Universiteit Leiden

Opleiding Informatica

Advancing Algorithms for the Multi-Objective Generation of Bicycle Routes in OpenStreetMap

Name:	Thierry van der Spek
Date:	27/08/2014

1st supervisor:	dr. M.T.M. Emmerich
2nd supervisor:	drs. R. van Vliet

BACHELOR THESIS

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

Abstract

Modern day routing planners are optimized to find the shortest route. This allows us to travel between two places efficiently. However, traveling is not only a means of transportation between two places but can be a recreational activity. In this work a way of generating bicycle routes that are not only based on distance, but consider other factors as well, is proposed. The additional usage of points of interest and amenities are used to generate routes. Furthermore a way of generating round trips is discussed. These options are combined with faster algorithms than the status quo in multi-objective bicycle route planning. The approach has been tested and visualized using data from OpenStreetMap.

Contents

1 Introduction

Route planning has been subject of research for over half a century. Dijkstra laid the foundation of modern day routing algorithms in Dijkstra's algorithm [6] from 1959. Since then many variations of this classic algorithm have been applied to different routing problems. Route planning has been applied to navigation applications, which have become part of our lives. Navigation applications come in the form of dedicated devices for cars, boats, planes and even bicycles providing turn by turn navigation as well as (web) application for computers to plan a trip beforehand like Google Maps and Bing Maps.

Many of these modern day applications focus on routing where distance or travel time is taken into account as the single factor to base the algorithm on. This factor however, could be any combination of distance, scenic value, energy consumption etc. associated with the roads. When driving a car most of these factors might not seem that important. Even though people do drive recreationally, driving is mostly seen as a means of transportation between point A and B. However, when cycling people might be interested in other factors than just distance or travel time. Cycling is seen as a means of recreation. For this group properties like scenic value or the ability to choose alternative routes are more important. Currently, limited research has been done on combining other factors than distance in routing applications.

It has however has been discussed before in [11]. In this thesis we extend the functionality of existing route planners by including:

- The addition of amenities or points of interest along the route
- Generation of alternative routes and a way of combining them to round trips
- Faster and more grained algorithms by integrating advances to the, as in the aforementioned thesis used, Dijkstra's algorithm

By adding these factors we create a routing application that can be used for leisurely bike rides where distance is no longer the only factor. We have implemented the generation of these multi-objective routes and present these in a web user interface, which allows for easy usage and understanding of the algorithms.

We have chosen to extend classic routing algorithms that take only a single objective as a measurement of cost. We add functionality, but limit ourselves to these algorithms. Faster methods for routing do exists, but are mostly based on preprocessing. These techniques and can be combined with the algorithms, but would make them more complex and harder to understand. The basic algorithms that are used are the ones explained in Section 4.

This thesis is the final report of the research done for the bachelor program computer science at Leiden Institute of Advanced Computer Science. This report refers to software written for this research. A copy of this software can be obtained upon request.

This thesis is structured as follows: In Section 2 we will elaborate about the background of routing and look into related work that has been done in this field of research. Section 3 will focus on the data that represents our map of the Netherlands and show how this data can be prepared to be later used in our routing algorithm. Section 4 will show the basic algorithms based solely on distance to give a core understanding of how these methods work. In Section 5 we will explain how we extend the fastest algorithm taken from Section 4 with multiple objectives and additional functionality. Section 6 will show some experiments done on the algorithms from Section 4 and 5. Section 7 will summarize our findings and discuss where our research could be extended.

2 Background and Related Work

The idea of generating bicycle routes using multiple objectives has been discussed by Hijmans [11]. In his thesis he discusses the idea of using several objectives instead of just distance for generating bicycle routes. The objectives he discussed are shortest distance, scenic beauty, cycleway and wind direction.

Shortest distance will always be a part of generating routes. We can take several objectives into account when generating routes, but in the end we have to generate a feasible and practical route. We can not generate a 300 kilometer detour just because it has scenic beauty if our start and end point are only 10 kilometers apart, except when planning a round trip. The second objective that is discussed is scenic beauty. Scenic beauty is a nice feature to have when cycling, but unfortunately it is hard to measure objectively. Different scenic attributes might be hard to quantify and compare. Furthermore, to our knowledge no such data source is freely available or even exists. This is why this objective was dropped.

Although scenic beauty was dropped as an objective, the objective cycleway that was used implicitly includes a part of this objective. Cycle way is a predefined bicycle node network in the Netherlands. This node network runs throughout the country and is selected on suitability for cyclists. They have safety requirements and are also selected on attractiveness of the surrounding landscape. [11] limited the graph to this node network. By limiting the graph to routes that are only tagged as cycleway the size of the routable network is reduced. This property also has the advantage of implicitly adding a constraint on usage of highways. This cycleway objective is an important feature for bicycle or walking routes, but in this thesis we will not restrict the graph to just this node network. We want to create a routing solution for every address in the Netherlands. This means we have to add our own constraint on the usage of roads excluding highways and other main car roads. More about this in Section 3.

The last objective that is looked into is wind direction. Wind could affect a cyclist enjoyability of his route. Planning a different route with less wind resistance could be preferred to a shorter route with more resistance. Hijmans showed that trying to avoid wind will not work since distance travelled to avoid wind or even benefit from it will be cancelled out by the work that it would take travelling back in the opposite direction given there is a constant wind speed and direction.

In this thesis an extension is made on the ideas of Hijmans. We will not focus on the objective cycleway and will omit the objective wind direction.

The generation of routes has been subject of research for a long time. Since the foundation for routing has been laid by Dijkstra [6], improvements have been made. One of the improvements is the A* algorithm [10]. A* is a generalization of Dijkstra, which uses a heuristic to estimate distance between source and target and is generally faster than Dijkstra. Although worst-case it is not better than Dijkstra's algorithm, with a good heuristic it will not expand out all the nodes and edges that Dijkstra's algorithm would and thus perform faster.

Another improvement to these algorithms is to use bidirectional search as first introduced in [3]. Bidirectional search runs two simultaneous searches, forward from the initial state and backward from the goal. An outline of these algorithms is given in Section 4.

There are several algorithms that outperform (bidirectional) A*. Most of these techniques are based on pre-processing the graph. An overview of most of those techniques is given in [4]. Since we are not focused solely on speed optimization we chose to stick to our classic routing algorithms and omit other speed optimization techniques.

3 Data exploration and pre-processing

For generating bicycle routes we will use real life geographical data from the OpenStreetMap (OSM) environment. OpenStreetMap is an openly licensed map of the world created by volunteers in a Wiki like form. This open source map contains imperfections and small flaws that vary according to the area [9], but due to its open source nature it is a good source of geographical data to use in our project.

