Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

On the Fly!
Automatic Running Route Generation

Bachelor's thesis

Tim Linggi

linggit@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Supervisor 1
David Stolz

Supervisor 2
Prof. Dr. Roger Wattenhofer

March 9, 2016

Abstract

Many people like to go jogging and use a variety of gadgets for that purpose, especially cell phones. However, most of the mobile applications that assist you which are currently on the market, usually only track you while running or give you predefined routes to run along. Therefore, you typically start by planning a route or selecting one created by someone else. We introduce a new mobile application to replace this – rather boring – way of planning your exercise. Instead of manual planning, our application dynamically generates a route based on user parameters, such as desired length and current coordinates of the user. To further improve user experience, the application responds to changes during the run by dynamically generating a new route that satisfies the original parameters.

Contents

Chapter 1

Introduction

The main motivation of this thesis is to replace the either cumbersome way of pre-planning your jogging exercise or using pre-defined routes with a new method: generating routes "on the fly". The user only needs to enter certain parameters (e.g. length), and then, the application generates a suitable route. Our application works in such a way that the user can start directly without waiting for the completion of the route generation, which enables the user to start running immediately while the calculation of the route is still in progress. Additionally, one will not run the same route over and over again. Some applications which try to apply this concept exist[1], but they yield rather unsatisfying results.

Our work processes the huge amount of raw geodata available to small parts which are more suited to work with. We use these small parts to generate routes and display them on the user's device. The computation-heavy work of extracting data and generating routes is done on a server, whereas the relatively light tasks of displaying routes and tracking are done by the client on a mobile device.

[1]For example plotaroute.com or routeloops.com.

Chapter 2

Overview

2.1 Open Street Map

We used the vast database of Open Street Map (OSM) [1] to populate our custom geo-database. In contrast to other providers of geodata, OSM is free to use and open source [2]. This has the big advantage that many small – maybe only locally known – paths are potentially only in the OSM database, since commercial projects usually care more about highways and generally larger streets. It is obvious that accurate highway information is not really important to generate good jogging routes. However, small paths are exactly what makes jogging routes interesting. Therefore, the OSM database is ideally suited for our goal.

However, also OSM has its drawbacks: As it is created by a very large community, there are quite a few wrong entries, and also the naming scheme is not used consistently (more in Section 2.5). Another issue of OSM is that the data is saved in a raw binary format (i.e. not human-readable). In order to work with this data, we first needed to condition and filter it (see Section 3.1).

2.2 Server

We implemented the server in Java, using the Tomcat framework [3]. The server takes the raw data provided by Open Street Map, generates routes, and sends them back to the client, as shown in Figure 2.1. The following paragraphs are

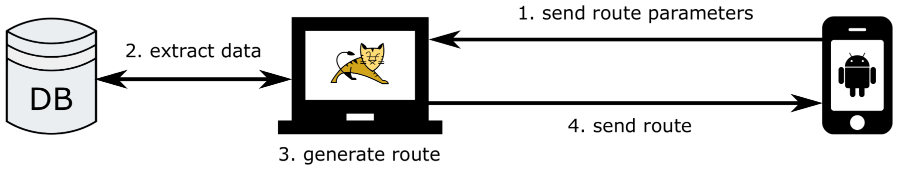

Figure 2.1: The structure of the application with the database, the Tomcat server and the Android client.

only a conceptual overview, for more details, refer to Chapter 3.

2.2.1 Data

The raw data can easily be downloaded from one of the Open Street Map mirrors [4]. The data is received in a binary file format which we preprocess for our purposes: we only keep the data relevant to us. This is done with the help of a tool called Osmconvert [5], which can be used to efficiently extract data of a PBF file [6] for a specified bounding box as either XML or CSV.

2.2.2 How to generate good routes?

With the extracted and preprocessed data, we generate the route(s). Of course, we could just generate random routes, but it is very unlikely that any user would be interested in such routes. Therefore, we asked ourselves the question of what makes a route *good*. We think that the answer to that question is to a certain degree subjective. However, there are some criteria which define a good route that most likely everybody could agree on. Given a start and an end point, as well as a length, we determined the following:

1. Actual start and end point as close as possible to the provided ones
2. Actual length close to the provided one
3. Minimize the number of small loops (as illustrated in Figure 2.2) in the generated route
4. Maximize the length of pedestrian friendly paths in the route

The last requirement turned out to be rather difficult to achieve. We therefore decided to focus on the first three goals and deferred the last one as a future improvement (see Section 6.1).

In order to achieve these goals, our algorithm first builds a (weighted, undirected) graph from the preprocessed data. Then it tries to find a path, which approximates a circle with correct start and end point. We chose this approach, because a circular route prevents (at least to a certain degree) the forming of loops (as mentioned in Criteria 3) and the length is easy to control.

It is important to note that this algorithm is only a heuristic and is not guaranteed to yield the optimal result.

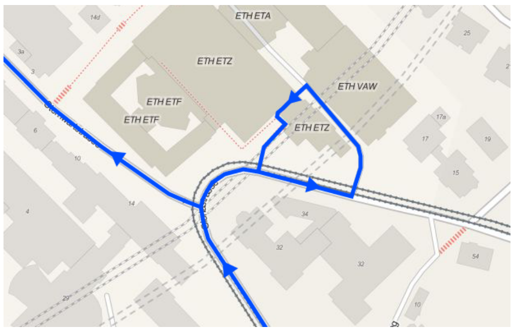

Figure 2.2: A small loop in a route.

