Internal Report 2012-2013-11 July 2013

Universiteit Leiden
Opleiding Informatica

Multi-objective Generation of Bicycle Routes

Bart Hijmans

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

There are a large number of bicycle route planners available that only output a single route. This paper explores the possibilities of making a routing algorithm that uses multiple objectives to produce several different routes that are objectively incomparable in quality, but each has their own relative merit. It looks at available data, possible objectives and finally tries to combine that into an algorithm to produce these more varied sets of routes. The challenges are to find realistic formulations of objective functions and to design new and modify existing route planning algorithms for multi-objective route planning with real-time constraints.

1 Introduction

The average person in the Netherlands owns 1.1 bicycles[14]. This is more than in any other country. Using those bikes Dutch people ride an average of 900 kilometers every year. Planning where exactly to go and how to get there is not a new idea. However most existing route planners focused only on the shortest routes. We do not always care about the shortest routes. Often we would prefer a slightly longer route if it avoids traffic and runs through a nice forest instead.

Throughout the country runs a network of well-marked bicycle paths, called the fietsknooppuntennetwerk. This bicycle node network (as literally translated) is marked with numbers from 00 to 99 and those numbers repeat. They are placed in such a way that there is never an intersection where the same number is in two different directions. These bicycle nodes are great because they make it easier for cyclists to find routes that a route planner gives them because there are signposts at each of these nodes with these numbers and directions to adjacent nodes. They also make the algorithm not care about the many thousands of streets through cities and villages that are very similar and not very interesting.

There are a number of route planners available. They range from Google Maps[4] to Falk[2] that only calculate the shortest routes, either over the bicycle node network or over all roads. The only exception is the route planner of the Fietsersbond[3]. It has a number of different route generating options, including "nature route" and "car sparse routes." How they work exactly is not public knowledge, but they seem to do something different and it is certainly an improvement over the other route planners.

What this project is about is finding multiple routes that the user can then choose between, based on multiple objectives, instead of one function with one specific deterministic solution that may not be the best option for what the user wants. The goal is to explore the possibilities for making a route planner capable of suggesting multiple good routes and combine that into a multi-objective algorithm. The research questions are as follows. What kind of data is available that can be used in bicycle route planning? How can that data be used to create meaningful objectives? And what kind of algorithm can be used to output multiple possible routes based on these objectives?

Chapter 2 will focus on the exploration of available data and how that data had to be processed. Chapter 3 will focus on the objective functions. The results of those two will be combined into an algorithm in Chapter 4. That algorithm is subsequently tested in Chapter 5. Finally Chapter 6 will focus on conclusions and future work that can be done to make it better.

2 Geographical Data Resources

An algorithm that generates routes needs geographical data to do anything useful. Though fabricated data was briefly considered, it just is not anywhere near as good as real-world data. There are a number of different mapping service applications available on the internet. They can be separated into commercial and open source maps, which will be discussed in Sections 2.1 and 2.2 respectively. Section 2.3 will talk about the format that the data is in and how that was converted, and Section 2.4 is about how the data had to be cleaned to avoid inconsistencies.

2.1 Commercial maps

Commercial mapping software include the aforementioned Google Maps, Bing Maps[1], Yahoo Maps[9] and several others. They all offer a limited number of queries for free and have a shortest path routing service for cars and sometimes for pedestrians and cyclists as well. However, none of these have the Dutch bicycle node network data that we want to use. So in order to get data on the distances between bicycle nodes, we need to gather the bicycle nodes and their GPS coordinates from another source.

When we get that data we will run into more problems. We still do not know which nodes connect to which other nodes. It would be easy to say that two nodes a and b are adjacent if there is no node c such that $d(a, b) = d(a, c) + d(b, c)$. Unfortunately there are often different ways to get from one node in the network to another and it is likely that in some cases the shortest route (which is the only thing these commercial maps calculate) between connected nodes runs at least partially over roads that are not part of the network. These problems are purely academic though, until we actually find data.

Unfortunately the data for these nodes is not publicly available from official sources. According to Stichting Landelijk Fietsplatform only about half of the fifty subnetworks let this data be freely available[15]. Those gaps would make for very odd and inaccurate routing. The only option seems to be entering the data manually, based on observations made while standing next to a sign with a gps-enabled device or from aerial photographs. This is where OpenStreetMap (OSM) comes in.

2.2 OpenStreetMap

Fortunately, data for the bicycle node network has already been entered manually into the collaboratively edited, open-source, OpenStreetMap[7]. Not only does OSM have all the bicycle nodes and their GPS coordinates, it also has data on the connections between

pairs of nodes, with a large number of potentially useful tags. It is also possible to calculate the distance over land over the network connections between pairs of connected nodes. This is more than we could have hoped to get from the commercial maps and there does not seem to be any reason to go back to them for now.

2.3 Data Formats

All the data in OSM is divided into three core elements. The first of which is the *node*. A node always has a unique id and a longitude and a latitude value and represents a point on the map. The next core element is the *way*. They have a unique id as well and consist of a list of nodes representing (segments of) roads. Nodes and ways together are enough information to draw the map, but for classification and ease of access there are also *relations*. A relation is an ordered list of nodes, ways and other relations. Each item in a relation may also have a role within the relation. All three of the core elements may have tags, but relations must have at least one to indicate what kind of relation it is.

OpenStreetMap has a number of different API's. An easy to use one that is solely for retrieving data is Overpass[8]. Overpass uses an XML-based query language that allows the user to easily access data. Since this API provided all the data that was needed for this project, no other APIs were considered, though some may work just as well.

The Dutch bicycle node network in OSM is found in sixty-five relations that each have data for one region. These regions seem to be different from the fifty regions mentioned before. The relations contain nodes that are the bicycle nodes in that region and relations that represent the connections from a bicycle node in that region to another node in the same or a neighboring region. It also has tags identifying it as a network of type *rcn* (regional cycling network) as well as for the province and country. Using these tags and recursion in Overpass, it is possible to retrieve all relevant data for the entire network.

The data for the relations that represent regions can be retrieved using the following query.

```
1 <query type="relation">
2   <has-kv k="type" v="network"/>
3   <has-kv k="network" v="rcn"/>
4   <has-kv k="addr:country" v="NL"/>
5 </query>
6
7 <print/>
```

We can subsequently find the other types of items we need by adding recurse calls on line 6, which has been left blank for clarity. The first item we need are the bicycle nodes. To get the data for these, the call we want to add is `<recurse type="relation-node"/>`, which gives us the data for all the nodes in each of the region relations. One of these nodes looks as follows.

```
<node id="336199688" lat="52.2294812" lon="4.4457029">
  <tag k="rcn_ref" v="20"/>
```

```
</node>
```

The information in this item is the unique node id, its latitude and longitude and the rcn_ref. The latter being the reference number that is on the signs as well. Many nodes have other key-value pairs that identify a source, a name, a specific object or building at that node's location or often one or more redundant tags identifying it as a part of the regional cycling network.