According to [11] there are several commercial geographical data sources available including Google Maps, Bing Maps and Yahoo Maps. Compared to the OSM environment they may contain less flaws, but public (free) access is limited to a certain number of queries. This is why the OSM environment was chosen as a geographical data source.

3.1 Data exploration

OSM provides all his geographical data in a XML structured way available for download. These XML files contain several elements. Those elements are node, way and relation. A node consists of an id, latitude and longitude and represents a point on the map. A way is an ordered set of nodes, which represents a road segment. Relations are a collection of possibly nodes, ways and other relations. A relation defines geographic or logical relationships between the elements. An example of a relation could be a bus route, which could contain bus stops at nodes and ways as part of the route.

The full network of the Netherlands as an XML file is roughly 22 gigabytes of data. The bicycle node network used in [11] only is a small subset of this data and is around 2 megabytes in size. The approach used of reading in the data into the program can not be used on such a large amount of data. This is why we have to choose a different approach.

3.2 Data pre-processing

The OSM data is commonly used in applications to render visually and generate custom maps.
The first problem that arises is that the standard OSM data is not suited for routing. For routing we will need a network where nodes are connected by edges. In the OSM data there are several roads that intersect each other geographically. All intersecting roads are cut off into sub-roads, which start or end at the intersection. In order for the network to be routable, the coordinates where those roads intersect should be the start or end node of a road. At this node a choice for 2 or more edges can be made. In the OSM data this is not the case.
To make the OSM data suitable for routing the data should be processed. This is illustrated in Figure 1:

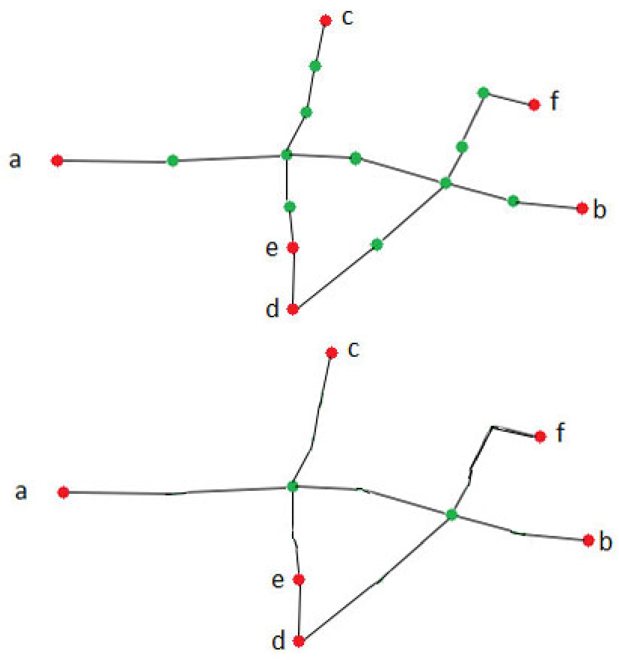

Figure 1: Original graph (top) and processed graph (bottom)

The original data on ways contains roads like (a, b), (c, d) and (e, f). The data is transformed to 8 edges (a, x), (c, x), (x, e), (e, d), (x, y), (e, y), (y, f) and (y, b). With a defined node as intersection we can use this data for routing.

When we use this graph in our routing algorithm, we will have to calculate the distance of edges. This is something we can easily pre-calculate and store. We calculate this using the Haversine formula. The Haversine formula calculates the distance between two coordinates on a sphere. This is slightly more accurate than Euclidean distance since it takes the curvature of the earth into account. Local height differences are not taken into account in both formulas, but since the Netherlands is mainly flat we choose to ignore those height differences.
The distance $d(i, j)$ between two consecutive nodes i and j is is calculated as follows:

$$d(i,j) = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_j - \phi_i}{2}\right) + \cos(\phi_i)\cos(\phi_j)\sin^2\left(\frac{\lambda_j - \lambda_i}{2}\right)}\right)$$

i, j are two consecutive nodes contained within the line string representing a road.
ϕ represents the latitudes of the nodes
λ represents the longitudes of the nodes
r is the radius of the earth. For our calculations it was set to an approximated 6378137 meters.

Since every edge is a line string of coordinates, the length of an edge is the sum of the distance between all consecutive nodes.

3.3 Data structure

According to the OpenStreetMap Wiki a common approach of storing OSM data for routing purposes is to convert and insert the data into a PostgreSQL database. PostgreSQL allows for efficient storage of the data and provides geospatial extensions. PostgreSQL is an open source relational database and has an useful extension called PostGIS. This extension allows for usage of geospatial functions, which makes handling spatial data easier and more powerful. These advantages could also be used for our routing. This is why we chose the same approach for our software. The newly generated edges from our preprocessing steps are stored in our relational database along with their respective lengths. For each edge we create a row in our table. The edges are stored as a coordinate string containing the latitudes and longitudes of the nodes within the road segment that represents the edge. Furthermore we store the coordinates of the edge's source and target node for easy access. Now that we have stored all of our edges we can route based on the distance.

3.4 Pruning of the search area

When we are searching a way or several ways between a source and target we generally travel towards those places and not away from them. We can try to omit roads that will certainly not be on the shortest path. Unfortunately it is not easy to say that roads may not be on the shortest path from source to target. We could pre-calculate this information for every node in the graph, but the amount of information that would have to be stored would be infeasible.

A method that has been proposed before is geometric pruning [13]. The idea is to store a geometric shape for each edge e that contains at least all nodes that can be reached by a shortest path starting with e. Although this method requires a lot of preprocessing, it can decrease the number of nodes visited in a search by up to 95% when using a effective geometric shape like a bounding box.

Another way of using bounding boxes within a real road network, which uses a less intensive preprocessing measure, is to use the longitudinal and latitudinal distance between the source and target. This method assumes that the shortest path has a high chance of being within the bounding box including source and target and add a certain margin. This reduces preprocessing to comparing nodes to a larger or smaller than criteria, but does not guarantee the optimality of the path at all times.

4 Routing based on distance

Before we start adding options to our algorithm we define the basic case, where we only route based on distance. In this Section we will discuss several algorithms that have been implemented for this basic case of routing. It is important to look at this basic case of routing before we start adding functionality. This helps us compare and gives a fundamental understanding of the steps taken in the algorithms. Furthermore the basic algorithm can be used to see if the algorithm is fast enough to be used by end users. Studies have found that delays greater than 10 seconds encourage users to believe that an error has occurred in the processing of their request [2]. This means that if the basic case of our algorithms would take more than 10 seconds to process and return our data it would not be fit to be used in our program. Section 6 shows a comparison of the time the algorithms took.
The following algorithms were implemented and tested.