2.3 Client

The mobile client is implemented for the Android platform [7]. We used Android, because it is very openly designed, free and programs can be written in Java. The client has two main activities, one with a user interface for entering the data about the route (such as start/end point and length) and one for displaying the route and tracking the user. For more details, refer to Chapter 4.

2.4 Communication

The communication between the server and the client is implemented with a REST API, using the data format JSON [8]. We chose JSON, because it is a very simple and compact data format, which is openly available. Also, it is widely used, so no compatibility issues should arise with possible future work.

2.5 Limitations

The data of Open Street Map is structured using tags [9] (see Section 3.1.1). Even though the guidelines for tagging are very detailed and cover most use cases, there is no guarantee that every element is tagged correctly or even tagged at all. Because we only use tags for filtering out non-pedestrian streets like highways, this was not too much of a problem, but if one wants to improve the algorithm (as outlined in Section 6.1), one would have to rely more on the information saved in tags.

Chapter 3

Server

3.1 Data

3.1.1 Structure of raw data

The raw data of Open Street Map is structured according to the rules defined in [9]. Its basic components are called elements, which in turn can be either nodes, ways or relations [10]. Nodes define points, ways define connections of nodes and boundaries and relations explain how elements work together. For our purposes, we only need nodes and ways between nodes, i.e., no ways which describe boundaries and no relations.

A node element contains its coordinates (longitude and latitude), as well as its id. All the other information, like the uploader's user name or version code, are not important for this work. A way element consists of at least two nodes (more precisely their ids) and a collection of tags. Tags can contain a variety of information about the way, but here, we only need to know, if a particular way is accessible for pedestrians (i.e. that it is not a highway or the like). Relations are completely discarded, since all the information we need is already contained in nodes and ways.

3.1.2 Conditioning of raw data

The raw data provided by the various mirrors of Open Street Map (we used the Swiss mirror [11]) is usually saved in the PBF format [6]. Since this is not a common file format, standard editors cannot read PBF files. However, there are tools available which extract XML and CSV files from a PBF file. In our work, we used Osmconvert [5]. Given two corners of the bounding box (the area from which we want to extract information), we can use Osmconvert to create a CSV file with all the ids and coordinates of the nodes, as well as an XML file (which has the ending .osm but still is an XML file), which contains all the ways with the corresponding ids of the nodes in the bounding box.

Given these two files, we wrote a small program that stores this information

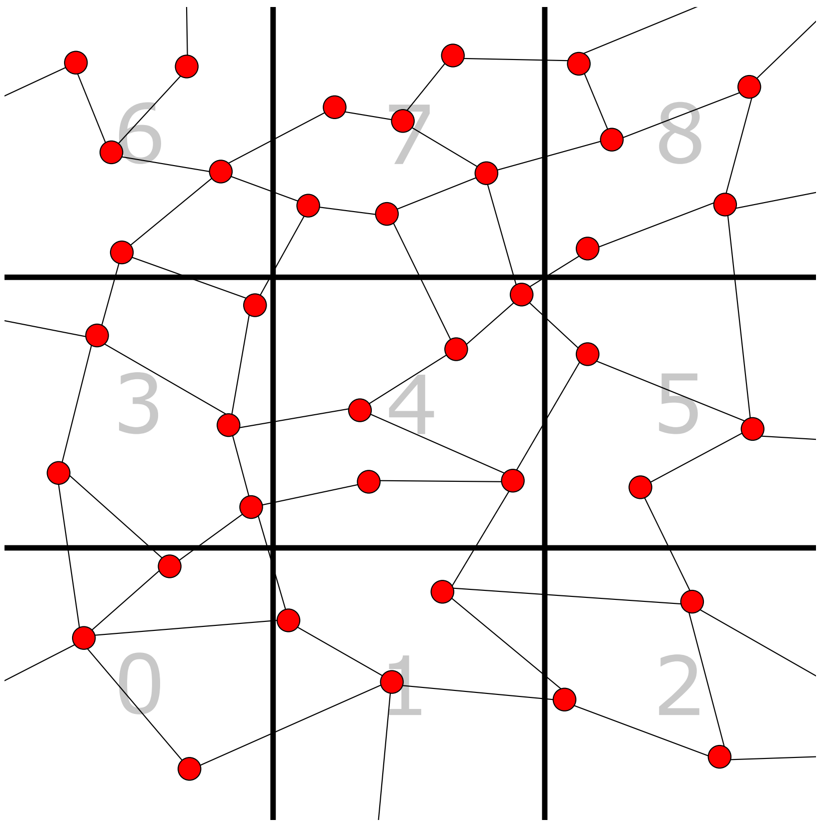

Figure 3.1: A graph grouped into numbered tiles.

(as nodes and edges) in a simple SQL database. Of course, an SQL database is not the optimal means to store geographical information, but for the sake of simplicity and because it performed well enough, we used it nevertheless.

Additionally, we grouped the entire area into square tiles with side length 500 meters and numbered them starting from the bottom left corner (as shown in Figure 3.1). We allocated the corresponding tile id to each node and stored that in the database as well.

3.2 Routing

The routing algorithm was the most difficult part of the thesis. To talk about its implementation and some choices we made, we use the requirements stated in Section 2.2.2.