Secondly, we need to get the relations that represent connections between those nodes. They can be acquired by adding `<recurse type="relation-relation"/>` to line 6 of the original query instead. An example of such a relation is listed below.

```
<relation id="571157">
  <member type="way" ref="30471337" role=""/>
  <member type="way" ref="30120191" role=""/>
  <member type="way" ref="30471239" role=""/>
  <tag k="network" v="rcn"/>
  <tag k="note" v="20-64"/>
  <tag k="route" v="bicycle"/>
  <tag k="type" v="route"/>
</relation>
```

As you can see the relation has a unique id and consists of three ways. It does not include any nodes as those can be retrieved from the ways. There are relations that do include nodes, but that is a deprecated system that has not been entirely phased out. The note key in this case lists the rcn_refs of the nodes it connects. However, because the notes are not used in drawing the map, they are much more likely to be inaccurate than the ways. Combined with the fact that the reference numbers are not unique, getting information from the ways is more reliable. The network, route and type keys tell us that this relation identifies a route for bicycles in the regional cycling network, which we already knew.

Thirdly, we need data for the ways. In order to get data for those we keep `<recurse type="relation-relation"/>` which got us the relations that represent connections between nodes and follow that line with `<recurse type="relation-way"/>` to get the ways in those relations.

```
<way id="30471337">
  <nd ref="45798403"/>
  <nd ref="336199680"/>
  <nd ref="336199682"/>
  <nd ref="336199684"/>
  <nd ref="336199687"/>
  <nd ref="336199688"/>
  <tag k="highway" v="cycleway"/>
</way>
```

This way is an ordered list of nodes with a unique id. There are a few things to note here. Firstly the highway key is used in OSM to determine the way a road is drawn on the

map. In this case it is a cycleway, which means it will be drawn thin and white, where a motorway would be drawn much thicker and blue in OSM's default view. A second thing to note is that the way listed here is the first way in the relation listed above and the last node in the way is the one used as an example as well. What this indicates is that even though the relation is an ordered list of ways and the way is an ordered list of nodes, those orders do not match in this case. Thirdly, it is important to note that ways with bends contain a lot more nodes than this one, to provide good approximations of how the road actually runs.

Lastly we need all the nodes in the ways as well, so we can calculate the distances over the road rather than just as the crow flies. For that, we need `<recurse type="relation-relation"/>` , `<recurse type="relation-way"/>` and `<recurse type="way-node"/>`, which gives us the nodes in the ways in the relations in the relations that represent the regions. These will also include the bicycle nodes again and the other nodes look just like the rcn nodes except that most have no tags at all.

2.4 Data Cleaning

Now that we have all this data the first thing we need to do is to find out exactly which nodes connect to which other nodes. The simplest way to do that would be to take the first node of the first way and the last node in the last way of a relation and check them against the rcn-nodes. However, as we noted before, the orders of the ordered lists do not always match. So we need to look at the first and last nodes of the first and last ways in a relation to see which of them are the bicycle network nodes. And because there is a possibility of a relation having only one way, we need to make sure we check the first node of the first way in a relation first, and the last node of the last way first, so we are guaranteed to get both rcn-nodes from that single way.

This is where the troubles start. It turns out that 800 out of 10761 relations do not have an rcn-node as the start or endpoint of either the first or last way. This is clearly a mistake and likely a consequence of the open source nature of OSM. Before we give up all hope though, we can try to find out exactly what is wrong. A short investigation of some of the relations reveals that there are numerous different problems. One relation ended on a roundabout that was not even on a part of the cycling network. Another ended on an rcn-node that was not listed in the regional subnetwork and after comparing it to a different map of the network for that region, turns out to have been mislabeled with the wrong reference number as well. These mistakes should, and may already, be fixed in the OSM database. The data used in this project is, at the time of writing, over four months old.

Some of the problems can be fixed however. A significant portion of the problematic relations end very close to an rcn-node. Connecting a relation to a node based on this proximity risks creating more errors, but when the distance is small enough, that risk is minimal. The average distance between two adjacent rcn-nodes is about 2.8 kilometers, so a cutoff point of half a kilometer seems fairly safe. After doing this through a semi-automated process, the last step was to remove all relations that connect a node with itself. In the end 455 relations had to be removed and as a consequence 212 nodes out of

7544 were not connected to anything and were removed as well to avoid strange behavior in the algorithm if they were used as the start or end nodes of a route.

An error rate of 7.4% in the relations leaves one to wonder what else could be wrong. There could be any number of nodes, ways and relations with significant errors, but that is not important. We now have data that at least resembles real world data and because OSM is open source, it might be that someone is correcting it as you are reading this. So there is hope for updates and improvements.

3 Objective Functions

The general goal of this project is to generate "nice" routes. Unfortunately, niceness is not a measurable thing. The question is what exactly it is that a cyclist looks for in a route? That is where the objective function modeling comes in. This chapter discusses a number of possible objectives, how they were formulated and why they were eventually chosen or discarded. We will start with the shortest path in section 3.1, followed by scenic beauty in 3.2. Thirdly we will discuss an objective called cycleway in 3.3 and finally we will look at the impact of wind direction in 3.4. There are also a number of possible constraints and objectives that were discarded for being incompatible with the algorithm. Discussing those outside the context of the algorithm makes no sense so they will be discussed briefly in Section 4.5 instead.

3.1 Shortest Distance

The main reason why shortest distance is important is that it adds a degree of reality. It may be a very interesting result if the nicest route from here to the next town over visits every city in the country before reaching its destination, but in the end a user will want to get to his destination before the end of the day as well.

In order to add the shortest path as an objective we need to calculate the distance between every node and all its neighbors. That should be the sum of the distances between every pair of consecutive nodes in every way in the relation that describes that connection. In order to take into account the curvature of the earth. the distances can be calculated using the haversine formula which reads as follows.

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Here ϕ represents the latitudes and λ the longitudes of the two points and r is the radius of the earth. This provides us with the distance over the surface of the sphere. This is important because at different fixed latitudes, a fixed difference in longitude represents a different distance. We assume the country is completely flat (besides the curvature of the earth) partly because it very nearly is, so any errors will be minor, and partly because the data for this is not available.

Now that we have this data it is easy to define the objective. The distance of a certain route is the sum of the distances of all the connections used, and we want to, at all times, minimize this distance.

3.2 Scenic Beauty

Outdoors, away from the noise of the cities and in the tranquility of nature is where we find cyclists on a nice day. And who could blame them? Scenic beauty is clearly something people are looking for in a bicycle route. So we should aim to provide it.

The problems start when trying to quantify the scenic beauty of a place as a number. One way it could fall apart is when we compare a pine forest and a field of flowers. If we decide fields of flowers are nicer, what happens in the fall when the flowers are gone and the forest is standing strong?

It gets worse when one person prefers heath and another prefers forests. It might be too hot in the sun for one and too cold in the shade for someone else. In order to make this less subjective we would need to introduce a large number of different criteria based on types of landscape, ranging from fields to forests. this would allow the user to pick which criteria he cares more or less about. And we have not even considered architectural beauty yet.

All of this is hypothetical though, because a database with this information does not actually exist — at least not anywhere freely accessible. And though it might be constructed manually or by image analysis algorithms, that is far from the scope of this project. It is unfortunate, but it seems that scenic beauty cannot be considered with the available data.

3.3 Cycleway

Given that we cannot model scenic beauty we would like to formulate another objective that represents the niceness of a route. We do not have to look very far because OSM provides us with a large number of possible tags. They range from illumination to width and from restricted vehicle types to street names. There is even a tag for different types of nature, but it is used for only one way and therefore not useful at all. And that seems to be a trend. The vast majority of tags are either woefully incomplete or otherwise too rare to use.