4.1 Dijkstra's Algorithm

Dijkstra's classic algorithm has been the basic algorithm for each of our implemented algorithms. Dijkstra's algorithm will construct better paths in an iterative manner. Choosing to extend the path in the direction of the lowest potential until a target is reached. This is done by storing the length of the shortest path found so far and by storing the preceding nodes of that path. In this thesis we will use almost the exact implementation that has been used by [11], which is Dijkstra's algorithm with a priority queue as proposed in [8]. The basic outline of this algorithm is as follows:

1. Let S be an empty set and let p_s, the potential of a vertex v, be ∞. Let p_s be zero.
2. Find the vertex v_0 which has the minimum potential in $V - S$ and add it to S. If v_0 equals to t, halt.
3. For all vertices v such that (v_0, v) is in E if $p_s(v_0)+l(v_0, v)$ is less than $p_s(v)$, replace the path from s to v with the path from s to v_0 and the edge (v_0, v). Let $p_s(v)$ be $p_s(v_0) + l(v_0, v)$.
4. Go to step 2.

When we imagine our source node s and target node t to be points in a 2d representation, Dijkstra's algorithm will explore nodes around s in all directions until t is found. A 2d visual representation of how the search area is roughly explored is shown in Figure 2 as the outer circle.

4.2 A* algorithm

The A* algorithm is similar to Dijkstra with one major difference: it uses a heuristic to estimate the lowest total cost. Like Dijkstra's Algorithm, when A* is running it chooses to extend the path in the direction of the lowest potential cost until the target is reached. However, the cost is no longer calculated as the distance between the nodes, but a combination of distance to the node that is to be explored and heuristic value. Using a good heuristic can drastically reduce the amount of nodes that are explored. The most common heuristic value is the Euclidean/Haversine distance between the node that is to be explored and the final target node. The total cost function will be a combined function of actual distance to the next node along the road and Haversine distance to the final target node. Using this heuristic value means that a preference is created to search in the direction of the final target node as straying away from it would increase the Euclidean distance thus increasing the heuristic cost. The basic outline of this algorithm is as follows:

1. Let S be an empty set and let p_s, the potential of a vertex v, be ∞. Let p_s be zero.

2. Find the vertex v_0 which has the minimum value of $p_s(v) + h_s(v)$ in $V - S$ and add it to S. If v_0 equals to t, halt.
3. For all vertices v such that (v_0, v) is in E if $p_s(v_0) + l(v_0, v)$ is less than $p_s(v)$, replace the path from s to v with the path from s to v_0 and the edge (v_0, v). Let $p_s(v)$ be $p_s(v_0)+l(v_0, v)$ and remove v from S if v is in S.
4. Go to step 2.

When we imagine our source node s and target node t to be points in a 2d representation, A* algorithm will explore nodes around s with a direction preference towards t until t is found. A 2d visual representation of how the search area is roughly explored is shown in Figure 2 as the inner ellipse. We can clearly see that the search space that is explored is smaller than that of the Dijkstra algorithm.

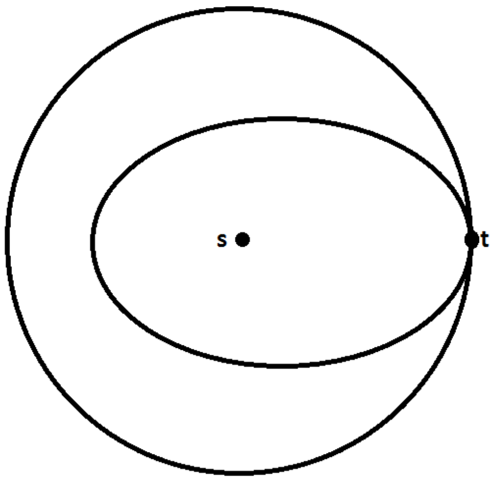

Figure 2: Search space visualization Dijkstra vs A*

4.3 Bidirectional Dijkstra

Both Dijkstra's algorithm and the A* algorithm are unidirectional. Classic (unidirectional) Dijkstra and A* algorithms start with source node s and expand until target node t is found. This means they only search from the origin s. The target t has a smaller role than s. With bidirectional search we use both s and t to search. Bidirectional search was first mentioned in [3]. We alternate between a forward search from s and a backwards search from t and continue to do so until the searches meet under a certain criteria. This is not when the searches meet for the first time per se. There could be a shorter total path than the sum of the paths that meet at first. We continue until two searches meet and there does not exist any length of a path that we have extended in the forward search plus any length that we have extended in the backwards search that is smaller than the current smallest path found from s to t. This works because any combination of to be explored paths that could potentially meet in the searches are longer than the current shortest path found so we can stop searching. The outline of this algorithm is as follows[12]:

1. Let S and T be empty sets and let $p_s(v)$ and $p_t(v)$, the potential of vertices v for s and t, be ∞. Let $p_s(s)$ and $p_t(t)$ be zero.
2. Find the vertex v_0 which has the minimum potential for s in $V - S$ and add v_0 to S. If v_0 is in T, go to step 7.

3. For all vertices v such that (v_0, v) is in E, if $p_s(v_0) + l(v_0, v)$ is less than $p_s(v)$, replace the path from s to v with the path from s to v_0 and edge (v_0, v). Let $p_s(v)$ be $p_s(v_0) + l(v_0, v)$.
4. Find the vertex v_0 which has the minimum potential for t in $V - T$ and add v_0 to T. If v_0 is in S then go to step 7.
5. For all vertices v such that (v, v_0) is in E, if $l(v, v_0) + p_t(v_0)$ is less than $p_t(v)$, replace the path from v to t with the edge (v, v_0) and the path from v_0 to t. Let $p_t(v)$ be $l(v, v_0) + p_t(v_0)$ and if v is in T remove v from T.
6. Go to step 2.
7. Find the edge (w, v) minimizing $p_s(w) + l(w, v) + p_t(v)$ such that w is in S and v is in T. The shortest path from s to t consists of the path from s to w, the edge (w, v) and the path from v to t if $p_s(w) + l(w, v) + p_t(v)$ is less than $p_s(v_0) + p_t(v_0)$. Otherwise the shortest path it consists of the path from s to v_0 and the path from v_0 to t.

We can again visualize our explored area as a 2d representation. This is done in Figure 3 This time we see that we search from both the source and the target. The total area that is searched (the sum of the 2 smaller circles) is smaller than the area of the bigger circle that visualizes the search space explored in Dijkstra's algorithm.