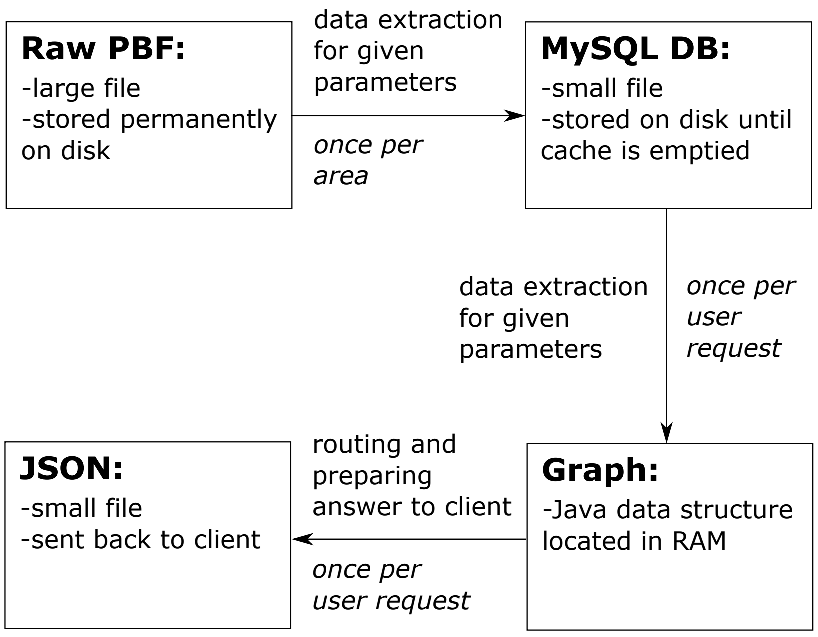
Raw PBF:
-large file
-stored permanently on disk
data extraction for given parameters
once per area
MySQL DB:
-small file
-stored on disk until cache is emptied
data extraction for given parameters
once per user request
Graph:
-Java data structure located in RAM
routing and preparing answer to client
once per user request
JSON:
-small file
-sent back to client

Figure 3.2: The dataflow on the server.

3.2.1 Concept

As mentioned in the requirements, generating a route requires a start and an end point (which can be the same) and a length. With this information, we extract a small database from the raw data (as described in Section 3.1), which includes all the nodes and edges around the required point(s). Since the database extract might be larger than actually needed, we first determine which tiles are required to avoid loading the entire database into memory. We then load the nodes in these tiles, as well as the edges between them, in a graph data structure. We use JGraphT [12] as a library for the graphs. Afterwards, the algorithm can easily work on this graph without accessing the database again. The entire dataflow is illustrated in Figure 3.2.

On an abstract level, the algorithm tries to approximate a circle around the target (end point). It does that, by penalizing edges which go further away from this hypothetical circle. Additionally, it tries to prevent loops, as this is a requirement, mentioned in Section 2.2.2, also by penalizing edges, which have been visited before.

3.2.2 Algorithm

Given the graph described in the last section, we can start the actual routing algorithm (Algorithm 1). We decided to use a depth-first approach. First, we need to determine the closest nodes in the graph to the provided start and end point, which is simply done by iterating over all nodes of the graph. We then need to define the penalty function. As described earlier, we try to approximate a circle around the target. For this purpose, we split the routing into two parts, which we call circle and end phase. In the circle phase, the algorithm tries to approximate a circle around the target by assigning larger penalties to edges, which point away from the circle. In the end phase, the penalties are assigned proportionally to the distance to the target. The end phase is needed because otherwise the algorithm would most likely never actually reach the target and keep circling around it.

Algorithm 1 Routing algorithm

```
function MAKEROUTE($graph, source, target, length, penalty\ function$)
    $currentVertex \leftarrow source$
    initialize $currentPath$
    zero penalties of all edges;
    while $true$ do
        $nextEdges \leftarrow$ edge set of $currentVertex$
        $nextEdge \leftarrow$ edge with smallest penalty in $nextEdges$
        update $currentPath$ with $nextEdge$
        $nextVertex \leftarrow$ other Vertex of $nextEdge$
        if $currentPath$ too long then
            reset $currentPath$
            continue
        end if
        if $currentPath$ in range && $target$ reached then
            return $currentPath$
        end if
        set penalties of edge set of $nextVertex$
    end while
end function
```

The penalty function consists of three parts. On one hand, it is dependent on the distance to the circle (or target respectively), as mentioned in the last paragraph. On the other hand, it factors in the number of times a node and an edge have already been visited and penalizes larger numbers. This prevents loops (as mentioned in Section 2.2.2, Requirement 3) and encourages the algorithm to take edges, which have not been visited before. Additionally, we added some noise, so that the algorithm does not return the same route for the same parameters every time it is called.

When a route has been found (with Algorithm 1), we remove any small loops that made it into the solution. However, this introduces the problem that routes which contain many loops are significantly shorter than required. For this reason, we initially multiply the length by a factor of 1.5 to request a longer route, which addresses this issue.

After testing the algorithm, we discovered that generating a single route was usually not enough, because many routes had either still some (larger) loops, or were too short or too long. For this reason, and because the routing algorithm (Algorithm 1) works rather fast (usually between 0.5 and 2 seconds), we decided to calculate many routes at once (12 to be exact) and choose the best to return. This approach worked well and, if more than one route, for example k routes, were required, one can easily choose the best k.

We further improved the calculated routes, by calling the entire algorithm multiple times (four at most) with slightly adjusted parameters, if no routes were found. For example, if all calculated routes were too short, we increased the requested length and vice versa. At this point, some sophisticated machine learning algorithms would be much more efficient and yield better results, however, we decided that this would go beyond the scope of this work.

Of course, looking at Algorithm 1, one can easily see that it might not terminate. To address this, we introduced a timeout, after which the process will be terminated and a new one started. The new one will most likely not be stuck with the same problem as the previous one, due to the added (random) noise in the penalty function.

