Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Smart Running Route Generation

Masterproject

Jan Schulze

schulzej@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Manuel Eichelberger
Prof. Dr. Roger Wattenhofer

30.11.2016

Abstract

In this thesis we introduce a model to assign and rank paths and routes with the objective to find the best and most beautiful of these routes that have some basic constraints. We show and discuss algorithms that find such best routes based on our model. We apply the theory in an Android application based on OpenStreetMap data, that finds optimized routes with adjustable length at any location for outdoor sports activities like for example running, walking or going for a walk.

Contents

Chapter 1

Introduction

Running is one of the most popular sports worldwide. Worldwide, millions of people are running regularly. There are other activities that are very similar. Examples are cycling, hiking or also just going for a walk. People do these sports for several reasons like for example fun, fitness and health or relaxation. While sometimes exploring the nature by oneself, and the freedom to be not restricted to certain routes, are key parts of the activities, often experiences and fun can be improved by selecting nice routes. For example, there are probably very few people who like to run next to a motorway instead of a track that is surrounded by a beautiful landscape. Bad road and path conditions also affect the experience negatively. And lastly, extremely steep paths are most of the time not welcomed and are only enjoyed by few sportsmen, especially if there is also a less steep path leading to the same destination. Better experiences not only result in more fun, but also in higher motivation, which in return can improve the athletic performance. In the following, we will take running as an example for all those activities.

If a runner wants to run a nice route, he can do several things to achieve this goal. If the runner knows the area well, he might be able to decide during the run where to go to get a nice route. This is actually a good approach, as it not only results in good routes but also distracts the runner from the physical efforts he is doing and therefore leads to better training results. But this approach works only if we assume that the runner knows enough of the area to create nice routes with his knowledge and is satisfied with restricting his running only to these areas. Furthermore, the runner also has to be able to concentrate enough to create those routes, which might not always be the case depending on the intensity of his training. Additionally, if we do not have a runner but for example a person going for a walk, the person probably does not want to plan ahead this much and wants to just enjoy the landscape instead of thinking about his route.

Of course, the runner can also follow this strategy if he does not know anything about the area he runs in. Because the runner does not know anything about the area he does not have to concentrate on the route and plan ahead as he only decides where to go next at crossings and forks of the path. In most cases

this strategy of deciding online where to go next without knowing the area will lead to at least decent routes. If the runner does not care about the landscape this will often be a sufficient strategy. But the runner might miss many nice routes as he only decides according to the near surroundings but if for example there is a very nice landscape behind a large street and some houses the runner cannot see, he will most probably miss it.

This strategy has one caveat which is that it is very difficult to get a route of a desired length. If we do not care about the length, this is no problem. But most of the time the length of the route does matter. First the runner might have a training plan for which he needs a certain intensity of his training. Also if the runner needs to be back at a given time a route with unknown length is disadvantageous. And lastly the runner does not want to be already exhausted if he is still far away from his home.

Another strategy that the runner can follow, is to stick to a nice route he already knows. As these are mostly routes he ran before, the runner might get bored after some time depending on how much variation he likes for his routes.

Luckily, the runner nowadays has many tools he can use to optimize his routes. At first, he can create his own route using maps. This is most likely too much work except if the user wants to stop by a certain place and even then it takes some effort to work out the best route. The runner can also choose from a wide variety of predefined or shared routes (e.g. the websites Runkeeper[1] for running, Contours[2] for walking or Runtastic[3] and MapMyRun[4] which provide both as well as an application as a website). The advantage of this method is that the runner will get nice routes which are preselected and tested. The disadvantages are that the routes most likely do not start where the runner wants to start. Also, there are not infinitely many routes in each area and therefore the runner is restricted to some routes with certain lengths. If the runner wants to run a route with a specific length, he will not have a wide choice or maybe even find no route that satisfies his denies. And last but not least, the routes are static, therefore they cannot be adapted or changed during the run.

Using other technologies for routing also does not solve the problem, as routing services like navigation systems, Google Maps[5] etc. are not aimed at nice routes but only at the shortest, fastest or most fuel-efficient routes from a start to a destination.

In our application, we implement a simple solution: Our application creates a preferably nice route and adjusts it to the user's preferences, like start point and length.

[1] runkeeper.com: Runkeeper - Find the best running routes on Runkeeper
[2] contours.co.uk: Walking Scotland, England, Wales
[3] www.runtastic.com: Runtastic: Running, Cycling and Fitness GPS Tracker
[4] www.mapmyrun.com: MapMyRun
[5] maps.google.com: Google Maps

Shop	Caffeine per litre	Volume	Price
Starbucks	$20\frac{mg}{l}$	0.5 l	5 $
Italian coffee shop	$100\frac{mg}{l}$	0.1 l	2 $
Coffee machine at the office	$10\frac{mg}{l}$	0.3 l	0.5 $

Figure 1.1: Example for coffee supply

1.1 Related Work

Previous Bachelor Thesis

This work is based on the bachelor thesis **On the Fly! - Automatic Running Route Generation** [1]. In that thesis, an application is presented, that generates routes automatically for the user. The routes are optimized to a length the user chooses. The shape of the routes is approximated with a circle. The type of roads used for the routes are not considered, thus even though we might get a route with the desired length, the probability that it will be a nice route which the user can enjoy is not very high.

1.1.1 Optimization

Optimization problems occur everywhere even in daily life. Where do we have to buy our coffee to get the most caffeine for our money? What is the shortest way to the supermarket? Which television do we buy - the cheapest one or the one with the largest screen?

There are problems which only have one variable that we have to optimize for example the first problem with the caffeine.

We get the solution intuitively by calculating the caffeine per money and choosing the shop with the highest value.

$$\frac{caffeine}{litre} \cdot \frac{litre}{money}$$

In the Figure 1.1 this is the coffee machine at the office with a value of $6\frac{mg}{\$}$. To be more formal, we define a so called objective function f for every optimization problem which we can maximize or minimize.

$$f : A \to \mathbb{R} \tag{1.1}$$

In our example, $f(x)$ is the objective function which shows how much $\frac{caffeine}{money}$ we get when buying the coffee at the location $x \in A = \{x_{starbucks}, x_{italian}, x_{office}\}$.

We solved the maximization problem with $\max_x f(x)$ by calculating all possible outcomes of f for every shop x and selecting the best.

Usually, a maximization problem can be turned easily into a minimization problem and vice versa by negating the objective function $\min_x -f(x)$, which of course returns the same results. If the codomain of f is not $\mathbb{R}$ but for example either strictly positive (or negative) $f : A \to \mathbb{R}$ we can also turn a maximization problem into a minimization problem by using the reciprocal of the objective function $\min_x \frac{1}{f}$.

Shortest Path Problem

Given a graph $G = (E, V)$ with its edges $e \in E$ and nodes $v \in V$. Every edge has a weight $d \in \mathbb{R}_{>0}$. We want to find a path from a node s to a node t such that the the cost function of the resulting path is minimized. Usually we have an additive cost function that adds up the weights of the edges contained in the path. We show how to solve this problem using Dijkstra's Algorithm.

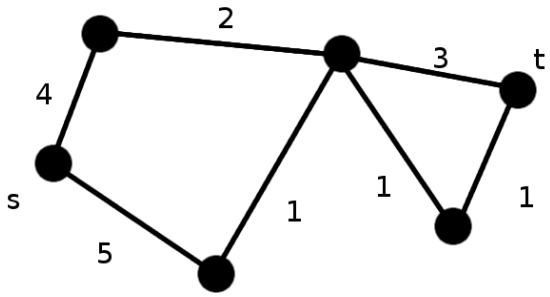


Figure 1.2: We convert the real world street network to a graph.

Dijkstra

Dijkstra's Algorithm implements a breadth first search on the graph G. In Figure 1.3 we show an example of the algorithm.

Multi Objective Optimization

The third example, about the question what television we want to buy if we consider the price and the size of the television, is a so called multi objective optimization problem as we have several objectives - in our case two - the price and the size of the screen, that we want to optimize.

Algorithm 1: Dijkstra's Algorithm

```
1 openList.add(s)
2 while openList not empty do
3     v = openList poll node with smallest cost
4     for neighbour in neighbours of v do
5         if openList contains neighbour then
6             neighbour.cost = v.cost + cost(v,neighbour)
7         end
8     end
9 end
```