There is one tag that stands out and that is the *highway* tag. Out of 111350 ways only 128 do not have this tag, making it by far the most complete tag available. The highway tag identifies the kind of road a way represents and its importance in the network[6]. It has a number of possible values ranging from motorway to residential and, most relevant here, *cycleway*. Of the ways on the network 31292 are marked as cycleway and this value means a road that is mainly or exclusively for bicycles. Pure cycleways in cities or villages are rare. There are often bicycle paths next to roads, but they would be classified as the type of road they are next to and not as cycleway. Though there are some true cycleways in cities, they are few and far between. Most cycleways are outside the cities, through nature, and paved in crushed seashells, and those are exactly the type of places we were trying to go.

Now we need to convert this very binary data into something that can actually be used. We could consider calculating the percentage of a route that is marked as cycleway. This

seems simple enough, but the problem is it does not scale well. The other criterion we have so far is distance and if we look at some possible values, we could have a route with a distance of ten kilometers and fifty per cent cycleway or a route with the same percentage marked as cycleway that is ten times as long. When using a linear weighting scheme, the balance of the two criteria radically changes over the course of generating a route, and exactly that balance is what we will be looking for. Problems also occur because the value of the attribute can both increase and decrease which breaks the algorithm. More on that in Chapter 4.

So we need a new idea. A way to make the cycleway attribute more predictable and more in line with the distance objective. Instead of using percentages, it is possible to use the absolute value instead. So we can change the attribute to the sum of the length of all the cycleways on the route in kilometers. Now the attribute will scale well with distance and it is nondecreasing. The only problem is now that making routes longer will make this attribute better — because we are trying to maximize it — and the best value will be for the route that manages to cover as many cycleways as it can find. However this is easy to solve by instead using the distance that is not covered by cycleways and minimize that. Now adding nodes to the route will only not make it worse if the new road segments are entirely cycleways.

Using this method it is possible for extremely long routes to be considered good, as long as they run over cycleways. This is not a problem, because those solutions can be discarded at the end, based on a maximum length defined by the user in absolute value or a percentage over the shortest path.

So the final definition for the cycleway attribute is the sum of the length of all route-segments that are not cycleways.

3.4 Wind Direction

Cycling against the wind is a lot more work than cycling with the wind. It would be nice if we could plan routes to avoid that. However, a preliminary and hasty implementation of this objective yielded no interesting results. Routes were barely, if at all, changed - even with a heavy focus on this objective over others. This begged the question, is it ever worth taking a longer route to avoid going against the wind? Before we get there though we need to think about the physics and how to synthesize that into an attribute.

We will need to assume a constant wind speed. A change in wind speed over the course of a few hours is hard enough for weather services to predict and taking that into account is not realistic. For the same reasons we will assume the wind direction is constant as well. We will also, for now, ignore the effects of friction. Friction is proportional to the length of the road and constant factors like the specific bicycle used. So the shortest path objective is already a good measure for friction.

In physics the wind affects the amount of work needed to move in a certain direction based on the component of the angle parallel to the direction traveled. There is no extra work needed when the wind is exactly perpendicular to the direction you are traveling and the maximum extra work is needed when the wind is in your face. With wind in

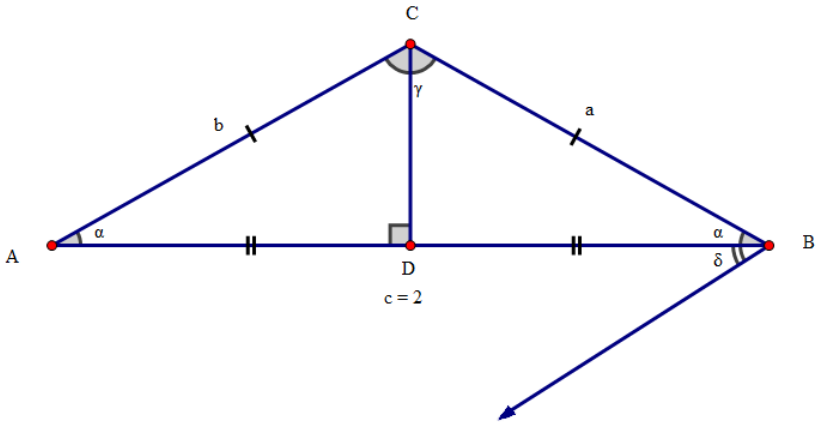

Figure 3.1

the back you will need less work to get to where you are going. The difference in work is proportional to the cosine of the angle between the direction you are riding and the wind. That angle is zero when the wind is in your face. We also need to multiply that by the length of the road segment because, clearly, going against the wind for a few meters is not as bad as going against the wind for a couple of kilometers. In physics this work is defined as a product of the distance and the force, so we have the following formula for the difference in work of a particular road segment:

$$cos(\delta) \cdot l$$

where δ is the angle of that road segment and the wind and l the length of the road segment. Let us now consider the situation in Figure 3.1

There we are trying to get from A to B and the angle δ represents the angle of the wind with the line c. So the work for the road segment c must be

$$cos(\delta) \cdot 2$$

Now let us consider the route through point C. Since the line from C to D divides the triangle into two right-angled triangles we can easily use trigonometry to calculate the length of a and b as a function of α, which turns out to be

$$b = \frac{\frac{c}{2}}{cos(\alpha)} = \frac{1}{cos(\alpha)}$$

and because of symmetry this is the same for a. The angles with the wind of the two route segments are $\delta - \alpha$ and $\delta + \alpha$ respectively. And because we take the cosine of these angles we do not have to worry about overflow. So, the formula for the extra work of $a + b$ is

$$cos(\alpha + \delta) \cdot \frac{1}{cos(\alpha)} + cos(\alpha - \delta) \cdot \frac{1}{cos(\alpha)}$$

This can be reduced using the following rule[16, Page 63]

$$cos(\alpha \pm \beta) = cos(\alpha)cos(\beta) \mp sin(\alpha)\sin(\beta)$$

And this works out as follows:

$$cos(\alpha + \delta) \cdot \frac{1}{cos(\alpha)} + cos(\alpha - \delta) \cdot \frac{1}{cos(\alpha)}$$

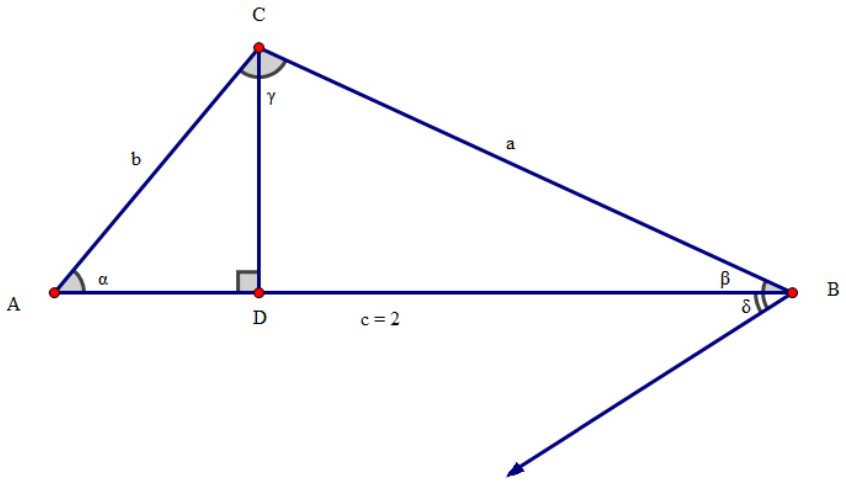

Figure 3.2

$$= \frac{cos(\alpha+\delta)+cos(\alpha-\delta)}{cos(\alpha)}$$

$$= \frac{cos(\alpha)cos(\delta)-sin(\alpha)sin(\delta)+cos(\alpha)cos(\delta)+sin(\alpha)sin(\delta)}{cos(\alpha)}$$

$$= \frac{cos(\alpha)cos(\delta)+cos(\alpha)cos(\delta)}{cos(\alpha)}$$

$$= cos(\delta)+cos(\delta) = 2cos(\delta)$$

This proves that a detour of this nature always has exactly the same amount of work associated with the wind, regardless of how long a detour someone takes.