3.2.3 Caching

Benchmarking showed that most of the time on the server side is spent on retrieving the database extract[1], and thus we decided to cache the extracts. For this purpose, we assigned a unique id to each of them and put the id in the response to the client. The client saves the id and the corresponding center and diameter of the bounding box. Therefore, the client can include the id in later requests, which are in the same area and not much larger[2]. The server then checks, if the id corresponds to a stored database and fits the criteria and, if so, uses this database for the routing algorithm.

This solution is a compromise between no caching at all – which of course would not need as much space – and pre-calculating database extracts for the entire raw data – which in turn would need an enormous amount of space. With our solution, a user who approximately starts their run in the same location with the same length multiple times, will only have to wait once for the database extraction and afterwards, the server can use the cached extract. On the other hand, users who start their routes at a new location every time they use the app,

[1]The conditioned geodata within the bounding box around the start and end point of the route to be used in the routing algorithm as described earlier.

[2]If the user requests a significantly larger route, the database extract will not contain the required geodata.

have to wait until the data is processed each time (which approximately takes additional 10 - 15 seconds).

While the route is being generated on the server, the user can already start running. After some time, the server will send the route back to the user's device. If the user is currently not following this route, a new route request would automatically be sent and, because of the caching, the request will only take a few seconds to complete.

Chapter 4

Client

4.1 User interface

4.1.1 Entering the route data

The interface which allows the user to input data about the route provides a map view (see Section 4.2), as well as text views for entering the coordinates for the start and, optionally, the end point, as shown in Figure 4.1. Since typing in coordinates is rather annoying, the user can use their current position (by pressing the corresponding button) or any other location (by clicking on the map) as well. Once all the parameters are set, the user can start the route generation, which leads to the next activity.

4.1.2 Displaying the route and tracking

Once the route generation is started, i.e., the client has sent the parameters of the required route to the server, the user can either start running or wait until the server sends the generated route back to the client. This may take some time initially (up to a minute, especially, if the database is not cached (see Section 3.2.3) and the requested route is long) but goes rather quickly (a few seconds) for subsequent queries.

After the client received a route, it displays it using Osmdroid's interface (see Section 4.2, [13]) as shown in Figure 4.2. Besides displaying the route on the map, the client also displays the total length, the remaining length, the total altitude (sum of absolutes of incline and decline) and altitude difference (difference of altitude between start and end point). The next segment is always indicated with a special color. While running, the client tracks the progress, i.e. which nodes the user has already passed, and updates the route accordingly. Of course, tracking your route visually while jogging is rather cumbersome. Therefore, we implemented audio navigation information: The device outputs basic audio signals: “Turn left” and “Turn right”. This is implemented using Android's TextToSpeech [14]. Additionally, the device outputs a short, acknowledging

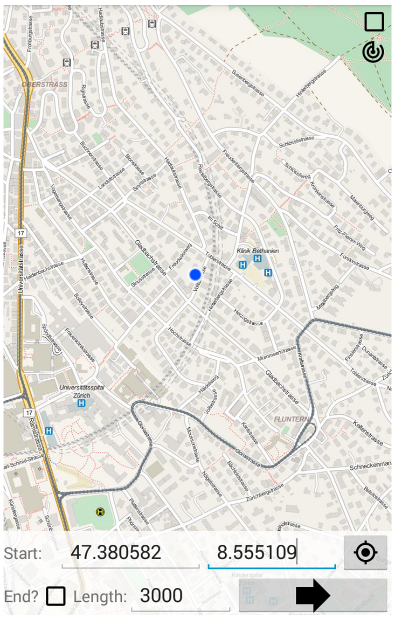

Figure 4.1: UI for entering route parameters.

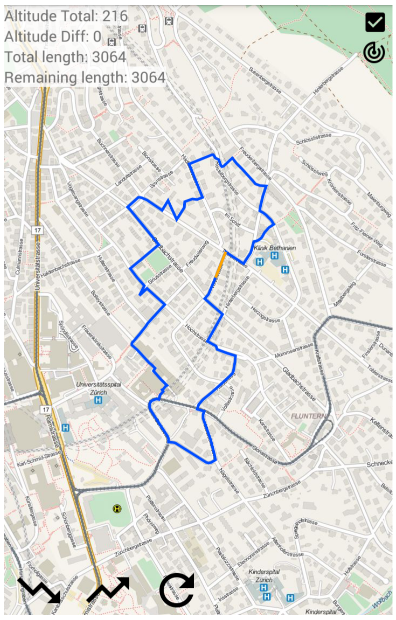

Figure 4.2: Displaying route and metadata.

sound if there is no turn to take to make sure, the user knows that the app is still running and tracking them.

When the user completed a route, i.e., when they reached the last node in the route, the complete run is saved into a database on the mobile device along with some metadata (like time and length) to be accessible later (see Section 4.1.4).

4.1.3 Requesting new routes

While a map is displayed, the user has four options to change the route:

1. Requesting a route with more total altitude
2. Requesting a route with less total altitude
3. Requesting any new route
4. Getting too far away from the current route (triggered automatically)

The first three options are triggered by the user pressing the corresponding button. For Options 1 and 2, the client requests a route with more (or less) total altitude. Of course, such a route cannot always be found, for example, if the route is already very steep and the user requests a steeper route. In this case, the user is notified that no better route was found and can continue running the current route.

If the user requests Option 3, a new route without any constraints for the altitude is requested. This corresponds to an initial route request with the current position as start point and the remaining length as length.