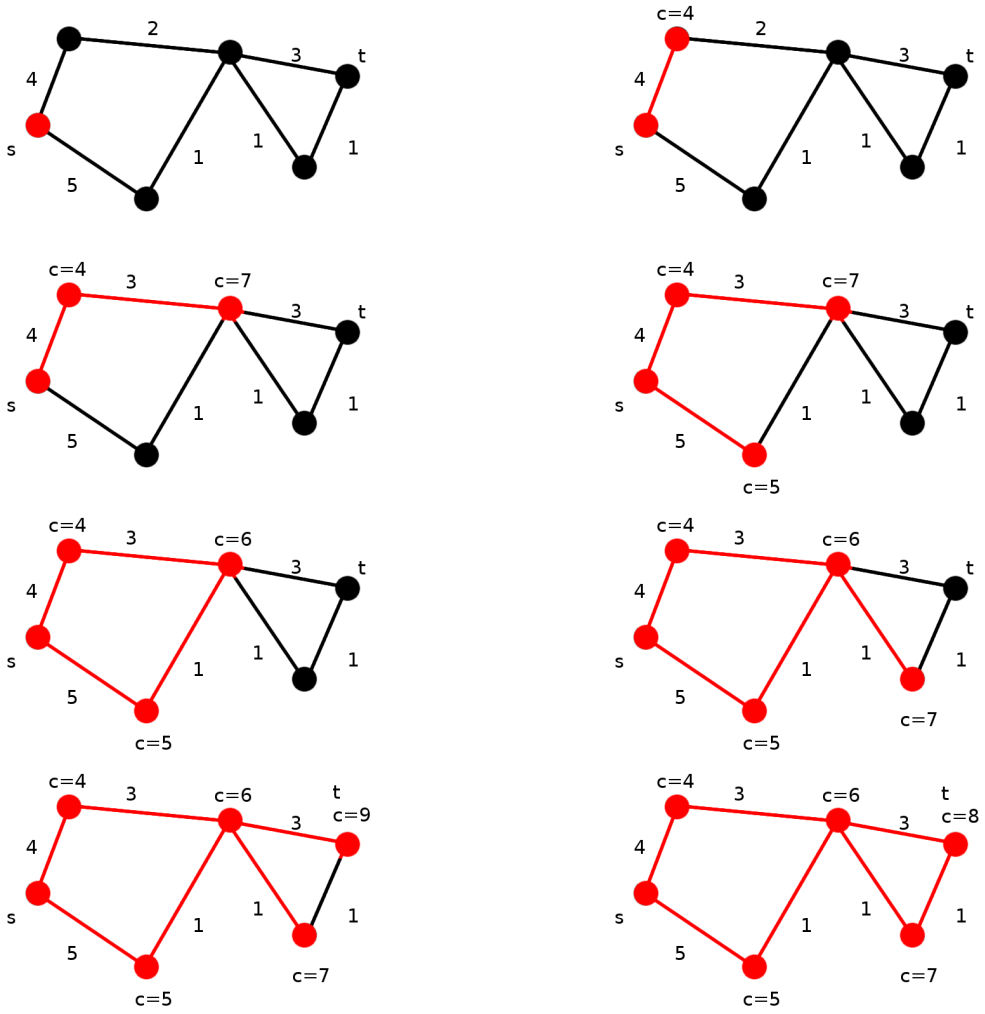


Figure 1.3: Example of Dijkstra's Algorithm

With the given information, we cannot necessarily find the best television, as the best television does not necessarily exist. For example, there might be one television that is the cheapest and another that has the largest screen. So we can have several constellations of television that are optimal in a way. We can find televisions that are clearly not optimal, for instance if we compare two televisions and one of them has a larger screen and is cheaper then this television is better than the other. We can express this kind of optimality using the Pareto Optimality. We use the definition from [2] that was partially taken from [3].

Definition 1.1. A solution is Pareto-optimal (i.e., Pareto-minimal, in the Pareto-optimal range, or on the Pareto front) if it is not dominated by any other solutions.

A vector x is partially less than y, or $x < py$ when: $(x < py) \Leftrightarrow (\forall i)(x_i \leq y_i) \wedge (\exists j)(x_i < y_i)$ x dominates y iff $x < py$.

If we want to get a unique solution we have to introduce an additional constraint. For example by deciding that we have 300 Chf that we want spend on a television that has a screen that is as large as possible.

Chapter 2

Theory

2.1 Problem Statement

We want to find the best routes from a start to a destination given a number of constraints. Most of the time the destination is the same as the start, which already makes our problem different to most of the routing problems out there. The route shall have a certain length the user can choose. Those best routes should be the routes which are the most fun and enjoyable to run for the runner. But how do we find them?

2.2 Modeling the Street Network

We model the street network as a graph $G = (V, E)$ consisting of undirected edges E representing the roads, paths etc. and the nodes V representing the crossings or meeting places of the those roads.

Figure 2.1: We convert the real world street network to a graph.

We assign two parameters to every edge. Every edge $e_i \in E$ has a length parameter $l_i \in \mathbb{R}_{>0}$ that represents the length of the according road and a second parameter $a_i \in \mathbb{R}_{>0}$ that we call the attractiveness. The attractiveness represents how nice or bad a route is depending on for example surroundings, road type or condition. The higher the attractiveness of an edge, the nicer the corresponding road section and the more enjoyable and fun it is to run on that path. For example, an edge $e_{nice} \in E$ that represents a beautiful path - like a path in nature with a spectacular view of the landscape - will have a high attractiveness,

whereas an edge $e_{bad} \in E$ that represents a street with several lanes and much traffic will have a low attractiveness. In our example with the two edges e_{nice} and e_{bad}, it holds that $a_{nice} > a_{bad}$. How the attractiveness is defined specifically, we show later in this chapter.

We define a route R to be a path on the graph G that is optimized for attractiveness, while having a given length. The route R may contain an edge several times.

2.3 Optimizing the Length

First, we explain how we can find a route R whose length l_R is as close as possible to the desired length l, not regarding the attractiveness. We define $\Delta l = |l - l_R|$ and minimize Δl. We can see that this problem is more complicated than the Shortest Path Problem. If we build up a path, Δl decreases at first and increases after the path gets longer than the desired length l. In the Shortest Path Problem, there exists an optimal path from every node to the end node. Thus, it is not necessary to look at all possible paths that exist from the start to the end. In our problem we cannot determine such optimal partial solutions and therefore, we have to look at all possible paths that exist between the start to the destination.

2.4 Optimizing the Attractiveness

Maximizing Average Attractiveness of Route

One possibility to define a model is to optimize the average attractiveness of the route. For this, we define the attractiveness for every edge a_i as follows: $a_i \in \mathbb{R}_{>0}$. The average attractiveness of a route R is a_R:

$$a_R = \frac{1}{l_R} \sum_{i|e_i \in R} a_i \cdot l_i \tag{2.1}$$

Therefore we try to get a route with many edges that have a high attractiveness. It is possible to have some edges with low attractiveness but following from the definition this will only be the case if therefore there are many nice routes so that the resulting route has still a high attractiveness. Note that this definition is very similar to the definition introduced in **??** but with the difference that we can compare the attractiveness of routes that do not have the same length.

Maximizing the Worst Edge of the Route

Another possibility is that we do not care as much about the average attractiveness as about the worst attractiveness a_{worst} of the edge e_{worst} that is contained in the route R. The attractiveness of a route R is therefore $a_R = a_{worst}, a_{worst} \leq a_i, \forall i | e_i \in R$. With this model, we want to prevent routes from having edges with a low attractiveness. But we do not have a guarantee that our route contains edges with a high attractiveness. Also, if the route has to lead through at least one edge with very low attractiveness because of the given conditions, this optimization method is useless and we need to do many adjustments to get more or less useful results.

Maximizing the Best Edge of the Route

A similar model is to maximize the best edge of the route. With this model we want to get a route that has at least one very good edge. The problem with this approach is that we have no guarantee that the over edges are somehow good. We might argue that if we find a very nice edge that there are probably other nice edges in the area but we do not have any guarantee for that. The worst case is that we have a very short edge with a very high attractiveness and all other edges have a very low attractiveness. Therefore, this model is not useful for our purpose.

Comparison of the Models

In our opinion, the average attractiveness fits the needs of our purpose the best. With the average attractiveness, the resulting route will have many nice edges. Although the route may contain some bad edges, this does not matter much because most runners will accept to have a short path that is not beautiful if they therefore can run most of their route on very nice and beautiful paths.

For only decent routes on the other hand, even if they do not contain bad paths, there is much less demand. Therefore, we do not maximize the worst edge of the route. Maximizing the best edge of the route does also not lead to good routes.

2.4.1 Maximizing the Attractiveness by Restriction to Few Important Edges

Even though we decided to use the average attractiveness, we will present a different noteworthy model that has some drawbacks and some benefits compared to the average attractiveness. We define an alternative model of the attractive-

ness as $a_{alternative} \in \mathbb{R}$ with a set of attractive edges $E_{attractive} \subseteq E$, a set of unattractive edges $E_{unattractive} \subseteq E$ and a set of neutral edges $E_{neutral} \subseteq E$ with

$$E_{attractive} \cup E_{neutral} \cup E_{unattractive} = E$$

$$E_{attractive} \cap E_{unattractive} = E_{attractive} \cap E_{neutral} = E_{neutral} \cap E_{unattractive} = \emptyset$$

The edges in $E_{attractive}$ have strictly positive values, the edges in $E_{unattractive}$ have strictly negative values and the edges in $E_{neutral}$ have an attractiveness of zero. Moreover we define the attractiveness to only take values in $\{-1, 0, +1\}$ and therefore treat all very good edges equal, all neutral edges equal and all bad edges equal.

Thus, we split up the set of edges in our graph into a subset of edges that we want to have in our path if possible, namely the edges in $E_{attractive}$, into edges we do not want to have in our path if possible, namely the edges in $E_{unattractive}$, and in neutral edges. An neutral edge with the attractiveness of zero can be interpreted as an edge that we do not care about whether it is in our route or not. We only assign very few edges to the set $E_{attractive}$ and very few edges to the set $E_{unattractive}$. This should be the best and the worst edges regarding the attractiveness of our graph. The majority of the edges are neutral so we do not care if those edges are in our route or not.

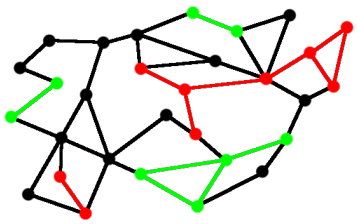


Figure 2.2: Example of a graph. Green edges represent good edges, red edges represent bad edges and black edges represent neutral edges.

With this approach we can reduce the complexity of the search for good routes drastically compared to the average attractiveness, as we only have to look at the clusters of good edges and how to connect them in a good way. The large drawback of this definition is that we loose the information about all the neutral edges which are the bulk of all edges. Thus this model is only useful if the average attractiveness is for any reason too complex to implement or if we really do not need these informations.

2.5 Optimizing the Route with Two Weights

Optimizing the route R according to both length and attractiveness yields another problem. As there are many possible Pareto-optimal routes R (see Figure 2.3), which route do we choose?

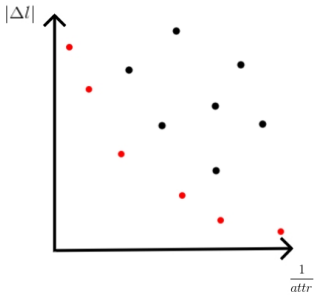


Figure 2.3: Example of the representation of attractiveness and length in a Pareto Graph. We marked the points in the Pareto-Front red.

