge 1> <replaced edge 2>
e.g. 22 104289 6 2 1.5 -1 -1
```

for an edge that is not a shortcut and $d = 3$

2.4.1 pbf-extractor

pbf-extractor is a tool to extract graph files from OpenStreetMap and SRTM data. The software extracts a graph that contains data on the distance, height ascent and bicycle unsuitability for cyclists, covering two of the four metrics in our model [Bar18b]. We forked the repository to adjust the code for the requirements of this thesis [Sin19a]. To cover all our criteria, we extended the set of extracted metrics by the unit metric, and added a new metric *UnsuitDist*, which merged the values of distance and unsuitability. In addition, we replaced the preset car edge filter with a bicycle edge filter in order to sort out motorways and motorway connections, but to include cycle paths and dirt roads.

2.4.2 multi-ch-constructor

multi-ch-constructor is a tool that creates multi-criteria contraction hierarchy graphs [Bar18a]. The software implements the core concepts of [FLS17], where the authors present an LP-based approach to efficiently decide the necessity of a shortcut when contracting a graph with multi-dimensional cost vectors. We use the tool to contract the graphs that have been extracted by the *pbf-extractor*, and use these graphs for our implementation.

3 Path Splitting

In [FLS16], the authors mention that there are paths for which no preference α exists such that the path is optimal in G_α. A path with costs $(5, 4)$ will never be optimal if there exists an alternative path between the same source and target with costs $(3, 2)$. In practice, this can only happen if $a)$ the user is not aware that his path is suboptimal, or $b)$ the path optimizes some metric(s), which are not represented by our model. We found that the amount of infeasible paths that came up in the real-world application was so much greater compared to the ones that could be explained, that we decided to implement the authors' suggestion to split such paths into smaller subpaths, which in turn can be explained. In the following, we will introduce the algorithm that we used in our implementation to split user paths and show that the calculated split always contains the minimal amount of subpaths.

We formulate the following requirement for our algorithm: Given a path p, divide p into a set of preferentially feasible subpaths, so that the length of each subpath is maximized. As any single edge always represents an optimal path for an α giving full weight to the unit metric, we can split any given path at every node to receive a valid path split, which means there is no path which we can not split in the just defined manner.

Our algorithm consists of two main loops. The outer loop iterates through the input path, setting the node $start$, which denotes the first node of the subpath we intend to find. In the inner loop, we utilize a binary search to find the longest feasible subpath originating from node *start*. Our search window is defined by the two variables low and $high$. Note, that line 12 calls the *Preference Estimator* introduced in chapter 2.1.2, but here we only want to explain a single path. If the estimator returns an α, we adjust our search window in order to find a bigger subpath, else we search for a smaller subpath. Since we explore more to the right every time we find a preference, the algorithm will indeed find the longest feasible subpath, until our search window is empty, i.e. $low \neq high$. When the inner loop breaks, we begin searching for the next subpath, whose starting node is the last node of the just found subpath.

Algorithm 3.1 Path Splitting

```
 1: procedure splitPath($path$)
 2:     $preferences \leftarrow \emptyset$
 3:     $cuts \leftarrow \emptyset$
 4:     $start \leftarrow 0$
 5:     while $start \neq path.length - 1$ do
 6:         $low \leftarrow start$
 7:         $high \leftarrow path.length$
 8:         $bestPref \leftarrow null$
 9:         $bestCut \leftarrow 0$
10:         while $true$ do
11:             $m \leftarrow \lfloor \frac{low+high}{2} \rfloor$
12:             $pref \leftarrow calcPreference(path[start..m])$
13:             if $pref$ then
14:                 $low \leftarrow m + 1$
15:                 $bestPref \leftarrow pref$
16:                 $bestCut \leftarrow m$
17:             else
18:                 $high \leftarrow m$
19:             end if
20:             if $low = high$ then
21:                 add $bestPref$ to $preferences$
22:                 add $bestCut$ to $cuts$
23:                 break
24:             end if
25:         end while
26:         $start \leftarrow bestCut$
27:     end while
28:     return $(preferences, cuts)$
29: end procedure
```

We can show that this algorithm will always find the minimal amount of explainable subpaths in a route. For this, let's consider a path p that our algorithm splits in three different subpaths s_1, s_2, s_3 with respective preferences $\alpha_1, \alpha_2, \alpha_3$. We now assume there exists a minimum amount of two subpaths s_1^* and s_2^* for p, which are explained by α_1^* and α_2^*, respectively. We then lead this assumption to a contradiction, therefore showing that the proposed algorithm is indeed optimal with respect to finding the minimal amount of preferentially feasible subpaths in a given route. There are three different cases we have to take into consideration.

Case 1 $|s_1^*| < |s_1|$

For case 1 we assume that the first subpath of the optimal split is shorter than the first subpath of the split returned by the algorithm, which means that $s_2 \cup s_3 \subset s_2^*$. Since it holds that every subpath of a feasible path can be explained with the same α, and we know that s_2^* is explained by α_2^*, it follows that $s_2 \cup s_3$ is also explained by α_2^*. As our algorithm always finds the longest feasible subpath, it would have found $s_2 \cup s_3$ and therefore returned two subpaths.

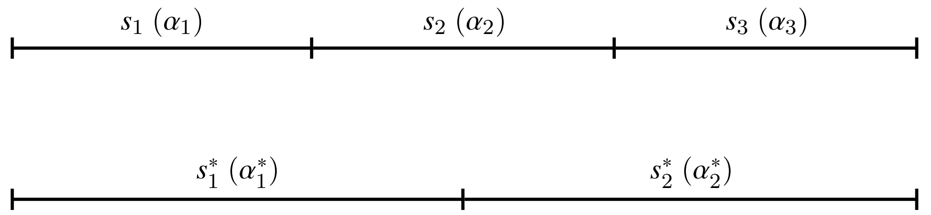


Figure 3.1: Setup for our proof, the upper split represents the assumed output of our algorithm, the lower one displays the optimal split of the path.

Case 2 $|s_1^*| = |s_1|$

If the first subpaths of both splits are of equal length, we know that $s_2^* = s_2 \cup s_3$. Since α_2^* explains s_2^*, it follows that α_2^* also explains $s_2 \cup s_3$, i.e. $s_2 \cup s_3$ is preferentially feasible. We know that the subroutine *calc_preference* will find an α, if the given subpath is feasible, which means it would have found an α for $s_2 \cup s_3$. Therefore, the algorithm would have been able to explain s_2^*, splitting p in two subpaths.

Case 3 $|s_1^*| > |s_1|$:

In the last case we assume that the first subpath of the optimal path split is longer than the first subpath returned by our algorithm. Since we utilize a greedy subpath search, we will always find the longest feasible subpath possible, which contradicts our premise that s_1^* is preferentially feasible, but longer than s_1.

4 Implementation

This chapter aims to introduce the implementation of the software, which was created for this thesis. We begin with an introduction to the architecture of the application, and will then proceed to present the different aspects and implementation details of the back end as well as the front end. Furthermore, used libraries and frameworks throughout the application will be mentioned and described. The source code for both the front end and the back end is publicly available on GitHub [Sin19c] [Sin19b].

4.1 Architecture

The application is based on the client-server model and features a server-side back end and a client-side front end. The main functionality of the back end lies in implementing the business logic and the algorithms which were discussed in the previous chapters. It also provides the necessary endpoints which are accessed by the front end. Communication between server and client is established via HTTP requests originating from the client. The front end itself implements no business logic and its sole purpose is providing a user interface to input and display the data that is sent to and received from the server.

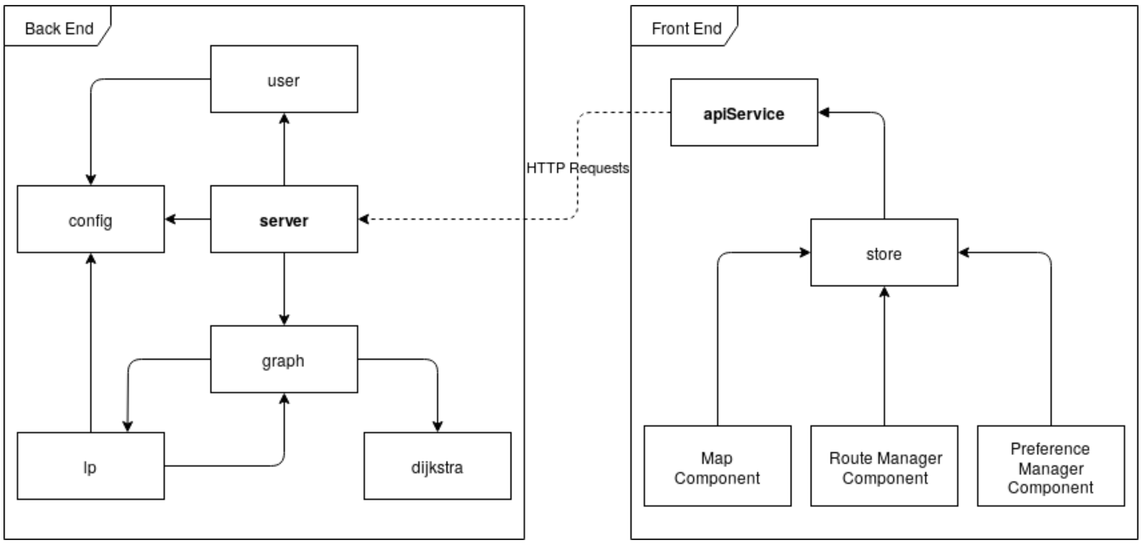


Figure 4.1: Architectural diagram

Figure 4.1 shows an architectural diagram of the entire software. Both client and server contain one module which handles HTTP communication. The *server* module of the back end implements a RESTful web server [RR07], which responds to the requests issued by the *apiService* of the front end. The back end of the application consists of various modules, each of which having its own area of responsibility. At the heart of these modules lies the *server* module, which apart from providing HTTP endpoints also handles the processing of requests by invoking functionality in the other modules. Communication between components in the front end originates from the three Vue components, which will be introduced in section 4.3.1 and represent the DOM of the front end. The *store* module contains and manages the application state of the client and additionally provides an abstraction layer between the DOM and the *apiService*.

4.2 Back End

The back end for the application implements the HTTP server and the business logic for the application. This includes user management, construction of the graph data structure and answering of requests related to the graph. It is written in the Rust programming language. Rust is a systems programming language, which was first introduced in July 2010. The language is statically and strongly typed and focuses on safety, especially safe concurrency [HP10].

In this section we mainly cover the different modules of the back end and mention used libraries and frameworks when necessary. We begin though, with a small paragraph providing execution instructions for the software.

Starting the Back End

Rust applications contain an *src* folder, which holds the source code for the application and is located in the root directory of the project. This folder contains the file *main.rs*, whose *main* function is the entry point of the application. Starting the software is done by running the command `cargo run --release [path/to/graph/file]` in the root folder of the project. This will build the software and run it automatically after completing the build. The command receives one parameter, which is the URL to the file containing the graph representation introduced in chapter 2.2. If the path is provided, the *main* function will start with initiating the parsing of the graph file and construction of the graph data structure. When this is done, the HTTP server will be set up and started, and the application is ready for use.

4.2.1 Modules

Rust enables the organization and scoping of functionality in an implementation with the help of modules. They are a way of grouping code for better readability and reusability, while also controlling the privacy of items with the two keywords *public* and *private*. The back end of the implementation for this thesis contains six different modules, which will be explained in the following sections.

config

The *config* module can be used by other modules to retrieve information about the configuration of the application, for example the port that the server should use. The values are stored in a file *config.toml*, which lies in the root directory of the project. Table 4.1 shows the initially set values for the app configuration.

The module implements a Singleton pattern, i.e. there will only ever be one instance of the *config* object, which is stored in a static variable. To request the configuration, a module calls the *get_config* method. When this function is called for the first time, the *config* module will parse the *config.toml* and create an object, which is stored in said static variable. All following calls to this function will return a reference to this object, which can then be used to access the configuration data.