Option 4 is triggered “on the fly”, if the user stops following the current route, i.e., gets too far away from the next node. This can for example happen, if the user wants to follow a nice track, which is different to the current route. In this case, the new route has the same constraints as Option 3. Option 4 guarantees that there is always a route available and that the user does not have to make U-turns, i.e., the user cannot run the wrong way (we do not want to give the user negative feedback). If they do not follow the route, a new one gets created automatically.

All the options mentioned above take the current position, the end point earlier defined and the remaining length as parameter for the route request.

4.1.4 Statistics

If a route is completed, information about it is saved in a simple SQL database. If the user wants to check on their completed routes, the user can click on the corresponding entry in a list view, which opens a map view displaying the route.

4.2 Osmdroid

For displaying maps, and with it, routes and points, we used the open source library Osmdroid [13]. It provides an easy interface for retrieving map tiles and centering at the current position. The provided map view made it very easy to include the map in the Android layout XML files. Additionally, we used the Osmbonuspack [15], a third-party library, which provides additional functionality, for example, an interface for a route and the possibility to display it.

Chapter 5

Testing

5.1 Beta

After testing the app ourselves, we decided to use the beta testing functionality of the Google Play Store [16]. This made it very simple to distribute the app to many people, since we could just send out a link, with which the testers could download the app without the need to set up any developer environments. Feedback can be sent per email or, if the app crashes, a stack trace is sent to Google, which we can evaluate.

Because this only provided a feedback about the app's usability and correctness but not about how a user interacts with the app itself and what routes they request and run, we decided to log their behaviour, namely:

- location updates every 7.5 seconds.
- route requests (which type and if one was successfully found).
- when a route is completed.

This yielded a lot of information, which can be used to improve the client and the server. Also, it gave us a feedback of how well our routing algorithm worked under different circumstances.

5.2 Evaluation

For evaluating our work, we focused on the quality, the latency and the success rate of the route generation algorithm. Since there were not enough testers, we locally tested the algorithm by querying routes with random parameters. As test area for the queries, we chose the canton of Zurich and its surroundings to cover rural and urban areas likewise. This was done on a virtual machine running Ubuntu with 8 GB of DDR4 RAM at 2400 MHz and two physical (resulting in

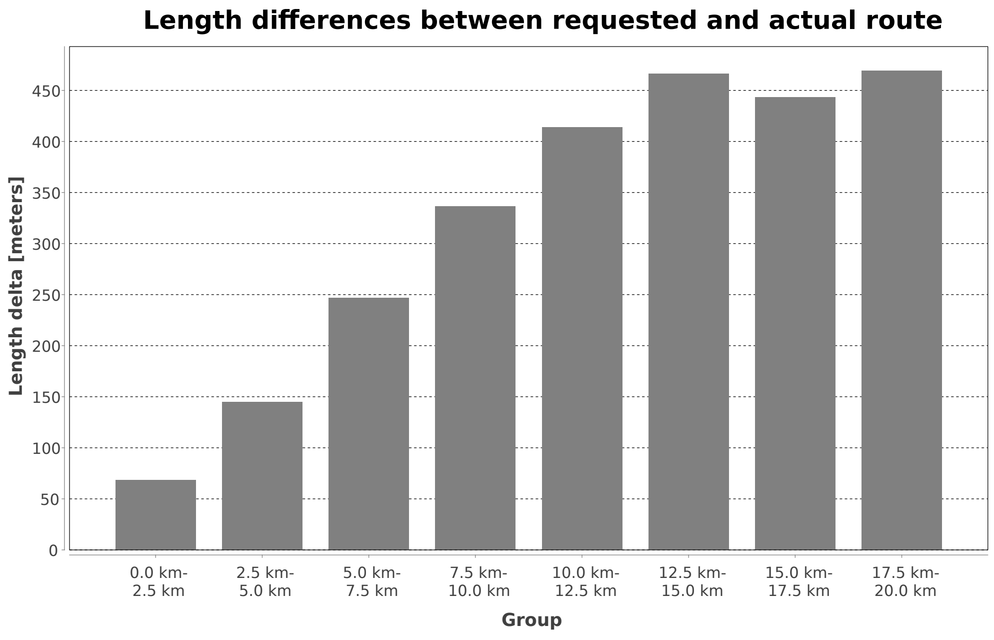

Figure 5.1: The differences between the requested length and the length of the generated routes.

four logical) cores of an Intel 5820K CPU at 4.0 GHz. The number of requested routes is around 1000.

We used the Criteria from Section 2.2.2 for measuring the quality of the generated routes, in particular Criteria 2 and 3, since Criteria 1 was not an issue to achieve and Criteria 4 was excluded early on. As illustrated in Figures 5.1 and 5.2, the difference between requested and actual length and the number of nodes visited multiple times increases, as the length of the route increases. The growth of the length delta was expected, since the routing algorithm accepts route lengths relative to the requested length (i.e., the longer the requested route, the larger the range of acceptable route lengths). At first, the number of revisited nodes seemed quite high, but one has to consider that a multiply visited street for example may contain many vertices. Nevertheless, these numbers are still quite high and should be reduced for better quality of the routes.