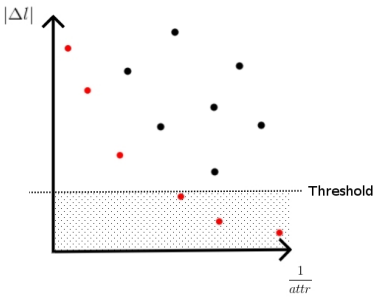


Figure 2.4: By introducing a length constrain, we get a region in which we can optimize over the attractiveness.

We have to define an objective function to be able to chose the optimal route R_{opt}. The attractiveness should be penalized if low and rewarded if high. The length of the route l_R should be close to l but for our purpose it is enough if Δl is not exactly zero but small. Therefore, we do not want to penalize small Δl much or at all. On the other hand, if Δl is large we want to penalize this very hard. We define our objective function as:

$$c = \begin{cases} a, if \Delta l \leq l_{thresh} \\ -\Delta l, if \Delta l > l_{thresh} \end{cases} \tag{2.2}$$

With this objective function all routes that fulfill the condition $\Delta l < l_{thresh}$ with a predefined threshold l_{thresh} is equal to the attractiveness of the route. This means that for routes that are close to the desired length we only consider the attractiveness of these routes to choose the best route. The region in which we optimize for this case in the Pareto Graph is shown in Figure 2.4. If we do not find any route which can fulfill the condition $\Delta l \leq l_{thresh}$ we select the route that is as close to the desired length as possible regardless of the attractiveness. Note that we have to set the threshold l_{thresh} thoughtful. If l_{thresh} is too small, we might not find routes that fulfill the condition and therefore select a route

that can have any attractiveness. If l_{thresh} is too large, we get routes that have a length that is much longer or shorter than the desired length. We show an example for a ranking using this objective function in Table 2.1.

Route	Δl	a_R	c	Rank
1	0	0.5	0.5	4
2	50	1.5	1.5	2
3	50	2.0	2.0	1
4	100	3.0	-100	6
5	60	1.0	-60	5
6	20	1.5	1.5	2

Table 2.1: The table shows an example of some routes. The length threshold is $l_{threshold} = 60$. We can see that route four is ranked worst because it does not fulfill the length condition even though it has the highest attractiveness. Route two and route six are ranked equally although route six has a smaller Δl.

All we have to do now is to maximize the objective function. Note that we have to set the length threshold l_{thresh} not too small as otherwise we will not find any routes fulfilling the length condition and therefore only rank the routes by their length.

2.6 Calculation of the Optimal Route

To calculate the optimal route, we first show an algorithm that returns the optimal route in relation to the objective function in Section 2.5. The algorithm uses a brute force approach as it builds up all possible routes and selects the route with the highest value for the objective function. During the building process we try to detect and sort out routes that cannot become the optimal route early to improve the performance of the algorithm. We assume for this algorithm that the route can contain any node at most once except the start and destination node if it is the same. This is due to the fact that we want to avoid crossings and that the route contains edges several times.

Algorithm 2: Brute Force Algorithm

```
   Result: Optimal Route
 1 Init
 2 bestRoute = null
 3 startRoute = [start]
 4 priorityQueue.add(startRoute)
 5 while priorityQueue not empty do
 6     currentRoute = priorityQueue.poll route with highest a · l
 7     if newRoute can be better than bestRoute then
 8         neighbours = currentRoute.lastNode.getNeighbours()
 9         for neighbour in neighbours do
10             newRoute = currentRoute.add(neighbour)
11             if currentRoute contains neighbour then
12                 if neighbour == end then
13                     if newRoute better than bestRoute then
14                         bestRoute = newRoute
15                     end
16                 else
17                     Deadend handling
18                 end
19             else
20                 if newRoute can be better than bestRoute then
21                     priorityQueue.add(newRoute)
22                 end
23             end
24         end
25     end
26 end
27 return bestRoute
```

In line 17 of Algorithm 2 we have to take care of the special case if the start and destination are the same and lie in a deadend. In this case, we will not find a *bestRoute* in Algorithm 2 because we have to pass the first edges of the route also at the end to get back to our starting point.

Figure 2.5: As the start and end point lies in a deadend, we have to take several nodes twice to complete the route.

Thus, as long as we have not found a route that consists only of unique nodes, we have to take care of the best route, that does not fulfill the condition, that we do not have several equal nodes. If the node we add to a route is already contained in the route, we therefore start the *Deadend handling*. The *Deadend handling* in line 17 of Algorithm 2 is shown in Algorithm 3.

Algorithm 3: Deadend Handling

```
1 bestRouteDeadend
2 if bestRoute exists then
3     continue
4 else
5     completeRoute = complete newRoute
6     if completeRoute better than bestRouteDeadend then
7         bestRouteDeadend = completeRoute
8     end
9 end
```

From the routes that contain several nodes twice, we only keep track of the best of them. To determine the best of them, we add the first part of the route to the end of the route, so that the beginning and the end of the route are equal. Then we keep track of the best route until we find a route that consists of unique nodes. If we do not find a route that consist of unique nodes, *bestRouteDeadend* becomes our best route.

Next, we look at line 7 and line 20 in Algorithm 2. In these lines we try to detect as early as possible whether a route can still become the best route or not. For this, we calculate the minimum distances and the maximal possible average attractiveness from every node to the end node.

First, we check if the route can still fulfill the length condition. After that, we check if the route can still be more attractive than the best route we have found so far. We can only sort out routes if we have already found a best route because a bad route even if it does not fulfill the length condition is better than no route. Therefore in line 6 we choose a route with a high attractiveness and a long length to find an attractive complete route fast. If we find such a route we can sort out worse routes. Afterwards, we can change to depth first regarding only the attractiveness of the routes if desired.

Algorithm 4: newRoute can be better than bestRoute?

Result: Whether newRoute can still fulfill conditions or not

```
1  Input:
2  newRoute

   // min distance from node to the end node:
3  minD = minDistToEndNode

   // max average attractiveness from node to the end node:
4  maxA = maxAvAttrToEnd

5  if | bestRoute.length-length | > threshold then
6      return true
         // no route found yet, that fulfills the length condition
7  end

   // route can still fulfill length condition?
8  lenBestCase = minD(newRoute.lastNode) + newRoute.length
9  bool cond1 = lenBestCase < length + tolerance

   // route can still be more attractive than best route
10 attrBestCase1 = (maxA(newRoute.lastNode)·(length−newRoute.length+tolerance)) / (length+tolerance)
11 bool cond2 = attrBestCase1 > bestRoute.averageAttractiveness

12 attrBestCase2 = (maxA(newRoute.lastNode)·(length−newRoute.length−tolerance)) / (length−tolerance)
13 bool cond3 = attrBestCase2 > bestRoute.averageAttractiveness

   // route can still fulfill both conditions
14 if cond1 AND (cond2 OR cond3) then
15     return true
16 end
17 return false
```

The minimum distances from every node to the end node and the maximal possible average attractiveness from every node to the end node are calculated

using Dijkstra's Algorithm in a preprocessing step in line 1 of Algorithm 2.

Algorithm 5: Init

```
1 Init minDistToEndNode
2 Init maxAvAttrToEnd
```

Algorithm 6: Init minDistToEndNode

Result: Initializes minDistToEndNode

```
1  minDistToEndNode = Map<node→ minDist>
2  predecessors = Map<node→ predecessor>
3  for node in allNodesOfGraph do
4      minDistToEndNode.put(node→ ∞)
5      predecessors.put(node→ null)
6  end
7  minDistToEndNode.put(end→ 0)
8  openList.add(end)
9  while openList not empty do
10     node = openList.popNode
11     neighbours = node.getNeighbours
12     for neighbour in neighbours do
13         newDistance = minDistToEndNode(node) +
           Distance(node,neighbour)
14         if newDistance < minDistToEndNode(neighbour) then
15             minDistToEndNode.put(neighbour→ newDistance)
16             openList.add(neighbour)
17             predecessors.put(neighbour→ node)
18         end
19     end
20 end
```

Algorithm 7: Init maxAvAttrToEnd

Result: Initializes maxAvAttrToEnd

```
1  maxAvAttrToEnd = Map<node→ maxAvAttr>
2  usedPaths = Map<node→ usedPathsForMaxAvAttr>
3  for node in allNodesOfGraph do
4      maxAvAttrToEnd.put(node→ 0)
5      usedPaths.put(node→ null)
6  end
7  openList.add(end)
8  while openList not empty do
9      node = openList.popNode
10     neighbours = node.getNeighbours
11     for neighbour in neighbours do
12         avAttrNode = maxAvAttrToEnd(node)
13         pathNode = usedPaths(node)
14         if pathNode does not contain neighbour then
                                                  // prevent infinite loops
15             pathNeighbour = pathNode.add(neighbour)
16             newAvAttr =
               (avAttrNode·pathNode.length+getAttractivity(edge=[node,neighbour])·Distance(node,neighbour)) / pathNeighbour.length
               if newAvAttr > maxAvAttrToEnd(neighbour) then
17                 maxAvAttrToEnd.put(neighbour→ newAvAttr)
18                 openList.add(neighbour)
19                 usedRoutes.put(neighbour→ pathNeighbour)
20             end
21         end
22     end
23 end
```

2.7 Computation of a Locally Optimal Route

The brute force version of the algorithm gets very fast to its limits of being able to compute the routes in reasonable time with increasing total path length. Therefore, we developed another algorithm that uses a different approach.

The Expanding Algorithm 8 initializes with a short route which we call the initial route. The route is then expanded every step of the algorithm by replacing a route part with a longer route part. As there are many route parts that can be replaced with a longer route part, we choose to replace the one that will give us the highest gain for the average attractiveness of the route. Note that this does not always have to be a positive gain as we always increase the length of the route. This algorithm does not necessarily find the optimal route as the Optimal Algorithm in 2.6 but rather a locally optimal route. Locally optimal does not refer to the area that the graph represents but to the starting conditions of the algorithm in our case the route we get during the initialization.