The next question is whether or not this only holds for a special case. An anomaly that only occurs when the roads have this exact pattern. For a more complex example, consider the situation in Figure 3.2.

The only change is that the angle β is now no longer necessarily the same as the angle α. The angles with the wind are now $\alpha-\delta$ and $\beta+\delta$ respectively. For the sides b and a we can use the law of sines [16, Page 109] which says

$$\frac{a}{sin(\alpha)} = \frac{b}{sin(\beta)} = \frac{c}{sin(\gamma)}$$

Since the angles of a triangle sum to π, $\gamma = \pi-\alpha-\beta$ and the lengths of a and b work out to be

$$a = \frac{2sin(\alpha)}{sin(\pi-\alpha-\beta)}$$

$$b = \frac{2sin(\beta)}{sin(\pi-\alpha-\beta)}$$

So the work associated with the wind direction is

$$\frac{cos(\alpha-\delta)2sin(\beta)}{sin(\pi-\alpha-\beta)} + \frac{cos(\beta+\delta)2sin(\alpha)}{sin(\pi-\alpha-\beta)}$$

$$= \frac{2 \cdot (cos(\alpha) \cdot cos(\delta) + sin(\alpha) \cdot sin(\delta)) \cdot sin(\beta) + 2 \cdot (cos(\beta) \cdot cos(\delta) - sin(\beta) \cdot sin(\delta)) \cdot sin(\alpha)}{sin(\alpha) \cdot cos(\beta) + cos(\alpha) \cdot sin(\beta)}$$

$$= 2 \cdot \frac{sin(\alpha) \cdot cos(\beta) \cdot cos(\delta) + cos(\alpha) \cdot sin(\beta) \cdot cos(\delta)}{sin(\alpha) \cdot cos(\beta) + cos(\alpha) \cdot sin(\beta)}$$

$$+2 \cdot \frac{sin(\alpha) \cdot sin(\beta) \cdot sin(\delta) - sin(\alpha) \cdot sin(\beta) \cdot sin(\delta)}{sin(\alpha) \cdot cos(\beta) + cos(\alpha) \cdot sin(\beta)}$$

$$= 2 \cdot cos(\delta) \cdot \frac{sin(\alpha) \cdot cos(\beta) + cos(\alpha) \cdot sin(\beta)}{sin(\alpha) \cdot cos(\beta) + cos(\alpha) \cdot sin(\beta)} = 2 \cdot cos(\delta)$$

This proves that once again the work difference caused by wind direction equals $2 \cdot cos(\delta)$ regardless of the values of α and β.

Now before we continue, we are going to have to find out what the work difference is if we travel this road in the opposite direction. The only things that change are the angles, the length of the route segments stay the same. And the angles change from being $\alpha - \delta$ and $\beta + \delta$ to $\pi - (\alpha - \delta)$ and $\pi - (\beta + \delta)$. And because $cos(\pi - x) = -cos(x)$ [16, Page 63], the work in one direction is exactly the additive inverse of the work in the other direction, so going over a road and back again over the same road yields a work difference of zero.

We can expand this result in two ways. First we can use a construction illustrated in Figures 3.3–3.6 to turn any detour from a straight line into a number of triangles like in Figure 3.2. Since the only things that are relevant to the work from the wind are angle and length, we can rearrange segments as long as we keep angle, length and direction intact. First we need to rotate our view so the straight road is exactly horizontal. Then we can disassemble a route by first cutting it up into all individual segments (from every corner to the next corner). We need to mark the start and endpoints of each of the road segments to make sure we travel them in the right direction and then sort them by the vertical component of their lengths. Now we can assemble a new route starting at the same starting point. First lay all road segments that are parallel to the straight road. Next, attach the route segment with the lowest vertical component followed by a route segment with the same vertical component in the other direction. If none is available, cut up one of the road segments to size. Note that this reduces the amount of road segments by at least one. Repeat this step until all route segments are used. We now have a new route from A to B that consists of exactly the type of detours that we have shown to have the exact same value for work as their straight road counterparts. And because we showed that traveling a road in the opposite direction yields the additive inverse of the work, any distance traveled left of A or right of B is canceled out because there will always be a route segment making up for it by going in the other direction. Note that this construction works for roads of any shape and size, as long as it consists of line segments.

The last thing we need to do is to observe that, given two routes from A to B, the work for both of those routes will equal the work of traversing a straight route and the work of traveling the two routes must therefore be the same.

Now that we have shown that the work needed because of the wind is equal for all routes with the same start- and endpoints, we can safely say this objective will not be useful for generating routes with one start- and one endpoint. It could be useful for comparing routes if there are multiple start and endpoints, but there is no point in using it during route-generation.

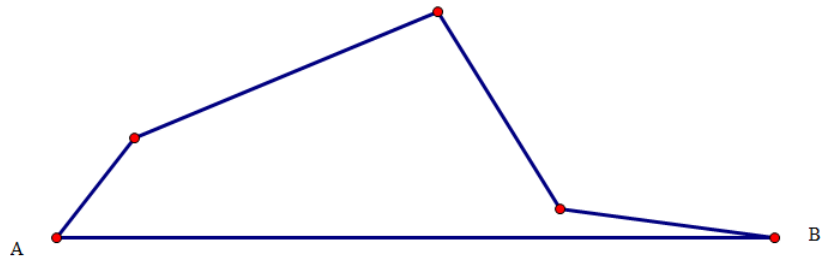

Figure 3.3: Original Route

Figure 3.4: Step 1

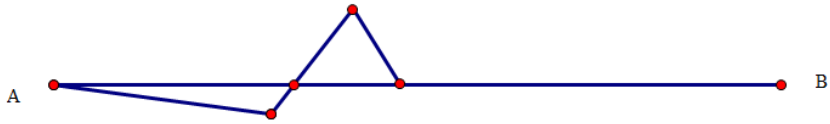

Figure 3.5: Step 2

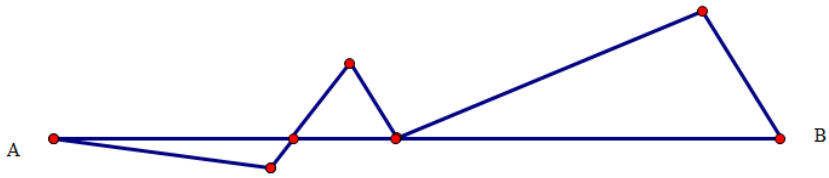

Figure 3.6: Rearranged route

4 Multiobjective Routing Algorithm

With data and objectives, it is time to work on the algorithm. First we draw inspiration from Dijkstra's classic routing algorithm in Section 4.1. We will discuss how the algorithm was modified in Section 4.2. We will discuss the use of Manhattan distance in the algorithm in Section 4.3 and the Chebyshev distance in Section 4.4. Finally we will discuss the possible objectives that were discarded for being incompatible with the algorithm in Section 4.5.