The success rate of the routing algorithm, as illustrated in Figure 5.3, was overall quite good. With the exception of the first group (0 - 2.5 km), more than 90% of the requests succeeded. However, in the first group, the success rate was only around 83%. We suspect the issue to be that the range within which routes of different lengths are accepted (plus and minus 7.5 %) is absolutely much smaller. For example, for a request of length 12 km, routes from 11.1 km to 12.9 km are accepted, whereas the range for a route of length 1.5 km is eight

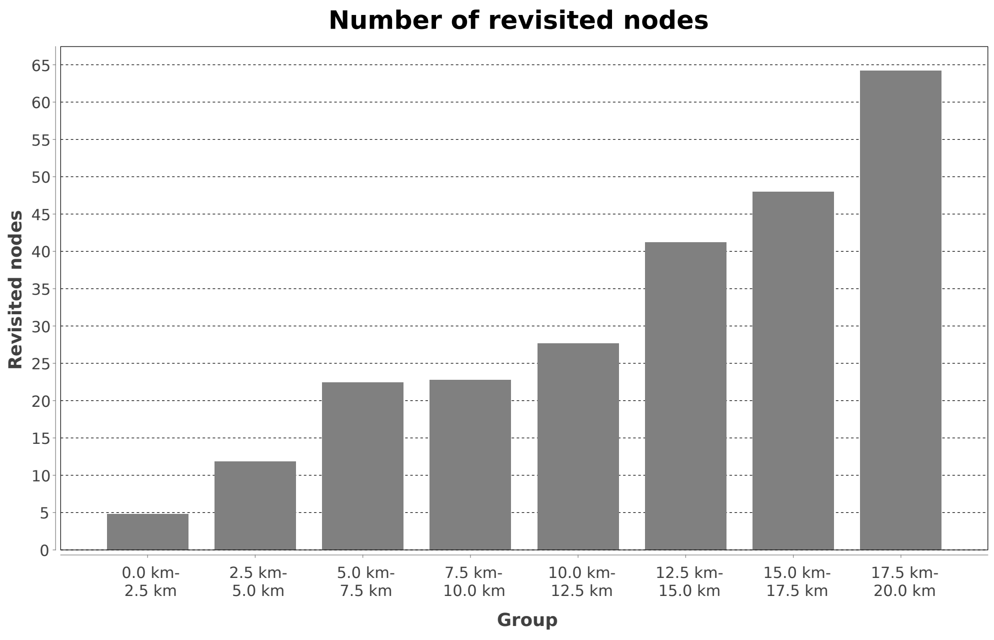

Figure 5.2: The number of revisited nodes in generated routes.

times smaller. For this reason, it is possible that disproportionally many short routes are discarded.

The time measurements and the average number of timeouts, as shown in Figures 5.4 and 5.5, were quite unexpected. Our assumption was that the longer the requested route, the longer the algorithm takes to complete and the more timeouts occur. However, this turned out to be only partly true. The number of timeouts increases, if the requested route is longer, but also if it is shorter. This can be explained with the same reasoning as in the last paragraph. It should be noted that a timeout does not mean that the entire query timeouted, but rather that one of the many invocations (as described in section 3.2.2) of Algorithm 1 did. However, this of course increases the overall latency significantly.

Overall, the routing algorithm performed quite well. Especially in the range, within which most users will query routes (5 - 12.5 km), the calculation time was low and the success rate high. Of course, there is still room for improvement, for example, in the number of revisited nodes and the success rate for small routes.

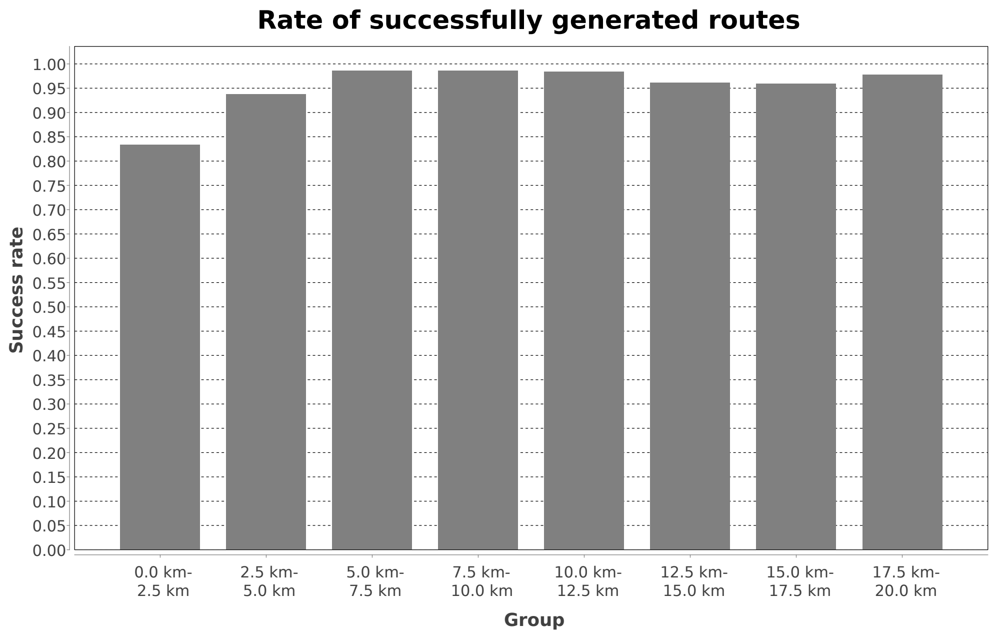

Figure 5.3: The rate of successful route generations.

