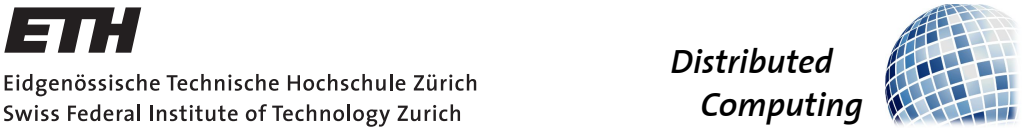


Worldwide Sports Route Generation

Semester Thesis

Johannes Weinbuch

jweinbuch@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Manuel Eichelberger
Prof. Dr. Roger Wattenhofer

December 23, 2017

Abstract

When doing endurance sports such as running or cycling, route selection can be a necessary evil. There is a tradeoff between the interestingness of different routes and comparability between results. To generate interesting routes of desired length within Switzerland, an app called *SmartRoute* has been developed in a previous thesis. In this work, the functionality of this application is extended to work on a worldwide scale. Furthermore, instead of relying on an Android app as user interface, a universal browser based solution has been implemented.

Contents

Chapter 1

Introduction

When doing sport, it is often desirable to have repeatability to compare activities. We can run, cycle or hike the same route multiple times and see easily, if we have improved. Over time however, this might get boring doing the same route over again. To conquer that, one could change the route every time. But this takes a fair amount of planning to have comparable routes. Alternatively, without much planning, there will probably be large differences in route lengths. There is a tradeoff between comparability and interestingness.

To combat that, an app to find good routes with a specified length has been developed in a previous thesis at the Distributed Computing Group (DISCO) [1]. This app works within Switzerland for the activities running, cycling, hiking, mountain biking and inline skating. In this work, the goal is to improve the system such that it works on a worldwide scale. After all, when regularly running in the own neighborhood, a map is not needed. Using the personal mental map, one can come up with a route very quick and even be slowed down by additional software. Even when just deciding spontaneously; with enough knowledge of the local paths one could still reach the desired route length.

By extending the existing solution to a world-wide scale, good routes are available even at places one might not know, for example on vacation. Using OpenStreetMap (OSM) data [2], it is possible to use worldwide information on nearly every inhabited place to select a route.

Apart from this extension in scale, a target is to improve the route finding to return better routes. While the basic principles of the algorithm have been left untouched, changes to the interpretation of the OSM data are introduced so that the generated routes exhibit better quality.

In the coming chapters, the steps taken to reach the goal of world-wide functionality and better routes are described. In Chapter 2, a short introduction to the existing solution is given as well as the analysis methods used to find improvements. These improvements are described in detail in the following Chapters 3 and 4. Chapter 3 contains the backend changes made that are transparent to the user. When employing these changes, the only thing a user should notice is

a faster run time of the route generation and worldwide functionality. In Chapter 4, the user facing changes are described. After having applied these changes, the results of a benchmark introduced in Chapter 2 are presented in Chapter 5. Finally, Chapter 6 will conclude this thesis.

Chapter 2

Methods

In order to extend the working area of the existing solution, the first step is to understand why it does not work yet. Only then, alternatives can be found and implemented. A basic requirement is to understand how the application works. An overview of this is given in Section 2.1. From reading the code alone it is however not always obvious where the slow parts are. Section 2.2 contains further analysis methods used in this work to make the discovery of bottlenecks easier and faster.

2.1 Overview Existing Solution

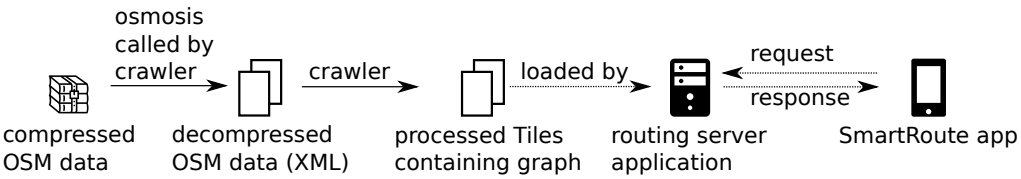


Figure 2.1: Data flow through existing solution

The previous solution consists of three parts: The *crawler*, the *routing server* and the *SmartRoute* app. To understand how the parts work together, we will follow the data flow from the source to the final route. This is also illustrated in Figure 2.1.

The first step is the conversion of OSM data to a file based format. The reason for this is that the original data contain much more information than necessary. There is no need to know where the political borders of villages are for example. Furthermore, the data are heavily compressed to enable transmission over the internet in a sensible amount of time. Uncompressed, the data for the whole planet would take up around 850 GB of disk space [3]. In the compressed case, using the *Protocol Buffer Format* [4], this is reduced to 37 GB. Since this version only works within Switzerland, an excerpt of the planet file has been used. It can be obtained either by extracting the data directly from the planet file or by

downloading it from a provider of extracts like Geofabrik [5] for example. This file has a size of around 300 MB.

The crawler gets supplied with a start location and a number of tiles to extract. A tile is defined in the WGS-84 coordinate system of Earth and is a square of 0.1 by 0.1 degrees. At the latitude of Switzerland, this corresponds to roughly 10 by 10 km. For each tile, the crawler starts a call to an external tool called osmosis [6]. This tool decompresses the data file and writes its contents to an Extensible Markup Language (XML) file. In this step, osmosis is configured to already drop some unneeded data.

The XML file is then run through osmosis again to get height information for the data. The height information come from the NASA Shuttle Radar Topography Mission (SRTM) data set. To get the height data and include it into the XML file, the osmosis-srtm-plugin [7] is used. For simplicity, both osmosis runs are summarized into one in Figure 2.1.

Finally the XML file is parsed. It contains elements called nodes and ways. The nodes contain an ID, coordinates and the height. The ways connect nodes to paths. They contain a number of node IDs and several tags. From this file, a graph can be built, with the nodes as vertices and the ways defining the edges. For each edge, a weight is calculated from the tags of the way. In this version, the graph was also modified to concatenate paths over multiple nodes with exactly two neighbors to a single edge. (In other words, only keep nodes where different ways share a node). The resulting graph represents the route network for each activity[1]. It is then serialized and stored in small files spanning 0.01 by 0.01 degrees, roughly a kilometer by a kilometer.

Now the routing server takes over. It is a Java web application running inside a Tomcat Server [8]. Tomcat takes care of the external interface and the request handling. It passes a request destined to the routing server to the web application. A request contains:

- start and end coordinates
- the route length,
- a (constant) routing type,
- the activity,
- user preferences for the importance of the environment, the elevation and the view.

On receiving the first request, the routing server loads all data files into memory. Until this is finished, all requests will return an empty route. After the

[1] Because the vertices of the graph and the nodes are so closely related, the terms vertex and node are going to be used interchangeable

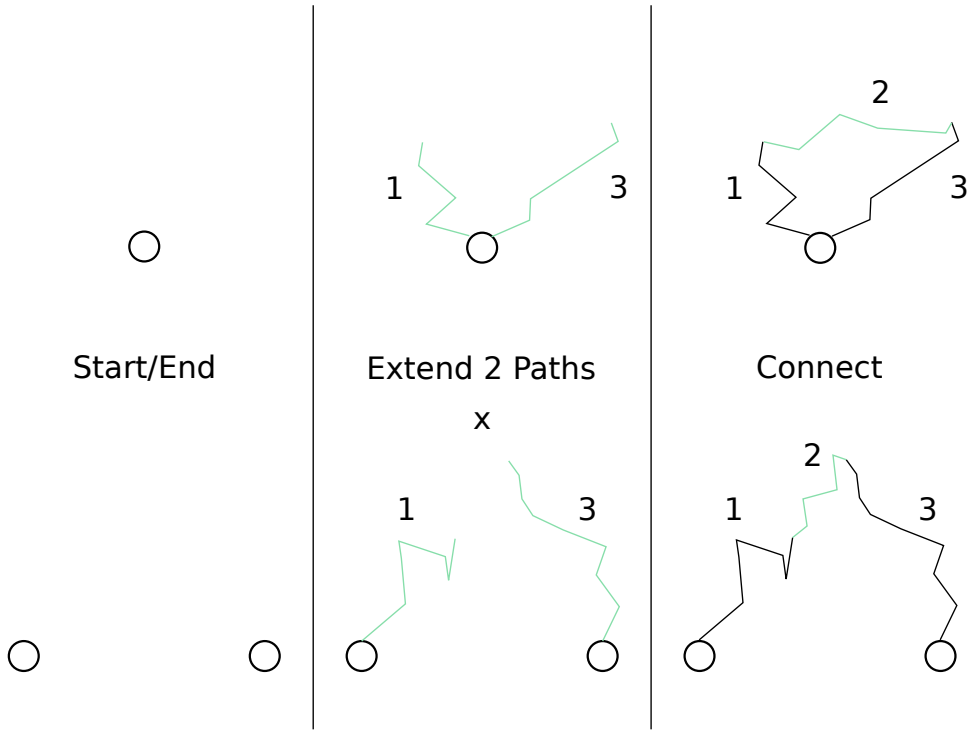


Figure 2.2: Illustration of routing algorithm

files are all loaded, a new request triggers a bunch of calculations. First, the IDs of the start and end point are determined. This is done by reading the data file containing the point again, looking at all nodes and returning the one that has the shortest distance to the supplied coordinates.

The rough procedure after start and end node are determined is illustrated in Figure 2.2. There are two main cases: starting and finishing at the same node, shown in the top part of the figure, or getting from node A to node B, shown in the bottom part of the figure.

In general, each route consists of three pieces, a first and a last one and the middle one to connecting the two. The first and last piece are generated by starting at the start and the end node. From these nodes, a path is generated along edges with high weights towards a target. This is called path extension and is illustrated in the middle part of the figure. The target in the case where start and end are different is a node where the route should connect. In the case where start and end are the same the target are randomly chosen directions. These pieces do not form a connected path yet. For that, the middle piece is responsible.

The middle piece is generated by an algorithm that is called *randomPoints*. Given the last nodes of the first and third piece, it tries to connect them. It

does so by selecting a set of points that lies between the two nodes. For that, it calculates the shortest path between them and adds neighbors with good edge weights. From this set, a number of nodes is drawn and then connected by the shortest path between them. Since there are many random decisions, the randomPoints algorithm is repeated for a certain amount of time as well as the whole procedure of generating three pieces. Out of the results, the route with the highest weight that lies within a length tolerance of 50 m plus 5 % of the route length is selected.

For more details on the workings of these algorithms, refer to the previous work [1, 9]. Where necessary to understand this work, they are explained in the following sections.

The requests are issued from the app called SmartRoute. It allows to select the request parameters in a way that is user-friendly, submits the requests to the server and displays the reply on a map with the ability to navigate the returned route. The requests follow the HTTP protocol, the parameters are transmitted by using the HTTP GET method (where the parameters are given through the URI). The app is closer examined and described in Chapter 4.

2.2 Testing Methods

Reading the code alone only helps so far. At one point, running the code and looking at how it behaves will lead to results much faster and easier. During this work, a benchmark to compare different versions of improvements has been written and is described in Section 2.2.1. This however only looks at the program execution as a whole, which is important for the final results but of limited use for intermediate development steps. There, a more fine-grained approach was necessary and a multitude of tools have been used.

To achieve the goal of worldwide route generation, the data storage needed, run time and memory requirements all need to be reduced. Thus, they need to be measured. While the data storage can be observed by looking at the file sizes, run time and memory usage is not as easy. For run time, it is possible to insert statements in the code to log the system time and calculate the time differences between them.