4.1 Dijkstra's Shortest Path Algorithm

Dijkstra's famous shortest path algorithm[12], first published in 1959, is as relevant now as it was back then. We will use a version of that algorithm with a priority queue[17], which has a computational time complexity in $O(|E| + |V|log|V|)$ where E and V are edges and vertices respectively, instead of the original algorithm's $O(|V|^2)$ computational time complexity. The priority queue prioritizes paths with the shortest length.

The code below uses three types of objects. First connections that contain a pair of nodes that that connection connects. Connections also have a function $NextNode(node\ a)$ that returns the node in a connection that is not a. Connections also have a length attribute that identifies the over-land distance between the two nodes. Second, we have nodes (including $StartNode$ and $EndNode$) that contain all connections that include that node. They also have a boolean called $Visited$ that is initially set to false. Lastly we have routes (P and Q) that have a Path element which is an ordered list of the connections used to get from the StartNode to the CurrentNode, the latter of which is also part of the route object. Routes also have a length attribute that is the length of the route. The priority queue sorts based on the length attribute.

```
1: for all connections C ∈ StartNode do
2:     P.Path := C
3:     P.CurrentNode := C.NextNode(StartNode)
4:     P.Length := C.Length
5:     add P to PriorityQueue
6: StartNode.Visited := True
7: while PriorityQueue not empty do
8:     Q := next route in PriorityQueue
9:     if Q.CurrentNode = EndNode then return Q
10:    if not Q.CurrentNode.Visited then
11:        for all connections C ∈ Q.CurrentNode do
12:            P.Path := Q.Path.Append(C)
13:            P.CurrentNode := C.NextNode(Q.CurrentNode)
```

14: $P.Length := Q.Length + C.Length$
15: add P to $PriorityQueue$
16: $P.CurrentNode.Visited := True$
17: **return** empty route

In words, what it does is it finds all nodes adjacent to the starting node and makes routes from the starting nodes to those nodes. Those routes are put into a priority queue which sorts them by length and the starting node is marked as visited (Lines 1–6). When that is done the algorithm repeatedly takes the route with the shortest total length out of the priority queue. If the last node in that route has been visited the route is discarded. Otherwise the algorithm generates copies of that route for each connection that contains its current last node. Those connections are each attached to one of the copies and they are put back into the priority queue. The old last node is marked as visited and the step concludes (lines 10–16). The algorithm finishes when a route reaching the end node is pulled out of the priority queue. That route is returned as the solution (line 9) and is guaranteed to be a shortest route from the start node to the end node.

4.2 Multi-objective Dijkstra's Shortest Path Algorithm

We will be modifying Dijkstra's algorithm for multi-objective route planning by making it into a restart algorithm. We need to be careful that none of the objective functions ever decrease in value when adding a connection to a route. that is already the case for the shortest path and cycleway objectives, but it is important to note for future objectives. The only things that need to be done to the actual route generation are adding a value for each of the objectives (in this case for distance and cycleway, recall that we want to minimize both) and sort the priority queue on a scoring function instead. This scoring function must generate a score for each route based on the values of the objective functions. Two of these scoring functions are discussed in 4.3 and 4.4.

The problem here is that with this scoring function we will generate only one route. Most of the time there will be multiple good routes and there is no way of objectively saying one is better than the other. There could for instance be a route that is 5km long of which 4km is not cycleway, and another route that is 8km long of which only 1km is not cycleway. One route is better in one objective and the other is better in another objective. What we can do is to compute a *Pareto front*[13]. This is — in the context of these bicycle routes — a list of routes that are not in all objectives worse than some other route. Routes that are not part of this Pareto front are said to be *dominated*.

Dijkstra's algorithm is deterministic as long as there is only one shortest route available. Assuming the scoring function is also deterministic (which both scoring functions will be), the multi-objective version will be deterministic as well. So in order to get different results we will have to change the scoring function for multiple runs of the restart algorithm. We will be doing this using *weights*. These weights are random real numbers between zero and one that are multiplied with the objectives. In this case, there are two objectives and we only need one weight w and use w for one objective and $1 - w$ for the other. However that does not scale with more objectives so a weight for each objective was used instead. These weights are fixed for a route, and new weights are generated for new runs

of the algorithm. Weights are very natural for this kind of algorithm, because they make objectives more or less important in a route, which is exactly what we want to see in solutions. So without adding the scoring function, the algorithm looks as follows. Note that the function frand() generates a random number between zero and one (excluding zero).

```
1:  function GenerateRoute
2:      w_1 := frand(), w_2 := frand()
3:      for all connections C ∈ StartNode do
4:          P.Path := C
5:          P.CurrentNode := C.NextNode(StartNode)
6:          P.Length := C.Length
7:          P.Cycleway := C.Cycleway
8:          P.Score := Score(w_1, P.length, w_2, P.cycleway)
9:          add P to PriorityQueue
10:     StartNode.Visited := True
11:     while PriorityQueue not empty do
12:         Q := next route in PriorityQueue
13:         if Q.CurrentNode = EndNode then return Q
14:         if not Q.CurrentNode.Visited then
15:             for all connections C ∈ Q.CurrentNode do
16:                 P.Path := Q.Path.Append(C)
17:                 P.CurrentNode := C.NextNode(Q.CurrentNode)
18:                 P.Length := C.Length + Q.Length
19:                 P.Cycleway := C.Cycleway + Q.Cycleway
20:                 P.Score := Score(w_1, P.length, w_2, P.cycleway)
21:                 add P to PriorityQueue
22:             P.CurrentNode.Visited := True
23:     return empty route
```

The only changes in the code are that lines 2, 7, 8, 19 and 20 are added, and that the priority queue now sorts based on score instead of length. The other part of the algorithm is the part that calls this GenerateRoute function.

```
1: for i =1 to n do
2:     R := GenerateRoute()
3:     if R not in Solutions then
4:         add R to Solutions
5: Remove dominated solutions from Solutions
```

As you can see this generates n routes and discards any duplicates. Afterwards any suboptimal solutions are discarded on line 5. This would also be the place to impose a maximum length by discarding any routes that exceed it.

4.3 Manhattan Distance to a Reference Point

The Manhattan Distance to a reference point[13, 5.4] represents the distance someone has to travel in Manhattan's city blocks. In that grid-structure it is impossible to go in

a diagonal line so the distance becomes the sum of the distances in one direction and in the perpendicular direction. This can be applied here with weights w and n objective functions o as follows.

$$score = \sum_{i=1}^{n} w_i \cdot o_i$$

In pseudocode this becomes as follows.

1: **function** SCORE(w_1, o_1, w_2, o_2)
2: **return** $w_1 \cdot o_1 + w_2 \cdot o_2$

The reference point here is the utopia point $(0, 0)$, which represents a route with a length and cycleway attribute of 0.