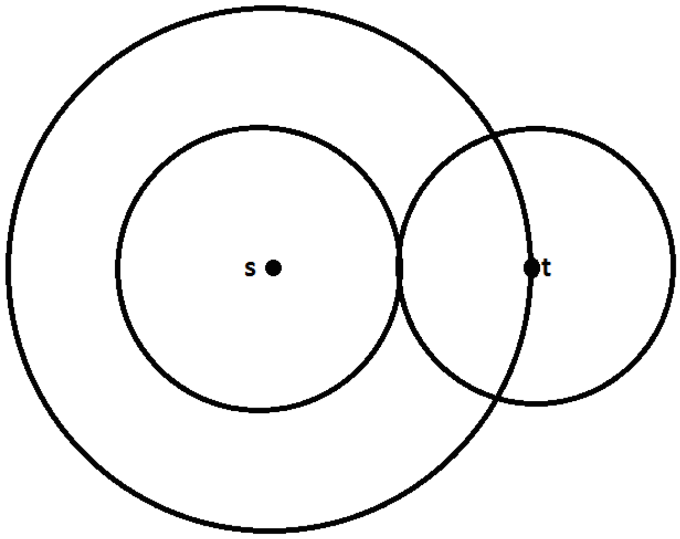

Figure 3: Search space visualization Dijkstra vs bidirectional Dijkstra

4.4 Bidirectional A* using distance only

In bidirectional A* we combine the A* algorithm with the bidirectional search method shown before. We have to use two types of heuristics: one that benefits the forward search and one that benefits the backward search. The forward heuristic can be the same as shown in A* search. The Euclidean or Haversine distance from the node that is to be explored to the target. For our backwards search we would just have to adjust this heuristic to use the distance to our source or the inverse distance to the target. This way the heuristics guide the search in the right directions. Our stopping criteria can still be the same as bidirectional Dijkstra.

The outline of the bidirectional A* algorithm is as follows [12]:

1. Let S and T be empty sets and let $p_s(v)$ and $p_t(v)$, the potential of vertices v for s and t, be ∞. Let $p_s(s)$ and $p_t(t)$ be zero.
2. Find the vertex v_0 which has the minimum value of $p_s(v) + h_s(v)$ in $V - S$ and add v_0 to S. If v_0 is in T, go to step 7.
3. For all vertices v such that (v_0, v) is in E, if $p_s(v_0) + l(v_0, v)$ is less than $p_s(v)$, replace the path from s to v with the path from s to v_0 and edge (v_0, v). Let $p_s(v)$ be $p_s(v_0) + l(v_0, v)$ and if v is in S remove v from S.
4. Find the vertex v_0 which has the minimum value of $p_t(v) + h_t(v)$ in $V - T$ and add v_0 to T. If v_0 is in S then go to step 7.
5. For all vertices v such that (v, v_0) is in E, if $l(v, v_0) + p_t(v_0)$ is less than $p_t(v)$, replace the path from v to t with the edge (v, v_0) and the path from v_0 to t. Let $p_t(v)$ be $l(v, v_0)+p_t(v_0)$ and if v is in T remove v from T.
6. Go to step 2.
7. Find the edge (w, v) minimizing $p_s(w) + l(w, v) + p_t(v)$ such that w is in S and v is in T. The shortest path from s to t consists of the path from s to w, the edge (w, v) and the path from v to t if $p_s(w)+l(w, v)+p_t(v)$ is less than $p_s(v_0)+p_t(v_0)$. Otherwise the shortest path it consists of the path from s to v_0 and the path from v_0 to t.

The heuristics for the backward search $h_s(v)$ forward search $h_t(v)$ are based on the Haversine distance between v for and either s or t.

The visualization of the search space is given in Figure 4. The search space is shown by the inner two ellipses. For comparison, the outer two circles represent the search space from Bidirectional Dijkstra from Figure 3 for comparison. The two searches from s and t search globally towards each other until they meet under the criteria described. This is not the first meeting point in the search per se as show in Figure 4, but it can be.

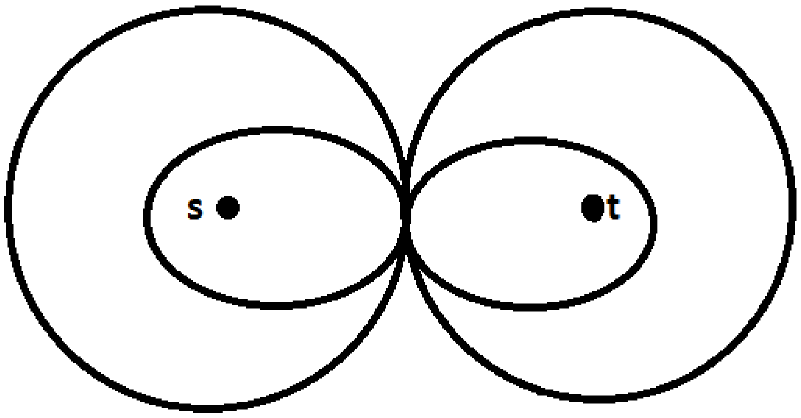

Figure 4: Search space visualization bidirectional Dijkstra vs bidirectional A*

5 Adding Objectives

5.1 Highway Avoidance

Since this research is mainly focused on bicycle routing an important property should be that the route can be constrained by roads that are usable by bikes. The data from the OSM data source contains tags about road properties. Road properties include a tag about the type of road. One of these properties is called "cycleway". The cycleway tag indicates if the road allows cyclists to drive on. Unfortunately due to its open source nature most roads are not tagged or tagged incorrectly as cycleway. If we would limit ourselves by using only these roads the amount of roads left would not be routable. Another tag is "motorway". This tag indicates if a road is road that can be used by cars. Another “clazz” tag indicates if that motorway allows only cars. By combining these tags we can exclude all highways and major motorways from the data. In Section 6.2 results are shown about excluding these roads from our routable graph, creating a cycle friendly route.

5.2 Alternative Routes and Round Trips

In leisurely bike rides it is not uncommon to return to the same place we started. People could for example go for a day trip on a bike or cycle for an hour. As most route planners focus on getting to one point fastest, users are left to use the same route back or find their own. This is why it is important to generate alternatives. These alternatives can be used as a way back or just a different way to the destination. Since the algorithms are deterministic the outcome is the same for the same input. This means when generating an alternative path we have to take a worse path than our initial one. When driving recreationally however this is not an issue.

Our first intuition is to use the second or third best paths as an alternative for our initial route, but this poses a problem. The k-shortest paths [7] are likely to only slightly vary from the best path. An example of this is shown in Figure 5. The best path between the markers is displayed as the straight line between the markers as seen on the left side. The second best path, as seen on the right side, takes a small detour upwards and then goes down and then merges with the best path again. This is why we have to choose a different approach.