In addition, profiler have been used. For this project, mainly VisualVM [10] was used. This tool allows to see the amount of time spent in each function and was invaluable to find the bottlenecks of the application. For analyzing the memory usage by the means of heap dumps, the Eclipse Memory Analyzer (MAT) [11] has been used.

2.2.1 Benchmark

To get comparable test results between version and test a wide variety of inputs, testing has to be automated. For this purpose, a set of scripts was written. The goal was to create a benchmark that can show whether the route generation works at all and how long it takes to get to a result or abort without result.

The benchmark consists of two shell scripts, *sendRequest.sh* and *sweep.sh*. The script sendRequest.sh takes following parameters:

- a name for identifying the run,
- the length of the route
- an activity,
- a longitude,
- a latitude.

The script constructs a request out of these parameters with the same start and end point at the given coordinates. Since the calculations for same start and end are very similar to the situation with different start and end, this was chosen as the easiest way of testing. The request is carried out by using the curl command and timed using the time command. To get the number of nodes in a response, all characters except for opening brackets are deleted. Because each coordinate in the response has exactly one opening bracket, a simple counting command gets the number of nodes in the returned route. This way, it is detectable whether the response is successful or contains an empty route. The time and result of the request are appended to a file which has the name of the name parameter to the script.

The sweep.sh script takes a name and a degree of parallelism as parameters. It consists of three nested loops to iterate over the parameters supplied to the sendRequest script. One loop iterates over 6 route lengths, between 5 and 100 km, another loop iterates over the available activities and a third one iterates over datasets of points. Such a dataset is a file with a list of coordinates. Inside the innermost loop, the script calls sendRequest.sh by using the command xargs. This allows to submit contents of a dataset as arguments to the script and to even use parallel execution of the script. In the case of parallel execution, it is however no longer possible to get from the execution times of single requests to the total execution time by simple addition, which is why in sweep.sh another timer is started to record the total time for all requests.

The last open question is how the points for the datasets have been selected. In order to avoid having to carefully fine tune a data set, points are generated points randomly with the help of the random coordinate generator at `http:`

(a) Brasilia, 1000 km, 100 points

(b) Lucerne, 1000 km, 150 points

(c) Kansas, 2500 km, 200 points

Figure 2.3: Examples of datasets used for the benchmark

//www.geomidpoint.com/random/. Three examples are shown in Figure 2.3. Random points were generated within increasing radii around the city Lucerne to get test coverage for most of Europe. To further test the application, a dataset with 200 points within a radius of 2500 km around Kansas City has been created to cover most of northern America as well as a file with 100 points 1000 km around Brasilia to have a test case where both latitude and longitude are negative.

Of course, it cannot be expected that a route is returned for each randomly chosen point on earth. Take for example a small radius around Lucerne. A point there can be completely within the Lake Lucerne or a steep flank of Pilatus where there are only few paths. There should be no routes for one or more activities. The datasets were however not cleaned of such points, as they provide valuable information about the behaviour of the application. Does it fail fast or does it take long to get no result? Resources are occupied even with requests that are bound to return an empty route. Therefore, it makes sense to test these situations as well.

Chapter 3

Backend

3.1 Data Structures

When implementing an algorithm, not only the complexity of the algorithm determines the run time complexity. When the underlying data structure does not fit the needs for the given problem, even good algorithms can be slow. In the coming two sections, examples of such situations will be given and the implemented solutions are described. The changes in this section concern both crawler and routing server, because the affected data structures are either shared across the boundary between crawler an server or occur in both of these applications.

3.1.1 List Implementation

In this first example, we will look at data stored in a *List*. An iteration through all values might look something like in Listing 3.1. Similare constructs have been used very frequently in the version of the previous work.

```
List<Object> a;
// Do something

// iterate over list
for(int i = 0; i < a.length(); i++) {
        Object currentObject = a.get(i);
        // Do something
}
```

Listing 3.1: Example of a Java List Style iteration

The chosen List implementation can make a huge difference when executing this loop. If a is an *ArrayList*, there is not much of a problem, the complexity of this loop would be $O(n)$, assuming there are n elements in the List. If however a *LinkedList* was chosen (as it was the case in most places of the existing code), the complexity is now $O(n^2)$, because for every element retrieved, on average $n/2$ elements have to be looked at before we are at the desired index. The fix for

this is to either use ArrayLists or to use an Iterator over the List. For this simple case, there is even a for-each syntax built into current Java versions as seen in Listing 3.2. Even without changing the List implementation from a LinkedList to an ArrayList, big improvements in runtime have been made with this simple change.

```
List<Object> a;
// Do something

// for each element in a do something
// each element is known in the loop as currentObject
for(Object currentObject : a) {
        // Do something
}
```

Listing 3.2: Example of a Java List for-each iteration

3.1.2 Graph Representation

While the list usage of the previous work mainly poses a run time problem, the chosen graph representation causes problems in storage size and memory usage. The whole graph is stored across several *maps*. A map is a data type that stores key value pairs, but the detailed implementation is not important here. The key point is what is stored in the map.

The previous solution uses maps to store the following data, each using the node ID as the key:

- a coordinate object storing two values for latitude and longitude
- the height
- a list of the neighbors
- a map containing the neighbors as keys and distances to them as values
- for each activity, a map containing the neighbors as keys and weights to them as values, as well as weights for the view and elevation.

All in all, there are 11 maps, each one containing the node ID as key. The ID is a 64 bit integer value. If only one map is used, 640 bit per node are saved for the IDs alone, or 610 MB for a million nodes. For comparison, a square of 1 degree by 1 degree in Switzerland has between one and two million nodes that are used. Even more savings could be expected, since there is only a single map instead of 11 and since keys for the map cannot be of a primitive data type, so a

Java Long Object had to be used instead. Therefore, one can expect the savings to be even bigger.

For this reason, this rather big change in the graph representation has been implemented. The graph is now represented by a single map. The keys are node IDs like before. The values are newly created *Node* objects. Each node object contains the ID, the coordinates, the height and a map with newly created *NeighborInfo* objects. This map contains the IDs of neighbors as keys. A NeighborInfo object contains the ID again, the distance to this neighbor and all the weights.

In this case, the node IDs are still used twice, once as key for the map and once within the object. The reasoning for this is to have self-contained objects. Even if a node is for example copied out of the map and used somewhere else, it should still be possible to uniquely identify the node without having to pass the ID along separately. The same goes for the NeighborInfo objects. Overall, there are still big savings while getting better maintainability. This also holds in face of the decision to store all nodes in the graph instead of only crossings. While it is not strictly necessary, the routes that are returned from the application profit from this change because they will no longer show routing artifacts as shown in the previous work [1] and displayed in Figure 3.1. The storage space for Switzerland using this new structure dropped from 2.7 GB to 1.4 GB. This is still a saving of around 50 % despite having many more nodes.

Another advantage only became obvious while implementing this change. The code size is reduced, as similar code for filling the maps with the different weights is now collapsed into a single procedure. Also, since now more data are contained within one object, it also became possible to encapsulate functionalities into the classes for the objects. For example, instead of looking up values within different maps and combining them to calculate an edge weight, there is a method call onto a NeighborInfo which returns the result of the calculation. This makes the code easier to understand and therefore better maintainable.

3.2 Storage

This section covers the portion of Figure 2.1 where the files are loaded by the routing server. As mentioned in Section 2.1, the data for the routing server of the previous version are stored in small files, each covering a square of 0.01 by 0.01 degrees. This amounts to roughly a square kilometer in Switzerland. This is combined with the data structure described in Section 3.1.2. Each of the 11 maps used to represent the Graph is stored in its own file. When loading the data for an area of 10 km by 10 km, over 1000 small files have to be read, most of them smaller than 1 KB. This causes major overhead and thus, two alternatives have been implemented. In Section 3.2.1, a first draft using a improved method using

Figure 3.1: Screenshot of routing artifact of previous work with omitted nodes shown in red [1]

files for storage is presented, Section 3.2.2 covers the finally evaluated solution of storage in a database. The storage engine can however be switched easily by recompiling the software with a different configuration.

3.2.1 Storage in Files

The first big improvement of file based storage is a direct consequence of the new data structure described in Section 3.1.2. Instead of storing 11 maps per small tile, there is now only one. Apart from this obvious improvement, more changes have been made.

The files in the previous version are written to disk using a serialization library called *kryo* [12]. The job of the library is to take an object and write it to a file without the user having to care about the representation. The user supplies a map and it gets written to a file. On reading, the same map comes out. There is however the possibility to provide a custom class that extends the kryo "Serializer" class, which is used in this new version. With that, the behavior of writing and reading can be specified exactly. This is depicted in Figure 3.2. The new data structure consists of a single map containing the node IDs as keys and the node objects as values. Because every node object still contains its own ID, the whole map-structure can be dropped for serializing. Instead, each entry of the map is written to the file: first the node ID, the coordinates, the height and then the number of neighbors. As the neighbors are stored in a map within the node, they can be stored in the same way: First write the neighbor ID, then the distance and the different weights. Of course, all these data are written in binary form to save storage space.

The reconstruction of the map is then done during reading. The info for one node are read, the node object is then constructed and added to a map that collects all the nodes. This way, it is even possible to change the data structure without having to re-generate all the data files, since the files only contain a bunch of node objects and only during deserialization, they are stored in a map. By changing the deserializer, they could be stored in a sorted List for example. Before it would have been necessary to re-generate all data files, because the map was stored as well.

At this point, the data storage is already much more efficient. Even with the changes so far, loading an area of 10 by 10 km, still requires 100 files to be read, which are still very small. So the tile area has been increased to 1 degree by 1 degree, which amounts to roughly 100 by 100 km in Switzerland. Since only routes up to 100 km are supported, there is now only the need to load a single file or up to four, if we are near the corner of a tile. The largest tile for Switzerland has a size of less than 330 MB, which is not a problem for SSD based storage.

This solution certainly eliminates most of the file system overhead for opening and closing files. There are however also disadvantages. The biggest is looking

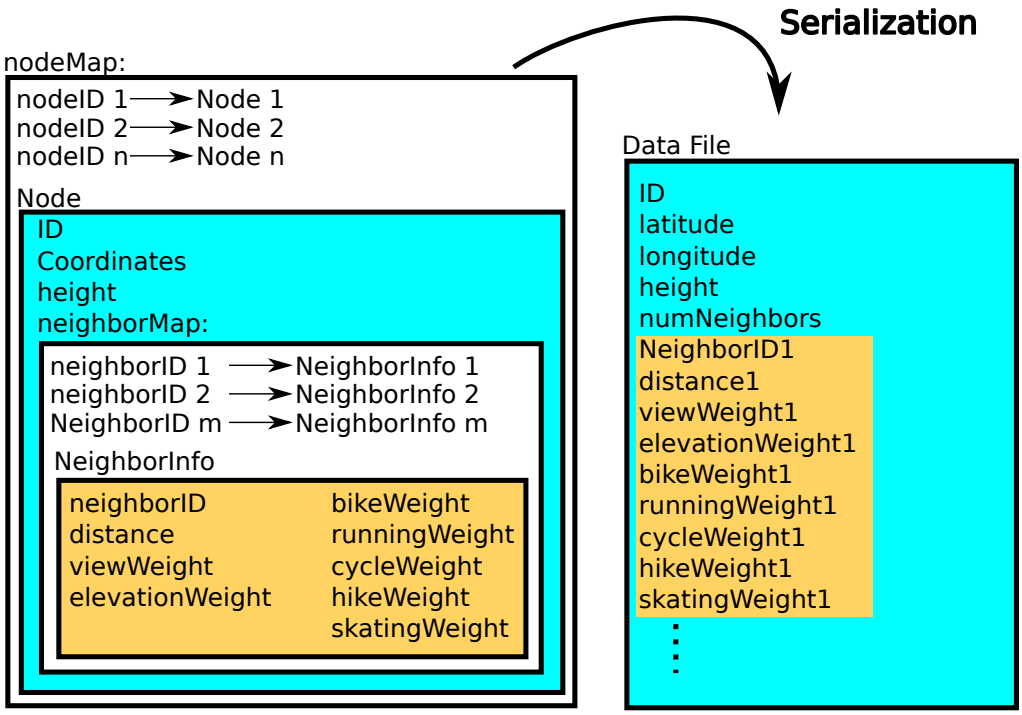


Figure 3.2: Illustration of Serialization

for the node IDs of the start and end points. As described in Section 2.1, the routing server gets passed the coordinates for the start and end points and first has to find the node IDs of the closest nodes. In the previous version, the server reads the corresponding 0.01 by 0.01 degree file again, looks at all the nodes and calculates the distance for each one. Since there are not many nodes in such a small area, this works rather fast. Doing this for a file with over a million nodes takes now much longer. But instead of trying to optimize a tradeoff between file system overhead for small files and the time to find the first and last nodes, another approach described in the next section has been implemented.