As a scoring function this has some nice properties. Since all weights are nonnegative, using Manhattan distance for this particular algorithm will always generate a non-dominated route. We can prove this as follows. Take a route j and a route segment k that can be added to it to form route l we find that

$$score(l) = \sum_{i=1}^{n} w_i \cdot o_i(l) = \sum_{i=1}^{n} w_i \cdot o_i(j) + \sum_{i=1}^{n} w_i \cdot o_i(k)$$

and we can conclude that the score of a route segment is fixed, and the score of a route is equal to the sum of the scores of its route segments. This is exactly the type of network where Dijkstra's algorithm always finds the lowest score route for. Now assume there is a route p that dominates the route we generated. That route has a lower value in every attribute than the route we generated. And because all objective function values and all weights are nonnegative, the score of p must also be lower. Therefore the route must have been explored before the route we generated, which is a contradiction.

There is a downside to Manhattan Distance though, and that is that it may not find all non-dominated routes. If we have three routes on a pareto front with their objective function values $k = (1, 2)$, $l = (2, 1)$ and $m = (1.9, 1.9)$. Then there would need to be weights w_1 and w_2 such that

$$w_1 \cdot 1.9 + w_2 \cdot 1.9 \leq w_1 \cdot 2 + w_2 \cdot 1$$

and

$$w_1 \cdot 1.9 + w_2 \cdot 1.9 \leq w_1 \cdot 1 + w_2 \cdot 2$$

If you plot this (plot available at[5]) you can easily see that this is impossible. Such a solution is called unsupported.

4.4 Chebyshev Distance to a Reference Point

Using the Chebyshev distance[13, 5.4] as a scoring function should change that. It represents the function

$$score = \lim_{x \to \infty} \sqrt[x]{\sum_{i=1}^{n} (w_i \cdot o_i)^x}$$

which works out to equal

$$max(w_1 \cdot o_1, ..., w_n \cdot o_n)$$

If we take our example from the previous paragraph it is easy to see that m will be the best solution at weights $w_1 = 1$ and $w_2 = 1$.

It is important to note that the Chebyshev distance might find *weakly dominated* solutions. Those are solutions that have the same value for one attribute and worse for all the others. Take for example two solutions with objective function values of $(10, 10)$ and $(10, 2)$ respectively. with all weights $w_1 \geq w_2$ the score of these two solutions is exactly the same and either may be chosen.

4.5 Discarded Objectives

There are a number of objectives that were considered, but quickly dismissed as being incompatible with the algorithm. We go over them briefly.

First, we need to discuss restaurants and landmarks. Someone might want to cycle past a windmill or have a bite to eat on the way. These are requirements on a route that are binary. Assume for the moment that someone wants to visit exactly one of these and assume we had data on these landmarks. We would need to find a way to look for routes that contain at least one of these. It is not possible to turn that into a non-decreasing attribute that has to be minimized, so we would need a different way to do this. The only other way would be to generate Pareto fronts for each eligible landmark consisting of the combined attribute values of two routes. One going from the start point to the landmark and the other going from the landmark to the end point. The number of routes generated this way is a multiplication of the number of routes in the first Pareto front and the number of routes in the second pareto front. Combined with the sheer number of restaurants and a potential for very long distances, we run into a bit of a combinatorial explosion and computation times will go far beyond what people are willing to wait for. It can be done, but it is very impractical. Of course if you pick a specific landmark it would not be a problem and though the program does not currently support this, it could be easily implemented by finding routes from the start node to the landmark, and then from the landmark to the endpoint.

Another objective, or rather, constraint we can discuss is imposing a minimum length. Doing this will again yield computation time issues. Throughout the run of the algorithm, it decides that a subroute is the best way to get from the start node to a specific other node, until it decides the best way to the end node. However, if we were to decide that route is not long enough and dismiss it, there is no guarantee the end node will ever be reached again. There may be only one connection to the end node. So we would have to dismiss the 'best' route to the node before it as well. We may have to do that for every node in the network in order to find the best solution with the right minimum length. In the end this will be equivalent to investigating every possible route until one is found to be the best, which is simply not possible in a reasonable time.

5 Case Studies and Discussion

All testing was done on an Intel Core i7 2600k processor at 3.4 GHz with 8 gigabytes of ram running Windows 7. The program was compiled using the GNU GCC compiler with -O2 optimization and no other compiler flags. It ran in a single thread on one virtual core.

5.1 Experiment 1 — Manhattan Distance

For this first experiment we would like to find out four things. Firstly we want to know whether or not the algorithm works. If that proves to be the case we want to know how many iterations we need to get all optimal solutions, check that the results are indeed all optimal for Manhattan distance as predicted, and lastly we want to see how many and how varied routes we get with Manhattan distance.

The experiment was done using the following settings:

Scoring function	:	Manhattan Distance
Iterations	:	100000
Start node	:	node 99 near LIACS in Oegstgeest.
End node	:	node 55 in Hilversum.

The results are displayed in the following table as well as in Figure 5.1.

	Length	Cycleway
1	78.4407	15.8795
2	78.65	14.1553
3	81.788	11.9241
4	83.367	10.8239
5	87.0759	8.97625
6	87.7373	8.68473
7	95.8029	7.69107
8	104.738	7.2658
9	106.944	7.17217
10	112.656	7.13784

The solutions were found at the following iterations:
0, 1, 3, 7, 8, 15, 19, 62, 410

The exact routes that each of these represent are not very important. We can observe that as the length of the route increases, the cycleway attribute decreases and that therefore none of these routes are dominated by other routes. Also, the shortest route we found here is longer than the shortest route found on the Fietsersbond planner. They have a shortest route over bicycle nodes of 73.52km.

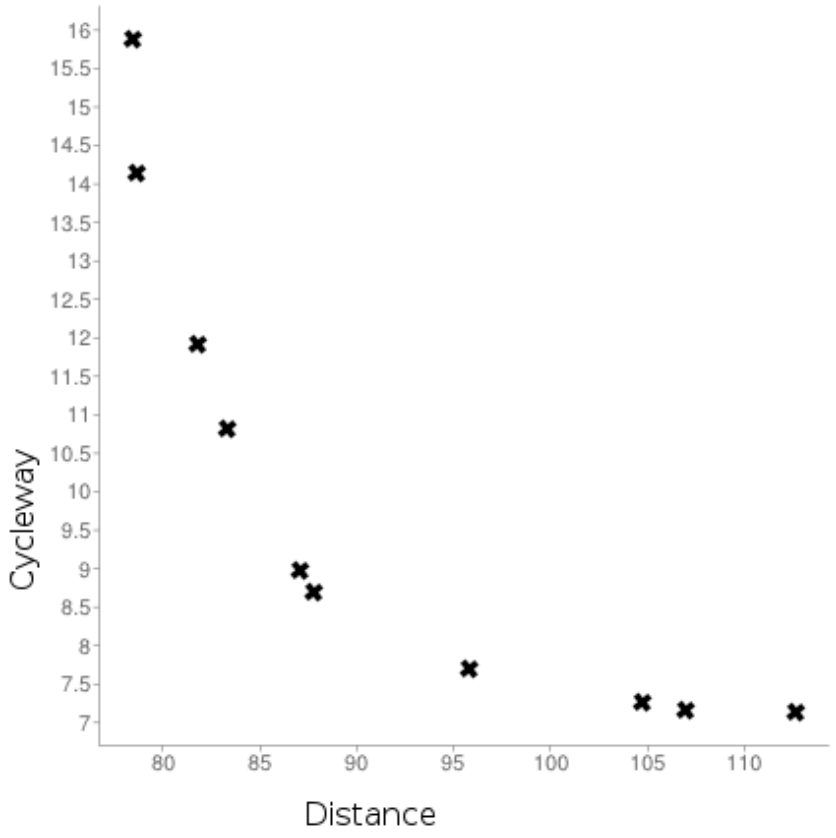

Figure 5.1

From this data it seems safe to say that the algorithm works. The discrepancy between these results and the results from the Fietsersbond is likely caused by a missing connection somewhere along the way. It also exactly matches what unmodified Dijkstra found. The results also seem nice and varied. And especially 2–5 seem like good trade-offs. It also seems that 500–1000 iterations is enough to find all or almost all possible solutions. But even with only about 20 iterations there should be a nice spread of solutions.