Key	Value
port	8000
database_path	database
edge_cost_tags	[Distance, Unit, Height, UnsuitDist]
initial_pref	[1, 0, 0, 0]

Table 4.1: Initial configuration for the software

user

All logic regarding user management, e.g. creation of users or updating the routes of a user is implemented in the *user* module. Every registered user in the back end will be represented by one *User* object, see Figure 4.2, which is created once the user registers himself via the front end of the application.

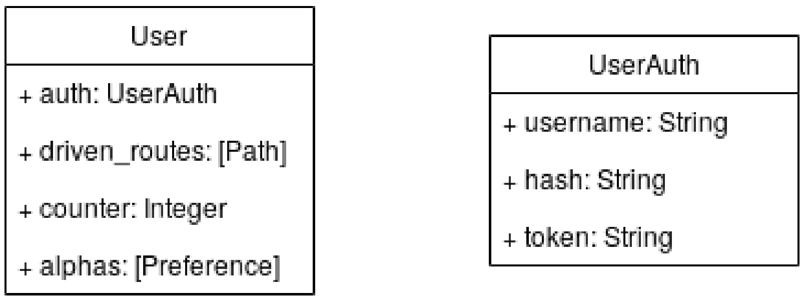


Figure 4.2: *User* and *UserAuth* classes

This object holds information like the driven routes as well as all preferences of the particular user it represents. Furthermore, every user object contains an *auth* object, which stores username, hash and token of the user it belongs to. When a user registers himself, we create a new object and set the username property to the value that was picked by the user. Since we do not want to save sensitive information like a user's password as plain text, we hash the value of the given password and save it to the hash property. When a user logs in and sends his username and password, we check if the hashes of the saved and given passwords match, and additionally if the username matches with the saved value. A successful login will set the token property in the *auth* object of the user and add the token value as payload to the response of the login HTTP request from the front end. This token is then added as a header value to every request from the front end (except logging in and registering), as we do not want to send the user's credentials in every request concerning business logic. The server then uses this token to find the corresponding user state in the database.

server

This module implements the server that handles the HTTP-Requests originating from the front end, e.g. logging in, fetching the user preference or getting a shortest path. It acts as the control center of the application, which means that the module will use the interfaces of the other modules in the project to process requests originating from the front end accordingly. The server was implemented using the *actix-web* library [Tea]. *Actix-web* is a Rust framework that provides functionality to implement an HTTP server. The library allows setting up a server via a builder-like pattern, where the different components are registered one after the other, which creates a concise and easily manageable code.

Using this pattern, we first register the application state for the server. This state holds the graph data structure that we constructed when starting the application, the URL which denotes the file we use to save said state to make it persistent, and finally, an array holding all users for our user management. Secondly, we register a CORS middleware to prevent CORS issues with browsers that call the endpoints of the server. This middleware adds an *allowed origin* header to every response that leaves the server. Next, we register another middleware component, which saves the previously mentioned user state to the respective file, every time the server finishes processing a request. Doing this every time we receive a request will guarantee that the persistent database is always up to date, even if there is a sudden crash of the server. Handlers for the HTTP-Requests are also registered with this builder-pattern, an overview of all the supported requests and routes is shown in Table 4.2. Lastly, we register the handler functions for the different requests. The handlers are split into two separate files in the *server* module: *auth.rs* and *routing.rs*. While the former contains functions that handle user log in and registration, the latter deals with any requests that are related to the topic of routing, e.g. managing user preferences or finding shortest paths. Request handlers may receive the application state as a function parameter, on which they have reading and writing access.

Endpoint	**HTTP Method**	**Description**
tags	GET	Gets the edge cost metrics
preference	GET	Gets user preferences
preference	POST	Sets user preferences
preference/new	POST	Adds user preference with standard values
preference/find	POST	Finds preference for given path
closest	GET	Gets the coordinates of the closest node in the graph to the given *lat* and *lng*
fsp	POST	Gets shortest path from given source and target
routes	GET	Gets all user routes
delete/{id}	POST	Deletes user route with given id
reset	POST	Deletes the entire saved data of the user
login	POST	Logs the user in and creates token
register	POST	Adds a new user

Table 4.2: Provided endpoints and their description

graph

The *graph* module on the one hand handles parsing the graph file and construction of the graph data structure, which is the first thing that is done when the software is executed. Secondly, the module implements operations on said graph data structure like calculating a shortest path given a set of waypoints, or finding the closest node that is represented by the graph given some coordinate. The path splitting algorithm introduced in chapter 3 is also implemented in this module.

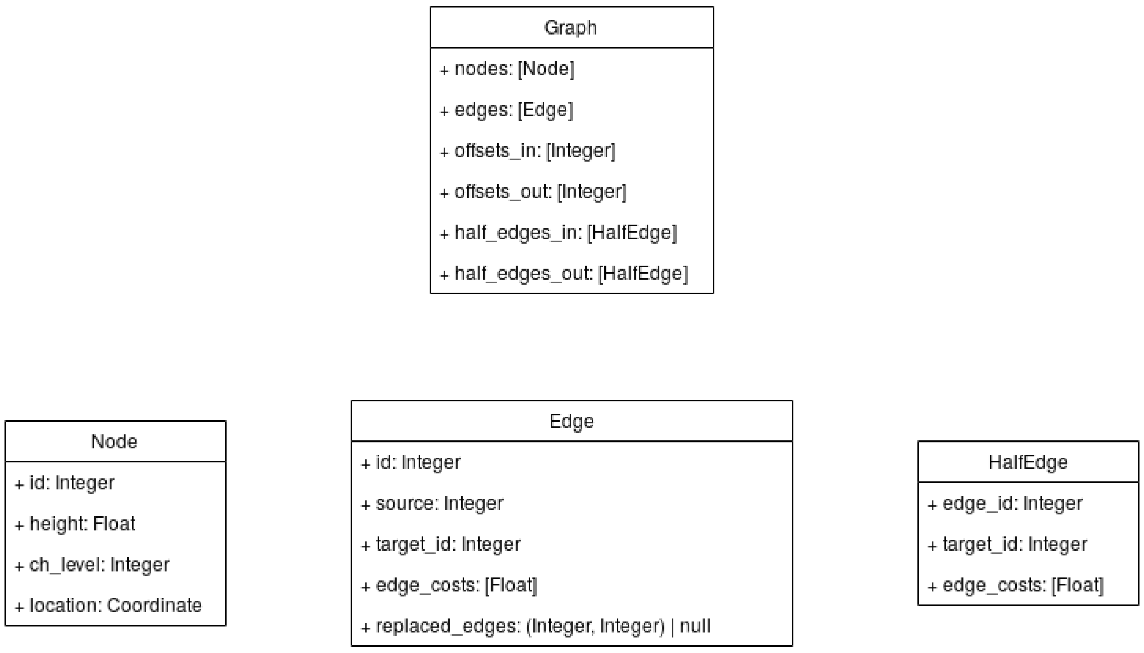


Figure 4.3: *Graph* class and its subclasses *Node*, *Edge* and *HalfEdge*

The result of constructing the graph structure is an object of class *Graph*, shown in Figure 4.3. Apart from the mandatory nodes and edges of the graph, this object contains the offset array graph representation introduced in chapter 2.2. Note, that we have divided every edge of the graph into two *HalfEdges*, which is required to run a bidirectional Dijkstra search on the graph. This also leads to two different offset arrays.

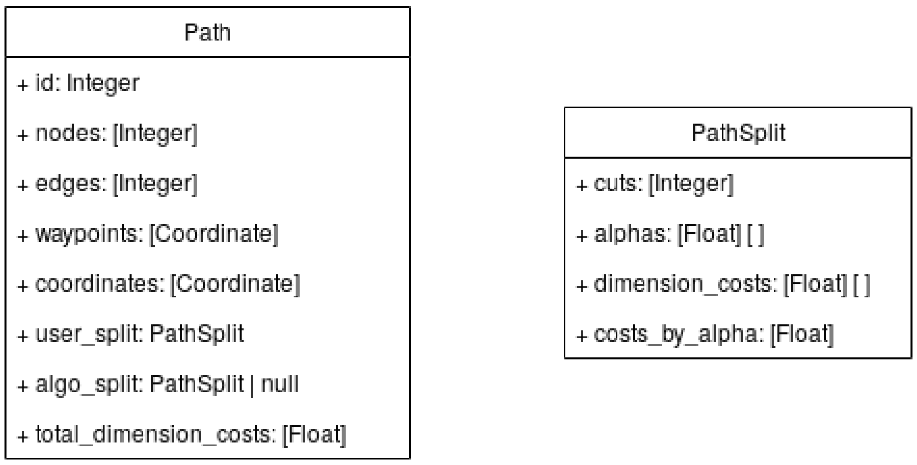

Path
+ id: Integer
+ nodes: [Integer]
+ edges: [Integer]
+ waypoints: [Coordinate]
+ coordinates: [Coordinate]
+ user_split: PathSplit
+ algo_split: PathSplit | null
+ total_dimension_costs: [Float]
PathSplit
+ cuts: [Integer]
+ alphas: [Float] [ ]
+ dimension_costs: [Float] [ ]
+ costs_by_alpha: [Float]

Figure 4.4: *Path* and *PathSplit* classes

The class *Path*, shown in 4.4 defines what path objects look like in our application. Besides the nodes and edges of the path, this object also contains two fields representing path splits. The field *user_split* represents how the user specified his path using the front end. When the path splitting algorithm has calculated a path split for a route, the property *algo_split* of that route will be set accordingly, otherwise it does not hold a value.

dijkstra

The *dijkstra* module implements the functionality to run shortest path queries on the graph data structure. There are two use cases in which we require shortest paths. Firstly, waypoint input of a user in the front end will run a shortest path query, the result of which will be sent back to the front end. Secondly, finding preferences for a given path requires calculating multiple shortest paths. The module implements a bidirectional Dijkstra algorithm on the contraction hierarchy graph, which is ran by calling the *find_path* method, passing a list of waypoints and a preference as parameters. The list of waypoints defines which intermediate points should be taken into consideration when calculating the route, while the preference denotes the weighting of the cost metrics for this route. If the amount of given waypoints is two, we have the case of a standard routing query, with the first waypoint being the source and the second one the target. Should the number of waypoints be greater than two, we know that the user has defined intermediate points which should be part of the resulting path. In this case, we execute multiple Dijkstra runs to compute the shortest path, one for every successive pair of waypoints in the list. For example, if we have waypoints $W = [n_1, n_2, n_3]$, we start two Dijkstra runs, one from n_1 to n_2, and the second one from n_2 to n_3. The results of these subpaths are then concatenated in an array and returned.

lp

The *lp-module* implements setting up the LP required for the *calc_pref*, which is called by our algorithm. To set up the linear program, the module makes use of the library *lp-modeler*, which acts as a wrapper for the GNU Linear Programming Kit (GLPK) [And12]. This library allows defining linear programming problems using familiar Rust syntax [Cav16]. The *lp* module defines a struct *PreferenceEstimator* whose constructor function sets up the basic LP defined in 2.1.3. We can then use an instance of this struct to compute a preference for a given path with source and target id. This function implements the algorithm presented in [FLS16], which was discussed in section 2.1.2.

4.3 Front End

This chapter will explain the front end of the application. Similarly to the previous section, which dealt with the back end, we will start with installation and execution instructions. Afterwards, we explain the basic concepts of the front end framework *Vue.js*, on which the application is based. Then, we will start going over the various components which make up the front end. We begin with the *Map Component* of the application, continue with the *Route Manager* and close the chapter with explaining the functionality of the *Preference Manager*.

Starting the Front End

Before running the application, all necessary dependencies have to be installed. Depending on the preferred package manager, this can be done by running the command `npm install` and `yarn install`, respectively. The package manager will then install the necessary dependencies defined in the file *package.json* to a dedicated folder called *node_modules*, and also create a file *package-lock.json*, which describes the exact dependency tree that was generated. For development purposes, the command `npm run serve` or `yarn run serve` is used to start a server that comes with ”Hot-Module-Replacement”, meaning that any changes to the source files will result in the server reloading only the module that was changed, which saves valuable time when developing [You18]. To build the application for production, one executes `npm run build`. This command will optimize and bundle the software. The resulting JavaScript and CSS files as well as the *index.html* will then be placed in a *dist* folder at the root directory of the project, along with any images and meta files. This folder can then be deployed by an HTTP server to make the application available to users.