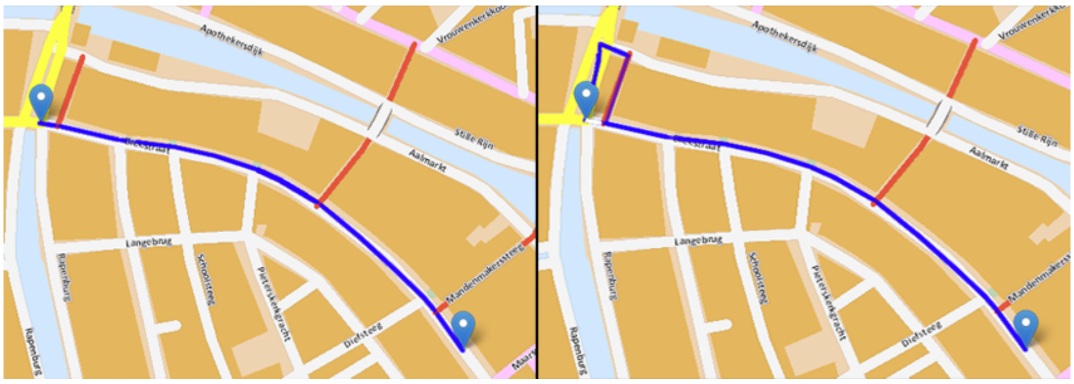

Figure 5: Using k-best routing algorithm to calculate an alternative route

For an alternative path our path should have only limited sharing with the original best path. Since the k-best paths do not consider this criterion, this method does not seam practical for computing alternatives or round trips.

A method has been proposed by [1] that focuses on limited sharing and local optimality. According to [1] limited sharing imposes an intuitive alternative without a significant overlap with the original route: $l(Opt \cap P) \leq \gamma \cdot l(Opt)$ where $l(Opt)$ denotes the sum of the lengths of the edges from the optimal (original) path and $l(Opt \cap P)$ denotes the sum of the lengths of the edges shared by path P and the optimal path Opt. γ is the factor or weight that defines the amount of overlap shared.

Local optimality invokes a reasonable constraint. A route is reasonable if it contains to unnecessary detours. If there is an easy path from A to B one would not take a detour through C if it does not improve the route.

This method works well, but there is more to think about when we think about using alternative routes for a round trip. We note that our round trip is not an arbitrary trip returning to A, but rather visiting point B as a point of interest and returning to point A through an alternative route. When we want to generate these routes using only limited sharing roads could be very close together providing the same sights. Bicycle roads could run parallel whilst being only several meters apart, especially in areas with a dense road network. This is why we choose a slightly different approach from [1].

Our bidirectional A* algorithm currently has an edge weight function that is solely based on distance. For calculating our alternative route we add an objective to this weight function. This objective is the distance between the closest edge (u, v) in the original shortest path, and our new edge (u', v'). Since our edges consist of line strings of coordinates, we can take the mean of the coordinates in our new edge (u', v') and compare the Haversine distance to the mean of coordinates for each edge in the original graph. The distance to the closest edge (u, v) will determine the score for our function.

Now the edges weight function is a combination of shortest distance and distance to the original graph. The distance to the original graph is a favorable feature so should be inverted to work as a weight function. This method works in providing an alternative route, but it creates a new problem. Since our nearest neighbor function is not limited, the optimum of the function could be obtained by straying apart from the original route, compensating the shortest path objective. A good alternative would stray from the original path, but not too far. We can use a desirability function on the distance to the original graph, as proposed in [5]:

$$d(Y_i) = \begin{cases} 1, & Y_i \leq L_i \\ (\frac{Y_i - U_i}{L_i - U_i})^{r_i}, & L_i < Y_i < U_i \\ 0, & Y_i \geq U_i \end{cases}$$

L is the lower limit on the distance
U is the upper limit on the distance
Y_i is our nearest neighbor distance
r_i is the gradient of the line

An example of this function is plotted in Figure 6:

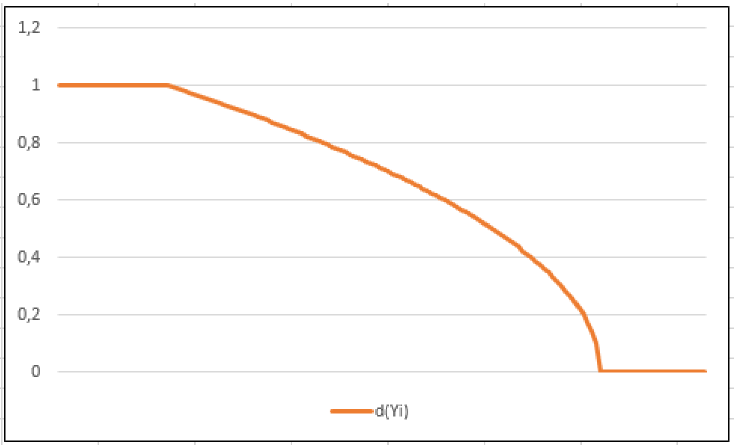

Figure 6: Example plot of $d(Y_i)$ with U set to 1 , L set to 0 and r_i set to 0.5

By using a desirability function on the distance we limit the maximum distance a path could stray from the original path. The nearest neighbor distance provides a decreasingly weight, imposed by the gradient of the line r_i, however it is limited to a minimum. At first straying from the original route is highly rewarded, but this is decreased the further we stray from it. At a certain point we gain no more from it. Our route could stray further, but since we have an limit to our function we will not get any benefit from it. Since the shortest path objective will limit the weight function on the other side, once the desirability function's minimum has been reached, the shortest path that does so will be chosen.

The limits of the desirability function can be varied according to the total distance of the original route. If the original route is longer, we will allow for a larger distance between the original and alternative path.
By combining the nearest neighbor distance (bounded by a desirability function) with the total shortest distance we have created a method to find an alternative path. By combining that alternative path with our original path we create a round trip possibility.

5.3 Adding Amenities

Recreational walking or cycling routes can be more enjoyable if the user has certain resting spots or amenities where they can rest along the way. Fortunately the OSM data contains data about these amenities. The amenities are tagged with a type like cafe, restroom or for example a picnic table. Not all of these amenities are suited as a resting place for the user. That is why we chose the amenities tagged: *cafe, toilet, pub, ice_ cream* and *drinking_ water* as suitable resting spots.

Unfortunately these amenities in OSM are not explicitly bound to a road. Each amenity has a geolocation defined by a coordinate string, but does not have a street name that it is bound to. To make these amenities part of our routing algorithm the amenity has to be part of the roads cost or score function. To solve this for each amenity holding the predefined tags we calculate the closest street segment that is in our database and store it. This way we can incorporate the amenities into our street's cost or score function. We assume that every one of these amenities is closely adjacent to a road so that it can be accessed from that road.