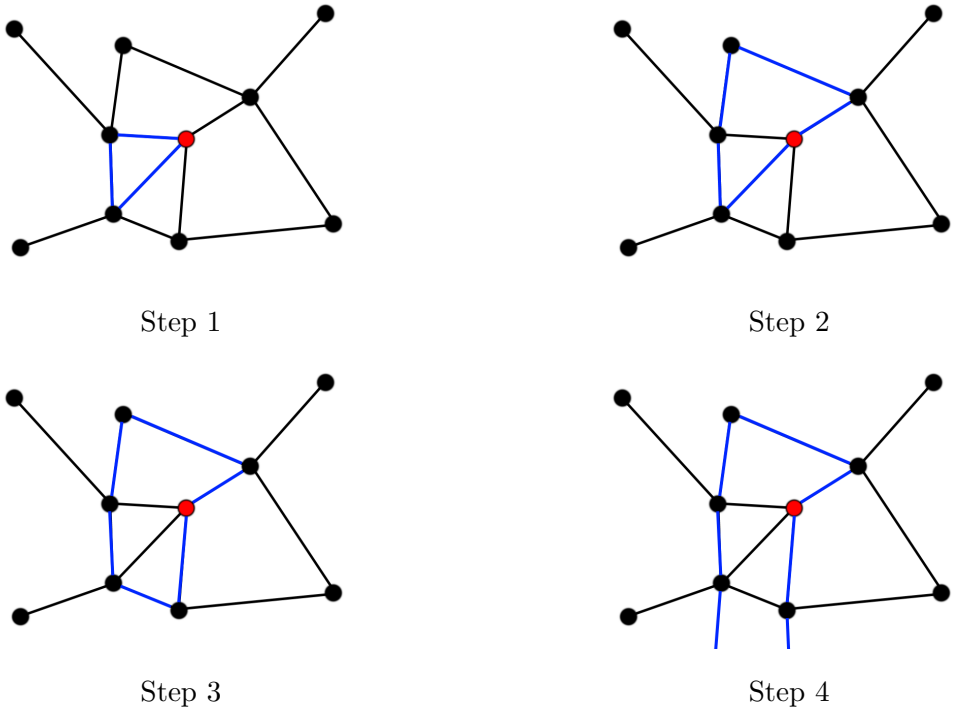


Figure 2.6: Example of the Expanding Algorithm

Thus, we can run the algorithm with any possible initial route in the hope of that we will get the optimal route. But doing this needs much calculation power and we try with this algorithm to decrease the need of resources. Thus, we try to find one good initial route to start with. If we are lucky, we still get

the globally optimal route and if not we assume that the locally optimal route we get is attractive enough to fulfill our purpose. If not all locally optimal routes are attractive enough to fulfill our purpose, we have to select the initial route carefully to be in an area of starting conditions, that lead to a good locally optimal route. We did not examine how to do such an estimation but this could be something that can be done in the future.

Algorithm 8: Expanding Algorithm

Result: An optimized route

```
1  route = initRoute()
2  finished = false
3  while not finished do
4      select best part of the route to replace
5      if route can be improved then
6          replace part in route
7      else
8          finished = true
9      end
10 end
```

Finding the best part of the route to replace is straight forward. We search for every part of the route up to a defined maximum length of a route part for the best replacement that increases the length of the route. We compute those best replacements using brute force to a certain depth. The depth indicates the maximal length of a replacing part.

2.8 Random Points Algorithm

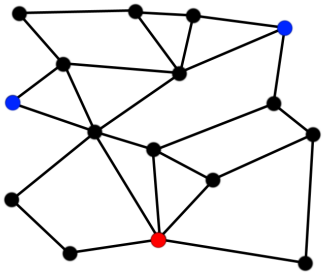
In the last algorithm we do not look for the one optimal route anymore. Instead, we just look for routes whose attractiveness is close the the attractiveness of the optimal route. With this algorithm, we decide to go one step further. The Optimal Algorithm in 2.6 is deterministic and returns the optimal route but is limited in its performance. The Expanding Algorithm in 2.7 is deterministic and returns a route that is at least locally optimal. So, the next step is to try a nondeterministic algorithm that returns a route that is not necessarily optimal but as attractive as possible.

For this, we developed the Random Points Algorithm. The algorithm is very simple: We choose and connect random nodes of the graph using the Algorithm 9. If two nodes that we want to connect are close enough, so that we can compute the connecting path directly we do this. Otherwise, we randomly select nodes between the two nodes and build a connection by connecting all nodes using this algorithm recursively. We compute this connection either with the Brute Force Algorithm 2, the Expanding Algorithm 8 or the Shortest Path Algorithm **??**. If we use the Brute Force Algorithm we get optimal connections between two nodes in relation to the attractiveness. A drawback if we use this algorithm is that the two nodes have to be very close to each other as the Brute Force Algorithm can only compute short routes. With Expanding Algorithm 8 we can create longer connections and are still optimal. The Shortest Path Algorithm does not guarantee at all that we get attractive connections. The idea behind using the Shortest Path Algorithm is, that because the shortest path between two nodes is so easy to compute, we can do more executions of an the algorithm and therefore have a higher chance of randomly finding an attractive route.

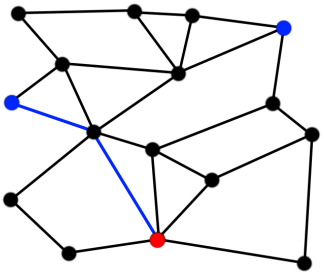
We do not assign every node the same probability to be chosen but a higher probability if the node belongs to attractive ways and a lower probability if the node belongs to unattractive ways.

If we run this algorithm once, the probability is very high that we do not get an attractive route. Therefore, we have to run the algorithm many times and choose the best route from the routes we get. We can do this because the algorithm is very fast.

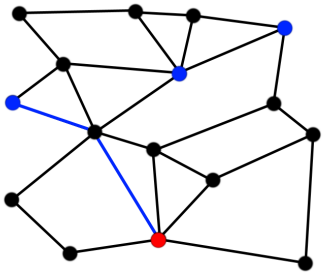
We can further adapt the algorithm if we do not select completely random nodes, but limit the set of nodes we can draw from to a subset of all possible nodes. If we do this intelligently, we can improve the expected attractiveness of the resulting route even further. We did not study how to select such a subset of nodes, but this can be done in the future. This could be done for example by dividing the edges of the graph in good, neutral and bad edges like we describe in Section 2.4.1. Then we select the subset as the set of nodes of those good edges.



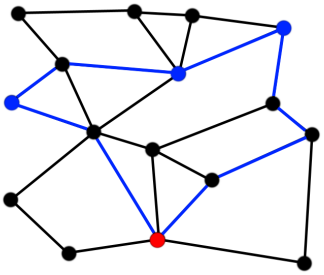
We choose randomly some nodes. The red node is our start and end node



If we can compute the connection of two nodes directly we do this.



If not, we choose randomly some additional points between the two nodes.



We continue until we finished the route.

Figure 2.7: Example of the Random Points Algorithm

Algorithm 9: Random Points Algorithm

Result: Random route

```
// Input:
start
end
n: number of target nodes
route = [start]
if start == end then
    randomNodes = get n random nodes
    targetNodes = order randomNodes
else
    // start != end
    if Distance(start,end) small enough to calculate Route directly then
        route = BFAlgorithm(start,end)
        OR route = ExpandingAlgorithm(start,end)
        OR route = shortestPath(start,end)
        return route
    else
        randomNodes = choose n random nodes between start and end
        targetNodes = order randomNodes
    end
end
targetNodes.add(end)
for node in targetNodes do
    currentNode = route.removeLast()
    routePiece = RandPointsAlg(currentNode,node,1)
    route.add(routePiece);
end
return route
```

Algorithm 10: Complete Random Points Algorithm

Result: A random attractive route

```
1  n = 2
2  bestRoute = null
3  while time < predefined time do
4      newRoute = RandPointsAlg(start,end,n)
5      if newRoute.length larger than length condition AND n > 2 then
6          n–
7      end
8      if newRoute.length smaller than length condition then
9          n++
10     end
11 end
```

2.9 Shape of the Route

With the algorithms, that we introduced so far, we can compute routes with a good average attractiveness. But sometimes, these routes, or at least some parts of them, are not what we expect to be a good route because the shapes of some parts of those routes are not desired.

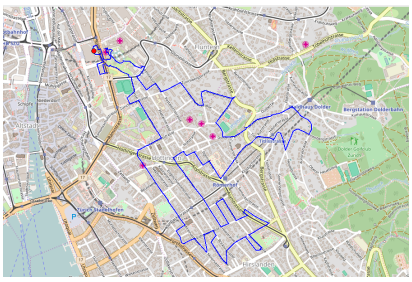


Figure 2.8: Example for a undesired route shape because the user has to run zigzag all the time.

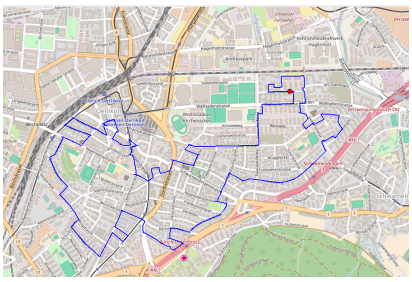


Figure 2.9: Although the overall shape is better than in figure 2.8 changing the direction often for no reason is not desired.

Figure 2.10: An enlargement of one of the unwanted artifacts from Figure 2.9.

Figure 2.11: Examples for bad route shapes

So, we also have to consider the shape of the routes, if we want to create nice routes. But finding a measure for determining whether a route has a good or bad shape is so complicated that it would probably be enough work for an entire new master thesis.

2.10 Smoothing the Route

We want to eliminate artifacts like these in in Figure 2.10. A generalization of these kind of artifacts are small loops as shown in Figure 2.12.

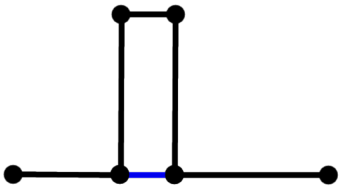


Figure 2.12: General example for small loop artifacts. Although a direct path exist we make a short detour, often around one or several houses. If the detour is very large this is not bad. But if it is short, this detour makes no sense. Therefore we want to eliminate these small loop artifacts.