4.3.1 Vue

Vue (pronounced ”view”) is an open-source JavaScript library for building user interfaces and single-page applications. A single-page application is a website, where the user experience is not interrupted by repeating calls to the server between successive page calls. Instead of loading the entire new pages from a server, they are dynamically replaced. *Vue* focuses on component composition, meaning applications are built by creating reusable modules, which can then be used by other components. For state management of the application, Vue developers offer a library called *vuex*. Vuex serves as a centralized store for all the components in the application, while enforcing rules that updates on the state are only done in a predictable fashion [You14]. Figure 4.5 shows a simple Vue component, which implements a login functionality. The UI consists of two text input fields and one button to submit the credentials to a server. The file consists of three different parts: *template*, *script* and *style*. Note, that a Vue component does not necessarily have to define all three of these sections. A file only containing the template region for example is also a valid Vue module.

```
<template>
  <div class="red-font">
    <input v-model="username" type="text" />
    <input v-model="password" type="password" />
    <button @click="login">Log In</button>
  </div>
</template>

<script lang="ts">
import Vue from 'vue';
import Component from 'vue-class-component';

@Component
export default class SimpleLogin extends Vue {
  private username = '';
  private password = '';

  private login() {
    console.log(this.username, this.password);
    this.username = '';
    this.password = '';
  }
}
</script>

<style scoped>
.red-font {
  color: 🟥red;
}
</style>
```

Figure 4.5: A Vue component implementing a simple login mask

Template

The *template* region registers the elements in the DOM of this component. It consists of exactly one HTML element, mostly a simple *div*, which contains all the elements needed for this component. Children components from the same project are also registered in this region, as well as components from external libraries. In our example of the login mask we first define the two text input fields.

Each of these inputs features a *v-model* property which is assigned a variable name. Doing this will activate the Vue reactivity system for the text inputs. If we now create two variables in the script code with the same names we used for the *v-model* properties, every input in the text fields will result in the values of the variables being updated automatically. Furthermore, changing the value of any of these variables in the code will result in an update of the value in the respective text input field, which makes Vue's *v-model* a **two-way binding**. Lastly, we define an HTML button, and register a listener for the *click* event of the button. The value of the listener points to a function, which we will also define in the code.

Script

The script section holds all the necessary logic which is needed for the component to work. It defines all used variables and functions referenced in the *template*, handles properties that are passed into the component by a parent, and imports and registers child components, which can then be used in the *template*. Here, we declare the two variables, which we have used for our *v-model* properties in the template region. Additionally, we define the login function, which is called when the *click* event listener of the HTML button is fired. When line 19 is executed, the console will output exactly the string values that the user has put into the text fields. At the end of the method, the values of *username* and *password* will be reset to an empty string, which will also clear the text input fields.

Style

The style section is used to define CSS classes which can be used in the template section. In this case, a simple class is defined, which changes the font in the login mask to have a red color. By default, the style region is public, which means every component can use the classes it defines. To make the classes private, which is recommended by the Vue guidelines, we add a property *scoped* to the HTML tag of the section, as shown in line 26.

4.3.2 User Interface

Figure 4.7 shows the main user interface (UI) of the front end, which provides the functionalities the user needs to input routes, manage routes and update preferences. The user interface is built with the help of the Vuetify front end framework. Clicking on the logout button in the top right corner of the UI will log the user out and redirect him to the login UI, shown in Figure 4.6 on the left. Here, a user can either login using an existing account, or create a new one by clicking the "Register" button in the top right corner, which will lead him to the register UI shown on the right side of the figure. A click on the login button will redirect the user back to the login page.

Vuetify Vuetify is a Material Design Component Framework crafted for Vue [LLC16]. The framework provides a vast range of different components, premade layouts and styling classes which can be used in Vue applications. Usable components range from simple buttons and text input fields over loading animation components to complex data tables. Thanks to the great documentation and vibrant community, which contributes to regular updates, Vuetify significantly simplifies and enhances the process of creating user interfaces.

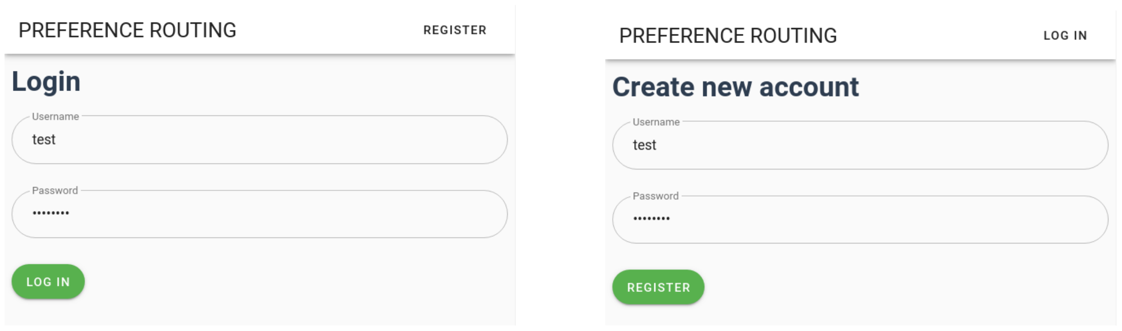
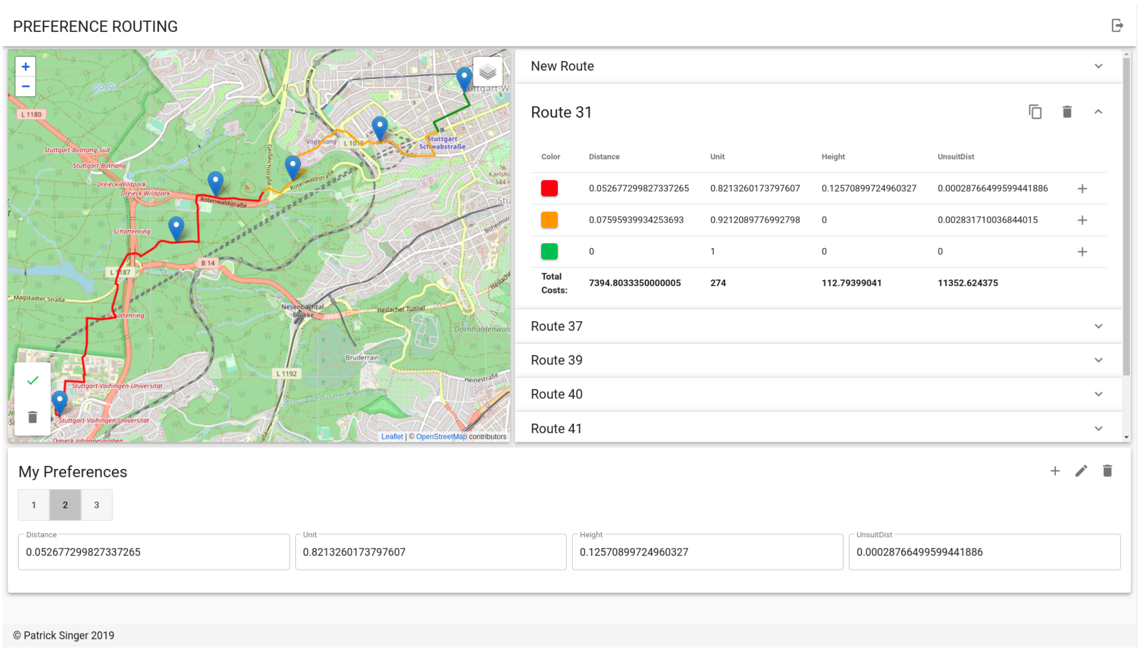


Figure 4.6: Logging In and Registering



Color	Distance	Unit	Height	UnsuitDist
🟥	0.052677299827337265	0.8213260173797607	0.12570899724960327	0.00028766499599441886
🟧	0.07595939934253693	0.9212089776992798	0	0.002831710036844015
🟩	0	1	0	0
Total Costs:	**7394.8033350000005**	**274**	**112.79399041**	**11352.624375**

Distance	Unit	Height	UnsuitDist
0.052677299827337265	0.8213260173797607	0.12570899724960327	0.00028766499599441886

© Patrick Singer 2019

Figure 4.7: View of the main UI

The main UI is built with a grid consisting of two main rows, which make up the whole user interface. The upper row contains everything that the user needs to input and examine his routes and is split into two columns. In the second row the user can find everything related to managing his personal preferences.

Map Component

Most of the user interaction with the application will take place on the map shown in Figure 4.8. The map is implemented using the *vue2-leaflet* framework, which wraps the components of the *leaflet* library, so they can be used in Vue applications in a modular way [Aga10] [Bou16]. We can use these components in the template section of the Vue files, which makes it easy to add event listeners, or add properties to the elements.

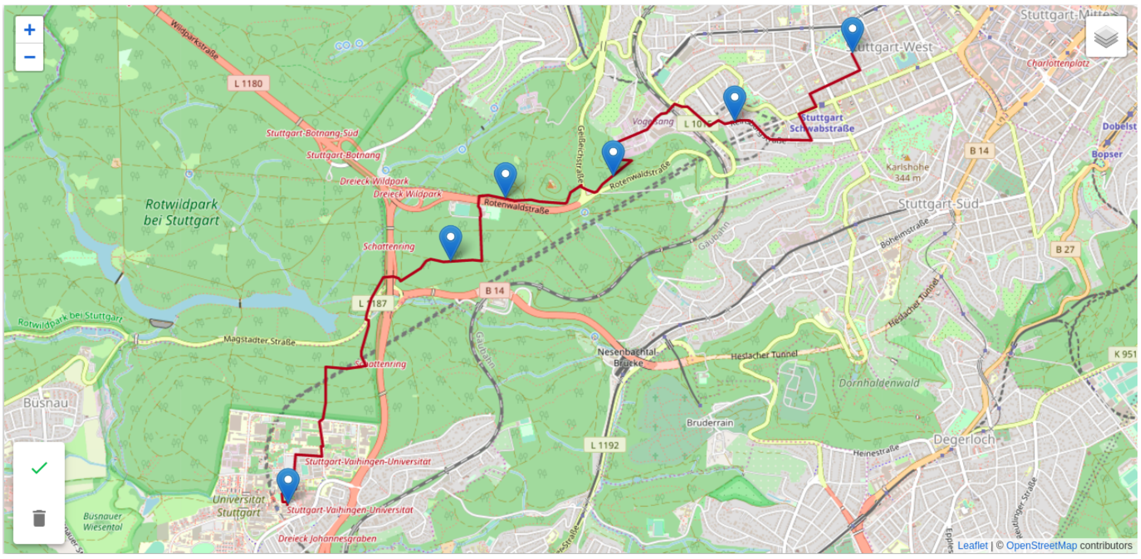

+
−
Rotwildpark bei Stuttgart
Stuttgart-West
Stuttgart Schwabstraße
Stuttgart-Süd
Degerloch
Universität Stuttgart
Stuttgart-Vaihingen-Universität
Busnau
Leaflet | © OpenStreetMap contributors

Figure 4.8: The Map component

Here, a user can add waypoints to his route by left-clicking on any location on the map. Adding a waypoint will set it as the new target, i.e. the ordering of the way points follows the order, in which they were added to the map. When the user adds a waypoint at some location on the map, this very location will most likely not be represented by an existing node in the graph representation, because our data structure only covers nodes on actual roads. Imagine for example a user clicking directly on his house, or the parking lot at his workplace. We therefore issue a request to the back end every time a user places a marker, sending the input coordinates as request payload. The back end then looks for the node that is closest to the given coordinates, and replies with its location. We then move the marker to the location we received from the back end, so that it represents an actual node in our graph. If the user is not satisfied with the position of a waypoint, he can click and hold the marker, to drag it across the map. Dropping it at some location will again update its position to the closest coordinate represented by our graph data structure. Removing a waypoint is done by right-clicking the corresponding marker on the map. Every addition, removal or dragging of waypoints will automatically send a shortest path request to the back end, if the number of markers is greater than two. On response, the front end will display the resulting path on the map with a brown line. Should the back end not be able to find a route, the user will receive a notification. Inserting intermediate waypoints on the route is done by clicking on a section of the route between two markers, and then dragging the resulting marker to the desired location. While the marker is dragged, the front end will repeatedly send shortest path queries to the server and display the resulting routes, which gives users immediate feedback on the impact that the repositioning of the waypoint has.