3.2.2 Storage in a Database

As we have seen in the last section, file based data storage has been greatly improved. There is however always the problem of choosing the area a file should cover. No matter the choice, there will always be some cases where another file size would have been better. If we choose a large area, short routes have to deal with an unnecessary amount of data. On the other hand, with a small area per file, there will be overhead in the file system when opening many files for a long route.

Instead of files, a database containing the graph has been set up. For this

use case, the *MongoDB* database system has been chosen [13]. Instead of storing data in tables as in more traditional Relational Database Management Systems (RDBMSs), MongoDB stores data as JavaScript Object Notation (JSON) documents. It is a so called "NoSQL" database. This means there is no schema and no tables. Instead of tables, JSON documents can be stored in collections. A JSON document as used in this project is shown in Listing 3.3. In this listing, we can see the building blocks of such a document. The simplest case are fields which follow the convention "name:value". In this example, the _id and height field follow this convention. The neighbors field is more involved. First, the brackets indicate an array of values. Then, indicated by the braces, new JSON documents make up the contents of the array, each one representing exactly one neighbor. It is however not necessary to fill an array with documents, as we can see in the coordinates field, where there are numeric values stored directly within an array. The reason for the seeming redundancy in the coordinates field will soon become clearer.


```
{
  "_id" : NumberLong("3357100045"),
  "coordinates" : {
    "type" : "Point",
    "coordinates" : [
      5.9907745,
      46.1446253
    ]
  },
  "height" : 372.09198458400533,
  "neighbors" : [
    {
      "nID" : NumberLong(951519442),
      "bikeWeight" : 10,
      "cycleWeight" : 1,
      "elevationWeight" : 4.953996495380659,
      "hikeWeight" : 10,
      "skateWeight" : 0,
      "viewWeight" : 1680,
      "runningWeight" : 3,
      "distance" : 5.257164478302002
    },
    {
      "nID" : NumberLong("3364720741"),
      "bikeWeight" : 10,
      "cycleWeight" : 1,
      "elevationWeight" : 5.02657069862113,
      "hikeWeight" : 10,
```

```
        "skateWeight" : 0,
        "viewWeight" : 1680,
        "runningWeight" : 3,
        "distance" : 11.92383861541748
      }
    ]
}
```

Listing 3.3: Example of a Node encoded in a JSON document

We can see from this example that this format closely resembles the new graph representation presented in Section 3.1.2. This is one reason for choosing this database system. It is rather intuitive to convert from a node object to a JSON document and back. Using a RDBMS, the nodes could not be stored in a single table row because of the varying amount of neighbors. Thus, a node would be spread across multiple tables and not form a self-contained unit, which would make the storage harder to understand.

Another reason for choosing MongoDB is the support for geospatial queries out of the box. The coordinates field in the listing contains *GeoJSON* object. When the data are written to the database in the crawler, a geospatial index is created over the coordinates field. This means, there is a quick way to use the location of nodes in a query to the database. The problem of finding the node IDs of the start and end position mentioned in the last section is solved with a single query which is shown in Listing 3.4 taken from the documentation [14]. Using such a query, the IDs are returned instantly for all practical purposes, independent of the number of nodes stored in the database[1].

```
{
  $nearSphere: {
    $geometry: {
      type : "Point",
      coordinates : [ <longitude>, <latitude> ]
    },
    $minDistance: <distance in meters>,
    $maxDistance: <distance in meters>
  }
}
```

Listing 3.4: JSON Query for finding a node close to a given set of coordinates

This query returns the nodes sorted by distance from the given coordinates. To get the closest node, looking at the first returned document is enough. The parameters for minimal and maximal distance are optional. The name of the

[1] This does not hold if the geospatial index does not fit into Memory. See Chapter 5 for details.

operator, "$nearSphere" indicates that it uses spherical geometry, so that the implementation of such functions can be avoided.

In principle, this query can also be used for getting all nodes needed for a routing request by restricting the maximal distance. There is however a more fitting operator called "$geoWithin". This operator takes a shape (for instance produced by a "$centerSphere" operator for points within a circle constructed on a sphere) and returns objects entirely within that shape. The difference between these two is that "$nearSphere" returns sorted nodes while "$geoWithin" does not. Therefore, the latter variant has a better performance.

The storage size depends greatly on the configuration of MongoDB. A first test for the area from 45N5E to 55N10E used around 60 GB of storage. It turned out that an old version of MongoDB was in use. A newer version using the new *WiredTiger* storage engine instead of the older *MMapv1* engine uses around 10 GB of storage. That is even slightly less than the file based storage for the same area, even though the database uses additional storage for indexes. Apart from setting the storage engine explicitly, the default settings are used and no further optimizations of the system for the database have taken place.

The greatest performance increase has been realized for short routes when comparing to the big files of 1 by 1 degree, because of the search for the start and end node IDs and the overall lower number of nodes. For longer routes, the files can be loaded faster than the database can load a large amount of nodes, so that even with the advantage of finding the start and end nodes quickly, the solution using files is still faster. This could be countered by trying to restrict the amount of nodes retrieved from the database further than in the current version.

3.3 Functional changes

In this section, changes made to the functionality of the program are described. Even though they change the behaviour, care was taken to keep such changes to a minimum. This way, the quality of the results should stay roughly the same while the efficiency improves.

3.3.1 Distance Calculation

At one point during development, the routing server was observed using most of its time to calculate the arctan function. This happened as a part of the distance calculation in spherical geometry, using the *haversine* formula [15]:

$$a = \sin^2(\frac{\varphi_2 - \varphi_1}{2}) + \cos(\varphi_1)\cos(\varphi_2)\sin^2(\frac{\lambda_2 - \lambda_1}{2}) \tag{3.1}$$

$$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a}) \tag{3.2}$$

$$d = R \cdot c \tag{3.3}$$

This formula calculates the distance d between two points on a sphere on a great-circle, which is a circle that connects the two points and has its center at the center of the earth. φ_1 and φ_2 are latitudes of the points and λ_1 and λ_2 are their longitudes. r stands for the radius of the earth.

These are obviously many trigonometric calculations for getting a distance between two points. Since only routes up to 100 km were supported, what would happen if we assume the earth as flat within such a short length? The first consequence would be that now the euclidean distance can be used, which reduces the calculations to two multiplications, one addition and a square root. But what about the error? It turns out that an analysis for this has been done already [16]. In this application, even for long routes, only distances up to 5-10 km have to be calculated, since the routes are calculated in smaller parts. For a length of 5 km, the error is less than 1 % even at 89°N.

To keep with the goal of changing the least amount of functionality, not all distance calculations were changed. According to the profiler, the great majority of calls to the distance calculation originated in the A* algorithm for calculating the shortest path between two nodes. Therefore, only this algorithm has been changed to use the euclidean distance. In fact, to even save the time for calculating the square root, the squared euclidean distance is used. The rest of the calculations do not have a big impact and have been left untouched. In a profiling run following this change, where a 60 km long route was calculated multiple times, there were 37.5 million lookups of the distance to a neighbor, which is precomputed during crawling, 2.7 million calls of the euclidean distance function and only 200000 to the haversine formula. This shows that there was little need to change the rest of the haversine calls.

3.3.2 Request Timeouts

The routing server can take a long time to complete a request, especially when there is a long route to calculate in difficult terrain. It might even be that no route is found. But even finding no route is better than having to wait for an indefinite amount of time. Failing fast is a positive quality. In Chapter 5, it will be shown that the previous version does not reliably show this quality. Thus, the handling of request timeouts has been improved.

The target was to specify a time, for example 15 seconds. When this time has elapsed, the routing server should abort and return the best route found so far if one was found at all. The old version has a timeout of 30 seconds, but fails to ensure it, because it was unintentionally reset each time the function for finding the shortest path was entered.

Instead of simply fixing this bug, alternative possibilities have been explored. When calculating a request, most of the time is typically spent in a multithreaded portion of the code. The main thread constructs a list of route finding tasks. Each task should compute one route starting with a different random starting direction and the best route should be kept. The main thread submits these tasks to a thread pool which executes them all, using as many processor cores as available. The result of each task is compared with the best result so far immediately after finishing.

In the previous version, the time elapsed is checked at a single point within the tasks and if it is too long, the Task is aborted. Still unfinished tasks would however still start execution up to this point. As an alternative, the thread pool has been given a timeout during construction so that the main thread could stop the thread pool. This solves the problem of unfinished tasks starting after the timeout. Running threads are also stopped, but not always. When a thread is blocked, for example when accessing a log handler, it will not be stopped. This occurs less frequently with more quiet log levels. Still, long running threads can be observed, even when the main thread already has moved on and returned a result for the request. This is bad in a second way, because it prevents the garbage collector from freeing up the data of the request. During benchmark situations, where a new request is sent as soon as the old one returns, hundreds of threads belonging to already finished requests are still there.

This has lead to the current solution. The thread pool still has the newly implemented timeout. This is complemented with each task checking regularly if there is still time for calculating as a safety net. In order to avoid similar bugs in the future, the type of the variable holding the timeout value has been set to final. This ensures that the value is set on the start of a request and not changed afterwards. Another lesson of this excursion is to keep logging in this multithreaded environment to a minimum.

3.3.3 View Calculation

In the old version of the crawler, the calculation of view weights has problems with efficiency as well as quality. A comparison of the functionality is illustrated in Figure 3.3. Starting from a node S, both solutions look at the terrain in 8 different directions as shown in the top down view. The directions are easy to calculate, since the SRTM height data already come in grid form of a 90 by 90 m grid.