5.2 Experiment 2 — Chebyshev Distance

For this experiment we want to find out how many, if any, new results using the Chebyshev distance gives. We leave everything else the same for ease of comparison, so the settings for this experiment are:

Scoring function	:	Chebyshev Distance
Iterations	:	100000
Start node	:	node 99 near LIACS in Oegstgeest.
End node	:	node 55 in Hilversum.

This experiment found 32 solutions, 18 of which are Pareto-dominated by other solutions found in the same experiment. Only the non-dominated solutions are shown in the table below and Figure 5.2 shows all 32 solutions.

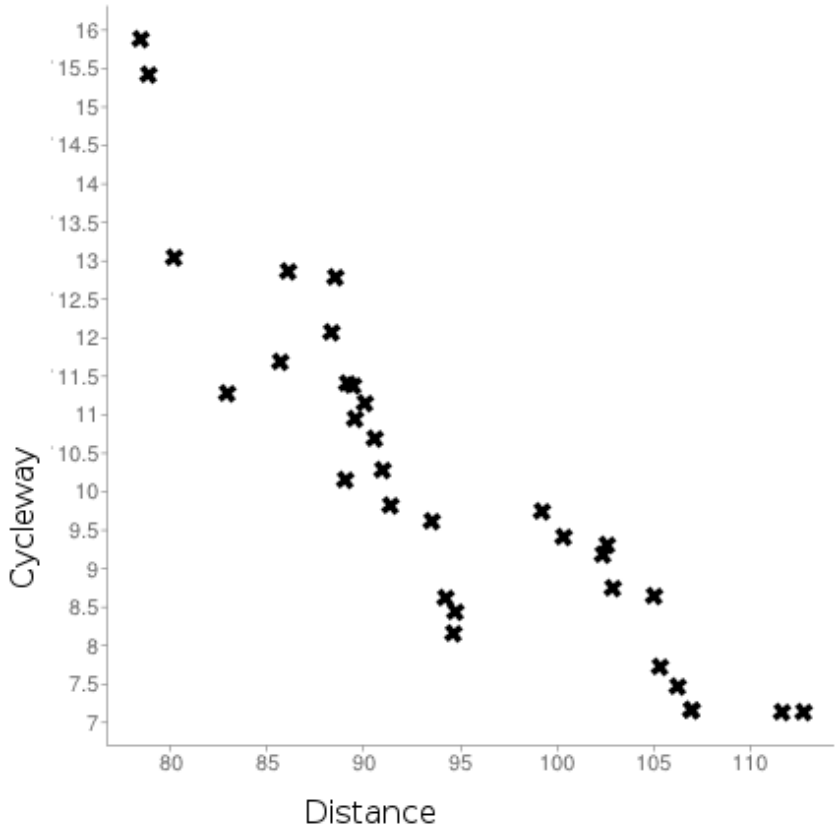

Figure 5.2

1	78.4407	15.8795
2	78.8655	15.428
3	80.229	13.0551
4	82.9422	11.2754
5	89.0876	10.1561
6	91.4326	9.83028
7	93.476	9.6134
8	94.2475	8.6166
9	94.6723	8.1651
10	105.289	7.723
11	106.282	7.46369
12	106.944	7.17217
13	111.639	7.14615
14	112.772	7.13784

The solutions here, dominated or otherwise, were found at:
0, 11, 18, 22, 24, 51, 57, 63, 64, 66, 193, 241, 251, 327, 351, 361, 421, 451, 489, 522, 737, 805, 939, 1057, 1909, 2090, 2097, 2149, 2540, 2675, 3136, 94891.

There are a few things we can observe from this. Firstly that it is apparently possible to find non-optimal solutions. It also seems to take more time to find solutions, even though there are apparently more solutions to be found and the odds of finding a new solution should be higher than they were with Manhattan distance.

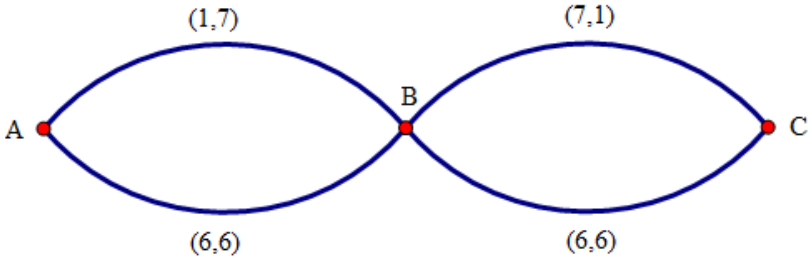

Figure 5.3

If we compare it with the Manhattan distance we find that 3, 4, 8, 9 and 13 are new, non-dominated solutions. But we also find that 14 is weakly dominated and that 2, 5, 6, 7, 10, 11 are dominated by solutions that were found using Manhattan distance. We also find that, from the previous experiment, only routes 1 and 9 were found.

It seems that Chebyshev distance performs significantly worse than expected. The reason for this may be seen in Figure 5.3.

Here we can see that the Chebyshev distance will follow the lower routes even though the upper routes have a much better result. With a large number of possible routes it is very likely that some dominant solutions will be impossible to find. It also makes it likely that other solutions will be Pareto-dominated. Combined with the longer computation time, it seems that Chebyshev distance on its own is significantly worse than Manhattan distance for this application.

5.3 Experiment 3 — Computation Time

For the third experiment we are interested in computation time and how that scales with with the length of routes generated. We also test for the number of routes generated with a cutoff point of 5 seconds computation time. The locations were chosen based on a rough route from LIACS to Groningen. We will use the following settings:

Scoring function	:	Manhattan Distance
Iterations	:	1000 per end node
Start node	:	node 99 near LIACS in Oegstgeest.
End node	:	Variable.

The results are listed in the table below as well as in Figure 5.4.