Removing a shown path from the map, which includes its coordinates and waypoint markers is done by clicking on the button showing a trash can in the lower left corner of the map. When the user has finished laying out his path, pressing the button with the check mark in the same corner will send a request to the back end which will then begin to find a preference for the current route on the map. Upon getting a response, the map will display the different subpaths of the route in different colors, as shown in Figure 4.9. Should the user now add another waypoint to the map, the colored subpaths will disappear again.

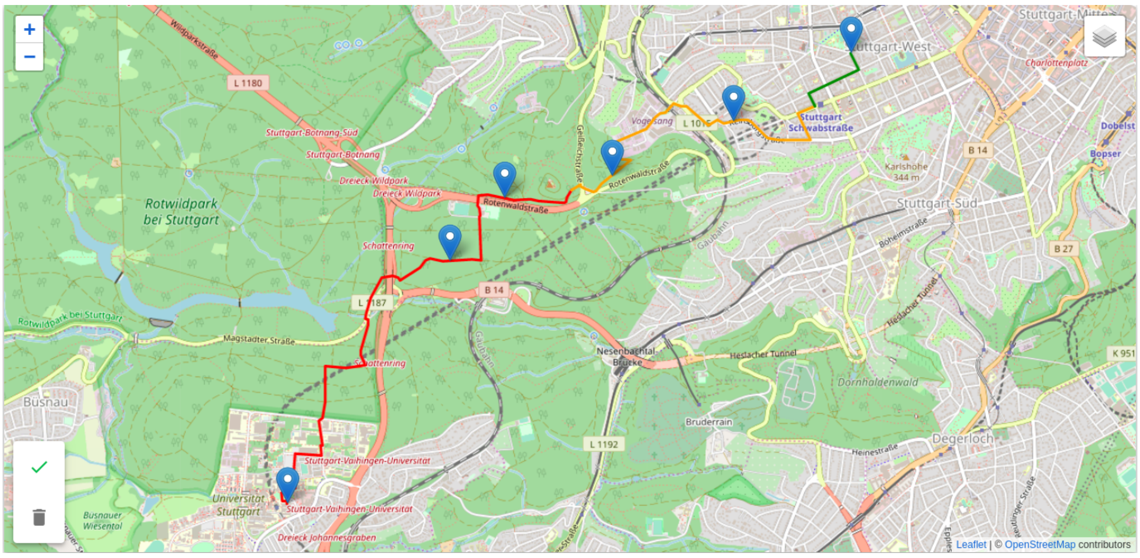
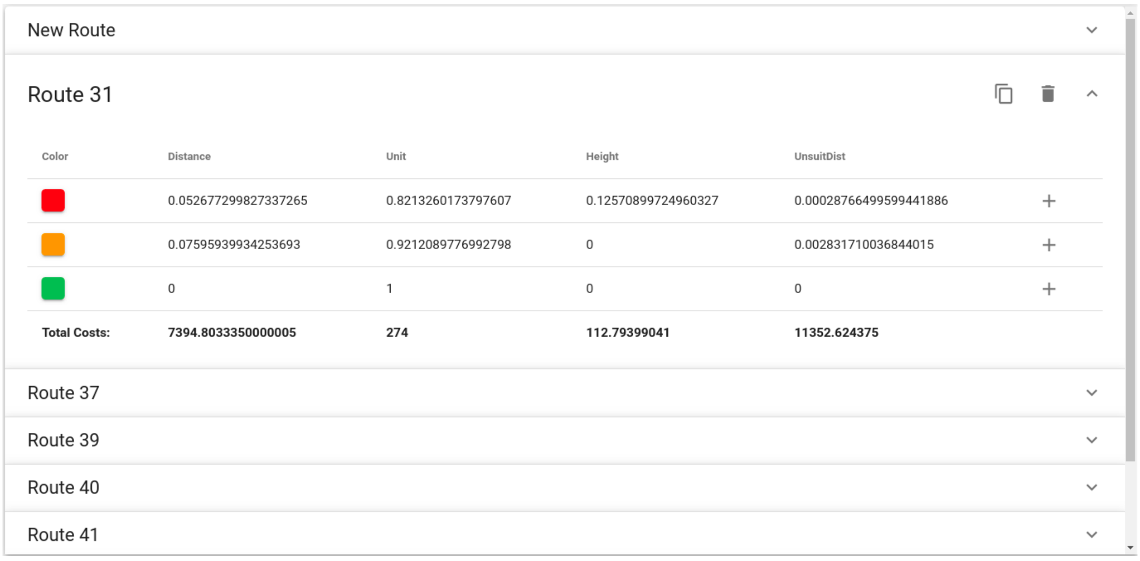


Figure 4.9: An example path split

Route Manager



New Route

Route 31

Color	Distance	Unit	Height	UnsuitDist	
🟥	0.052677299827337265	0.8213260173797607	0.12570899724960327	0.00028766499599441886	+
🟧	0.07595939934253693	0.9212089776992798	0	0.002831710036844015	+
🟩	0	1	0	0	+
Total Costs:	**7394.8033350000005**	**274**	**112.79399041**	**11352.624375**	

Route 37

Route 39

Route 40

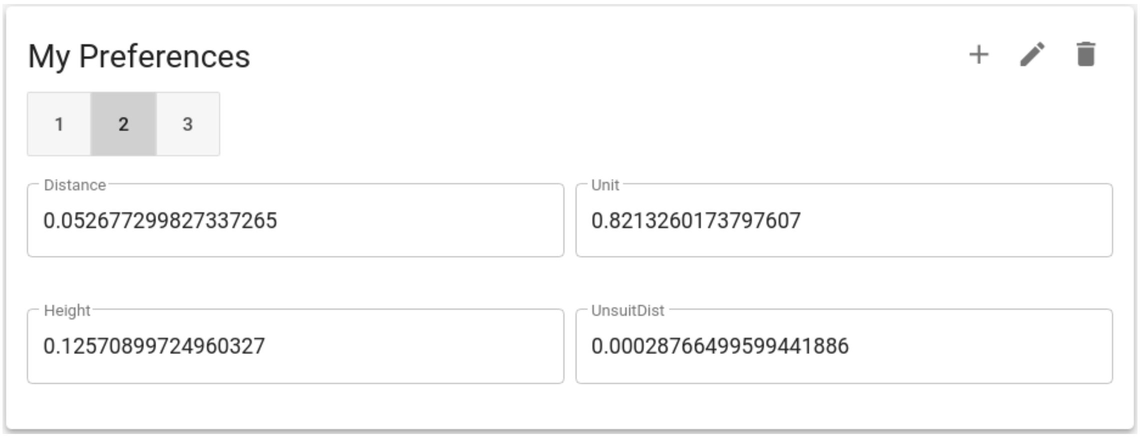
Route 41

Figure 4.10: The Route Manager

An overview of all routes driven by a user is provided by the route manager component shown in Figure 4.10. Here, the routes, which were explained by the algorithm are bundled together in a list. Clicking on a route will show it on the map and expand its details. There can only be one route selected at a time. The detailed view of a route contains a data table, which shows the different subpaths of the selected route. The data table will contain one row for every subpath, which shows the weights in the corresponding preference vector for each metric. A little colored square at the front of the row will additionally show, which segment on the map belongs to which preference. In case there is some preference that the user would like to add to his personal ones, he can click the plus button to add the preference to his profile.

The list will always contain a route called ”New Route”, which is the one that is used for placing and removing waypoints. If the user clicks the check mark button on the map, this ”New Route” will be explained and saved to the driven routes of the user in the back end, which will add it to the list. If the route has been successfully explained and saved, the ”New Route” will be cleared for the next user input. In case the user already has an explained route, but wanted to see how a different arrangement of the waypoints would change the resulting subpaths, a click on the corresponding button in the top right corner will copy the selected route into the ”New Route”. Doing this will copy all waypoints and therefore relieve the user of doing any duplicate work.

Preference Manager



Distance: 0.052677299827337265

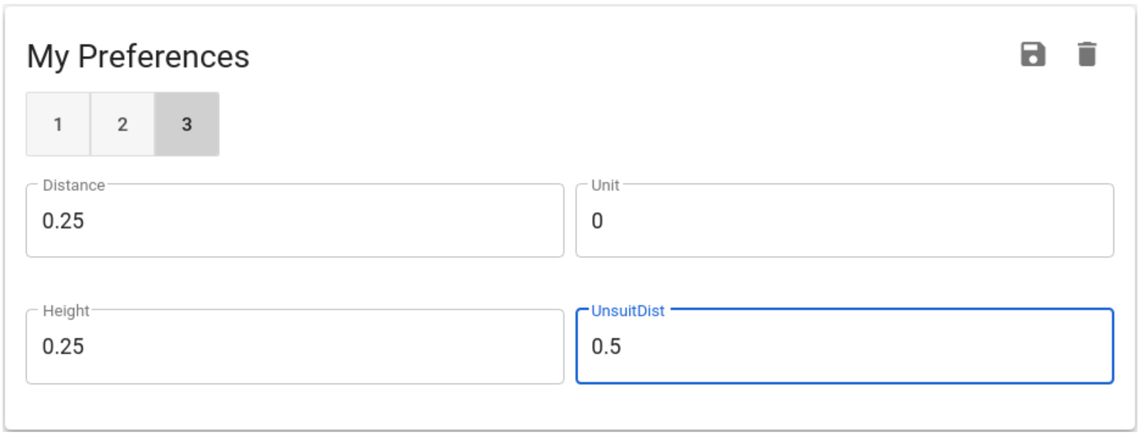
Unit: 0.8213260173797607

Height: 0.12570899724960327

UnsuitDist: 0.00028766499599441886

Figure 4.11: The Preference Manager

The preference manager shown in Figure 4.11 allows users to manage their own individual set of preferences. It lists all user preferences in a button group, and displays the values of the currently selected preference in text input fields. The buttons in the top right corner of the component allow the user to add a preference, edit the preferences, or delete the selected preference. Apart from adding a calculated preference from the route manager, a user may also add a custom preference by clicking on the plus symbol. This will append a new preference to the end of the button group. It is also possible to edit the preferences by clicking on the pencil button, which will allow the user to edit all listed preferences. Both adding a preference and editing the list will swap the plus and pencil buttons out for a save button, as seen in Figure 4.12. In case some of the values do not lie between 0 and 1 or do not add up to 1, the button will be deactivated. When the user is done changing values, clicking on the save button will save the changes to the back end. Removing the selected preference is possible by clicking on the button with the trash can symbol in the top right corner of the component.



My Preferences

1 2 3

Distance: 0.25

Unit: 0

Height: 0.25

UnsuitDist: 0.5

Figure 4.12: Editing a Preference

If the user adds a waypoint to the map, the currently selected preference will be used for the calculation of the subpath between the last way point and the one that has just been inserted. Deleting an intermediate waypoint in an existing path will replace the two preferences of the subpaths left and right of the deleted way point with the currently selected preference.

5 Case Study

To examine the applicability of our software for real-world usage, we tested the implementation on three bike routes driven by a hobby cyclist. We compared the results coming from our software with the actual decision process behind the route planning. Our test person is cycling for more than 20 years now and has driven countless routes in many different regions. According to his own account there is no specific preference in mind when it comes to planning a bike route, but there are some aspects, which are preferred or avoided. Firstly and most importantly, the test person mainly plans out circular routes, which means that source and target in a given route are identical. The ways there and back in such routes are chosen to have as little overlapping edges as possible. Secondly, the cyclist opts to avoid long stretched downhill runs in a route. To meet this criterion, he simply adjusts the direction of travel to turn downhill runs into ascents. Lastly, the test person avoids any climbs towards the end of a route to bring the journey to a leisurely finish, oftentimes leading to detours with respect to path length. Regarding preferred road types, the subject avoids car roads, especially primary and secondary roads, and favors cycle paths.