More interesting is the behaviour along such a line. Both versions count the number of grid tiles that can be seen. The left part of the figure indicates that by using orange or green color for unseen or seen nodes. The old version counts until a grid tile is higher than the starting tile. This is indicated in the right part of the figure with the short orange dashed line. The total view weight is then

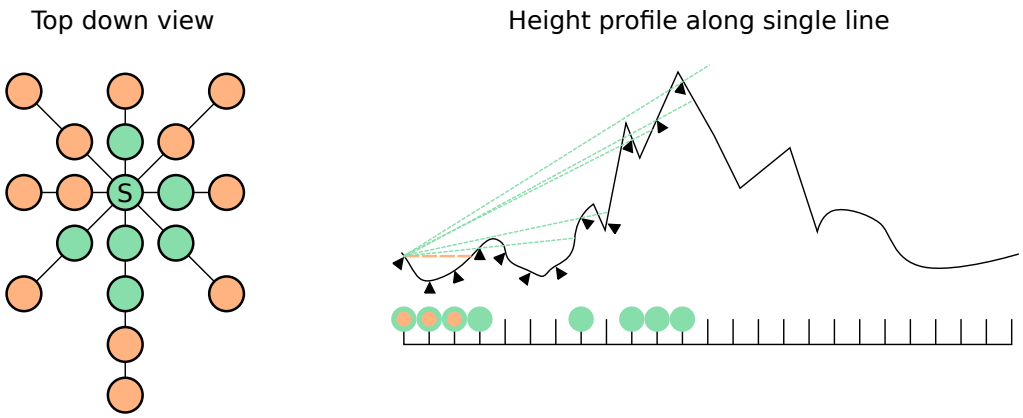


Figure 3.3: Comparison of view calculation old and new version

calculated as shown in Equation 3.4. The formula was taken out of the code, in the previous thesis [1], the fifth power was not mentioned. The reasoning for this way of calculation is not entirely clear. In Figure 3.4, we can see that as soon as 20 tiles (or 1800 m) are reached, the view weight does not grow significantly anymore. This is a rather short distance for a view, since it is quite common to oversee larger distance, especially from hills.

$$weight = 1 + \sum_{directions} \left(\frac{\left(1 - \frac{1}{1+count}\right)^5}{2} \right) \tag{3.4}$$

Another problem with this previous approach is that it neglects that one can also look upwards. If there is even a slight hill in front, the great view towards a high mountain range behind it is not taken into account. An exemplary mountain range is sketched in the right part of Figure 3.3. The measure underneath represents the grid in one direction with marks to indicate when a point is counted. The black triangles point to the location in the terrain directly above the grid points to help indicating the height.

To correct the behaviour of the previous version, a new way of calculating this line of sight has been developed. We still follow the directions on the grid. At the start, we initialize a slope with a value of negative infinity. Each tile is counted, if its height is above a straight through the starting point with the given slope. some of these straights are indicated by the dashed green lines in the figure. If a tile is counted (shown by the green circle on the measure in the figure), the slope is updated. This goes on, until either 1000 tiles have been checked or until the height needed is above 9000m. Furthermore, instead of calculating the resulting view weight according to Equation 3.4, only the sum of all the counts for each direction is stored. Instead of using the old calculation for the view weights, the count is scaled by a fixed number. To obtain the number,

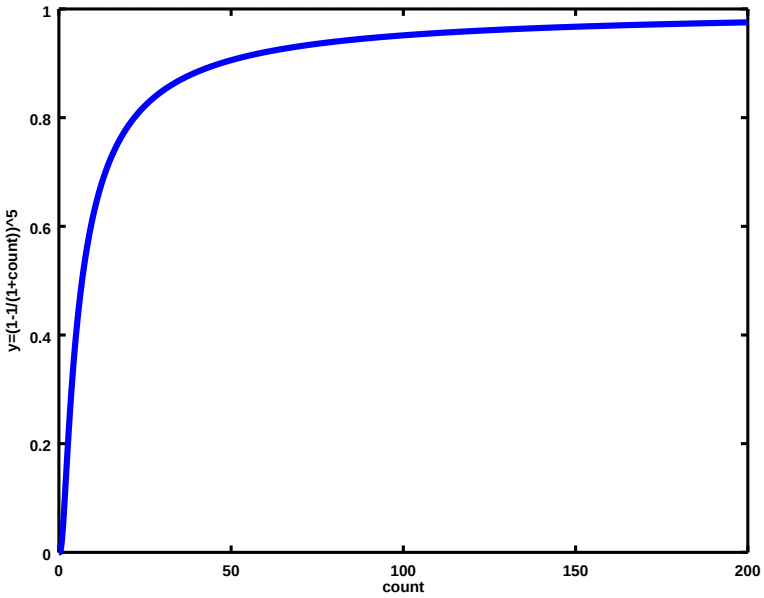


Figure 3.4: Plot of $(1 - \frac{1}{1+count})^5$

100000 nodes have been sampled from the database at random and the resulting average view count calculated (Which amounts to around 107). The number has been chosen to bring the average view weight to roughly the same size as the average activity weight of the same nodes.

The other part of the view calculation is efficiency. To get the grid, the previous version generates a XML file of the same form as OSM data, containing a node for each grid tile. This XML file is fed to osmosis in order to get height data via the osmosis-srtm-plugin (see Section 2.1 for an explanation). After that, the XML file is parsed, which means loaded into memory as a whole and the height data are extracted. Only then, the view calculation was could be out.

For the new version, a parser for the SRTM data files was written, since the format is simple. Each data file contains the data for a square of 1 by one degree as 16 bit signed Integers. The values are organized in rows from west to east and the rows are present in direction from north to south [17]. Using this description, a tile, its 4 direct neighbors and its 4 diagonal neighbors are read into a big array (which is still way smaller than a corresponding XML file). For each grid tile in the center, the view calculation is executed an the result is stored in a second, smaller result array. For each node of the routing data, the coordinates are converted to the nearest array index and the value is added to the node. This way of calculating is of course most efficient when the OSM data are dense, such that at least one node of the routing graph lies in each tile of the result array. In more thinly settled parts of the world, there might be better strategies.

3.3.4 XML Parsing

Weight calculation

The Crawler is responsible for converting the OSM data to a graph with nodes and edges with weights. As already mentioned in Section 2.1, the XML file contains elements called nodes and ways. The nodes can be directly translated into the nodes for the graph, as they have an id and a location. They can also contain tags, but they are not relevant for the calculation of edges and weights, because the tags on nodes only refer to that single point referred by that node and do not necessarily have to refer to a way. For examples, single trees can be marked by a node that is tagged accordingly.

For the edges, the ways are the important element. They have an ID and contain a list of node reference elements. In addition, they contain tags as well. The tags are the most important thing in determining how to add the way to the graph. A way could for example also represent political borders for a village. This way must not be added to the graph. So, how can be decided if and how to add a way to the graph? This will be answered using the example in Listing 3.5.

```
<way id="3977794" version="18" timestamp="2016-10-30T11:19:00Z"
uid="74900" user="FischersFritz" changeset="43281412">
  <nd ref="20599757"/>
  <nd ref="2872818310"/>
  <nd ref="20599756"/>
  <nd ref="20599755"/>
  <nd ref="1676882153"/>
  <nd ref="4471205059"/>
  <nd ref="4471205055"/>
  <nd ref="20599753"/>
  <nd ref="2872542380"/>
  <nd ref="4471205051"/>
  <nd ref="2872542394"/>
  <nd ref="4471205035"/>
  <nd ref="1676882154"/>
  <nd ref="20599751"/>
  <nd ref="2872542401"/>
  <nd ref="20599750"/>
  <nd ref="2872542426"/>
  <nd ref="20599749"/>
  <nd ref="2872517007"/>
  <nd ref="2872513095"/>
  <nd ref="2872513805"/>
  <nd ref="20599746"/>
  <nd ref="2872513807"/>
```

```
  <nd ref="20599743"/>
  <nd ref="2872513844"/>
  <nd ref="20599742"/>
  <nd ref="2872513858"/>
  <nd ref="20599740"/>
  <nd ref="20599739"/>
  <nd ref="20599738"/>
  <nd ref="20599736"/>
  <nd ref="2835671390"/>
  <nd ref="2835674307"/>
  <nd ref="20599734"/>
  <tag k="name" v="Locherbodenstrasse"/>
  <tag k="access" v="yes"/>
  <tag k="bicycle" v="yes"/>
  <tag k="highway" v="unclassified"/>
  <tag k="surface" v="asphalt"/>
  <tag k="motorcar" v="no"/>
  <tag k="motor_vehicle" v="destination"/>
</way>
```

Listing 3.5: Example of a way in Switzerland

The way has several attributes in its opening XML tag. They are however not needed for routing purposes and are thus discarded. The list of "nd" elements contains the node IDs in the way in the right order. This defines the edges that this way would create. In this case for example, node 20599757 is connected with node 2872818310, and so on. The only thing left to interpret are the tag elements. They have two attributes, a key, denoted by "k" and a value, denoted by "v". For the purpose of this work, the highway key is the most important one, since all kinds of paths are denoted as a highway. This holds from the biggest motorways of a country to the smallest hiking trails far off in the countryside. Therefore, already in the previous version, all ways that are not a highway are filtered out by osmosis as well as nodes not part of a highway.[2]

The tags in this example listing indicate the quality of the route. The first tag, the name is not relevant for the edge weights. The access is especially interesting if it is set to no otherwise it is not too interesting. The next tag allows bicycles, so the weights for cycling and mountainbiking should get a bonus. The value of the highway tag does not give any additional information, while the surface information is valuable for the inline skating weight.

[2]For the sake of completeness, another element called relation also is filtered out. A relation is to a way what a way is to a node. In other words, a relation contains a set of ways and can used for example to designate regional, national or international cycling or hiking routes. They provide however little additional information, as the key information about the road quality is contained within the tags of the ways.

It is easy to see, how this information in the tags can be used to construct a weight. Simply store desired tag value combinations in a table and look up the weights on encountering a tag during parsing. This is the strategy of the previous version. The values are stored in a JSON file for each activity. The issue with the method of the previous version is that the weights are initialized to the value 1 if the highway tag is present and 0 otherwise. On encountering a tag, the current weight is multiplied with the stored weight from the file, which can also be negative. If only one undesired tag with negative weight were to be encountered, this method would work, but with two or any even number, the negative signs would cancel out.

As a consequence, multiple changes were made to this mechanism. The first and most important one was to change the weight calculation to additions to solve the issue with canceling negative weights. Further, the mechanism of table lookups has been changed. Instead of storing the values within a JSON file, the values have been imported to the database, since it also works with JSON data and allows further analysis and easier management. In order to gain more flexibility, a separate weight table has been created for each value of the highway tag until 97.5 % of the highway tag values were covered. This statistical information was gathered through taginfo [18].

Depending on the value of the highway tag, weights could already be given based only on that or based only on combinations of other tags. For example, a residential highway is most likely usable for all activities, so all weights get a little bonus. Walking along a secondary highway is probably not a good idea, unless it has a sidewalk. So the “secondary” value of the highway tag would get a zero weight on its own for everything but a mountainbike or bicycle. Other activities could still get boni from other tags.

By using a separate value table for each value of the highway tag, it is also possible to get very fine grained control over the weights for the graph. One use case is for example to give a sidewalk along a tertiary highway a better weight than a sidewalk along a primary highway, since there is probably less traffic.

To assign weights to the tables during XML parsing, a configuration switch has been added to activate an interactive command line parser. If a combination of tags and values is unknown, it asks whether to create a new entry for weights or not or to ignore this combination for the current session. The advantage of that is that the user assigning the weights gets to see real world tag combinations, can look them up and get an idea of the quality by examining the location. This way, much can be learned about the information present in the OSM data. The biggest disadvantage of this procedure is its slowness. Since the weight information are distributed across around 15 tables, there are many different combinations that all should be entered by hand, which takes a lot of time and patience. As a remedy, a single table with weights applying to all highway values was introduced. At the moment, this global table covers mainly values

prohibiting ways due to private or forbidden access. The other tables were filled with the most common combinations by hand, using again taginfo.