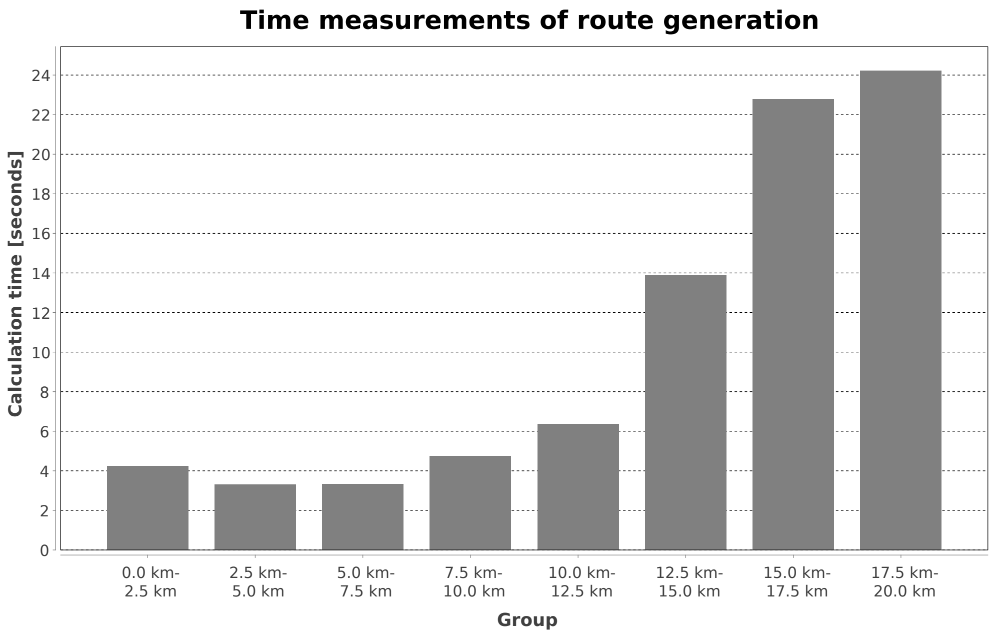

Figure 5.4: The average runtime of the routing algorithm.

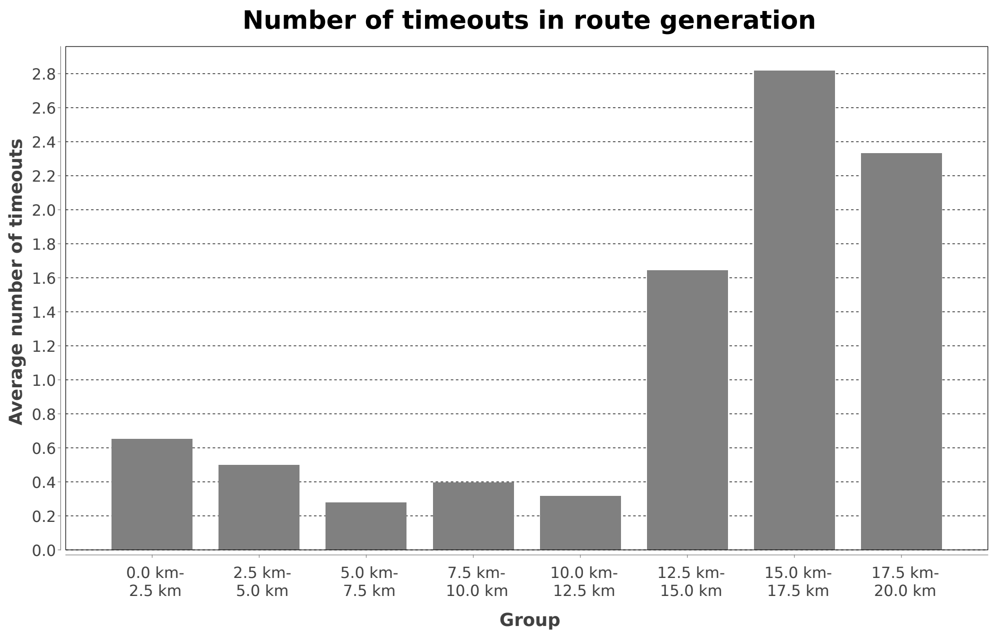

Figure 5.5: The average number of timeouts of Algorithm 1.

5.3 Example routes

This section contains some examples of generated routes in the surroundings of Zurich.

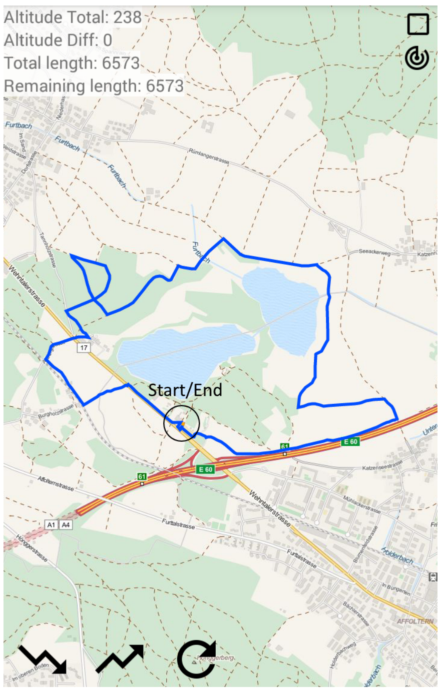

Figure 5.6: 6.5 km route around a lake.

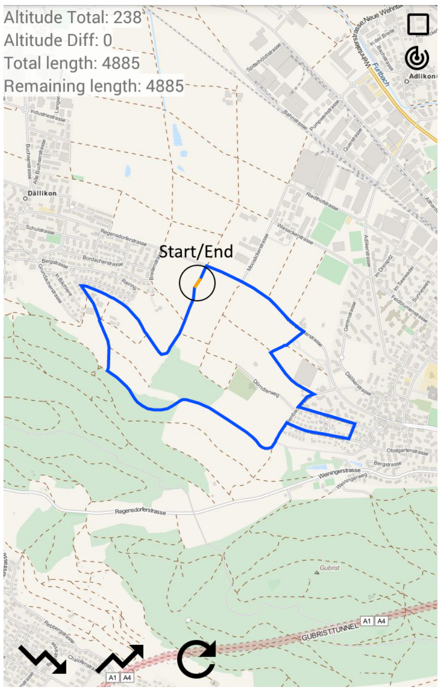

Figure 5.7: 4.9 km route near the city limits.