5.1 Design

For this case study, we reproduced three bike routes of the test person by carefully laying out the waypoints of each route using the front end of the application. We then let the path splitting algorithm find a set of subpaths and preferences, and investigated *a)* How well the found preferences resembled the route planning criteria of the cyclist, and *b)* How some of the resulting subpaths lined up with specific stretches of the original paths, which we point out in the next section. Apart from being of different length, we sought to vary the regions through which the routes pass in order to take into account territorial conditions such as the coverage of cycle paths in a given area.

5.2 Used Routes

In the following, we will go over each route and explain the underlying decision process of the cyclist when appropriate. Additionally, we will point out certain properties of the routes, which should be kept in mind and come back to them when we analyze the results. Since the collected routes begin and end at the same place, we have cut the area around these locations out of the figures to preserve privacy of the test person. Travel direction is indicated by a brown arrow in the respective figures.

Route 1

The first route of our case study, which totals about 43.8 kilometers in length, is shown in Figure 5.1. This route features a very long stretch of road, indicated by the two blue strokes in the figure, which runs parallel to the street B 294. Going from top to bottom, this stretch rises mildly but steadily in height throughout the entire length. Since our test person tries to avoid long descending runs, the travel direction of this route was chosen to go upwards from the source as to take this stage as a climb. Shortly before the route comes to an end, we encounter a sudden bend in the path, indicated by the black circle in the figure. The reason for this is the subject's tendency to make detours in order to avoid climbs towards the end of a route.

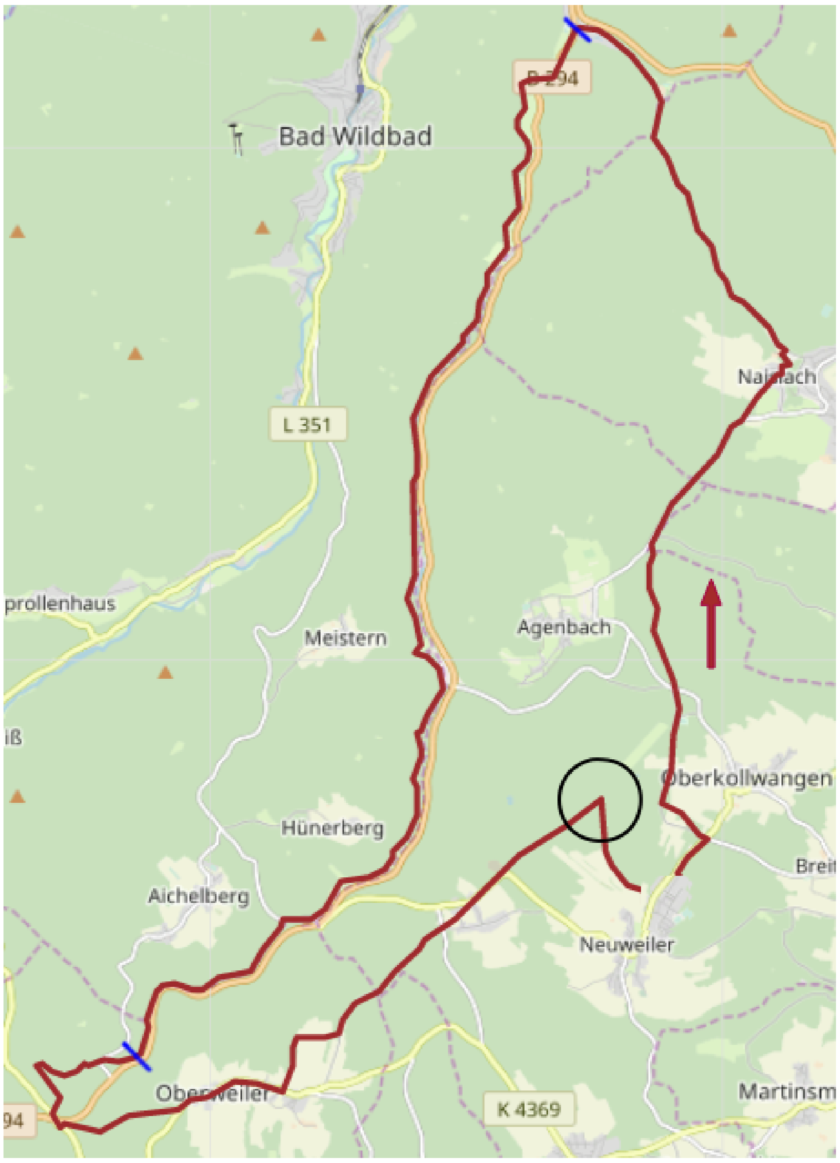


Figure 5.1: Route 1 (≈44km)

Route 2

The second route totals a length of approximately 37 kilometers. We added two blue strokes to the map that highlight a small subpath of the route, which is extremely steep and not very long (in traveling direction). There is a possibility to avoid that climb with a detour, but this would lead to a much greater travel distance. Similar to route 1, we find that the test person has again taken a detour to avoid ascents as the route comes to an end, again indicated by a black circle in the figure.

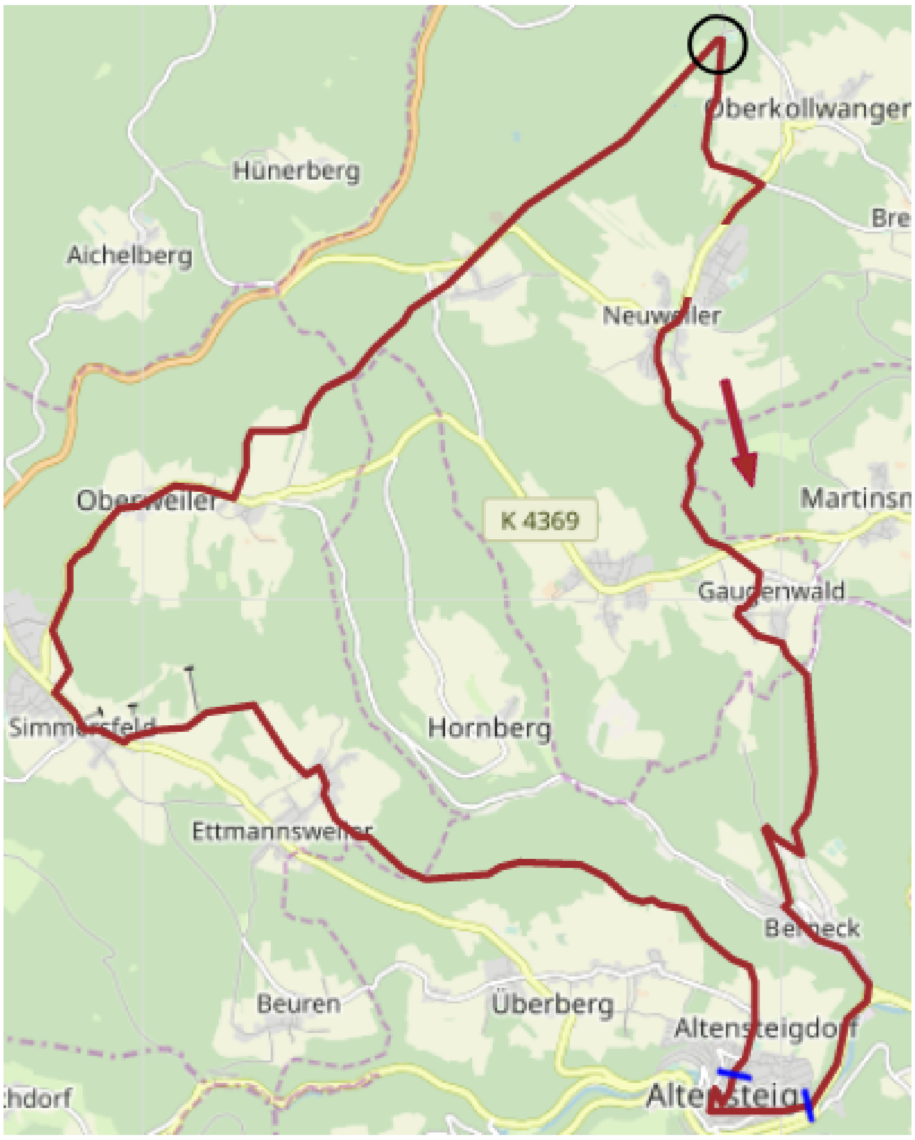


Figure 5.2: Route 2 (≈37km)

Route 3

The third and last route is the biggest one with a total distance of 54.5 kilometers. Again, we encounter a lengthy route segment, this time beginning in Nagold and following the Nagoldtalradweg, which runs parallel to the primary road B 463. While the previous two routes showed a detour towards the end of the journey, here there is no necessity of such a detour, since there is no ascent in the last section of the route.

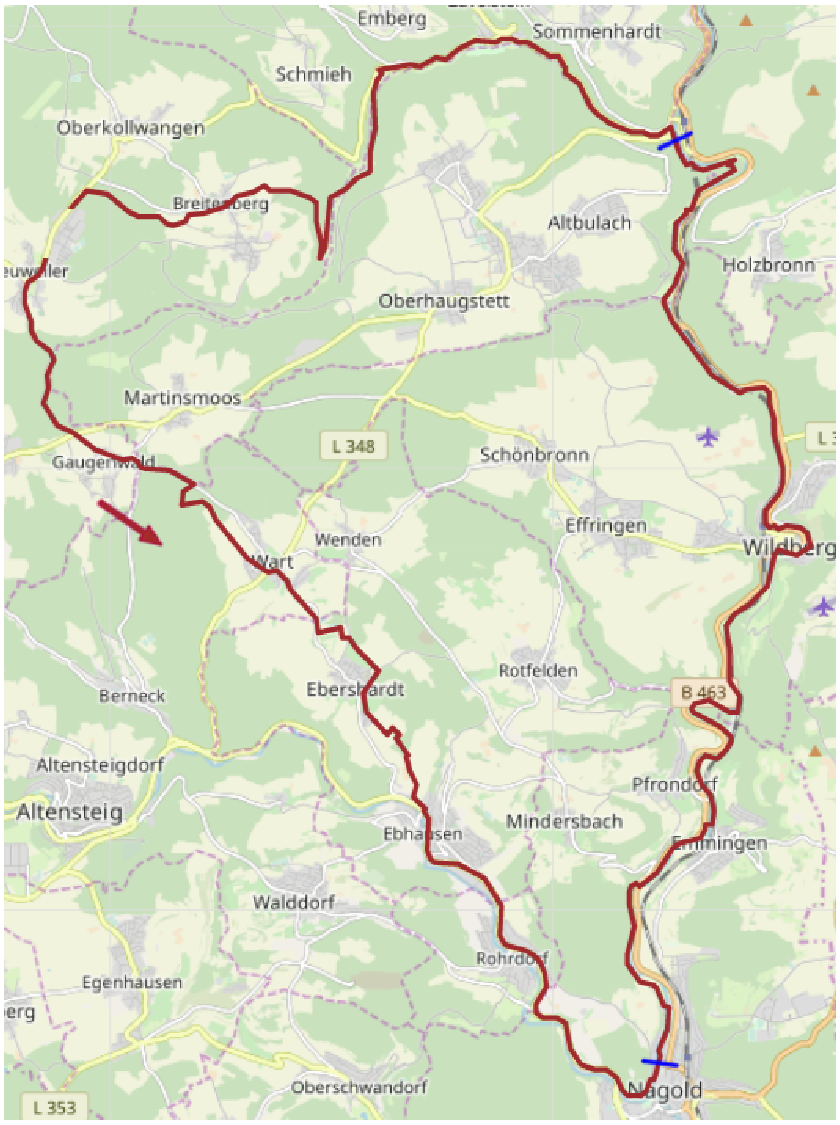


Figure 5.3: Route 3 (≈54km)

5.3 Expectations

Regarding *a)*, we have set very modest expectations. One reason for this is the vast range of different metrics that could have impact on the decision process of a cyclist, compared to the relatively low amount of four metrics, that are covered by our model. Furthermore, we doubt that the test person's preferences introduced in the first paragraph of this chapter can be sufficiently expressed by preferences. Since the test person tries to avoids car roads, we expect the *UnsuitDist* metric to be the dominating factor in the preferences that are returned by the software.