The different activities were assigned weights based upon their characteristics. For hiking and mountainbiking, there are scales for difficulty in the OSM data. If a difficulty is present, it can be assumed that the path is very well suited for that activity. For low to medium difficulties, big boni have been assigned, the highest difficulties are penalized since we want to avoid sending users into dangerous terrain without planning. Cycling is less penalized for big roads with much traffic and got boni for cycleways. Inline Skating is heavily constrained to low traffic situations with good road surface, but could use footpaths as well as cyclepaths. Running also tries to avoid heavy traffic, but is not confined to good road surfaces.

Memory efficiency

During a late stage of development, the crawler no longer has been executed on a personal development computer but on the server that should host the final application eventually. No significant problems were expected, as both machines use the same operating system with the same version of Java. This turned out to be a wrong assumption. On the server, the Java Virtual Machine (JVM) has been observed to terminate with an out of memory error because of garbage collection overhead. This means that the code was spending more than 98 % of the time in the garbage collector and less than 2 % actually executing code. Even allocating more memory for the JVM on the Server did not help. The XML file to be parsed only has a size of 500 MB. Even when loading that into memory as a whole, one would expect this to work when 11 GB of memory are available on the server. This problem has not occurred on the development computer, because it has 64 GB of memory.

For diagnosis purposes, the JVM has been instructed to generate a heap dump on out of memory errors. On Analysis, it turned out, that the XML parser library was using over 9.5 GB of memory. Since many XML files are parsed in the same run of the program, a memory leak was seems to be the logical conclusion. It is unclear where the leak was caused and if the code of this project was responsible or the library. To get the code to run on the server, the XML parsing library has been changed to StAX [19]. The main difference to before is that instead of loading the whole XML document into memory, this library works in a sequential way. The XML is read, and upon encountering an element, an event is produced. It is in the responsibility of the calling code to poll the events one at a time and handle the information accordingly. Such events are for example an opening or closing tag.

Since the OSM data are rather simple, it was not hard to replace the old calls to the parser with two new event handling functions for nodes and ways.

Also, since the ways are parsed sequentially now, it has become unnecessary to store a list with all ways. Instead, as soon as a way is encountered, its tags are evaluated to calculate a set of weights and the edges in the graph are added. After that, the way is discarded, which saves memory as well. Note that this works efficiently, because in the OSM data, the nodes are saved before the ways. That means, as soon as we are at the first way to parse, we know that we have all nodes that we need to know.

Chapter 4

Frontend

As we have seen in the last chapter, the backend part of the application experienced many changes. Why is the frontend part also relevant? The answer is that the existing frontend part is out of date. It even has been removed from the Google Play Store. First, the issues with the existing Frontend will be explained, then the solution of them.

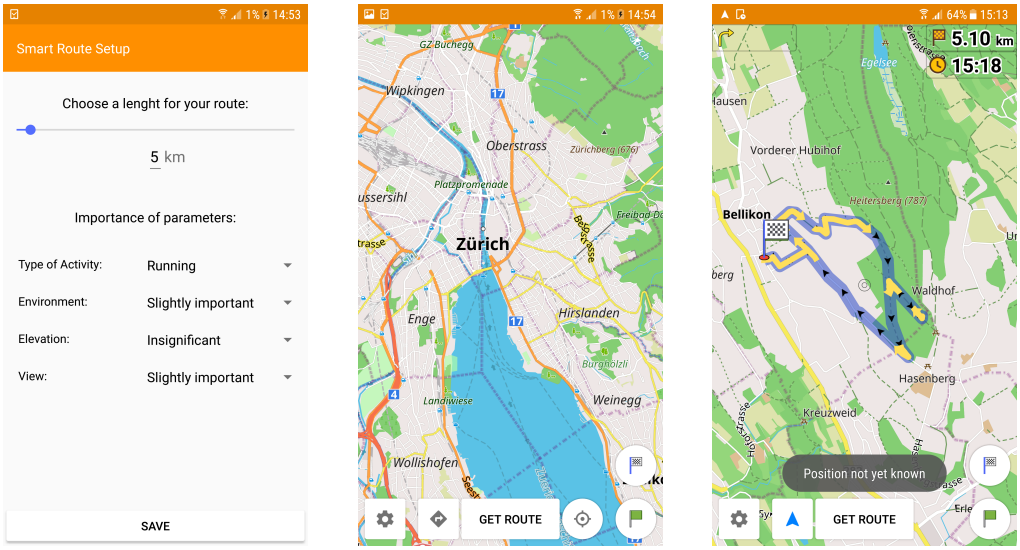
4.1 Frontend of the existing solution

The existing frontend of the previous version is an Android app called SmartRoute. This app is based upon *OsmAnd* [20]. OsmAnd is a navigation app that uses OpenStreetMap data as a base. This app has been extended with buttons to select start and end point of a route and a settings screen to set the parameters for the desired route. This can be seen in Figure 4.1

This fork of an existing app has several problems. The first problem is that it was removed from the Google Play Store because of issues with the necessary device permissions. Since not too many changes have been made to the original code of OsmAnd, the cleanest solution would have been to rebase the changes onto a current version of OsmAnd. However, this would still result in a lot of work, because OsmAnd is developed very actively. At the time of writing, OsmAnd has diverged from the used version with over 3000 commits to the master branch of their version control system. This implies that in a year from now, the same problem could come up again: A new Android version that requires slight adaptations could be released and then the app would probably be pulled again from the store.

A second problem is the licensing. While the code of OsmAnd is licensed under the GPL, the User Interface (UI) is not [21, 22]. So, while it is no problem to modify the code and even reupload or resell it (As long as the modified code is obtainable), written permission is needed to put a modified version with the original UI to app marketplaces such as the Google Play store.

Finally, another problem is that the SmartRoute app is only running on



(a) Settings Screen (b) General Interface (c) Route generated with new Backend

Figure 4.1: Screenshots of SmartRoute App

Android. OsmAnd works on Android as well as iOS. Furthermore, it would be desirable to support even more devices such as dedicated navigation devices, smart watches etc.

These problems led to a decision to introduce a completely new frontend as described in the next section.

4.2 Universal Frontend

The solution that has been chosen for all the problems mentioned in the previous section is to drop the existing SmartRoute app. Instead, the frontend is now realized by a simple HTML page extended with the JavaScript *OpenLayers* map library [23]. Figure 4.2 shows a proof of concept of the new Frontend.

This web page works very similar to the old SmartRoute app. The user can select a start and end point on the displayed map. This map uses OSM data as well. The control elements below allow to select the parameters in the same manner as in the SmartRoute app. The difference here is that the user can adjust his priorities for a nice environment, hills on the route and the view in a continuous way instead of selecting predetermined values. A click or tap on the "Get Route" button sends a request to the routing server. Once the request is finished, the result is displayed on the map and the "Download Route" button gets activated. This button starts a download of a GPX-File containing the

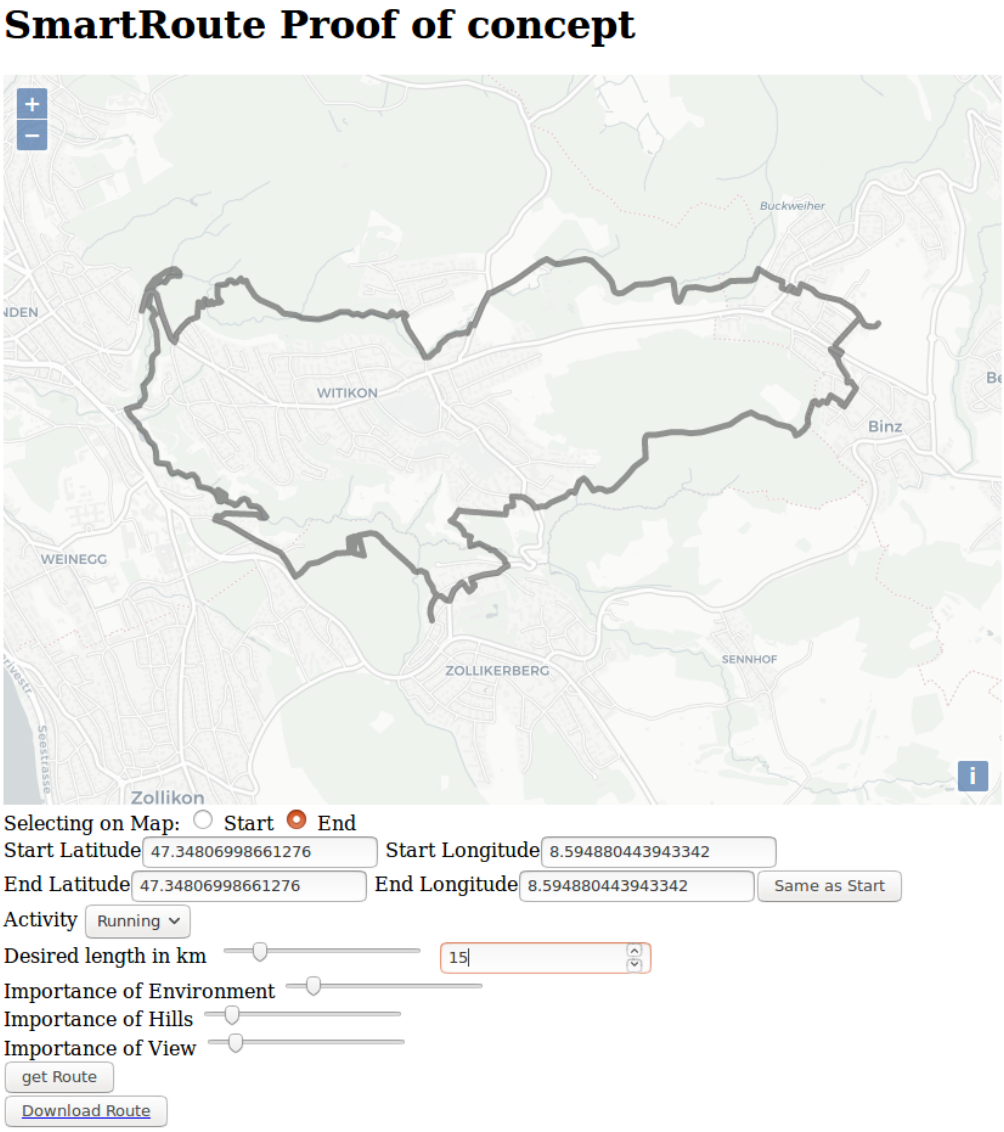


Figure 4.2: Screenshot of new frontend displaying a running route in the vicinity of Zurich. Displayed are the contents of a browser window on a desktop computer

displayed route (See Appendix A for a short introduction to the GPX format). In short, this solution extends the compatibility from a single mobile operating system to every browser that supports HTML5 and JavaScript. The only change to the routing server is the addition of a second output option to generate GPX data instead of a list of coordinates.

This GPX file can be opened by the current, unmodified OsmAnd app. But since the format is open and universal, there are many apps and devices that support a GPX import. The file can for example be downloaded on a desktop computer and then loaded to a dedicated navigation device. This might even improve safety, as no phone battery is drained during the activity which is useful in case of emergency.

Chapter 5

Results

For testing, routes have been generated originating from random points, as stated in Section 2.2.1. For the tests, the database has been loaded with the data of the countries marked in green on Figure 5.1.

These countries have been put into the database using the crawler with data excerpts from Geofabrik [5]. The reason for using excerpts instead of the whole planet is that the osmosis run in the crawler where the input data is split into tiles works much faster when there are fewer tiles to generate. The smaller the extent of the area imported, the faster the osmosis call. These data have been imported to the database within one week, with potential for more savings by using available provinces of countries instead the full countries.