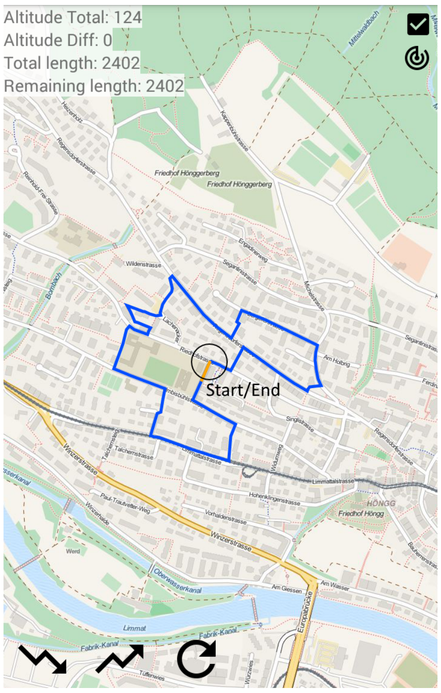
Altitude Total: 124
Altitude Diff: 0
Total length: 2402
Remaining length: 2402

Figure 5.8: 2.4 km route in an urban area.

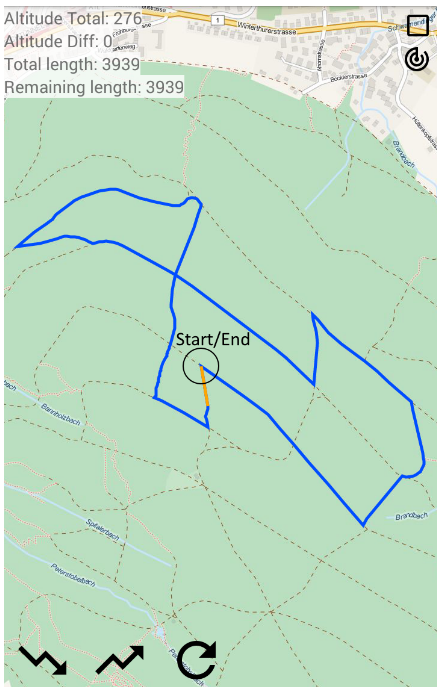
Altitude Total: 276
Altitude Diff: 0
Total length: 3939
Remaining length: 3939

Figure 5.9: 3.9 km route in the forest.

CHAPTER 6

Future work

6.1 Data

In this work, we made very few restrictions on what data we used for routing. In particular, we only filtered highways and used anything else as equally important. Of course, this is suboptimal, because jogging on a remote track is much nicer than on a large street.

In order to pick more such tracks, one would need to use the tags provided in the Open Street Map data (as described in Section 3.1.1) and alter the routing algorithm that it prefers smaller over larger paths (or however one wants to define the quality of a jogging route). Of course, this means that one has to rely on the consequent tagging of ways, which in turn creates new problems as described in Section 2.5.

6.2 Routing

As described in Section 2.2.2, the routing algorithm is only heuristic. Thus, there is a lot of room for improvement. Ideas include applying machine learning algorithms to tune the parameters of the routing algorithm or incorporating user feedback about the generated routes.

6.3 Evaluation

As described in Section 5.1, we collected a lot of data which we evaluated. However, the collected data was from a small group of people. To get more representative results, one would need a larger testing group. With such data, one could most certainly detect some trends amongst the users and adjust the application accordingly.

Bibliography

[1] Open Street Map. https://www.openstreetmap.org. Accessed: March 9, 2016.

[2] Open Street Map Copyright. https://www.openstreetmap.org/copyright. Accessed: March 9, 2016.

[3] Tomcat. http://tomcat.apache.org/. Accessed: March 9, 2016.

[4] OSM Mirrors. http://wiki.openstreetmap.org/wiki/Planet.osm#Planet.osm_mirrors. Accessed: March 9, 2016.

[5] Osmconvert. http://wiki.openstreetmap.org/wiki/Osmconvert. Accessed: March 9, 2016.

[6] PBF. http://wiki.openstreetmap.org/wiki/PBF_Format. Accessed: March 9, 2016.

[7] Android. https://www.android.com/. Accessed: March 9, 2016.

[8] JSON. http://www.json.org/. Accessed: March 9, 2016.

[9] Open Street Map, Map Features and Tags. http://wiki.openstreetmap.org/wiki/Map_Features. Accessed: March 9, 2016.

[10] Open Street Map Elements. http://wiki.openstreetmap.org/wiki/Elements. Accessed: March 9, 2016.

[11] Open Street Map Mirror Switzerland. http://planet.osm.ch/. Accessed: March 9, 2016.

[12] JGraphT. http://jgrapht.org/. Accessed: March 9, 2016.

[13] Osmdroid. https://github.com/osmdroid/osmdroid. Accessed: March 9, 2016.

[14] Android TextToSpeech. http://developer.android.com/reference/android/speech/tts/TextToSpeech.html. Accessed: March 9, 2016.

[15] Osmdroid Bonuspack. https://github.com/MKergall/osmbonuspack. Accessed: March 9, 2016.

[16] Android Beta. https://support.google.com/googleplay/android-developer/answer/3131213?hl=en. Accessed: March 9, 2016.