As far as *b)* is concerned, we are cautiously confident to receive some meaningful subpaths from the application. Since detours in a route tend to run in a different direction than the previous path segment, we expect the preference estimator to find preferences, that cover the path up to the node from which such a detour is taken, maybe even a bit further, and then proceed with the next segment.

Route 1

Regarding Route 1, we expect to see a long subpath explaining the marked stretch in Figure 5.1, since the course of the route does not show any sudden twists or anomalies, which could make it infeasible. We expect a high weighting of the distance metric for this stage, but as the primary road B 294 should be shorter than the path that was driven by the subject, we also wager that the algorithm will assign the *UnsuitDist* metric some weight to balance out the high bicycle unsuitability of the primary road. Regarding the detour at the end of the route, we expect the algorithm to return at least two preferences, since the detour seems suboptimal for any metric.

Route 2

Our main point of interest concerning route 2 lies in the short but intense climb highlighted by the blue strokes in Figure 5.2. Here, we expect to see this section explained by a preference assigning the distance metric a high value and the height metric a value of 0. Similar to the first route, we do not presume that the last stage of this route can be explained by a single preference, as the detour taken is simply too extensive to be optimal for a preference.

Route 3

For the last route, we expect to see a big part of the marked subpath in Figure 5.3 explained by a single preference. Since the route taken here (Nagoldtalradweg) is a cycle path, we guess that the resulting preference will have high weight values for the metric *UnsuitDist*, but we estimate that the distance will also be weighted.

5.4 Results

In the following sections we will examine the path splits, which were calculated by the algorithm. We show a figure of the path split as well as a view of the route manager for each route. The results are then compared to the test person's actual preferences as well as the expectations we set for each route.

Route 1

Immediately when looking at Figure 5.4, we find a long subpath covering the major part of the stretch in the route we mentioned previously. The route manager in Figure 5.5 shows that the weight of the preference for this subpath is split between the metrics distance and *UnsuitDist*, just as we expected. Regarding the last part of the route, we examine that the algorithm was indeed not able to find a preference, which explains the entire finish of the route. Note, that the preference manager shows a weight value of 1 for the height metric of the last orange colored path, but could not quite explain the entirety of the last path segment.

Route 2

Regarding the results of route 2, shown in Figure 5.6, we observe that the small path segment with the heavy ascent mentioned previously has been split into two subpaths. The purple subpath confirms our expectation that the weighting of the height metric will be 0, while the other weights are split between distance and unit. Interestingly, we find a weighting of 1 for the height of the following red segment, which is still part of the ascent. Towards the end of the route, we find various subpaths, yet we can observe that some of these indeed have a respective preference, which maximizes the weight value of the height metric, namely the bigger blue subpath and the red subpath, which is the second to last one.

Route 3

Concerning the last route displayed in Figure 5.8, we again notice a long subpath covering a great deal of the before mentioned Nagoldtalradweg. The preference returned from our algorithm for this segment shown in 5.9 turns out to be the default one. For the remaining part of the cycle way path, we encounter the red preference, which does not assign weight to the *UnsuitDist* metric, only to the distance.

Overall, we found that our algorithm returned a wide variety of different preferences for the routes. Since distance is often a metric that is optimized by users subconsciously, we find a lot of preferences which assign weight to this metric. However, we did not encounter many high weightings of the *UnsuitDist* metric, although we used cycle routes and the test person aims to avoid car roads.

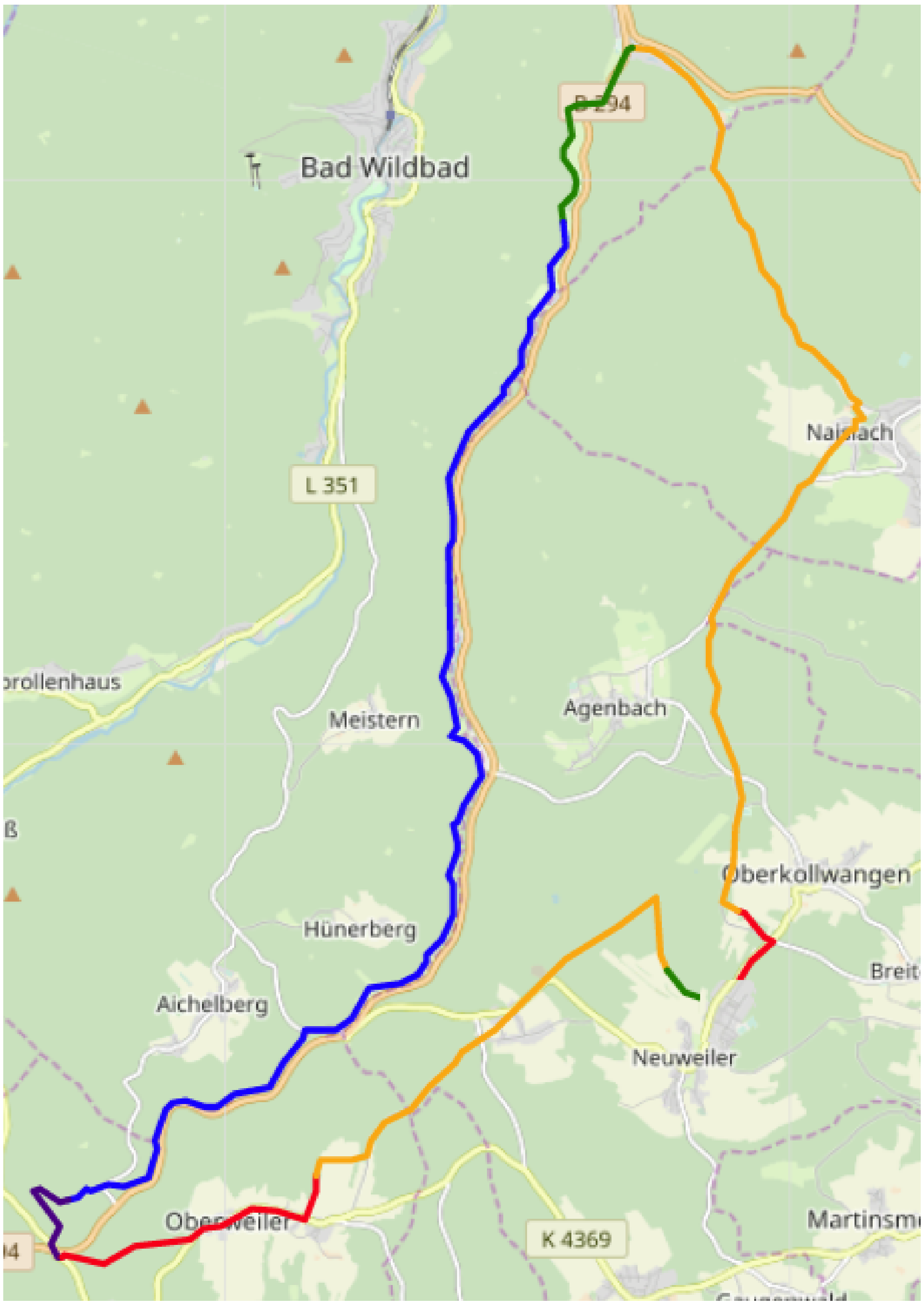
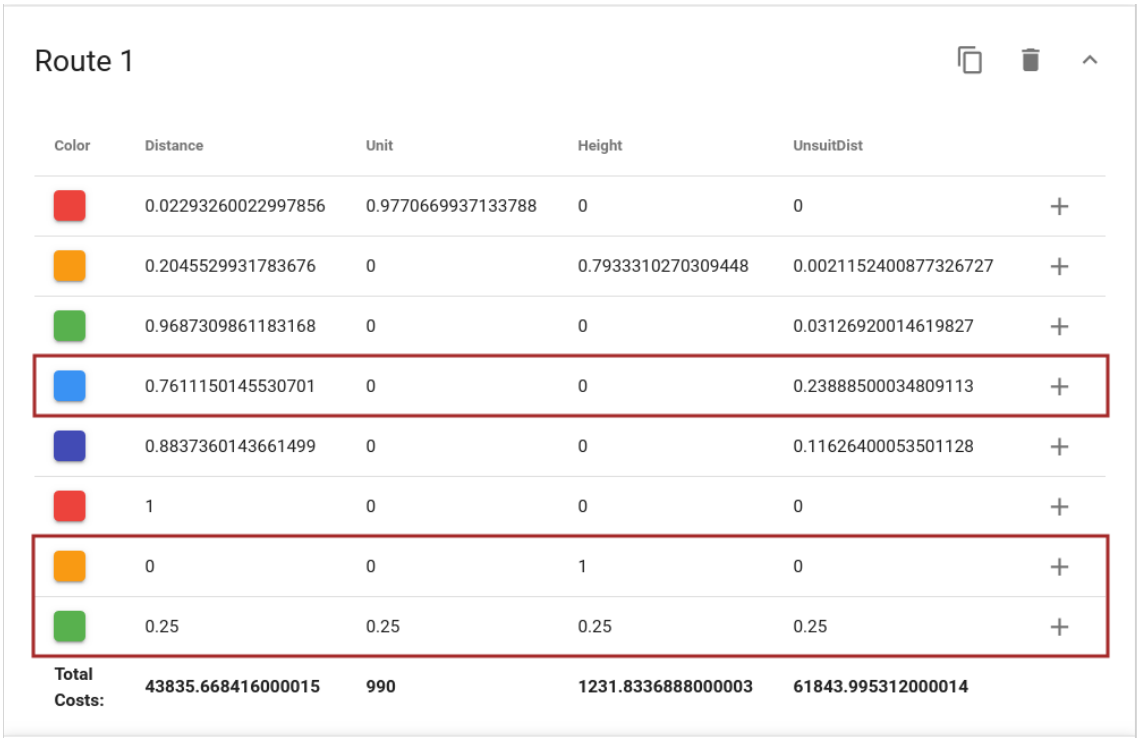


Figure 5.4: Path Split of Route 1



Route 1

Color	Distance	Unit	Height	UnsuitDist	
🟥	0.02293260022997856	0.9770669937133788	0	0	+
🟧	0.2045529931783676	0	0.7933310270309448	0.0021152400877326727	+
🟩	0.9687309861183168	0	0	0.03126920014619827	+
🟦	0.7611150145530701	0	0	0.23888500034809113	+
🟪	0.8837360143661499	0	0	0.11626400053501128	+
🟥	1	0	0	0	+
🟧	0	0	1	0	+
🟩	0.25	0.25	0.25	0.25	+
Total Costs:	**43835.668416000015**	**990**	**1231.8336888000003**	**61843.995312000014**	

Figure 5.5: Preferences for Route 1

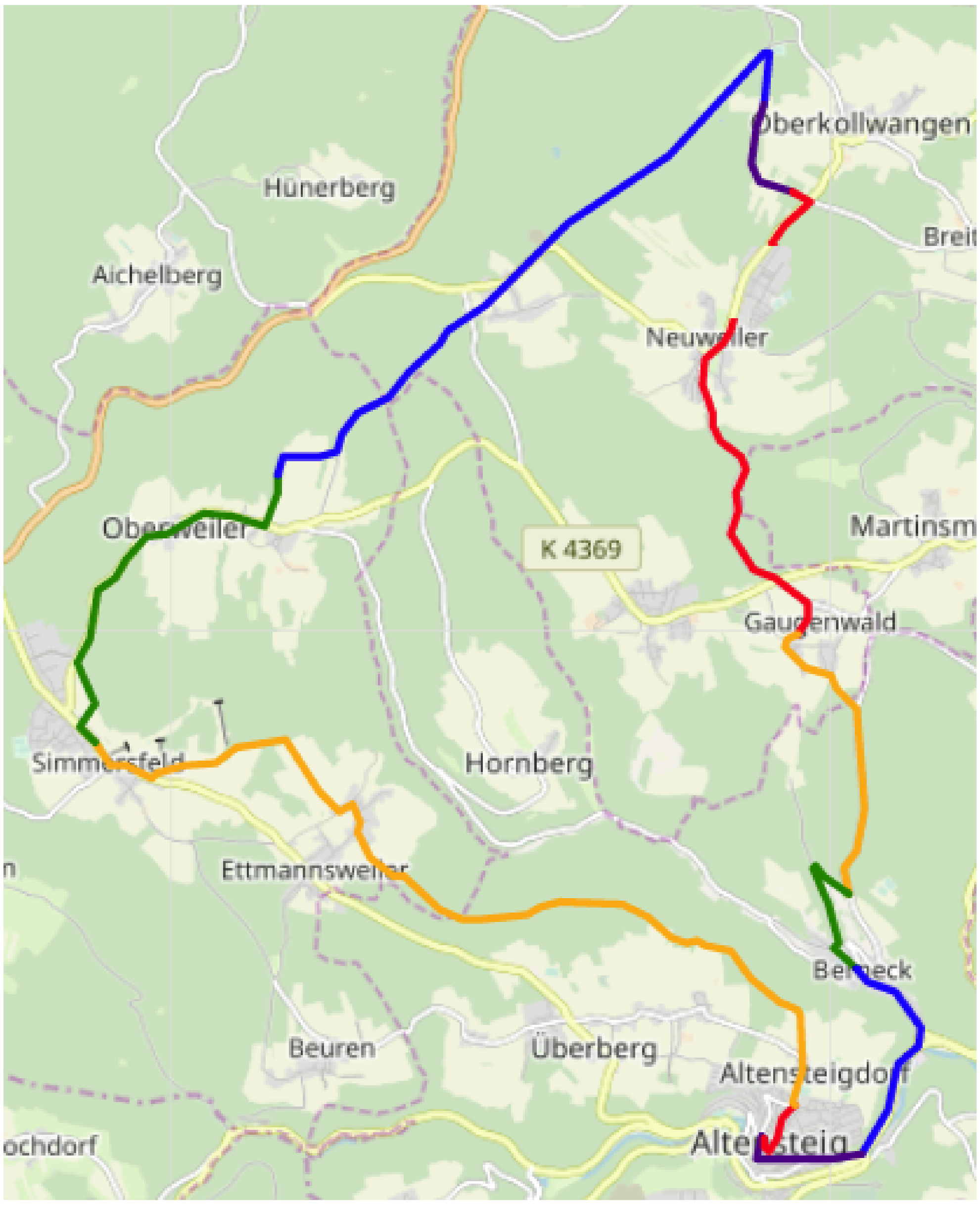
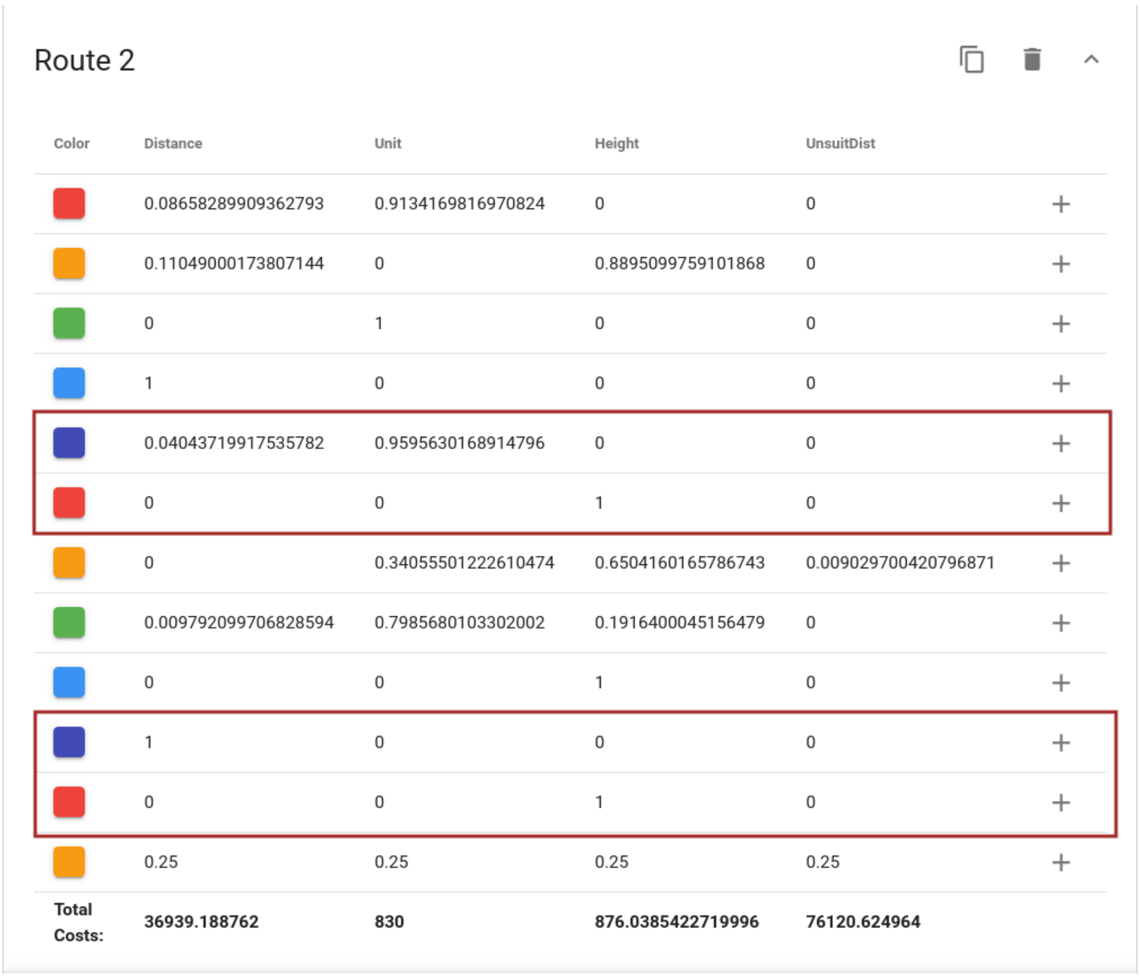


Figure 5.6: Path Split of Route 2



Route 2

Color	Distance	Unit	Height	UnsuitDist	
🟥	0.08658289909362793	0.9134169816970824	0	0	+
🟧	0.11049000173807144	0	0.8895099759101868	0	+