The graph stored in the database contains 533 million nodes taking up storage space of 75 GB. The database reports the uncompressed data size to be 260 GB. The OSM statistics of 20th December 2017[1] indicate that this amounts to 12.6 % of the total number of nodes for the whole planet. The nodes are however not exclusively used in ways with the tag "highway". According to the same statistics, there is a total of 424 million ways for the whole planet. Out of that, only 113 million or roughly 25 % have the highway tag [18]. In addition, only nodes that actually have a weight greater than zero for any activity are included in the graph. Motorways are for example not included. Furthermore, most of the missing area has not a lot of data. The compressed data size for the excerpt of Africa is 2.1 GB, which is less than Germany alone (3.1 GB). The excerpt for Oceania, which includes Australia and the surrounding territories takes up around the same space as Austria (540 MB). Thus, from the perspective of storage, this new solution can easily store the whole world and that this test used most of the available data.

The geospatial index, which is essential for retrieving the nodes from the database takes up 7.8 GB, the mandatory (but unused) index on the IDs takes 6.6 GB. This is problematic on the available hardware. For the database to work properly, at least the used index as well as the working set (the data requested

[1] http://www.openstreetmap.org/stats/data_stats.html

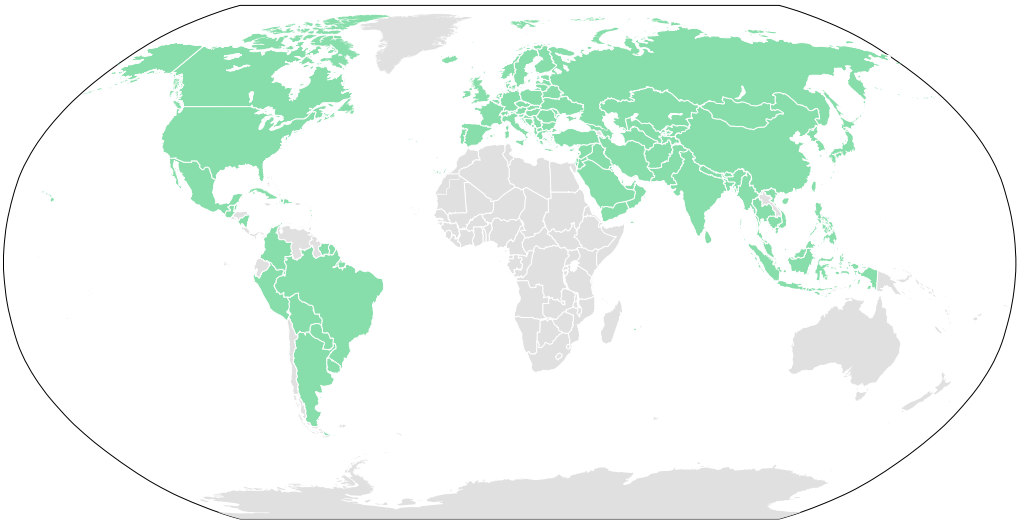


Figure 5.1: Map with countries imported to the database

at the same time) should fit in memory. In addition more memory for usage as cache is preferred.

The server available for these tests is a virtual machine with 14 GB of memory. At the time of the first tests it had however only 11 GB. In this situation, when running the routing server and the database in parallel, the database process has been killed by the Linux kernel out of memory killer. Restricting the available memory to the routing Server to 4.5 GB and the database to 5.5 GB has prevented the activation of the out of memory killer, but the database still crashes or hangs sometimes because it can not allocate more memory. In order to run tests despite this resource shortage, the routing server has been moved off to the development machine for the benchmark, so that the routing server still runs locally but queries the database on the virtual machine. The database with full data does not fit in storage of the development computer. With 11 GB on the virtual machine and the database running without a limit, the out of memory killer still kills the database process sometimes on long routes (80 or 100 km). With 14 GB, this does not happen anymore[2]. This way, the minimal requirement of 14 GB for the database has been found. The routing server (running on the development machine with 64 GB total memory) has been observed using up to 6-7 GB under the load of 1 request at a time, but works fine with a restriction to 5 GB. Compared to the previous version, where the routing server had to be allowed to use 10 GB to serve Switzerland, 20-25 GB for world wide route generation is a great improvement. Of course, more memory will benefit the caching of the database.

[2]The database can still get slow on long routes, but resorts now to swapping instead of running out of memory

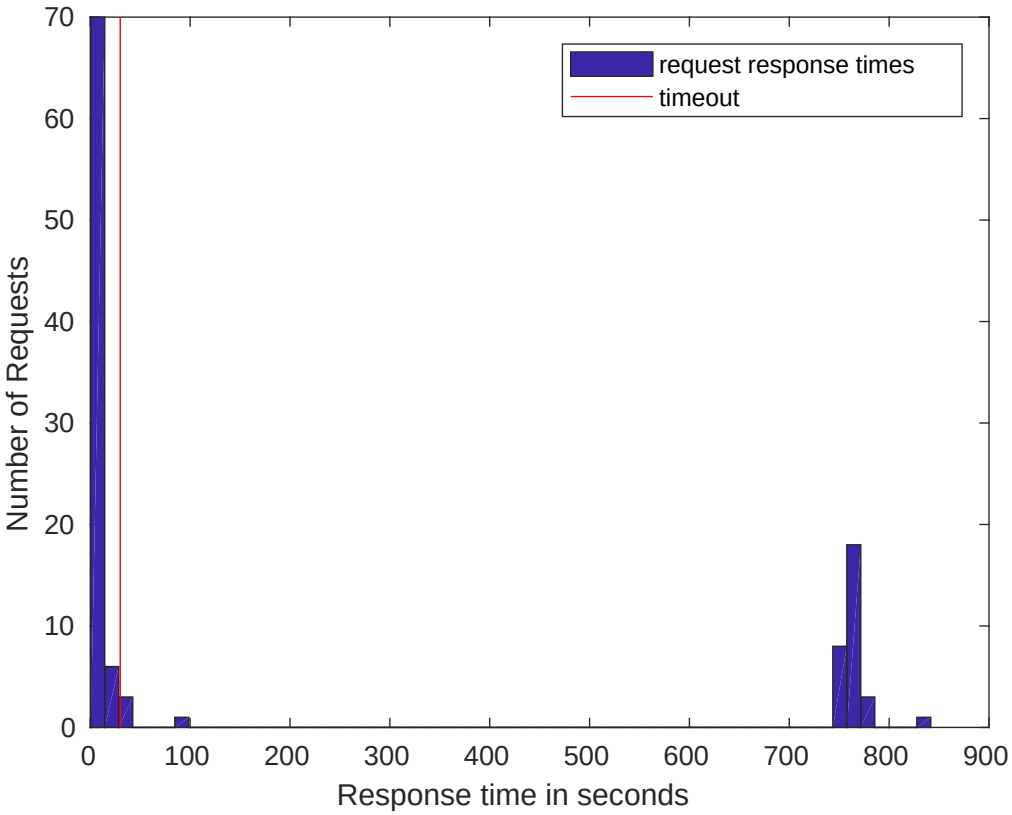


Figure 5.2: Response times of previous version for routes with length of 5 km within 100 km of Lucerne

5.1 Performance of previous Solution

For comparison purposes, before deploying the new solution to the virtual machine, the previous solution has been tested with the same benchmark intended for the new solution. For this test, 30 points within 100 km of Lucerne have been randomly selected as described in Section 2.2.1. The first request has completed in 30 minutes and 40 seconds, which is expected because of the loading of all data. After 7 hours, the test has been aborted.

The reason for this is that with the exhibited performance, the test would not have finished in reasonable time. In the remaining 6 and a half hours, only 110 requests were completed. All completed requests have a route length of 5 km and either mountain biking, running, cycling or hiking as activity. A Histogram of the response times is shown in Figure 5.2.

The red line shows the timeout value of 30 seconds. The 31 requests that have taken more than 600 seconds all have returned an empty route, the others have been successful. If 110 requests take already more than 6 hours, there is

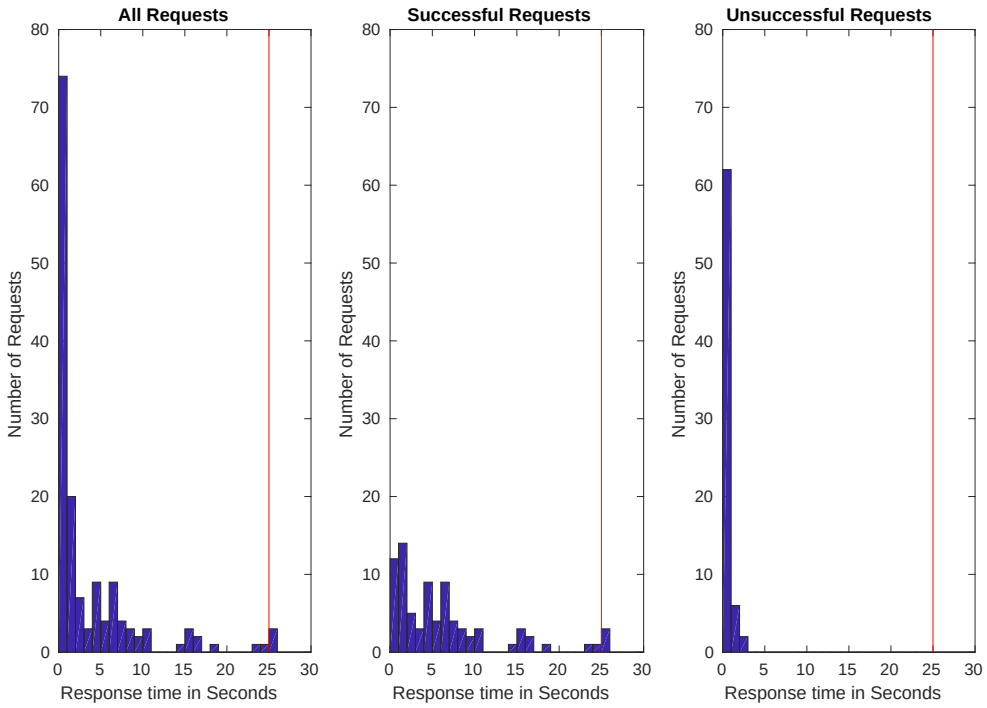


Figure 5.3: Response times of new version for routes with length of 5 km within 100 km of Lucerne

no point in forcing the whole benchmark to run with longer routes. The average time of the successful requests is 6.6 seconds, which is acceptable. 17 Requests even returned a route in under one second, which is very fast. Failed calculations are to be expected, but without proper timeout control, the expected times for the whole benchmark become unreasonable large.

5.2 Performance of new Solution

First, consider the direct comparison to the old solution in Figure 5.3. Running the same test, it takes 539 seconds for 150 requests or 3.6 seconds per request on average. Out of the 150 requests, 80 were successful. The requests returning no route have been summarized in Table 5.1. In contrast to the previous solution, requests returning no route return almost immediately. Furthermore, there are requests that have run into the timeout, but obviously terminated quickly after encountering the time out.

On the first glance, the number of returned routes seems worse than in the previous case. 80 successes out of 150 requests is a success rate of 53 % for the new solution, while the old solution has 79 out of 110 or 71 % success rate.