In the algorithms that calculate routing based on distance a distance function is used and we strive the minimum cost from point A to B. By adding amenities we combine this distance function with the amenities. However we have to keep in mind that we want to minimize distance and we prefer amenities. To combine the amenities with the distance we can use the desirability function from Section 5.2 again. When there are no amenities we get a high constant cost which decreases with the amount of amenities on the road. The cost function now becomes a combination of the distance function and the desirability function over the amenities. We can put dynamic weights on distance and the amenities functions and choose the importance and scale of both factors.

5.4 Visualization of Results and User Interaction

An important part of usability is the possibility to visualize our data. This is not only important for our end user but helps to understand our algorithms as well. When creating a graphical user interface or web interface it is important to create an understandable, easy to use and fast environment. To make this environment accessible to multiple devices a web interface is practical. Most web languages are interpreted languages. This means that instructions are executed directly without previously compiling the program beforehand. These languages are good to visualize data and process small amounts of data, but are not fast enough to do large calculations required for our algorithm. To do calculations like this we need a compiled language.

We chose C++ as programming language to do our main calculations. Our visual aspect consists of a Javascript and HTML web layer. These languages however are client side languages. This means that in order to use the software we would have to transfer most of our data to the user which is simply not feasible. This is why we also had to create a server side aspect. This server side aspect is a php script which communicates with the C++ program through inter process communication, more specifically a named pipe. The php script also communicates with the Javascript layer through Ajax and transfers the result of the C++ program to that layer. By combining these technologies we utilize the visual aspects of web languages and the processing power of compiled languages.

The web interface allows the user to lookup any street in the Netherlands. The street is then marked as a source or a destination location. By clicking on plan route the user is presented with the results in a sidebar and he/she can select the preferred route from it and visualize it on the map.

6 Results

6.1 Routing Based on Distance

The four routing algorithms are tested for performance running on a dual-core 2.4 GHz Intel Core i5 (I5-520M) processor with 4GB of DDR3 SDRAM. The code is compiled using GNU g++ version 4.6 using the pqxx library. The times were calculated running each algorithm 100 times and taking the average time it took to calculate each route.

The routes are selected for various reasons. The first two routes (Den Helder - Maastricht) and (Middelburg - Groningen) are near worst case routes. They stretch from the north-west to the south-east and from the south-west to the north-east of the Netherlands. These are some of the longest routes we can imagine as they are all the way cross-country. The routes (Amsterdam - Rotterdam) and (Leiden - Den Haag) are chosen such that they route through a dense network of roads within the conurbation "Randstad". The algorithms will have many roads to process when routing in this area compared to a more rural area. The last routes (Raalte - Vriezenveen) and (Dokkum - Kollum) are chosen as they reside in more rural areas of the Netherlands. This means that there are fewer roads to route around in these areas.

Table 1 shows the results of the performance measurements in milliseconds. The times are the actual running times of the algorithms and do not contain preprocessing times or the time it takes to visualize the routes. Best results are bold.

Source	Target	Distance (km)	Dijkstra (ms)	Bidirectional Dijkstra (ms)	A* (ms)	Bidirectional A* (ms)
Den Helder	Maastricht	276.6	3122	2049	2822	**2047**
Middelburg	Groningen	316.2	3526	2794	2545	**2487**
Amsterdam	Rotterdam	65.4	1390	1246	**613**	983
Leiden	Den Haag	13.6	466	**186**	273	274
Raalte	Vriezenveen	27.4	448	**183**	295	342
Dokkum	Kollum	12.5	178	**61**	142	128

Table 1: Performance routing based on distance

The results clearly show that improvement can be made by using a different approach from classic Dijkstra. We see that all techniques are faster than the original Dijkstra. The results between each speedup technique vary, but it looks like longer distances benefit from the A* and Bidirectional A* methods and Bidirectional Dijkstra seems to work better on slightly shorter distances. An example of the visualized route between Raalte and Vriezenveen is given in Figure 7:

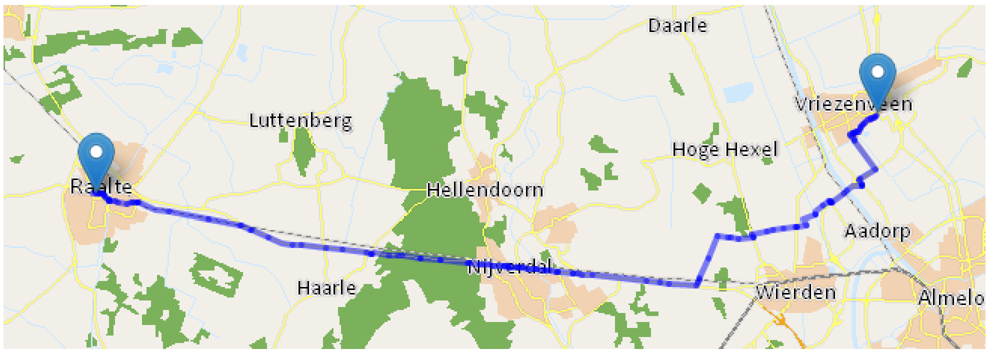

Figure 7: Route visualization example

6.2 Highway Avoidance

To see if our algorithm can actually avoid highways we can not analyze every single route to see if it avoids highways. This is why we test the small subset of routes from Section 6.1 and enable highway avoidance. We then visually analyze the generated routes for usage of highways.

Source	Target	Original Distance (km)	Highway Avoided Distance (km)
Den Helder	Maastricht	276.6	281.5
Middelburg	Groningen	316.2	326.9
Amsterdam	Rotterdam	65.4	65.8
Leiden	Den Haag	13.6	13.74
Raalte	Vriezenveen	27.4	29.5
Dokkum	Kollum	12.5	13.61

Table 2: Highway avoidance

When we visually inspect these routes we see that they all avoid highways thus following the set constraint. An example of this highway avoidance is given in Figure 8 where the source and target are the same as in Figure 7 but highways are avoided. Furthermore we see a small increase in distance from avoiding highways. The increase in distance may only be small because of a dense road network.