🟩	0	1	0	0	+
🟦	1	0	0	0	+
🟪	0.04043719917535782	0.9595630168914796	0	0	+
🟥	0	0	1	0	+
🟧	0	0.34055501222610474	0.6504160165786743	0.009029700420796871	+
🟩	0.009792099706828594	0.7985680103302002	0.1916400045156479	0	+
🟦	0	0	1	0	+
🟪	1	0	0	0	+
🟥	0	0	1	0	+
🟧	0.25	0.25	0.25	0.25	+
Total Costs:	**36939.188762**	**830**	**876.0385422719996**	**76120.624964**	

Figure 5.7: Preferences for Route 2

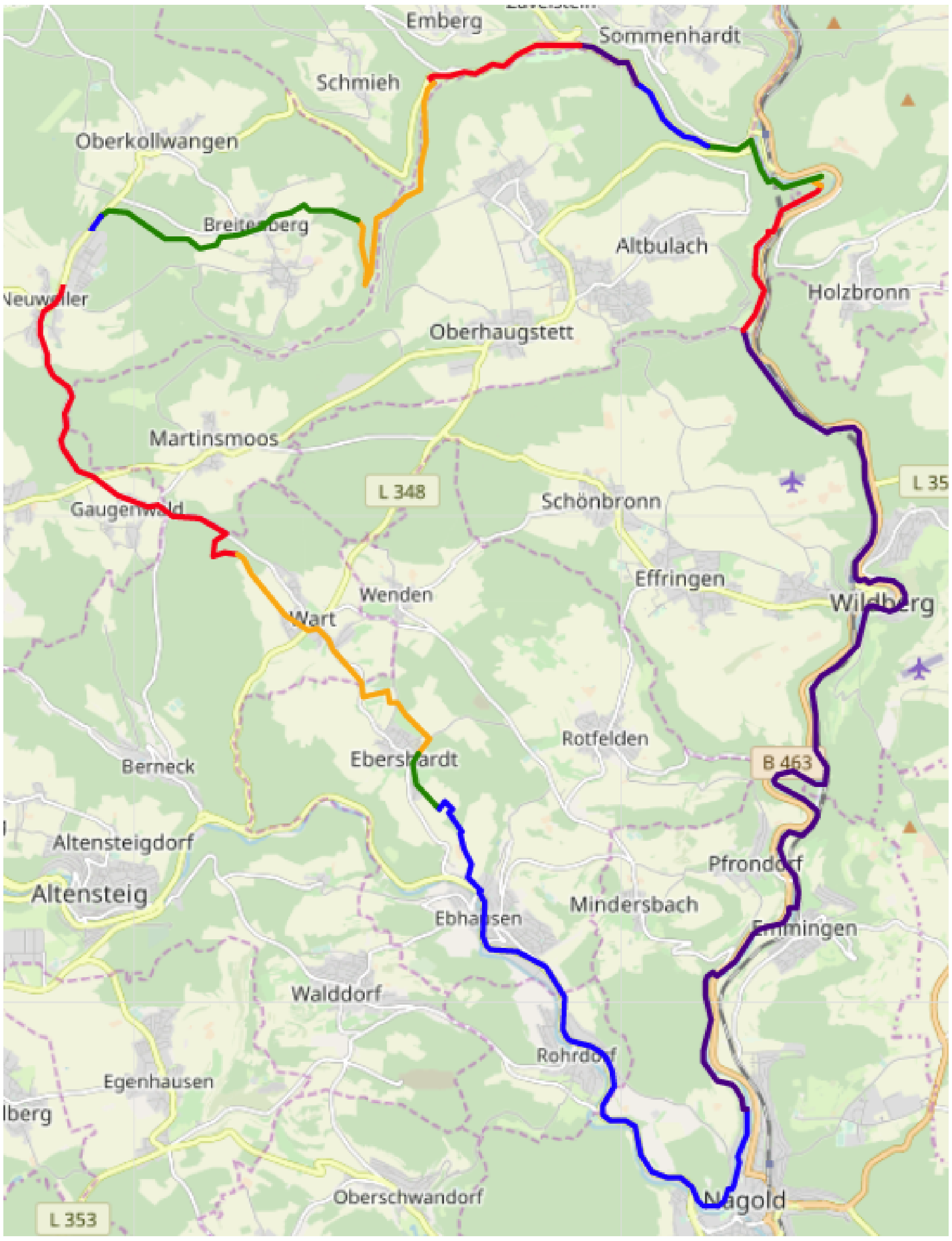
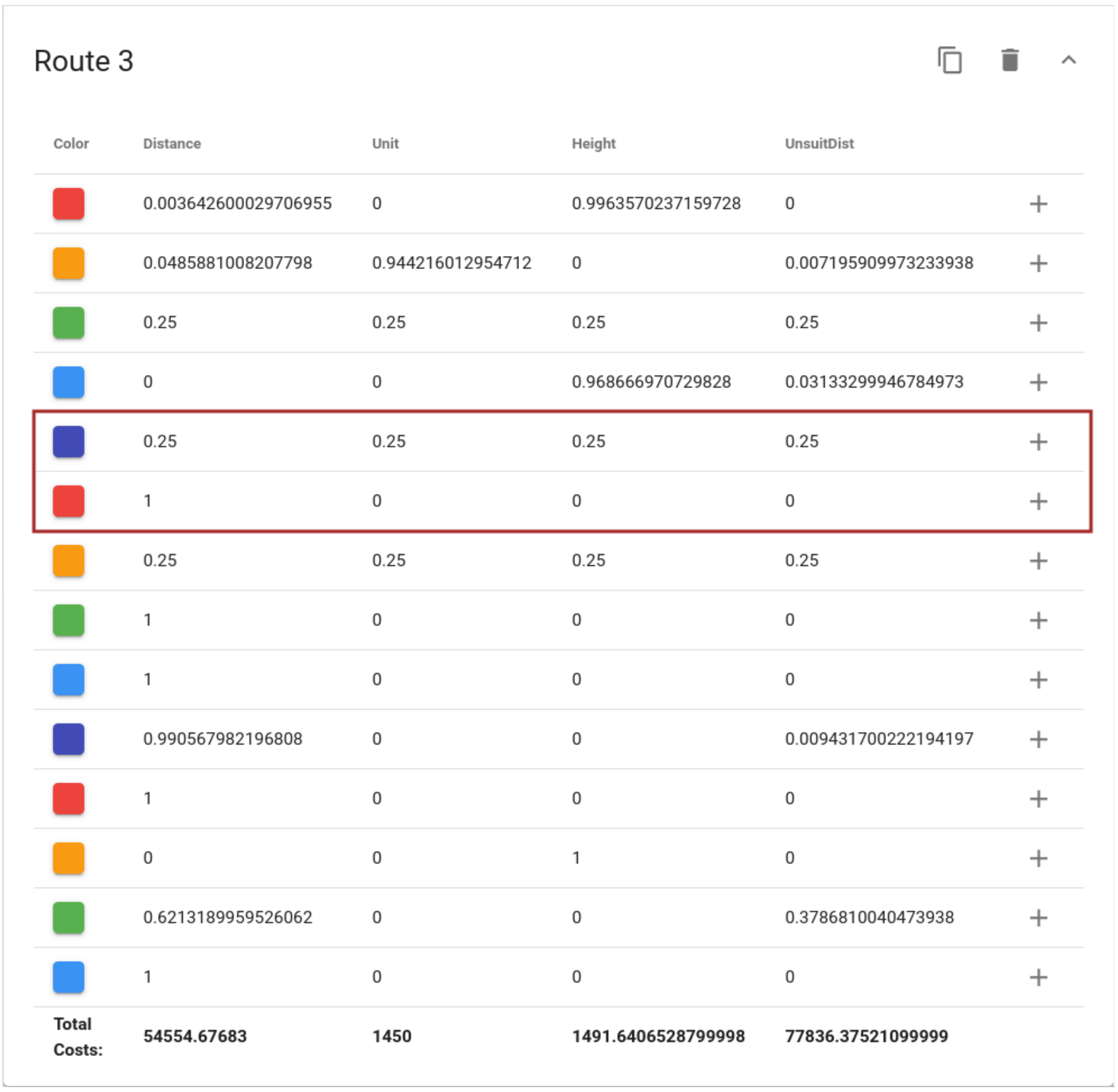


Figure 5.8: Path Split of Route 3



Route 3

Color	Distance	Unit	Height	UnsuitDist
🟥	0.003642600029706955	0	0.9963570237159728	0
🟧	0.0485881008207798	0.944216012954712	0	0.007195909973233938
🟩	0.25	0.25	0.25	0.25
🟦	0	0	0.968666970729828	0.03133299946784973
🟪	0.25	0.25	0.25	0.25
🟥	1	0	0	0
🟧	0.25	0.25	0.25	0.25
🟩	1	0	0	0
🟦	1	0	0	0
🟪	0.990567982196808	0	0	0.009431700222194197
🟥	1	0	0	0
🟧	0	0	1	0
🟩	0.6213189959526062	0	0	0.3786810040473938
🟦	1	0	0	0
Total Costs:	**54554.67683**	**1450**	**1491.6406528799998**	**77836.37521099999**

Figure 5.9: Preferences for Route 3

6 Algorithm Runtime Analysis

In this chapter, we aim to examine the runtime of the application. Our points of interest here are the different factors that have impact on the runtime of the path splitting algorithm. We decided against the idea of creating random waypoints that simulate user behavior to test our application. The main reason for this is that these random paths simulate actual cyclist behavior very poorly, since arbitrary waypoints mostly lead to utterly nonsensical routes, which would give us no real insight on how the software behaves when we input real-world routes. We therefore conduct our analysis with the cycle paths from our case study.

Specs

The evaluation was conducted on a single core of an Intel i5-8250U CPU with 1.6GHz and 16GB RAM. The Rust compiler *rustc* was running on version 1.37.0. The source code was compiled in release mode.

Results

We received the following values by calculating the mean over 25 runs of our algorithm for each of the three routes. Table 6.1 shows the results for the routes of our case study. We were interested in how the number of subpaths and the total length of the route impacted path splitting performance. While the runtime clearly increased with a greater number of subpaths in the route, we also observe that total route length impacts the time taken to find a subpath.

Furthermore, our results showed that for each subpath, the algorithm called *calcPref* on average 8.2 times, the subroutine itself took on average $925ms$ to calculate a preference. While calculating a preference, we on average had to solve the LP 2.04 times, which means we also added 2.04 constraints to the LP and ran 2.04 ± 1 Dijkstras (depending on whether the LP found a solution or not). Dijkstras took on average $378ms$, while it took merely $3ms$ on average to solve the LP to optimality.

	Route 1	**Route 2**	**Route 3**
Length	$\approx 44km$	$\approx 37km$	$\approx 54km$
# Subpaths	8	12	14
Total Time Taken	$61.08s$	$86.36s$	$109.2s$
Time to find Subpath	$7.14s$	$6.87s$	$7.35s$

Table 6.1: Runtime evaluation results for the three paths

7 Conclusion and Outlook

In this bachelor thesis we presented an algorithm, which is able to calculate a preference explaining some given route by splitting the route into feasible subpaths. We utilized this algorithm in a software, which features a user-side front end and a server-side back end. The application allows users to specify their cycle paths using the front end of the application, while the back end of the software calculates path splits and manages users. Our case study showed that the algorithm could return some meaningful preferences and subpaths for real-world cycling routes.

7.1 Conclusion

When attempting to explain a real-world bicycle route, we found that the algorithm presented in [FLS16] could scarcely find a feasible preference for the four metrics used in our model. One possible explanation for this is the relatively low amount of edge metrics that we took into consideration for the implementation. Comparing these four with the countless other metrics suited for cyclists, we recon that adding more metrics to our model increases the probability of finding a feasible preference. The case study presented in chapter 5 has shown that the algorithm, even when just considering four metrics, is able to deduce some very interesting preferences and subpaths, a few of the path splits represented stages of the bike route very well. We have seen that the runtime of the algorithm naturally increases with increasing path length as well as increasing number of subpaths.

7.2 Outlook