Activity	Number of empty routes
Mountain Biking	12
Running	10
Cycling	15
Hiking	9
Inline Skating	24
Total	70

Table 5.1: Summary of requests returning empty routes

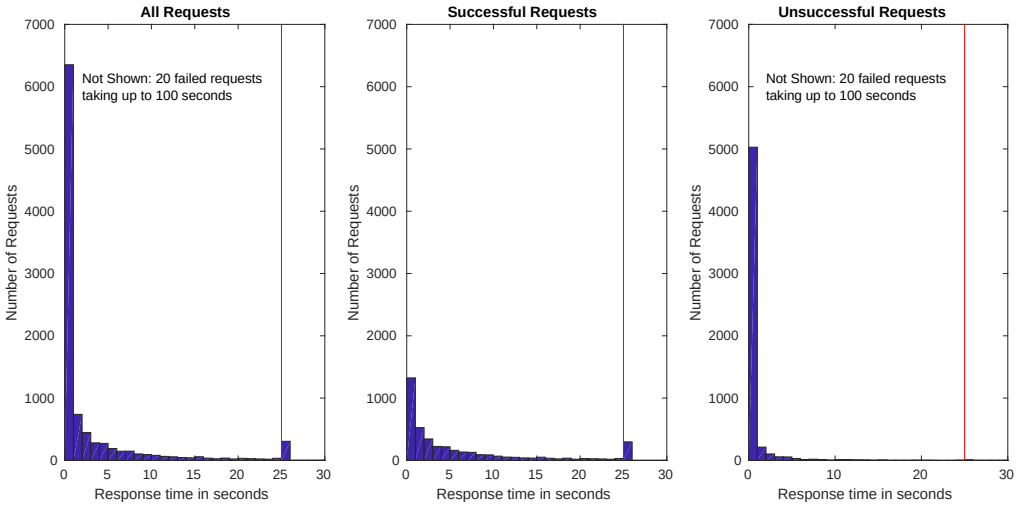


Figure 5.4: Response times of new version for routes with length of 5, 15, 30 and 50 km, acquired within 1000 km of Lucerne, 1000 km of Brasilia and 2500 km of Kansas City

However, the previous solution has not been tested for Inline skating at all and the weights for the different activities are different. Table 5.1 shows that the new solution is most successful for running and hiking and least successful for inline skating. This makes sense, since footpaths usable for hiking or running can be found almost anywhere. Inline skating requires a good road surface and ideally not much traffic, which is not as common.

Next, we will look at all routes with lengths 5, 15, 30 or 50 km as these work flawlessly with regards to memory consumption of the database. Even when randomly accessing locations over large areas of the world, the memory consumption does not exceed far beyond 7 GB and stays strictly below 8 GB. Figure 5.4 shows the same behaviour as Figure 5.3. Most of the 9720 requests stay well within the timeout. Only 20 take longer than 35 seconds, with the longest taking 100 seconds. Out of the 9720 requests, 4072 or 41.8 % were successful.

Location	Requests	Successful Requests	% Successful	Average Time [s]
Lucerne	4320	1786	41.3	5.13
Kansas	3600	1409	39.1	1.52
Brasilia	1800	877	48.7	0.96
Total	9720	4072	41.8	3.02

Table 5.2: Comparison of success rates across different areas of the world for short routes

Table 5.2 shows, how the new solution performs across different geographical areas. The success rate does not differ too much, it is however interesting to note, that within 1000 km of Brasilia the chance for success is highest. The region around Lucerne contains the highest number of timed out requests and 18 of the 20 requests taking more than 35 seconds. Multiple of these long requests originate from the same points for different activities and from routes that are either 30 or 50 km long. If only request are considered that have taken less than 25 seconds, the average reduces to 3.49 seconds, which is still the longest time of the considered areas. A possible explanation for this behaviour is that Europe is more densely populated than the United States or Brazil[3]. Accordingly, the road infrastructure can be expected to be more dense as well which would lead to more data being present per area.

The benchmark also has been run for routes of length 80 and 100 km, but only for part of the area around Lucerne. The reason is that even with the additional memory for the database, heavy swapping could occur. This did not result in a crash (which would easily detectable) but slow operation. This slow operation does not trigger a timeout in the external library querying the database, even though it is set to 30 seconds. Because the execution is stuck in the external library call, the own request timeout of the routing server is of no use either. The result is that requests running well over an hour have been found before restarting the database solved the problem temporarily.

Since it was not monitored when exactly the swapping started, the results are qualitative at best. Nevertheless, a short summary shall be given. In total, 2097 requests have been worked on. Out of that, 323 returned a route, 203 with 80 km and 120 with 100 km length. This success ratio is smaller than with the short routes, because request timeout is started before reading from the database starts. This means there is less time for calculation when the database is slow. The calculation of the successful 80 km long routes has taken 20.7 seconds on average, the calculation of 100 km long routes 21 seconds.

[3] https://en.wikipedia.org/wiki/List_of_countries_and_territories_by_population_density

5.3 Performance of parallel Execution

The performance of parallel execution of multiple requests has not been measured in the previous version because the start coordinate as well as the route length is stored in a static variable. Since Tomcat executes requests in parallel in multiple threads, this static variables are subject to a race condition and will cause erratic behaviour when overwritten.

This has been corrected in the new version. It is possible to send requests in parallel. Because of the unintended situation with splitting the routing server and the database to different machines, not much testing has gone into parallel execution. To test if it works, the benchmark has been run with a dataset of 150 points within 1000 km around Lucerne and two requests in parallel. The success rate in both cases is very similar, 30.0 % for sequential requests, 29.3 % for the parallel case. The average time for a request, calculated by dividing the total time by the number of requests, is 3.1 seconds for the sequential case and 2.3 seconds for the parallel case.

This improvement is not because unused processor cores are now used. The route generation uses all available cores even for a single request. Since the next request is sent as soon as one is finished, they two requests executed at a time are not at the same stage of calculations in general. This means one request can query the db, while the other is calculating a route for example. This leads to better usage of resources like network bandwidth or processing time instead of waiting for the benchmark to store the result and process the next input before sending the next request. In the parallel case, the database also uses more than a single core for processing multiple requests at the same time.

Because of the limited resources, no more testing has been undertaken on this matter. Of course, when calculating multiple long routes in parallel, the memory requirements are only going to grow.

5.4 Quality of Routes

In the last sections, the criteria for evaluation have been strictly confined to measurable data points. The quality of a route is much harder to quantify. A good route has of course the right length. This is ensured strictly by the code, even with the previous version. If a route is within the tolerance of the length, it is preferred, even if there are routes with much higher weights, but slightly out of tolerance. Since this functionality has not been changed, the new solution still exhibits the same adherence to the tolerance of 5 % of the total route length plus 50 m.

Changes have been made to the weight calculation and the frontend for selecting weight preferences. It cannot be automatically determined if a route is

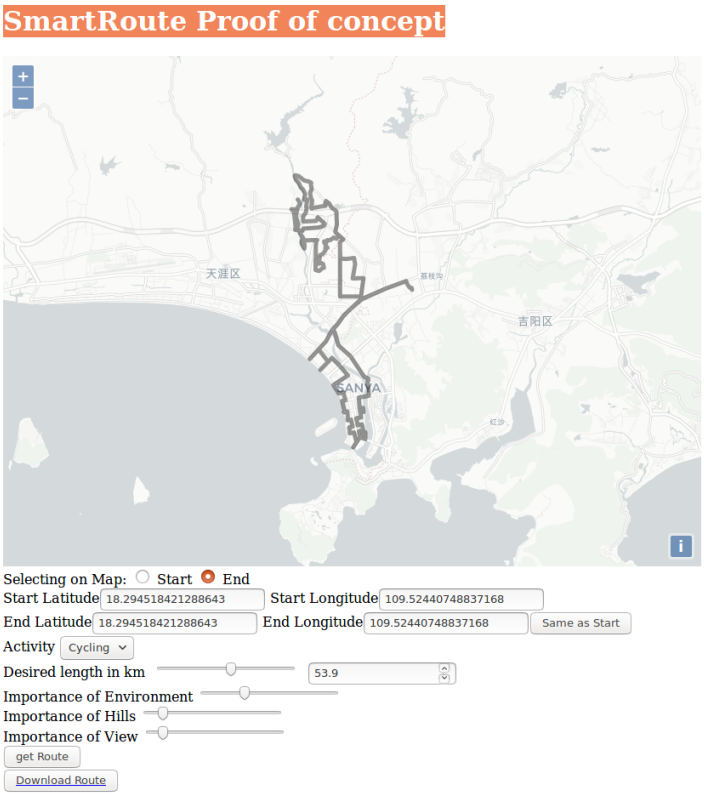


Figure 5.5: Cycling route in Sanya with focus on Environment

a "good" route. So we are left with manually examining generated routes. As an example, the Chinese coastal city Sanya has been chosen. Figure 5.5 shows a cycling route with heightened focus on the environment. Thus, the route follows roads with high values for the cycling weight which is achieved for example by cycle lanes.

Figure 5.6 shows a route of the same length starting at ending at the same point. Here, elevation changes have been prioritized. It can clearly be seen that one path of the initial extension has left towards the mountainous area to the north, but the randomPoints algorithm connecting the paths did not find a way to connect the paths without backtracking

Figure 5.7 finally shows the route generated at the same spot when the view is important. The furthest view can be expected along the coast. The selected route accordingly goes directly to the see from the starting point and follows the shoreline.

This example shows that routes with different qualities can be generated. Whether they are good or not can not be told beforehand. The final