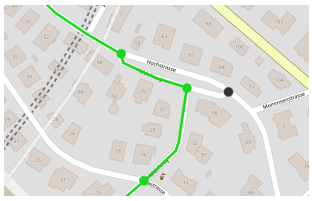
To eliminate some of these artifacts, we do as follows. We design an algorithm that checks if there exist such artifacts and replaces them with the direct path, if possible.

Algorithm 11: Route Smoothing

Result: Remove unwanted artifacts from route

```
1  for node in route do
2      routePiece = [node]
3      while length of routePiece is smaller than a threshold do
4          nextNode = route.getNextNode(routePiece.lastNode)
5          routePiece.add(nextNode)
6          if distance of first to last node in routePiece is smaller than a
           percentage of the length of routePiece then
7              SP = shortestPath(routePiece.firstNode,routePiece.lastNode)
8              if SP.length smaller than routePiece.length then
9                  route = replaceInRoute(route, SP)
10             end
11         end
12     end
13 end
```

An artifact that we might want to remove.



The artifact is removed and replace with the shortest path.

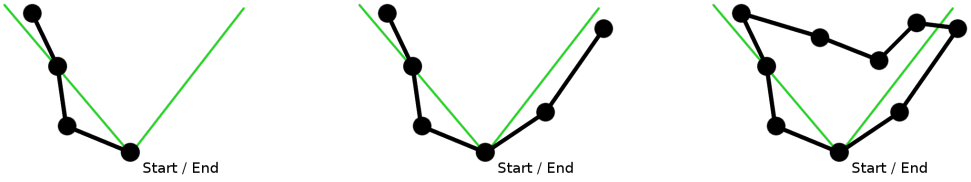
Sometimes there are reasons why we have artifacts. In this example there does not exist a path that prevents this artifact as there is no shorter path because of the river.

Figure 2.13: Example for the Route Smoothing Algorithm

2.11 Triangle Algorithm

In all algorithms we covered up to now, we only consider the attractiveness and the length of the route. But as we have seen in section 2.9 we also have to consider the shape of the route. Ideally an algorithm creates an attractive route such that we do not need additional algorithms like for example the Smoothing Algorithm 2.13. But developing a measure of quality for the shape of a route is such a complicated task that it can easily take a whole new project. Therefore, we keep it simple and try to approximate the shape of our route with a triangle. Thus we ensure that the shape of our route will not be too bad.

The algorithm works as follows. At first we create two branches in different random directions. Next we select pairs of nodes each containing one node from the first branch and one node from the second branch. We calculate the connections of the nodes so that we get many slightly different routes. From those routes, we take the ones that fulfill the length condition and choose the most attractive route.



Step 1: Create First Branch | Step 2: Create Second Branch | Step 3: Connect Branches

Figure 2.14: Simple example for the Triangle Algorithm

The process in which we shorten the two branches in line 8 is straightforward. Additionally we try to reduce the need for shortening the branches by choosing a good estimate of the branch length in line 6 and 7. We connect the branches in line 12 either with the Expanding Algorithm 8 or with the Random Points Algorithm 10.

For creating the branches, we have tried out several ways to do so. At first we tried keeping the branches as close to a given half line as possible. We show this in Figure 2.15. Therefore, we create the half line starting at the start node in the given direction. In the beginning, the branch only consists of the start node. In every step, we choose the best neighbor of the last node of the branch. We determine the best neighbor as follows. For every neighbor we calculate the distance from the half line $dist_{Vert}$. We want to stay close to the half line therefore we penalize large distances from the half line by calculating the weight with the multiplicative inverse of $dist_{Vert}$. Then, we calculate how much further we get away from the start node in the given direction. This distance $dist_{horiz}$

Algorithm 12: Triangle Algorithm

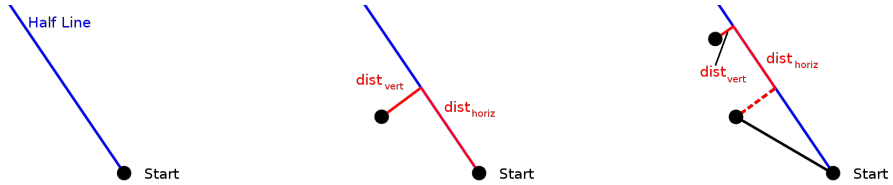
Result: Route with a nice shape, at least a bit optimized

```
Input
start
length
angle = random angle in [30,120] degree
direction = random angle in [0,360] degree
branch1 = createBranch(start, direction, branchlength)
branch2 = createBranch(start, direction + angle, branchlength)
shorten branch1 and branch2 so that
route = branch1 + connection of ends of branch1 and branch2 + branch2
can fulfill length condition
bestRoute = null
routes = connect the last few nodes of the branches and save the resulting
  routes
bestRoute = select best route out of routes
```

is the length of the projection of the line that has the current node as the start point and the neighbor as the end point on the half line. As we want to reward neighbors with a large $dist_{horiz}$ we multiply the weight with $dist_{horiz}$. Because we also want to have attractive routes, we want to allow the branch to use edges that have a high attractiveness, although the corresponding neighbor of that edge has a short $dist_{horiz}$ or a large $dist_{Vert}$. Therefore, we multiply the weight function with the factor a^b where a is the attractiveness of the edge and b is a parameter to adapt how strong we want to weight the attractiveness. In practice we discovered that $b = 1.5$ is a good choice. The weight for a neighbor is thereby:

$$w = \frac{dist_{horiz}}{dist_{vert}} \cdot a^b \tag{2.3}$$

The problem with this method is that most of the time we do not get very nice shapes and have to do much postprocessing to smooth the routes. Often, this is very difficult or not even possible as can be seen in the Examples 2.16.



Initialize the half line

Calculate distances for neighbor of the start node. Select best neighbor.

Continue until branch is long enough.

Figure 2.15: Simple example for the Create Branch Version 1

Algorithm 13: Create Branch Version 1

```
1  Input:
2  start
3  directionAngle
4  length
5  halfLine = halfLine starting at start in directionAngle
6  branch = [start]
7  while Branch smaller length do
8      currentNode = branch.lastNode
9      neighbours = currentNode.neighbours
10     bestNeighbour = null
11     bestWeight = null
12     for neighbour in neighbours do
13         distVert = Distance(neighbour,halfLine)
14         line = line with start currentNode and end neighbour
15         distHoriz = ProjectionOnHalfLine(line).length
16         attr = edge(currentNode,neighbour).attractiveness
17         weight = distHoriz/distVert · attr
18         if weight > bestWeight then
19             bestNeighbour = neighbour
20         end
21     end
22     branch.add(bestNeighbour)
23 end
24 return branch
```

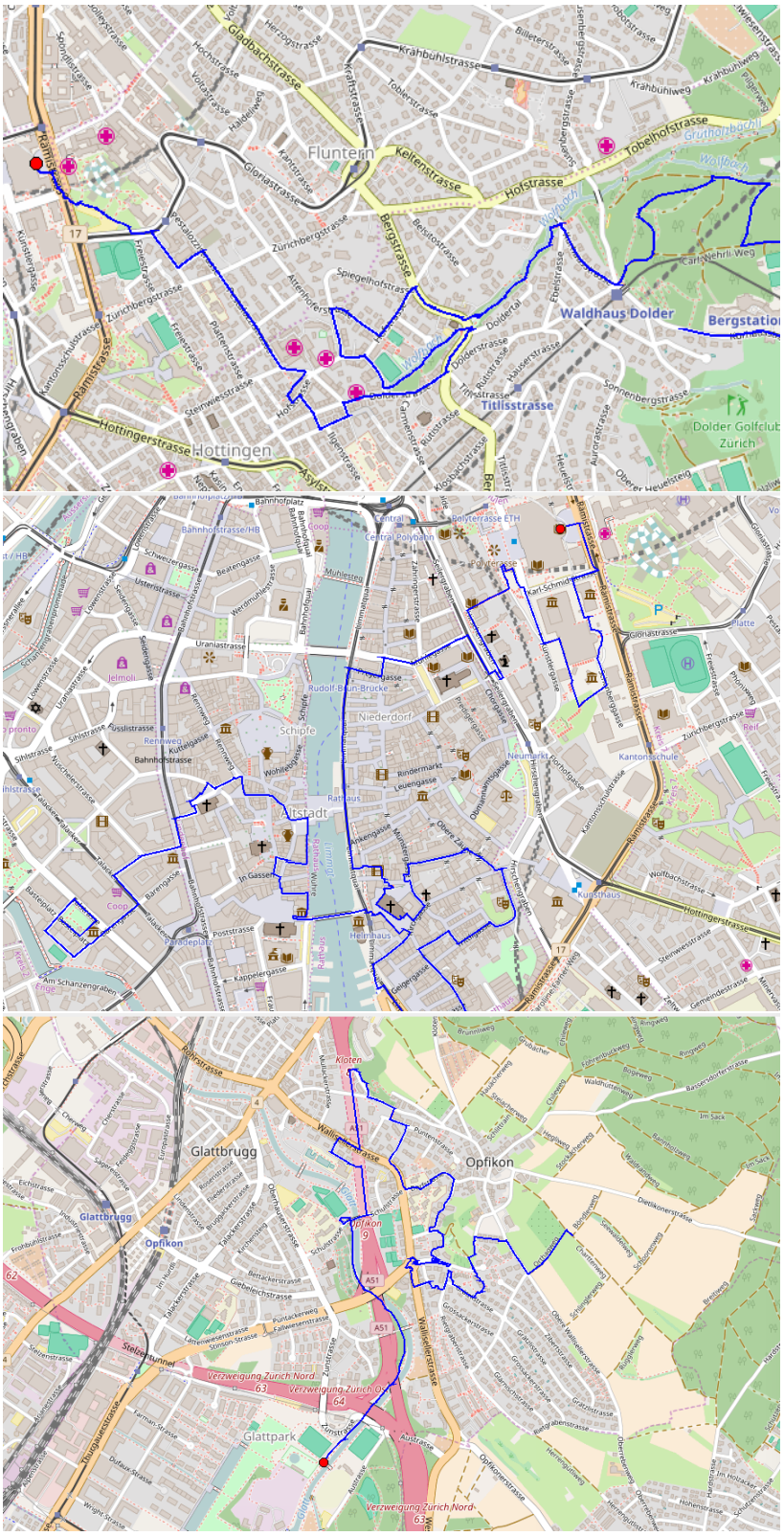
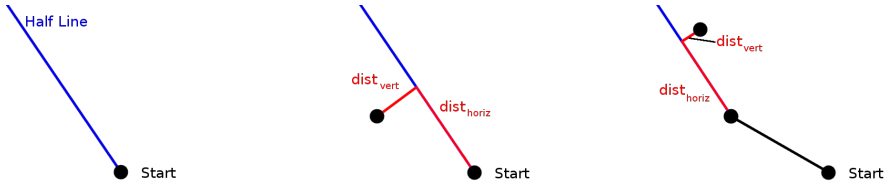


Figure 2.16: Examples for bad branches created with the Create Branch Version 1 Algorithm 13

In the next method, we create each branch by always selecting the same direction starting at the last node of every branch. This means that we do exactly the same as in the Create Branch Version 1 Algorithm 13 except that we create at a half line in every step starting at the last node of the branch. We illustrate this in Figure 2.17. With this method we get much better and sufficiently good shapes of the branches.