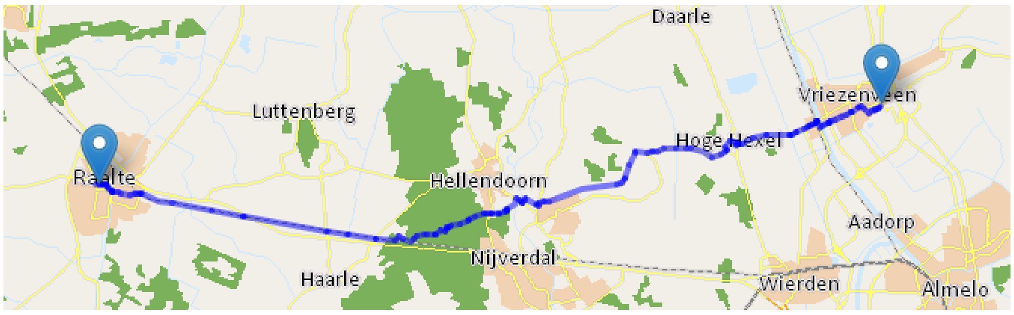

Figure 8: Highway avoidance example

Figure 8 shows a slightly different route. The difference with the original route is around 2 kilometers. This difference is fairly high in rural areas because it has a less dense road network. In other more dense areas cycle paths or small roads exist close to highways, which lead to a smaller distance.

6.3 Alternative Routes

To measure the quality of an alternative route we have to define a good alternative route. [1] defined some aspects of a good route. It must be substantially different from the optimal path and must not be much longer. They must also feel natural to the user, having no unnecessary detours. We extend the definition of “substantially different” a bit. A route should not only be substantially different from the original route by having less overlap, but should also stray away from it. With a dense road network, two roads could easily run parallel providing a close to optimal alternative. When cyclists want to take a roundtrip taking an alternative route back they should be able to take a totally different alternative. This does affect the criterion that the alternative must not be much longer. The more we stray from the optimal path implies that we compromise in distance. For all routes measured in Section 6.1 we calculate two alternatives.

Original Distance (km)	Alternative Distance 1 (km)	Diff (km)	Diff(%)	Alternative Distance 2 (km)	Diff (km)	Diff (%)
276.6	282,8	6,2	+2,2	283,9	7,2	+2,6
316.2	323.9	7,7	+2,4	325.1	8,9	+2,8
65.4	67.9	2,5	+3,8	70.6	5,2	+8,0
13.6	19.3	5,7	+42,2	27.0	13,4	+99,0
27.4	33.0	5,6	+20,4	40.1	12,7	+46,4
12.5	18.2	10,4	+45,6	22.9	10,4	+83,2

Table 3: Alternative route distances

When looking at the percentile difference between the alternative routes and the original we see quite a difference. The differences vary between 2,2 and 99 percent. The absolute distance however only varies between 2,5 and 10,4 kilometers on the first alternative route and between 7,2 and 13,3 kilometers on the second alternative. The absolute distance difference does not increase with the total distance, which causes a smaller percentile difference. This all is caused by our rule that our alternative must differ substantially from the original route and it does so by taking a detour.

When we visually inspect our alternative routes as shown we see that they indeed differ substantially from the original route. The alternative routes show no unnecessary detours. An example is given in Figure 9.

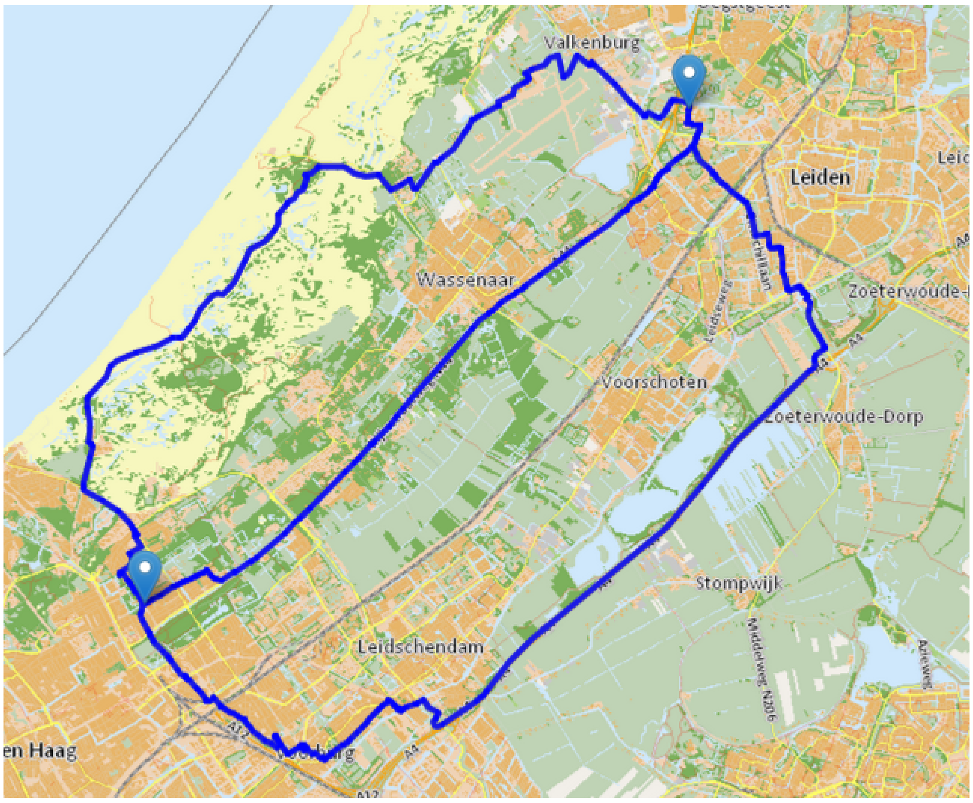

Figure 9: Alternative routes example

6.4 Adding Amenities

Unfortunately the OSM data does not provide a lot of registered amenities. This means that in order to meet amenities we will have to reduce the edge weights of the edge on which the amenities reside. Unfortunately, due to the nature of the algorithms, just reducing the weight only does not guarantee the road being visited. The extra costs involved in leading towards that edge could easily cancel out the reduced edge weight. This caused the algorithm only to visit those edges when it is a close alternative in distance to the original distance. Normally we would not want to travel a long distance just because we can visit a picnic place or toilet, so this method would work. However, due to the lack of amenities the routes differ rarely from those without amenities.

We can enforce visiting an amenity by adding it as a fixed constraint. This is shown in Figure 10. The amenity is shown as a park bench. We can see the direct route from source to target on the right and the route visiting the amenity on the left. The example shows that it does work. We can calculate the difference in length between the original route and the one visiting amenities and set a maximum variation. This way we can enforce amenities in our route.