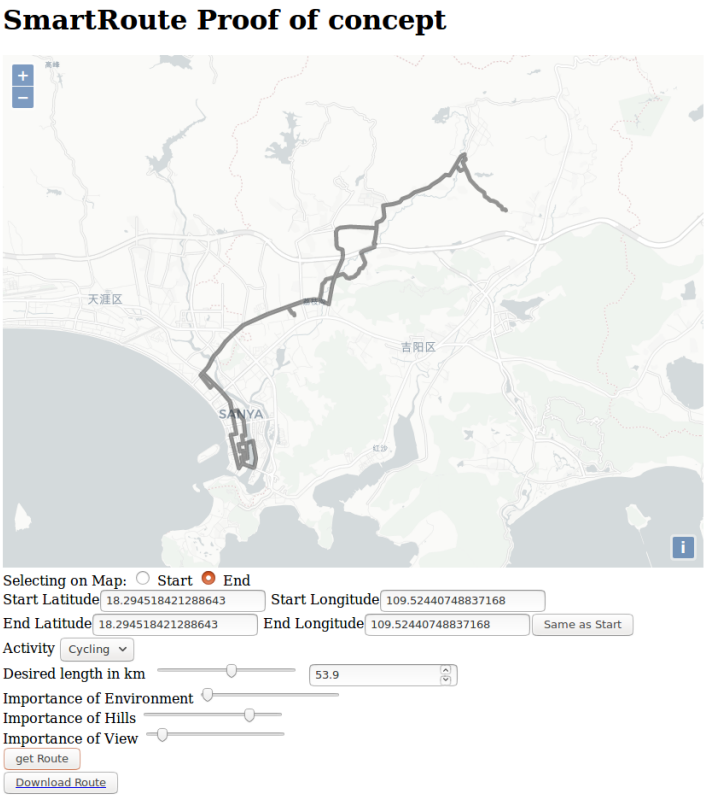


Figure 5.6: Cycling route in Sanya with focus on Elevation

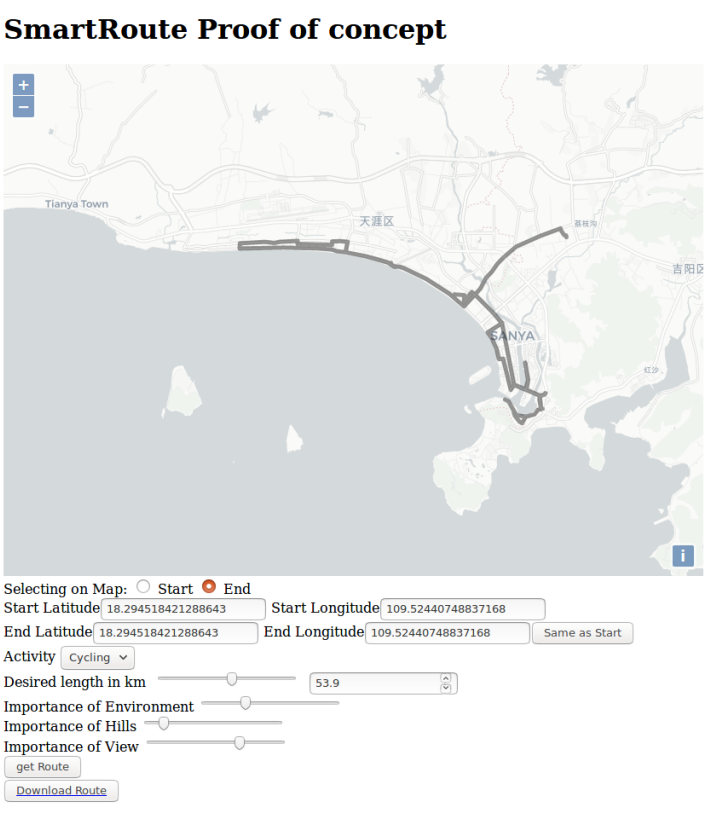


Figure 5.7: Cycling route in Sanya with focus on View

Chapter 6

Discussion and Future Work

The changes to the existing system in this work have enabled the routing server to generate routes on a worldwide scale. This goal has been reached. Due to limitations in hardware, the new solution even has been distributed across two machines. This has a disadvantage, because instead of a small request and its reply, the whole working set of the graph for a request is transmitted over the internet. Despite that, the old version still has been outperformed.

Distribution of storage and processing might be prove valuable in the future. This way, the machine used for data storage can be scaled independently from the machine used for data processing. MongoDB supports distribution across several machines and several Tomcat instances could connect to the same database. In principle, scaling this version up by using more hardware is possible.

However, there are other solutions that could be implemented before using more and better hardware. First, the nodes extracted from the database could be constrained more. Instead of extracting nodes within a circle around points, a rectangle or polygon could be constructed by taking more information into account. For example, when calculating a route from point A to B with a desired length that is only slightly longer than air distance, a small rectangle along the direct connection would return fewer nodes than a circle.

Perhaps memory can be saved, if instead of relying on a database for geospatial queries, file based storage is used and a custom way to index the files is found. Such an index could prepended to the files. Because of this possibility, switching from a database to file based storage has been implemented by a simple configuration switch.

The problems with memory usage occur only with long routes. There are two simple immediate solutions that do not require further development work to reduce it. The first solution is to restrict routes to shorter distances. The other possibility is to regularly restart the database as a maintenance operation.

Apart from performance, it has also been shown, that the new weights for activities can produce nice routes. But there are still ideas for improvement. A new weight based on user experiences could be introduced. Users having

completed a route could give feedback. The view calculation could be extended to take weather based visibility into account[1]. When it is foggy, the view weight has no relevance. The view calculation could also take the sun position into account to avoid looking into the sun.

The current proof of concept for the frontend does work, but it is not very pretty. To attract more users, responsive design that adapts to the display is required. Another part of the frontend not discussed so far is the map display. At the moment, the map graphics are loaded from servers of the company Carto[2]. This is free for up to 75000 map views per month. While this number is probably not reached in the near future, the need for map graphics could arise and alternatives should be reevaluated.

A final point that has not been addressed in this work is the question of updates. The OSM data are always changing. Instead of deleting all data of a specified area and reimporting the current data, a more efficient way would be to parse OSM changesets to keep the database up to date with the source data instead of having one big update. The question for this will be how the changeset can be applied to the selection of OSM data present within this application.

[1] For example by querying the weather API of https://openweathermap.org/.
[2] https://carto.com/

Bibliography

[1] Damman, S.: Outdoor Sports Route Generation (July 2017)

[2] OpenStreetMap contributors: Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org (2017)

[3] OpenStreetMap Wiki Contributors: Planet.osm - OpenStreetMap Wiki. http://wiki.openstreetmap.org/wiki/Planet.osm accessed 2017-12-09T15:35.

[4] OpenStreetMap Wiki Contributors: PBF Format - OpenStreetMap Wiki. http://wiki.openstreetmap.org/wiki/PBF_Format accessed 2017-12-18T14:27.

[5] Geofabrik GmbH: OpenStreetMap Data Extracts. https://download.geofabrik.de/ accessed 2017-12-09T15:45.

[6] Henderson, B., Wolschon, M., Newman, K., Baebler, S., Sommer, C., Topf, J., Jacobs, A., Klement, J., et al.: Osmosis. https://github.com/openstreetmap/osmosis accessed 2017-12-09T16:09.

[7] Graf, F.: SRTM Plugin for OpenStreetMap's Osmosis. https://github.com/locked-fg/osmosis-srtm-plugin accessed 2017-12-09T16:23.

[8] Apache Software Foundation: Apache Tomcat. http://tomcat.apache.org/ accessed 2017-12-09T17:48.

[9] Schulze, J.: Smart Running Route Generation (November 2016)

[10] Sedlacek, J., Hurka, T.: VisualVM All-in-One Java Troubleshooting Tool. https://visualvm.github.io/index.html accessed 2017-12-09T20:09.

[11] Eclipse Foundation: Memory Analyzer (MAT). http://www.eclipse.org/mat/ accessed 2017-12-17T17:04.

[12] Esoteric Software: Kryo - Java binary serialization and cloning: fast, efficient, automatic . https://github.com/EsotericSoftware/kryo accessed 2017-12-12T14:49.

[13] MongoDB, Inc.: MongoDB. https://www.mongodb.com/ accessed 2017-12-12T15:53.

[14] MongoDB, Inc.: $nearSphere - MongoDB Manual 3.6. https://docs.mongodb.com/manual/reference/operator/query/nearSphere/ accessed 2017-12-12T16:57.

[15] Movable Type Ltd: Calculate distance, bearing and more between Latitude/Longitude points. https://www.movable-type.co.uk/scripts/latlong.html accessed 2017-12-13T23:03.

[16] Salonen, J.: Geographic distance can be simple and fast. http://jonisalonen.com/2014/computing-distance-between-coordinates-can-be-simple-and-fast/ accessed 2017-12-13T23:03.

[17] : SRTM Topography. https://dds.cr.usgs.gov/srtm/version2_1/Documentation/SRTM_Topo.pdf accessed 2017-12-14T17:37.

[18] OpenStreetMap contributors: taginfo. https://taginfo.openstreetmap.org accessed 2017-12-20T18:20.

[19] Oracle: Streaming API for XML. https://docs.oracle.com/javase/tutorial/jaxp/stax/index.html accessed 2017-12-17T17:26.

[20] OsmAnd Contributors: OsmAnd - Offline Maps and Navigation. http://osmand.net/ accessed 2017-12-09T14:34.

[21] Stallman, R.: GNU General Public License. http://www.gnu.org/licenses/gpl.html

[22] Shcerb, V., Pelykh, A., Mueller, H., Zibrita, P., et al.: OsmAnd License

[23] OpenLayers Contributors: OpenLayers. https://openlayers.org/ accessed 2017-12-22T18:19.

List of Acronyms

DISCO . . . . Distributed Computing Group

GPX . . . . . GPS Exchange Format

JSON . . . . . JavaScript Object Notation

JVM . . . . . Java Virtual Machine

MAT . . . . . Memory Analyzer

OSM . . . . . OpenStreetMap

RDBMS . . . . Relational Database Management System

SRTM . . . . . Shuttle Radar Topography Mission

UI . . . . . User Interface

XML . . . . . Extensible Markup Language

Appendix A

Use of GPX files in this work

The structure of the GPX (or GPS exchange format) file generated by the new frontend is simple. Listing A.1 shows an example. It is an XML file with a structure as defined by the schema at the location mentioned in the gpx tag. Most of the possible features of a GPX file are unused. In the gpx tag, only a creator attribute and information for validating the XML are supplied. A file name based on the activity and the time of the request is added as metadata. The rest of the file is taken up by a track consisting of a single segment with the trackpoints containing the route. For more information, refer to the standard at http://www.topografix.com/gpx.asp.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpx xmlns="http://www.topografix.com/GPX/1/1" version="1.1"
     creator="SmartRoute Route Generation Tool"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.topografix.com/GPX/1/1
     http://www.topografix.com/GPX/1/1/gpx.xsd">
<metadata><name>Route_cycling_2017-12-22T22:15:28.gpx</name></metadata>
<trk><trkseg>
  <trkpt lat="18.294619" lon="109.524411"></trkpt>
  <trkpt lat="18.295332" lon="109.524261"></trkpt>
  <trkpt lat="18.295408" lon="109.523493"></trkpt>

  <!-- Points in between omitted -->

  <trkpt lat="18.295408" lon="109.523493"></trkpt>
  <trkpt lat="18.295332" lon="109.524261"></trkpt>
  <trkpt lat="18.294619" lon="109.524411"></trkpt>
</trkseg></trk></gpx>
```

Listing A.1: Example of a GPX file generated by the new frontend

Appendix B

Task Description

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Prof. R. Wattenhofer

SA:

Worldwide Sports Route Generation

A large number of outdoor sports apps exists on the market. Many of them allow you to track your routes and evaluate your workouts. Some apps enable sharing routes between users and few tools exist for automatic route generation. However, the latter are mostly quite simple and yield unsatisfying results.[1] The typical exercise procedure therefore starts with manual planning or selection of a route. As this is an undesired overhead, people tend to use the same routes over and over again, although this is rather boring.

We think that route generation for outdoor sports like workouts, hiking or even just sightseeing can be done in much better ways using a smartphone. In past theses, we created an Android app[2] which generates routes for running, biking, hiking and other outdoor sports in Switzerland.

The goal of this thesis is to extend the app's functionality. Most importantly, it should be possible to generate routes anywhere in the world. Further ideas include adapting routes on the fly, when a user deviates from the foreseen path, and improving the pathfinding for different activities.

Requirements

- Creative thinking and good programming skills are advantageous to successfully work on this topic.
- The student(s) should be able to work independently!

Interested? Please contact me for more details!

Contact

- Manuel Eichelberger: manuelei@ethz.ch, ETZ G97

[1] http://www.plotaroute.com/
http://www.routeloops.com/

[2] *Smart Route* App in the Google Play Store: https://play.google.com/store/apps/details?id=sd.smartroute

Detailed Project Outline

In the following, we outline the tasks of the student (on the right side you find a rough estimate for the time that we allocate to the respective task):

- Comparison of existing tools and literature research. ($\star$)
- Optimize existing algorithm and data structures to extend the geographical coverage up to worldwide route generation. ($\star\star\star$)
- Extend edge attractiveness metric. For instance allow specific height profile settings and consider meteorological data. ($\star\star\star$)
- Balance routes for different activities and user settings. This may include improving existing metrics such as the view attractiveness. ($\star\star$)
- Adapt UI of existing app to new features and clean up code base ($\star$)
- Write a report ($\star\star$)
- Present your findings ($\star$)

Extensions

If the student makes good progress, there are several ideas for extending the project:

- Adaptation of routes on the fly, when a user deviates from the proposed route.
- Determine an edge's attractiveness based on user data from the Internet or paths users actually choose.

Student's Duties

- One meeting per week with the advisor.
- A final presentation (15 min) of the work and results obtained.
- A final report, presenting work and results.

Appendix C

Declaration of originality