Initialize the half line

Calculate distances for neighbor of the start node. Select best neighbor.

Define a new half line and continue until branch is long enough.

Figure 2.17: Simple example for the Create Branch Version 2

The last step is to use one of the algorithms that we introduced above. We can use the Expanding Algorithm 8 or the Random Points Algorithm 10. We can select the node closest to the end point of the line segment that starts at the start point and points into the given direction. Alternatively, we can define a subset of nodes from which we want to choose a random node. If we have selected the end node we can simply use the Expanding Algorithm or the Random Points Algorithm to get a path from the start to the end node.

Algorithm 14: Create Branch Version 2

```
Input:
start
directionAngle
length
branch = [start]
while Branch smaller length do
    currentNode = branch.lastNode
    halfLine = halfLine starting at currentNode in directionAngle
    neighbours = currentNode.neighbours
    bestNeighbour = null
    bestWeight = null
    for neighbour in neighbours do
        distVert = Distance(neighbour,halfLine)
        line = line with start currentNode and end neighbour
        distHoriz = ProjectionOnHalfLine(line).length
        attr = edge(currentNode,neighbour).attractiveness
        weight = distHoriz/distVert · attr
        if weight > bestWeight then
            bestNeighbour = neighbour
        end
    end
    branch.add(bestNeighbour)
end
return branch
```

Algorithm 15: Create Branch Version 3

```
Input:
start
directionAngle
length
targetNode = closest node to length in directionAngle
OR
targetNode = random node in subset, subset defined accordingly
// use algorithm to get branch:
branch = RandPointsAlg(start, targetNode, 1)
OR
branch = ExpandAlg(start, targetNode, 1)
return branch
```

2.12 Dynamic Route Updating

The algorithms above all calculate a nice, but static, route. But what happens if the user leaves the given route - be it either because for example he spontaneously decides to run somewhere else during the run or because he unintentionally ran the wrong way. We also want our application to handle this case. Updating the route not at all is no option as the user then has no use of the application for the rest of his run. Even if he finds back to the route later and follows it to the end, the actual distance he runs is generally different from the desired length. Sending the user back to the given route the way he came is also not desired, as this is most likely not what the user wants. Generating a new route is an option, but we want to try to avoid creating a new route because the user decided to run the route in the beginning and therefore we try to keep the updated route similar to the initial route. So we want to find a path back to the route without using edges that the user already ran. Then we adapt the resulting route so that it fulfills the length condition again.

We define the path the user ran as $R_{finished}$ and the path of the route that the user did not run yet as $R_{to\,go}$. As long as the user stays on the route R it holds that $R = (R_{finished}, R_{to\,go})$. We want to find a path that connects the node the runner is currently running towards with a node in $R_{to\,go}$. To implement this, the first thing we do is we take the edge the user did not run which is the first edge from $R_{to\,go}$, and the edges from $R_{finished}$. We are not allowed to use these edges for the path that shall connect both routes. Next, we connect the current node with all nodes of the $R_{to\,go}$ that are in a reachable radius. The routes we get do not necessarily fulfill the length condition anymore. Therefore, we have to adapt the length of those routes. To do this, we use the Expanding Algorithm 8. If a route is too short, we expand the route like in the Expanding Algorithm. If the route is too long, we use the same principle as in the Expanding Algorithm

but instead of increasing the length we decrease it.

Algorithm 16: Update Route

Result: Updated route

```
Input:
RouteRan
RouteLeft
forbiddenEdge = routeLeft.firstEdge
add forbiddenEdge to list of forbidden edges
add all edges in RouteRan to list of forbidden edges
currentNode = RouteRan.lastNode
targetNode = routeLeft.firstNode
bestRouteLeft = null
while targetNode is close enough to currentNode do
    nextTargetNode = routeLeft.getNextNode(targetNode)
    targetNode = nextTargetNode
    connection = connectNodes(currentNode, targetNode)
    newRouteLeft = connection merged with routeLeft
    while newRouteLeft.length does not fulfill length condition do
        newRouteLeft = adaptLength of newRouteLeft
    end
    if newRouteLeft better than bestRouteLeft then
        bestRouteLeft = newRouteLeft
    end
end
if bestRouteLeft fulfills length condition then
    return bestRouteLeft
end
return new created route
```

Chapter 3

Implementation

In the previous chapter we introduced some algorithms which we can use to create the routes we need. To get a running application we still have to get the data for the algorithms for example about the street networks or the attractiveness. Then we have to develop an application for the user devices. At last we have to set up a server to either supply the application with the data that the application needs to calculate the routes or to supply the application with the routes directly.

3.1 Client Side versus Server Side Computation

For the application and the server we have to make a decision. Do we want to do server side or client side computation. Both methods have advantages and disadvantages. If we run the algorithms on the devices of the users the application works without internet access. This is beneficial for the cases if the user does not have internet access on every part of the route for example in a forest or if the user has no internet access at all for example not only if the user has no mobile internet but also if the users wants to use the application in a foreign country. The disadvantage is that the mobile devices our application runs on usually have much less computational power and might also have few available data storage. In addition the application also works if our server is down for some time.

Server side calculation therefore has the opposite advantages and disadvantages. The application does not work if the user has no internet access. Also the application does not work if the server is down so we have to assure that our server runs stable and has as few and short downtimes as possible. In contrast we can run the algorithms much faster or more often as the server has higher calculation power than the devices of the users. Additionally the application does not need much storage space on the users device.

At first we tried to implement client side computation of the routes. Unfortunately we had some problems with implementing the algorithms fast enough on the android operating systems. Especially the fast reading from the stor-

age into the memory was much slower than on normal Linux operating systems. Therefore we implemented a server side computation of the routes.

3.2 Data

In this part of the chapter we explain how we get our data. Important for the data is that it is not only locally restricted data. Even if those data has a high quality like for example the traffic data of the engineering office of the Kanton Zurich[1] we need data that is globally (or at least nearly globally) available to implement our application. Having many different sources of data for many different areas of the world leads to too much manual data maintenance effort and also to the problem that our application does not work equally well in every part of the world.

Street Network Data

We take the data about the street networks from OpenStreetMap[2]. OpenStreetMap is an open source project that provides map data of the whole world under the Open Database License. The advantages of using OpenStreetMap are that we can access all data freely in contrast to commercial organizations like e.g. Google Maps[3] that do not publish their data. We download the data from one of the mirrors and extract the data we need using the tool osmosis[4]. The raw data we get are so called *ways* and *nodes*.

Those *ways* and *nodes* have tags that explain what those *ways* and *nodes* represent. The *ways* are an ordered sequence of node ids that can represent anything from streets to buildings to country borders. As we are only interested in the street network we select only the *ways* that have the *highway* tag as this is the tag that shows that the according *ways* are streets, paths, motorways etc. Next we only select the nodes that belong to those *ways*. The parameters of the *nodes* that we need are the node id to match the *nodes* to the *ways* and a coordinate. We now get a graph representing our street network by putting the data together. The *ways* usually consist of many nodes to represent the shape of the *ways* correctly.

In the graph for our street network we want to have as few edges and nodes as possible. Therefore we merge as many edges as possible so that we get a graph that is as small as possible like in figure 3.2. Additionally we sort out edges that represent streets that we do not want to use for our routing for example

[1] Tiefbauamt Zurich: www.tba.zh.ch/internet/baudirektion/tba/de/laerm.html
[2] openstreetmap.org: OpenStreetMap[4]
[3] maps.google.com: Google Maps
[4] github.com/openstreetmap/osmosis: Osmosis

```
<node id="453781" version="6" timestamp="2012-11-22T21:53:31Z" uid="334389"
user="ueliw0" changeset="13991903" lat="47.3921507" lon="8.5033225">

<way id="4310913" version="21" timestamp="2014-07-18T05:38:59Z" uid="555807"
user="Internethias" changeset="24213710">
    <nd ref="455590"/>
    <nd ref="2968383388"/>
    <nd ref="249092469"/>
    <tag k="ref" v="1;3"/>
    <tag k="name" v="Pfingstweidstrasse"/>
    <tag k="oneway" v="yes"/>
    <tag k="highway" v="primary"/>
    <tag k="maxspeed" v="50"/>
</way>
```

Figure 3.1: Example of a *node* and a *way* from OpenStreetMap[5].

motorways or private property. We have to keep the data about the shapes of the streets because we need the exact shapes to draw the routes.

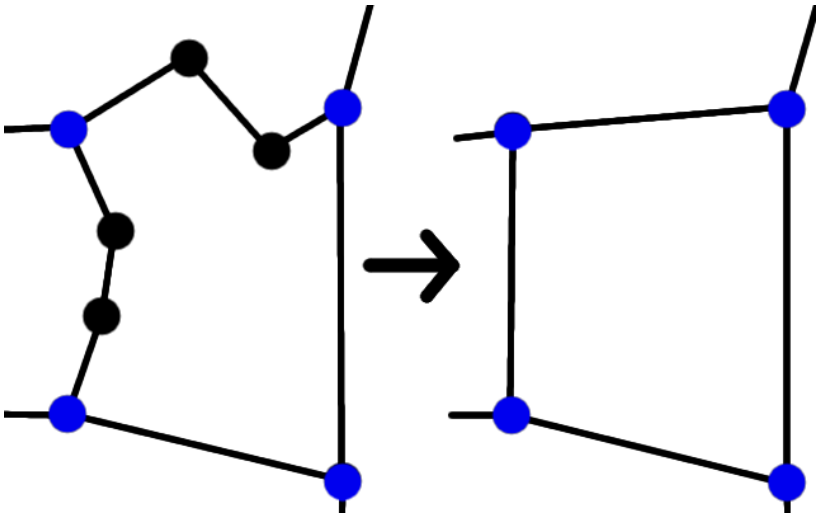


Figure 3.2: We merge as many edges of the graph as possible.

Attractiveness Data

The only thing that is still missing in our street network graph is the attractiveness. The attractiveness represent how enjoyable and how much fun it is to run a certain path. How enjoyable a certain path is depends on many aspects.

tag	weight
highway:primary	0.1
highway:footway	2.0
tracks:steps	0.1
maxspeed:10	3.0
maxspeed:50	1.0
natural:forest	5.0
motor_ vehicle:no	5.0

Figure 3.3: Example of tags of a *way* from OpenStreetMap and how we weight them.

Additionally this is different from person to person. We try to select the most important and general aspects of a path.

Attractiveness of the Path

An important aspect of how nice a path is, is the path itself. For example a track, on which only pedestrians are allowed, is much nicer than a sidewalk next to a main road with many lanes and much traffic. Also, the direct surroundings of the path affect the attractiveness directly. A path surrounded by trees usually is much nicer than a path leading through an industrial area.

We can easily get those data, as our data we take from OpenStreetMap already contains these informations. All *ways* have several tags that describe the type of path that they represent and some information regarding the path. All we have to do is to take the tags that we need, weight them and calculate the attractiveness for the path. We calculate the attractiveness for the path by multiplying the weights that we assigned to all the relevant tags.

The Topography

Another aspect that influences the attractiveness of a route is the topography. We can divide this aspect in two subaspects, first the elevation of the path and second the view. As OpenStreetMap only provides very few topographic data we had to take the topographic data from another source. We take our data from the Shuttle Radar Topography Mission (SRTM) [5]. At first we wanted to work with the data itself but we found a much simpler solution with the osmosis-srtm-plugin[6] for osmosis[7] which is an easy tool to download the SRTM

[6] github.com/locked-fg/osmosis-srtm-plugin: Osmosis SRTM Plugin
[7] github.com/openstreetmap/osmosis: Osmosis

Elevation	$w_{elevation}$
>5%	0.75
>10%	0.5
>20%	0.1

Figure 3.4: Example of how we penalize elevation.

data.

The steeper the path is the more exhaustive it is to use the path. Thereby it does not matter much if we want to go uphill or downhill. Therefore paths that are too steep should suffer a penalty. For this we introduce an additional weight $w_{elevation}$ that decreases if the slope is too high.

The view is another parameter that influences the attractiveness of a path. The better the view the more attractive the path is. To get a measure for the view we model the view by looking at the topographic data in eight directions. We show an example in figure 3.5 If we look in one direction we look how far we can go until we find a point that has a higher height than our current location. We count the number of points in each direction $n_{Direction}$. The larger $n_{Direction}$ is, the further this point is away and the better our view is. The better the view in the more directions the better is the overall view. The better the overall view the more attractive is the path. Thus we define an additional weight based on the view w_{view}.

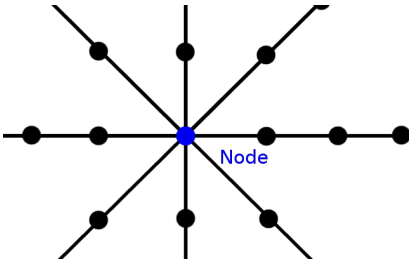
$$w_{view} = 1 + \sum_{D \in Dircetions} (1 - \frac{1}{1 + n_D}) \tag{3.1}$$

Thus if we have no view $w_{view} = 1$. The better the view gets, the larger gets w_{view}.

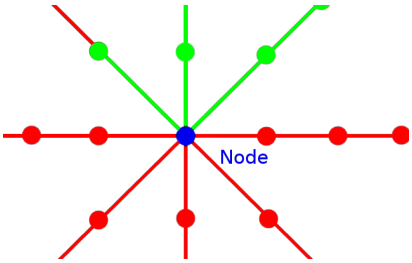
Merging the Weights

We have now the attractiveness of each path and for each path additional weights. The easiest way to merge those is to multiply the attractiveness with the weights. As the traffic, the elevation and the view is not equally important for every user we show an easy method to personalize those weights. The user can rate the three parameters traffic **t**, elevation **e** and view **v**. The user can set each parameter to **not important** = 0, **normal** = 1 and **very important** = 2. We can calculate the adapted attractiveness of a path by:

$$a_{personalized} = a \cdot w^t_{traffic} \cdot w^e_{elevation} \cdot w^v_{view} \tag{3.2}$$



We check how far the view from a certain node is in eight directions.



In this example we have a view in three directions represented by the green edges.

Figure 3.5: Example of how to determine the view.

3.3 Android Application

We developed an appliaction called **Smart Route** and have published it in the Google Play Store [8]. As we use the data of OpenStreetMap for our routing we also use OpenStreetMap to display those routes. We use externally rendered map tiles that are provided by Mapnik[9] to display the map. For this we use the libraries osmdroid[10] and the osmbonuspack[11].

The graphical user interface provides all necessary tools for the applications purpose. The application keeps track of the users movement and displays it on the map. The user can choose that the application follows the users position on the map automatically and it is also possible to rotate the map in the direction the phone is held/the user is watching.

For the creation of a route the user selects a start on the map, chooses the desired length and then requests a route. This request is then send to the server that responds with a route. This route is than displayed for the user.

Currently the application itself does not do any of the steps of the route calculation itself. Yet it is possible to run our routing algorithms also on the phone but the routing process is significantly slower than on more powerful computers/servers. For more details see the chapter 4.

[8] play.google.com/store/apps/details?id=js.myroute: Smart Route
[9] mapnik.org: Mapnik
[10] github.com/osmdroid/osmdroid: Osmdroid
[11] github.com/MKergall/osmbonuspack: Osmbonuspack

3.4 Server

For our server we use a simple tomcat[12] implementation. The server itself has two purposes, hosting the preprocessed data and generating routes on demand. If a user demands a route with the application the server receives a request with the necessary parameters. With a simple Java web applet the server generates a route using the Triangle Algorithm introduced in section 2.11 in chapter 2.

If in the future the application will get the function that it can calculate the routes locally only few lines of code are necessary to change the function of the server so that the server provides the preprocessed data for the application to download. The route generation can then be either turned off completely or kept for slow devices, very large routes or saving mobile data volume if the application does not have the data of the area stored internally and would have to download it.

Currently we have preprocessed the whole data for Switzerland thus our application can be used everywhere in Switzerland. But theoretically we can generate routes in every region of the world as long as OpenStreetMap has stored enough data for the specific region. We just need to preprocess the data of the desired region and put it on our server.

[12] tomcat.apache.org: Apache Tomcat[6]

Chapter 4

Evaluation

4.1 Performance Analysis

In the following we will analyze the performance of the algorithms which we introduced in chapter 2.

The Brute Force Algorithm

At first we want to take a look at the Brute Force Algorithm 2. How good does the algorithm perform? We use a personal computer running Ubuntu (800MHz, 8 CPUs) for these measurements. We get the average times in Figure 4.1 for the computation of a routes with several different length parameters. We use the same start and end points for these measurements. Then, we examine if it makes a difference whether the start and end point are close to each other or not. We show the corresponding graph in figure 4.2. As we can see there is merely a difference whether we use the same start and end point or not.

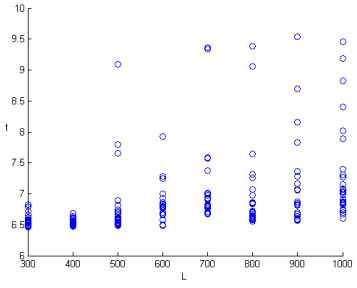


Figure 4.1: We plot the length on the x axis versus the time on the y axis for the Brute Force Algorithm2. We use the same start and end point.

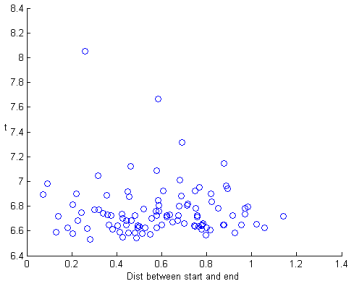


Figure 4.2: A plot of the distance between start and end point on the x axis and the time on the y axis. Using a fixed distances of TBC

4.2 Quality Analysis

Quality of Data

Unfortunately the data we use are not perfect. OpenStreetMap is an open source project that is build mostly on the work of voluntary helpers. As the data is therefore not controlled by a company that has many resources to validate and check the correctness of the data, it takes time to find and correct man made errors in the data.