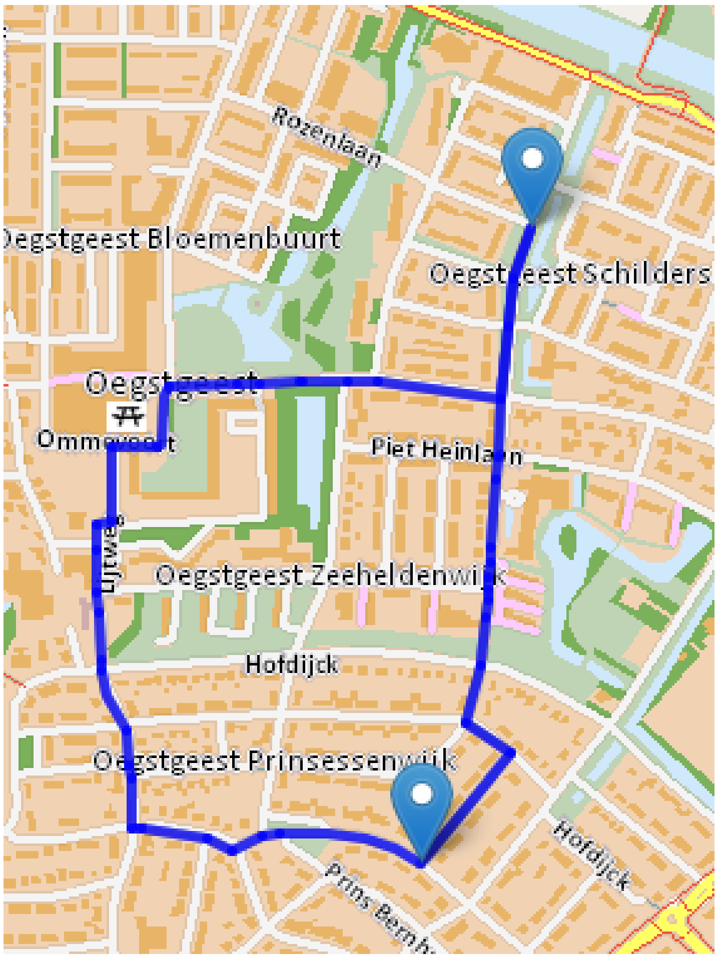

Figure 10: Route with amenity

7 Conclusions and Discussion

We see that by approaching routing in a different way from traditional distance based methods we can generate alternative routes for cyclists that are not only based on distance. The OpenStreetMap data unfortunately is not perfect for routing. It is developed to visually generate maps. By preprocessing the data as seen in Section 3.2 the OpenStreetMap data can however be adapted to do so. By splitting roads into segments and calculating the length of these segments we have created a routable graph. Converting the raw XML data from OpenStreetMap and converting and importing the roads from it into a PostgreSQL database helped for easy access of the data and speedup.

By combining the processing power of a compiler language and the visual aspects and visibility of a web interface we see that we can create a flexible routing interface which performs fast enough calculations to be handled by web users. By porting our software to a powerful web server it can easily be used in an online environment for users to look up a route. By easily selecting the street the user wants to depart from or arrive at and being presented three alternative routes in a simple manner the user can plan his route.

We show that for the Netherlands we can generate routes within 5 seconds, which is reasonably fast. Comparing the Dijkstra, A*, Bidirectional Dijkstra and Bidirectional A* algorithms we see that the latter three are an improvement speed on classic Dijkstra, but there is no clear winner for all routes. To expand this approach to continental size maps other preprocessing techniques could be needed to keep calculating within a reasonable time.

By filtering our database on specific OSM motorway tags we were able to filter out roads that our not suited for cycling. This approach turned out well and only showed a small increase in route length. We should note that alternative roads could however still lie next to motorways, which could make it less enjoyable for cyclists.

A new technique for generating alternative routes and round trips has been presented. We show that we can stray away from the original route and generate a good alternative. On smaller trajectories this caused a detour that was quite a lot longer than the original route percentage wise, but in absolute numbers was not that much longer. On longer routes we saw about the same increase in length, but compared to the total distance it seemed a lot less. It shows that an alternative route that is significantly different does not have to be all that far, just far enough.

We show that we can embed amenities in our algorithm, but have to say we could not test this thoroughly due to the lack of amenities from our data source. This caused our route to diverge a lot to visit just a low amount of amenities. When we decrease the weight on this objective it would not visit any. Having more amenities the algorithms would find routes that are closer to the original one and visit more amenities as well.

The algorithms shown can be extended with other objectives. One could think of measuring the amount of nature (green areas) around the roads or add multiple points of interests. Such a data source, to our knowledge is currently not freely available or is very limited, but could easily be added to the algorithms. When it is the algorithm is flexible enough to easily add these objectives.

The addition of amenities and round trip possibilities focused on bicycle routing, combined with an intuitive simple user interface are an addition to the current market where applications are too much focused on a single shortest distance from source to destination.

References

[1] Abraham, I., Delling, D., Goldberg, A. V., and Werneck, R. F. Alternative Routes in Road Networks. *Journal of Experimental Algorithmics 18*, April (2013), 1–20.

[2] Bouch, A., Kuchinsky, A., and Bhatti, N. Quality is in the Eye of the Beholder : Meeting Users Requirements for Internet Quality of Service. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (The Hague, 2000), ACM New York, pp. 297–304.

[3] Dantzig, G. B. *Linear Programming and Extensions.* Princeton University Pres, Princeton, 1963.

[4] Delling, D., Sanders, P., Schultes, D., and Wagner, D. Engineering route planning algorithms. *Algorithmics of large and Complex Networks 2*, 1 (2009), 117–139.

[5] Derrenger, G., and Suich, R. Simultaneous Optimization of Several Response Variables. *Journal of Quality Technology 12*, 4 (1980), 214–220.

[6] Dijkstra, E. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik 1*, 1 (1959), 269–271.

[7] Eppstein, D. Finding the k Shortest Paths. *SIAM Journal on Computing 28*, 2 (1999), 652–673.

[8] Fredman, M. L., and Tarjan, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM 34*, 3 (July 1987), 596–615.

[9] Haklay, M. How good is volunteered geographical information? A comparative study of OpenStreetMap and Ordnance Survey datasets. *Environment and Planning B: Planning and Design 37*, 4 (2010), 682–703.

[10] Hart, P., Nilsson, N., and Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2 (1968), 100–107.

[11] Hijmans, B. *Multi-objective Generation of Bicycle Routes.* Bachelor thesis, Leiden University, 2013.

[12] Ikeda, T., Hsu, M., and Imai, H. A fast algorithm for finding better routes by ai search techniques. In *Vehicle Navigation and Information Systems Conference* (Yokohama, 1994), IEEE, pp. 291 – 296.

[13] Wagner, D., and Willhalm, T. Speed-up techniques for shortest-path computations. In *Symposium on Theoretical Aspects of Computer Science* (Aachen, 2007), vol. 2, Springer Berlin Heidelberg, pp. 23–36.