End node	Shortest path	Computation time	Routes in 5s	Total routes
16 in Alphen a/d Rijn	24.7854 km	0.862 seconds	5	5
57 in Mijdrecht	40.2658 km	1.616 seconds	6	6
55 in Hilversum	78.4407 km	4.619 seconds	10	10
25 in Voorthuizen	119.519 km	9.774 seconds	13	14
18 in Zwolle	169.798 km	20.032 seconds	15	18
30 in Steenwijk	216.454 km	29.654 seconds	18	23
85 in Groningen	295.601 km	45.390 seconds	20	29

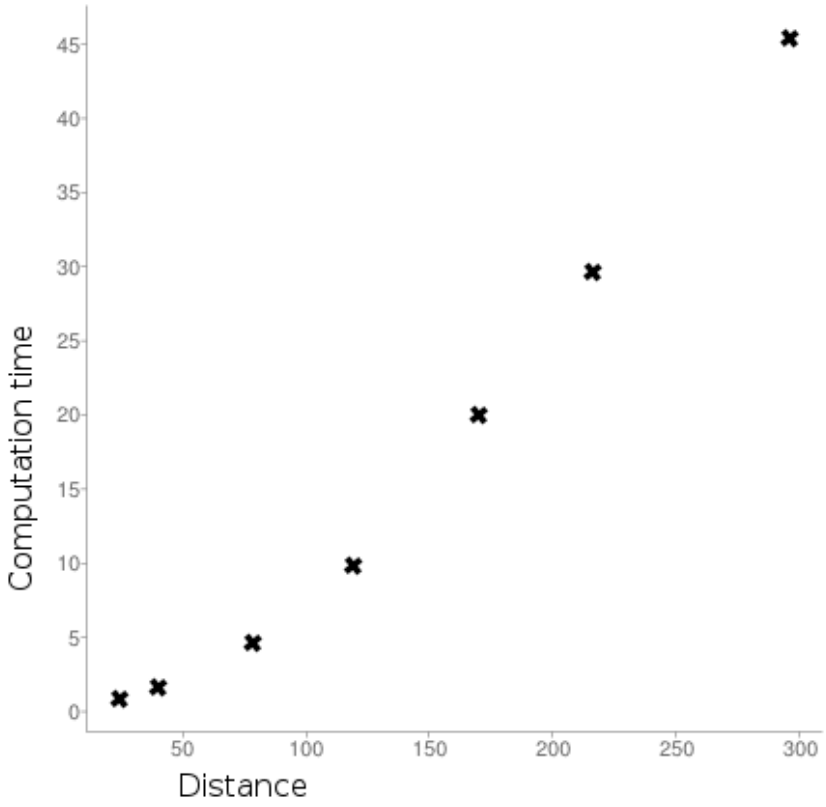

Figure 5.4

From these results we can observe a few things. Firstly there are more Pareto-optimal routes to nodes that are farther away. Secondly, the increase in computation time seems to start off quadratic and then settle down to an almost linear pattern. Lastly even with the shortest distance to Groningen being almost 300 kilometers, the algorithm still finds more than half of the solutions in 5 seconds.

The result that the number of Pareto-optimal routes increases as the distance increases is not at all surprising. There are many more small detours to make for longer routes.

The increase in computation time seems to become linear because we are reaching the borders of the country. Since the algorithm checks all nodes that are "closer" to the start node than the end node, one expects the computation time to be related to the surface of a circle with a radius of the distance to the end node. However, if we draw that circle with LIACS as its center and make that reach to the different end points, we would find that increasingly larger areas of the circle are not within the borders of the Netherlands, and the algorithm does not try to make routes that leave the country. Only the area of the circle that is also in The Netherlands should be counted and that is likely what accounts for this apparent leveling out.

The fact that the algorithm finds the majority of routes even for a distance of almost 300 kilometers in five seconds means that it, at least when used in The Netherlands, will always find routes in a reasonable time. It does not really matter that not all possible routes are found, as giving the user too many routes would be overwhelming and unnecessary.

6 Conclusions and Future Work

In summary we found that the algorithm works well and runs reasonably fast. Using Chebyshev distance to a reference point did not yield very nice results, but Manhattan distance worked very well and gives a nice spread of results. We will now discuss the research questions.

The first research question was "what kind of data is available that can be used in bicycle route planning?" It turns out that available data was very limited and surprisingly inaccurate. The fact that there is no data publicly available about whether or not a road runs through a forest is discouraging, and it has severely limited the potential of this project. It is clear though that the Fietsersbond does have the kind of data we would have liked to have, and since they work with many volunteers anyway, it might be possible to work with them and their data to use the techniques described here to improve their route planner. So for future work, contacting them should be first on the list.

The second research question was "how can that data be used to create meaningful objectives?" Because of the limited data available the options for objectives have been limited as well, to the point where only one objective besides shortest path could be constructed. Directing cyclists to paths designated solely to cyclists is still a good improvement over what was available before this project started though, and that is something to be satisfied about. Also the result that the effect of wind direction is determined solely by the start and end points and not affected by the route between them is good information to have. Even though it did not provide a useful objective it saved a lot of time trying to construct one.

The last research question was "what kind of algorithm can be used to output multiple possible routes based on these objectives?" It turns out that the restart multi-objective version of Dijkstra's shortest path algorithm with a priority queue that has been developed in the previous chapters works very well. It provides the user with a range of possible routes to their destination exactly as intended. Even though the algorithm is not capable of generating unsupported solutions, it generates enough solutions for a user to choose from.

Data can always be added to and corrected, the algorithm is flexible enough to allow for the addition of new objectives and it just works. The goal was to create multi-objective bicycle routes with a nice spread of options for the user and this algorithm does just that. There is absolutely no reason to believe the algorithm would not work just as well with more objectives and it is great that, at least with Manhattan distance, it will never generate dominated routes. Although it is true that there cannot be any guarantees that the algorithm will find all routes on the Pareto-front, it does not really matter. The user will not want every possible route anyway or he might as well plan it himself.

To turn this into an actual working tool it would need a graphical user interface that

allowed the user to easily select the nodes he wants for start and end, and it would be important to give the user a limited choice of routes with maps. Especially when more objectives are added the number of different routes generated may grow substantially. Those routes may be selected using an algorithm like NSGA-II[11] or SMS-EMOA[10], but all that is future work. For now though it is safe to conclude that the algorithm works exactly as intended and that multi-objective bicycle routes can make a good addition to already existing routing software.

Bibliography

[1] Bing Maps. http://www.bing.com/maps/.

[2] Falk. http://www.falk.nl/.

[3] Fietsersbond Routeplanner. www.fietsersbond.nl/fietsersbond-routeplanner.

[4] Google Maps. https://maps.google.com/.

[5] Graph for unsupported solutions. https://www.desmos.com/calculator/jobf0upjs6.

[6] Key:highway. http://wiki.openstreetmap.org/wiki/Key:highway.

[7] OpenStreetMap. http://www.openstreetmap.org/.

[8] Overpass API. http://www.overpass-api.de.

[9] Yahoo Maps. http://maps.yahoo.com/.

[10] Nicola Beume, Boris Naujoks, and Michael Emmerich. SMS-EMOA: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653 – 1669, 2007.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[12] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[13] Michael Emmerich and Andre Deutz. Multicriteria optimization and decision making principles, algorithms and case studies. 2006.

[14] fietsen.123.nl. Hoeveel fietsen zijn er in nederland. http://www.fietsen.123.nl/entry/12658/hoeveel-fietsen-zijn-er-in-nederland, jan 2011.

[15] Landelijk Fietsplatform. Landelijke Routedatabank.

[16] I.N. Bronstein, K.A. Semendjajew, G. Musiol, H.Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 1st edition, 1993.

[17] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming*, volume 1843. INFSYS Research Report, 2006.