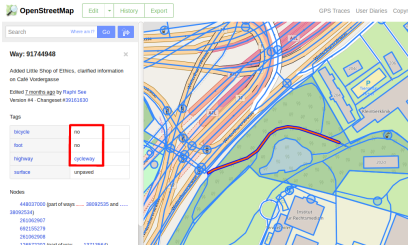


Figure 4.3: Example of errors in the data...tbc Example of errors, e.g. irchel radweg

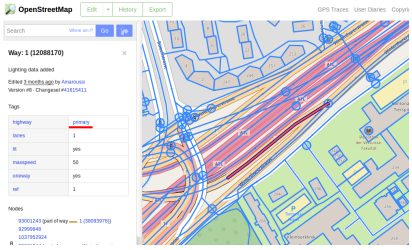


Figure 4.4: Example of errors in the data...tbc Example of errors, e.g. irchel radweg

Especially non-consistent tagging of roads is a large source of error. Still the data is good enough to show that our algorithms work and to create routes that are useful not only in theory but in real life.

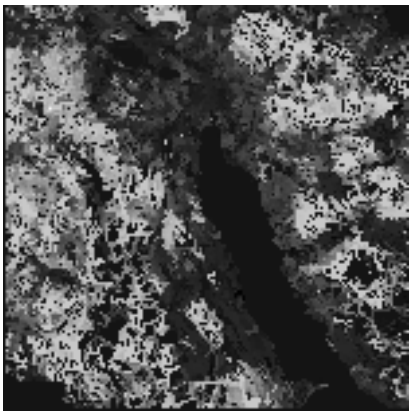
In contrast the height data we use from SRTM[5] has a very high quality and is also very accurate as the accuracy is 3-arc-seconds what corresponds to about 90 meters depending where on earth we exactly are.

Quality of Attractiveness

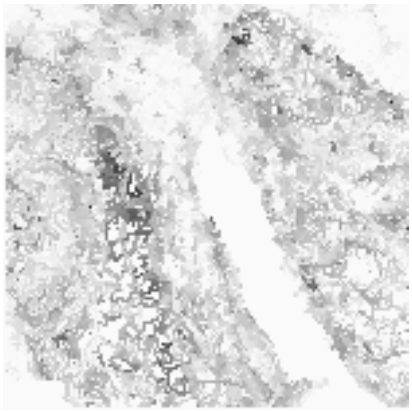
In Section 3.2 we have defined various ways to get a measure for the attractiveness of routes. We have to check if our measure does not contradict the real world. To check this, we have created a map that represents the attractiveness of the region of Zurich in Figure 4.5. For every pixel we have calculated an attractiveness by averaging the attractiveness of the paths that lie in the pixel. Looking at Figure 4.5 we can see that there are no areas that in the real world are completely different from the information in the attractiveness maps As this is only a very coarse check whether our attractiveness data is good or not, we cannot state that our data are very good but they seem usable and at least decent.

[
caption]Normal Map of Zurich, taken from Google Maps[a]

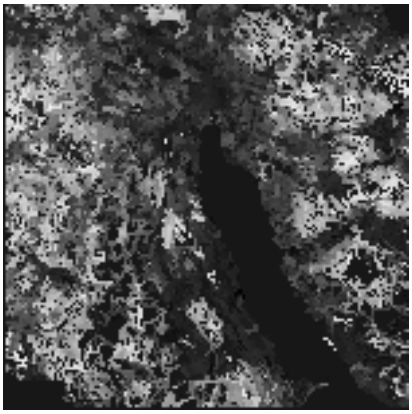
[a]maps.google.com: Google Maps



Corresponding attractiveness of the roads and paths in the area.



Corresponding attractiveness of the elevation weights in the area.



Corresponding attractiveness of both roads and elevation weights.

Figure 4.5: In these figures we show how attractive the areas in and around Zurich are according to our weights. The brighter the pixels are the more attractive are the paths in these area on average.

4.3 Summary

From the algorithms we introduced in section ?? the Triangle Algorithm 12 is the best choice for our demands. It is fast even for large routes and the drawback that it does not return good routes for short distances does not matter as our goal is to create longer routes than routes that are only one or two kilometers long. Therefore, we have implemented the application with the Triangle Algorithm.

As mentioned in section 3.1 we first tried to implement the application with client side computation. As we were not able to accomplish this in some time, we decided not to waste more time on this, and instead implement a server side computation. Nonetheless, we believe that the advantage of the client side computation that no internet access is required, once the data of the area was downloaded, like mentioned in 3.1, justifies to give it another try. Theoretically, depending on the size of the data it should be possible to accomplish a much faster reading of the data on Android.

4.4 User Feedback of the Application

We have run a closed alpha test with the members of our group. After that we have published the application called **Smart Route** in the Google Play Store [1]. As we did not have the time to advertise the application we do not have (at least we believe so) any users, that were not asked by us to test the application, yet.

[1] play.google.com/store/apps/details?id=js.myroute: Smart Route

Chapter 5

Conclusion

In this thesis we have introduced a model for the attractiveness which allows us to create attractive routes. We have introduced several different algorithms. We have seen that the computation of an optimal and therefore best route is only feasible for very short distances. For longer distances we have to trade off the optimality for faster computation of the routes. But this is not as severe as it sounds as our non-optimal algorithms create sufficiently attractive routes.

We have developed an Android Application that implements the Triangle Algorithm. This application provides the user with routes at any given start point in Switzerland. The application can theoretically be used in any area of the world as long as there is enough data available in OpenStreetMap for this area. We just have to preprocess the data and add it to the server.

The Quality of our data is good. Even though there exist wrong data as shown in Section 4.2 most of the time this does not affect our routes. In the few cases, in which we get a bad route or a route with a section that cannot be used by pedestrians or bikers, the user just can demand a new route quick and easy.

5.1 Future Work

Model of the Attractiveness

We could choose a different version of the model for the attractiveness. Maybe it might be possible to create a model of the attractiveness by using Fuzzy Logic (see for example [7]) and by defining three sets of edges - good, bad and neutral edges like in Section 2.4.1 - but not as a normal set but as Fuzzy Sets. Then the edges are not strictly good or bad but instead have a higher or lower affiliation to one set. We could also choose another maybe better cost function if we study how this can improve our routing.

Algorithms

The algorithms can be improved further. For the Expanding Algorithm we can study how to initialize the algorithm best to get the best possible locally optimal route.

In the Random Points Algorithm we can study how to select a good subset of nodes to choose our random points from so that we get the best possible results for our routes. For example, an algorithm can be implemented that rewards routes that move away from the start point fast and do not approach the end point until the route is long enough. This can be easily based on the Brute Force Algorithm, for example.

Also an algorithm might be able to build a route with a nice shape without using such a trick as in the Triangle Algorithm 12. This might be accomplished for example by penalizing every angle between two edges, the larger the angle the larger the penalty. Also, if two edges are not close to each other in regard to the route, we can penalize them if they are too close in regards of their real distance.

We also can try to develop completely new algorithms for example an evolutionary (see for example [8]) algorithm. In fact we already have everything we need to sketch a simple evolutionary algorithm which we do in Algorithm 17. If we want to combine routes this might not be possible with every routes. But as all routes have the same start and end point it is very likely that many the routes have some identical nodes so that we can recombine some parts of these routes easily. We could also try to define some more complex recombination method that allows to combine any two routes. If we want to evolve a route we can do this already. The Expanding Algorithm allows us to grow the route and if we redefine it also to reduce the size of the route. If this is not enough, we can define some other algorithms to evolve a route for example by replacing a part of the route with one of our algorithms.

Android Application

As already mentioned above, one task to do is to implement client side computation at least for medium routes. Additionally we did not yet accomplish to implement the automatic route update so that it runs completely stable.

The Data

Our data can also be improved. We could improve our graph, that represents the street network. As this graph in some areas still has a very large density of nodes per area, we can decrease the size of the graph by sorting out unnecessary edges for example edges that we really do not want to use for our routing because they

Algorithm 17: Sketch of an evolutionary algorithm to generate routes

```
1  Routes = get some initial routes, e.g. random Points Alg or Triangle Alg;
2  while time is not larger than predefined time do
3      for some actions do
4          routesToRecombine = take best + some random routes from
           Routes;
5          combinedRoutes = recombine(some routes in routesToRecombine);
6          for route in combinedRoutes do
7              if RandomBool then
8                  route = evolve(route);
9              end
10             newRoutes.add(route);
11         end
12         Routes = select best and some random routes from newRoutes;
13     end
14 end
```

have such a low attractiveness. We have to take care though that we can only do this in parts of the graph that are connected good enough so that routing will be still possible without those edges without any problems. We also can ask our users if we can get anonymized data about the routes they actually run. With this data, we can improve the street network graph, e.g. finding new edges that we do not have in the graph yet due to faulty data or deleting edges that cannot be used by pedestrians and that is only in our graph because of faulty data. We also can adapt our measure of attractiveness according to the real running behavior of the users. The attractiveness can also be improved further by adding more influences of the surroundings of an edge. For example, information about positive surroundings, like nature, a lake, forests but also negative surroundings like close large streets with much traffic, that is not directly given for the edges can be added. This can for example be done by calculating the attractiveness of an area by taking all these things into account. Then, the edges in these areas are adapted accordingly. We did some experiments with calculating such an area score for traffic density but unfortunately did not have the time to implement it.

Bibliography

[1] Linggi, T.: On the fly! automatic running route generation. online at disco.ethz.ch/theses.html (3 2016)

[2] Bentley, P.J., Wakefield, J.P. In: Finding Acceptable Solutions in the Pareto-Optimal Range using Multiobjective Genetic Algorithms. Springer London, London (1998) 231–240

[3] Goldberg, D.E., Holland, J.H.: Genetic algorithms and machine learning. Machine Learning **3**(2) (1988) 95–99

[4] OpenStreetMap: Openstreetmap (2016)

[5] Rabus, B., Eineder, M., Roth, A., Bamler, R.: The shuttle radar topography mission a new class of digital elevation models acquired by spaceborne radar. {ISPRS} Journal of Photogrammetry and Remote Sensing **57**(4) (2003) 241 – 262

[6] Apache: Apache tomcat (2016)

[7] Zadeh, L.: Fuzzy sets. Information and Control **8**(3) (1965) 338 – 353

[8] Bäck, T., Fogel, D., Michalewicz, Z., eds.: Evolutionary Computation 1: Basic Algorithms and Operators. Institute of Physics Publishing, Bristol (2000)