First and foremost we think that adding more metrics contributes significantly to the algorithm's potential of explaining user routes, since every added metric increases the range of possible solutions for the LP, which makes it more likely that the algorithm finds a preference. In addition, we see room for improvement considering the runtime of the path splitting algorithm. Currently, processing a medium sized cycling route ($\approx 30km$) takes more than a minute, with the Dijkstra runs being the greatest factor contributing to the runtime, see chapter 6. During the calculation of a path split for a given route (which took three minutes for route 3 of our case study), the user receives no information regarding the progress of the algorithm, provided that he has no access to the logs of the back end. As this limits user experience significantly, future work could extend the algorithm to return a subpath and preferences as soon as one is found, because they will not change regardless of how the algorithm splits the remaining route. Furthermore, our algorithm relies on a greedy binary search, only looking for the longest possible subpaths, but there are other approaches to find feasible subpaths. For example, we could try to minimize the amount of unique preferences when splitting a path, hoping that the user plans certain subpaths with the same preference. Since we focused on splitting paths, because we could not explain them wholly, we did not pay much attention to the idea of grouping user routes into meaningful groups, as presented in [FLS16]. While joining paths as a whole into groups seems barely doable in a real-world scenario, we see potential in grouping preferences rather than routes. Finally, carefully laying out a route in the front end of the application is very time consuming. Route 3 of our case study for instance required 80 waypoints until it was specified to our satisfaction. We therefore think that adding a functionality to import routes to the front end is beneficial for user experience, since users tend to track themselves by using smartphones or GPS sensors anyway, there is plenty of data to be imported available.

Bibliography

[Aga10] V. Agafonkin. *Leaflet — an open-source JavaScript library for interactive maps*. 2010. URL: `https://leafletjs.com/` (cit. on p. 33).

[And12] Andrew Makhorin. *GLPK - GNU Project - Free Software Foundation (FSF)*. 2012. URL: `https://www.gnu.org/software/glpk/glpk.html` (cit. on p. 29).

[Bar18a] F. Barth. *Multi-CH-Constructor*. 2018. URL: `https://github.com/Lesstat/multi-ch-constructor` (cit. on p. 18).

[Bar18b] F. Barth. *Pbfextractor*. original-date: 2017-10-10T14:56:06Z. Apr. 2018. URL: `https://github.com/Lesstat/pbfextractor` (cit. on p. 18).

[Bou16] M. Bouchaud. *Vue2 leaflet library*. 2016. URL: `https://korigan.github.io/Vue2Leaflet` (cit. on p. 33).

[Cav16] J. Cavat. *Lp modeler written in Rust*. 2016. URL: `https://github.com/jcavat/rust-lp-modeler` (cit. on p. 29).

[FLS16] S. Funke, S. Laue, S. Storandt. "Deducing Individual Driving Preferences for User-aware Navigation". In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPACIAL '16. Burlingame, California: ACM, 2016, 14:1–14:9. ISBN: 978-1-4503-4589-7. DOI: `10.1145/2996913.2997004`. URL: `http://doi.acm.org/10.1145/2996913.2997004` (cit. on pp. 11, 13–15, 19, 29, 53, 54).

[FLS17] S. Funke, S. Laue, S. Storandt. "Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees." In: *SEA*. Ed. by C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, R. Raman. Vol. 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 18:1–18:13. ISBN: 978-3-95977-036-1. URL: `http://dblp.uni-trier.de/db/conf/wea/sea2017.html#FunkeLS17` (cit. on pp. 17, 18).

[FRC+07] Farr Tom G., Rosen Paul A., Caro Edward, Crippen Robert, Duren Riley, Hensley Scott, Kobrick Michael, Paller Mimi, Rodriguez Ernesto, Roth Ladislav, Seal David, Shaffer Scott, Shimada Joanne, Umland Jeffrey, Werner Marian, Oskin Michael, Burbank Douglas, Alsdorf Douglas. "The Shuttle Radar Topography Mission". In: *Reviews of Geophysics* 45.2 (May 2007). ISSN: 8755-1209. DOI: `10.1029/2005RG000183`. URL: `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2005RG000183` (visited on 03/29/2018) (cit. on p. 16).

[Gmb18] G. GmbH. *Geofabrik Download Server*. 2018. URL: `http://www.rust-lang.org` (cit. on p. 15).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In: *WEA*. 2008 (cit. on p. 17).

[HP10] G. Hoare, T. R. Project. *The Rust Programming Language*. 2010. URL: `https://www.rust-lang.org/` (cit. on p. 25).

[LLC16] V. LLC. *Vue Material Design Component Framework*. 2016. URL: https://vuetifyjs.com/ (cit. on p. 32).

[MS08] K. Mehlhorn, P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. 1st ed. Springer Publishing Company, Incorporated, 2008. ISBN: 9783540779773 (cit. on p. 15).

[RR07] L. Richardson, S. Ruby. *Restful Web Services*. First. O'Reilly, 2007. ISBN: 9780596529260 (cit. on p. 23).

[Sin19a] P. Singer. *Pbfextractor*. 2019. URL: https://github.com/sinpat/pbfextractor (cit. on p. 18).

[Sin19b] P. Singer. *Preference Routing Web Client: Specification of Trajectories and Learning of User Preferences*. 2019. URL: https://github.com/sinpat/preference-routing-web (cit. on p. 23).

[Sin19c] P. Singer. *Preference Routing: Specification of Trajectories and Learning of User Preferences*. 2019. URL: https://github.com/sinpat/preference-routing (cit. on p. 23).

[Tea] T. A. Team. *Actix - Actor System and Web Framework for Rust*. URL: https://actix.rs/ (cit. on p. 26).

[You14] E. You. *Vue: The Progressive JavaScript Framework*. 2014. URL: https://vuejs.org/ (cit. on p. 30).

[You18] E. You. *Vue CLI*. 2018. URL: https://cli.vuejs.org/ (cit. on p. 30).

All links were last followed on October 30, 2019.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature