Karlsruhe Institute of Technology

Automated Generation of Destination Maps

(Map Sketching with Shortest Paths)

Bachelor Thesis of

Yordan Boev

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Dr. Martin Nöllenburg
	Prof. Dr. Peter Sanders
Advisors:	Dipl.-Inform. Benjamin Niedermann
	Dipl.-Inform. Andreas Gemsa

Time Period: 1 July 2014 – 14 November 2014

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 14th November 2014

Abstract

Main subject of this paper are destination maps and their generation using software. A destination map resembles a hand-made sketch, which purpose is to show a person how to easily get to a fixed destination. Such maps achieve better readability by including just the most important streets and places which directly help the reader navigate to the end point. Our goal is writing an application that uses various algorithms and as little as possible input and interaction from its user to create such maps. Handpicking which elements should be in the destination map is probably ideal, but also requires a lot of time and effort. This is why we believe in the usefulness of an application, capable of analyzing the surroundings for any given point and constructing such map based on different criteria. We are not the first ones to come up with an automated solution for this problem and this paper contains some reviews and references to previous work done on this topic.

After defining what a good destination map is according to us, we will describe the whole process our application follows in order to achieve this result. The paper presents each step of our creation process and reveals the algorithms and ideas that stand behind it. The information flow follows the order of execution in our application, making it easier to follow why and how each step is done.

Contents

1. Introduction

A destination map is a special kind of map that usually covers small to medium areas in size and as its name suggests focuses around a particular *destination point*. Such maps are most likely to be seen on posters, visit cards and web sites and their main purpose is to show the reader how to get to a particular address. Traditional street maps do contain all the information needed to allow a driver or a pedestrian to get where he wants, but reading them requires substantial amount of time and effort. By knowing the destination's location the map can be significantly simplified to contain just the streets that are most likely to be used to get there. Such maps built around a pin-pointed location are called *destination maps* and often resemble a hand drawn sketch. They are not required to be exact as long as the deviation in them does not affect the navigation process. A good destination map can be easily read and interpreted by a person. This is why containing simply shaped routes and less unnecessary details is considered a big plus as long as it does not cause confusion in the map reader. Usually destination maps cover areas not bigger than couple of kilometers in diameter. Some examples created by Bing[1] can be found on Figure 1.1.

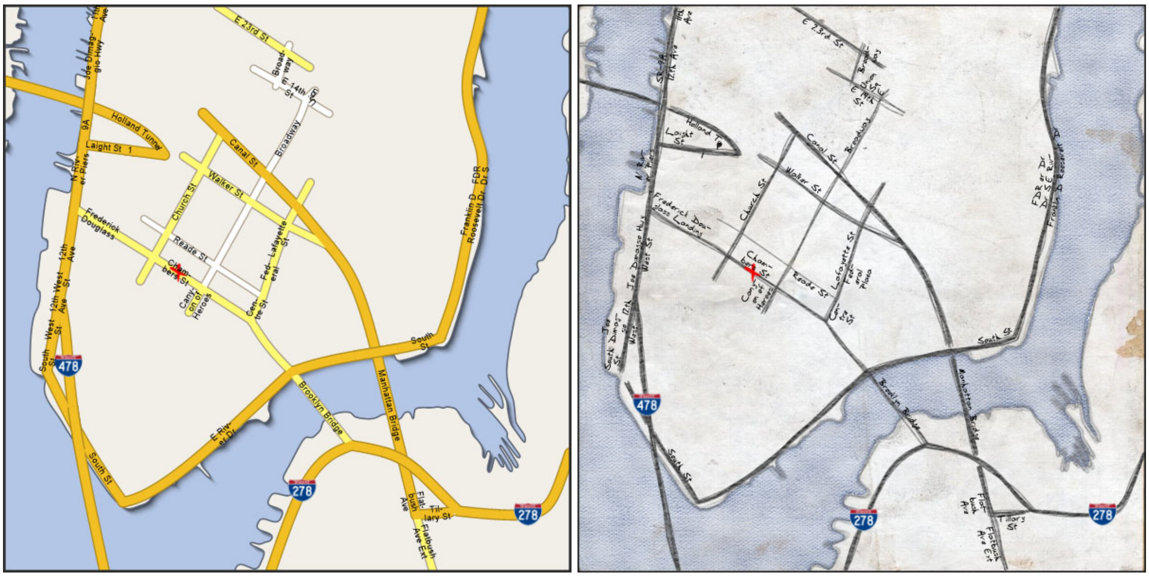


Figure 1.1: Bing Destination Maps.

[1]Bing Maps App: Destination Maps http://www.bing.com/maps/?FORM=Z9LH3

Often people use handmade sketches to describe how to get to a familiar place to someone else. While almost any such drawing can be classified as a destination map, our focus here will be on professionally made destination maps. These can be created by painters and graphic designers, but with the advance of technology, various algorithms and computer programs capable of constructing them have emerged. This thesis describes one possible approach for automated creation of destination maps by presenting a chain of underlying algorithms and analyzing the results each of them produces. We have written a computer program that implements these algorithms and give map data can produce a destination map. It represents a prototype for testing our ideas and we will refer to it as *our application*. Later in this paper we will analyze in depth how this computer application works and what are the steps it take to create a destination map. More information about our prototype along with some screen shots can be found in the examples chapter at the end of the document.
An individually created destination map allows for unique selection of the roads being displayed as well as hand made design that ensures visual pleasure and good readability, but requires a substantial amount of work time from the person creating the destination map. This is why we believe in the usefulness of an application that automatically creates such maps in a matter of seconds. So our problem is determining a series of steps that lead to a destination map that can compete with a hand made one in both usefulness and readability and implement this steps into a prototype application. We are not the first ones to come up with this idea, in 2010 Johannes Kopf et al. [KAS+10] published his vision of automated destination map creation. The process described there can be generalized in three major steps: road selection, road simplification, drawing of the map. Our application for destination map creation also follows these three major steps, but implements them in a different way than Johannes Kopf did using a unique combination of underlying algorithms. The road selection is the first one and is done with the help of shortest paths algorithms and evaluation of each street. It determines the selection of routes being displayed in the map and is the foundation on which the next two steps operate. The path simplification sacrifices detail in order to make the shape of each route that leading to the destination simpler. This is crucial, because it increases readability and helps the map reader easier navigate and remember which streets to take by presenting them with a less complicated curve. The last step is to visualize the resulted graph layout as a sketch. We believe that a destination map created following this model can be worthy rival of a hand made one. The next chapters describe the algorithmic blocks used in our application and follow its work flow by explaining each move it makes toward the final goal - the destination map. There is no constraint on the size of the area on which the application operates, but it usually is not bigger than a country or a vast region. The actual destination map covers a smaller area in this region which is usually a couple of kilometers wide (normally 3-20km, but can go up to 100-200km and more if the user wants). Depending on this size the steps take different amount of time to complete which rarely exceed a couple of seconds. This is completely satisfying when you consider how long it takes to create a hand-made map.

2. Related Work

This chapter will briefly outline previous attempts for constructing an automated destination map tool and any important work considered connected to this project. When we started with the project, there was already a number of computer programs, mostly online applications, capable of creating a destination map or something similar to one [1] [2] [3]. One of the most popular and the one that we will review here closely was created by Johannes Kopf at al. The paper describing how they approached this problem is called Automatic Generation of Destination Maps [KAS+10] and achieves very good results when compared to actual hand-made destination maps. Their solution breaks down the problem into three smaller subproblems and handles them separately:

1. Selecting the relevant subset of roads.
2. Road simplification and improvement.
3. Actual drawing of the map and using textures to improve visual comprehension.

Their first step aims to select a subnetwork of roads that are more likely to help reach the given destination. These three steps describe the approach:

- Visibility rings: They first compute concentric rings of highways, arterials and residential streets around each destination. These visibility rings form the hierarchy associated with navigation to a destination.
- Traversable routes: To produce complete traversable routes, we use a shortest path algorithm to connect the rings to the destination. We also connect all highways entering the boundary area of interest to the destination.
- Road extension: Some selected road segments are extended to provide additional context. For example, if two disjoint segments of the same road have been selected, we add the segments between.

The second step, which is responsible for improving the layout of the selected subnetwork of roads, defines an energy function to do so. This function defines the cost of a layout and is calculated as a weighted (with coefficiets) sum of 5 terms. For a complete explanation of

[1] Bing treasure maps: http://blogs.bing.com/maps/2010/12/09/destination-maps-for-your-holiday-new-years-parties/
[2] Scribble maps: http://www.scribblemaps.com/
[3] Google map maker: http://www.google.com/mapmaker

these function and its terms please refer to [KAS+10, page 6]. The first term punishes a false intersection and increases the cost of the function if the distance between two edge that do not intersect becomes too small. The second term forces each edge to maintain a minimum length on the screen. The next term makes sure the length of every edge stays relative to the surrounding edges and thus help keep the visual ration of the whole sketch. The last two remaining terms keep the orientation of each edge in the layout by punishing deviations and wrong angles. After having this energy function defined and a start layout produced by the previous subnetwork selection they aim at finding a local minimum for this function and thus providing a better view for the reader over the selected subset of roads.
Their final step is to draw the actual map. The approach they take is to draw the subnetwork along with some additional detail included. For example, water or street names. A variety of different texture sets allows for differently looking drawings of the same subnetwork of roads. This makes the generated maps aesthetically pleasing and gives variety of options so that the map can be matched to its purpose.
The approach used by Johannes Kopf at al. can be summarized in three major steps: road selection, road simplification and drawing. This exact model is used by most automated tool for generating special purpose maps, but by putting different ideas and algorithms behind each step, one can shift the focus of the map and get very different results.

2.1 Force Based Graph Embedding

Our approach to improve the street layout[4] on the destination map is inspired by spring embedder techniques and force based graph drawing algorithms. These algorithms are usually used to force a graph to spread over a certain area and thus improve its visual representation. One of the first examples of a spring embedder is the 1984 algorithm of Eades [Ead84]. It uses a mechanical model where vertices are replaced by steel rings and edges are represented by stretched strings. After being placed in some initial layout the springs forces move the rings so that the energy of the whole layout is minimized. The spring forces are implemented by defining attractive and repulsive forces between the vertices. The size of these forces depends on the distance at which the vertices currently are and the size of the available place where the graph is being drawn. The idea is that vertices that are connected with an edge and are further away should attract each other in order to minimize the edge's length. Vertices not connected with an edge should repel each other. This is done by defining repulsive forces between such vertices and aims to make the graph spread and use the available space given efficiently.
This idea is further developed by Fruchterman and Reingold in 1991 [FR91]. The attractive (f_a) and repulsive (f_r) forces in their embedder are defined as follows:

$$f_a(d) = d^2/k,\ f_r(d) = -k^2/d,$$

where the variable d is the distance between the two vertices and k is defined as:

$$k = \sqrt{\frac{available\ area}{number\ of\ vertices}}$$

and is referred to as optimal pairwise distance. These embedders work by starting at some initial layout that is most likely not optimal and can be significantly improved. They achieve this by calculating the energy for the current layout as sum of the forces applied at each vertex. By performing a number of iteration they aim to minimize the whole energy of the layout and thus provide a aesthetically pleasing distribution of the vertices. At each

[4] Under street layout we understand a graph representation in a 2D euclidean plane.

iteration the forces are newly calculated at each vertex, which is then displaced (moved) a little according to the direction and the strength of these forces.
The spring embedders and force based graph layout algorithms have developed greatly as of today. There exist many variations that involve using different forces, predefined rings, k-centers approximation, predefined local neighborhoods. A summary of these techniques and algorithms is provided by Stephen G. Kobourov in his Spring embedders and Force Directed Graph Drawing Algorithms from January 2012 [Kob12].
Most of the spring embedding algorithms presented in Kobourov's paper operate on unweighted graphs and just aim to spread the graph efficiently by starting at some initial layout that is randomly generated (vertices are placed at random locations and edges are drawn as lines between them) and we cannot say much about this initial state. This presents a difference with our line of work. Our graph layout consists of points (vertices) and streets (weighted edges) and already finds itself in a reasonably good state, since they are not placed randomly. We can't apply the force embedders reviewed by Kobourov directly, because they are not designed to work on such special case and would change the graph layout in an unexpected way. Vertices and edges should not be moved/altered too aggressively and the initial layout shape should be preserved. We achieve this by changing the model a bit and defining our forces in such a way that they work for weighted graphs. Most spring embedders just aim to place the vertices in some efficient way, but our task demands that we keep the initial shape of the map and that vertices keep their relative position according to adjacent vertices. This is important because, after all the destination map is used for navigation and must not deviate from the real map up to a point where it is no longer useful. We achieve this layout transformation by defining forces and iteratively adjusting vertices according to these forces.

2.2 Path simplification

In order to achieve simplicity and make our map easier to read we are going to need a path simplification algorithm that transforms an embedded path in the plane into a similar one with less detail (less lines). As Andreas Gemsa et al. state in the paper "On d-regular Schematization of Embedded Paths" [DGN+14] the problem of path simplification in the field of cartography is well studied and one of the most popular solutions is presented by Douglas and Peucker [DP73]. This algorithm, also known as the *split-and-merge algorithm*, takes as input a polyline (a continuous line composed of one or more line segments with points at the end of each segment) and returns a simplified curve that contains only a subset of the points defining the original one. The algorithm the the divide and conquer technique and works as follows:
The algorithm accepts all points from the polyline and automatically marks the first and the last one to be kept. It then analyzes the points between the first and the last one and selects the one that is furthest from the line segment defined by these two end points. If this distance is smaller than a predefined ϵ then all points except the first and the last can be discarded and the polyline can be represented just by the line between the two endpoints. However if the distance is greater than this ϵ the furthest point is included in the simplification and the same procedure is called recursively for the two sub-polylines in which the original polyline is divided by this point. So in this case the algorithm calls itself one time with the start point and the furthest point and another time with the furthest point and the end point (thus marking the furthest point as kept). The naive algorithm implementation has a worst-case complexity of $\mathcal{O}(n^2)$).
The Douglas-Peucker algorithm is a classic example of a polyline simplification algorithm that provides approximation of the original line. We have chosen to use a similar algorithm with a different criteria that say which inner points from the polyline get removed. The

idea is to measure the area difference between the actual polyline and the approximated one. The algorithm is described in Chapter 6.

3. Preliminaries

In this chapter we summarize the algorithmic knowledge and the terminology needed to understand the mechanisms and solutions described in the next chapters.

3.1 Graph related knowledge

The reader is expected to have basic knowledge of graph theory. Our main focus will be Dijkstra's algorithm [Dij59] because later on we are going to use a slightly modified version of it. Dijkstra's algorithm operates on a weighted, oriented graph, which is defined as $G := (V, E, \omega)$. In this definition V represents the set of *vertices*, which are also sometimes called *points*. The set E represents the edges between the nodes. An edge $e = (v_1, v_2)$, where $v_1, v_2 \in V$ is simply a pair of two nodes and denotes that there is a connection from the first to the second one. The last component gives our graph its 'weighted' property and is a cost function $\omega \to \mathbb{R}_{\geq 0}$ which assigns a numerical value to every edge. Generally the cost function does not have to be non-negative, but for convenience reasons and the fact that our graphs will all have non-negative edge costs, we will assume that ω is not negative. This property is also required for the next algorithm to operate correctly.

3.1.1 Dijkstra's algorithm

Dijkstra's algorithm [Dij59] was introduced and published in the 1950s by Edsger Dijkstra as a solution to the single-source shortest path problem. For given graph $G = (V, E, \omega)$ and a source vertex $s \in V$, this algorithm computes the shortest paths from this initial vertex s to every other reachable vertex in the graph. Dijkstra's algorithm starts its computation at the source vertex s and maintains two functions: *dist* and *pred*. The *dist:* $V \to \mathbb{R}_{\geq 0}$ function returns for every vertex the minimal length of the path from the source vertex to this vertex found so far. If no path to a certain vertex $v \in V$ is jet discovered, then $dist(v) = \infty$. Note that the values of the *dist* function change throughout the computation as new paths are discovered or as old ones are improved. As the name itself implies, the *pred:*$V \to V$ function gives for each vertex its predecessor on the shortest path from the source vertex to it. The *pred* function is only used to reconstruct the found shortest paths as sequence of nodes and edges. Consider:

$$s \to v_1 \to v_2 \to \cdots \to v_{k-1} \to v_k$$

to be the shortest path from s to v_k, then:

$$dist(v_k) = \omega(s, v_1) + \omega(v_1, v_2) + \cdots + \omega(v_{k-1}, v_k)$$

and

$$pred(v_1) = s, pred(v_2) = v_1 \ldots pred(v_k) = v_{k-1}$$

Algorithm 3.1: DIJKSTRA

Input: Graph $G = (V, E, \omega)$, source node s
Data: Priority queue Q
Output: Distances dist(v) for all $v \in V$, shortest-path tree of s given by pred($\cdot$)

```
   // Initialization
1  forall v ∈ V do
2      dist(v) ← ∞
3      pred(v) ← null
4  Q.INSERT(s, 0)
5  dist(s) ← 0
   // Main loop
6  while Q is not empty do
7      u ← Q.DELETEMIN()
8      forall (u, v) ∈ E do
9          if dist(u) + ω(u, v) < dist(v) then
10             dist(v) ← dist(u) + ω(u, v)
11             pred(v) ← u
12             if Q.CONTAINS(v) then
13                 Q.DECREASEKEY(v, dist(v))
14             else
15                 Q.INSERT(v, dist(v))
```

Before Dijkstra's algorithm starts its computation the initial values of the functions *dist* and *pred* are defined as follows (lines 1-5, algorithm 3.1):

$$dist(s) = 0 \text{ and } dist(v) = \infty, \text{ for every } v \in V, v \neq s$$

$$pred(s) = -1 \text{ and } pred(v) = \texttt{null}, \text{ for every } v \in V, v \neq s$$

At the start of each iteration the algorithm choses a node $p \in V$ so that:

- p has not been chosen in any of the previous iterations
- $dist(p) \neq \infty$
- the $dist(p)$ value is the smallest among all vertices that fulfill the previous two requirements

We are going to call p an *update-vertex* for the current iteration (line 7, algorithm 3.1). It is easy to see that on the first iteration the update-vertex is always s, because it's the only one with $dist \neq \infty$. Furthermore every vertex is chosen to be update-vertex at most one time. After we have selected p as the update-vertex for the current iteration it is then certain that the value of $dist(p)$ equals the cost of the shortest path between s and p in our graph G and cannot be improved anymore. In practice, this property is often used to terminate execution earlier and avoid unnecessary computations.

The iteration is then completed by updating the values of the *dist* and *pred* functions(lines 8-15, algorithm 3.1). We do this by iterating over edges $(p, q) \in E$ and updating our sets as follows:

$$dist(q) = \min\left(dist(q), dist(p) + \omega(p, q)\right)$$

If we have improved the value of $dist(q)$ by using the edge (p, q), then we also set p to be the predecessor of q:

$$pred(q) = p$$

The algorithm terminates if no update-vertex can be selected in the current iteration. As we stated above at each iteration we choose an update-vertex and every vertex can be used only once as update-vertex. This gives us $|V|$ iterations at most. On each iteration we have to select this update-vertex, which can be done in $\mathcal{O}(\log(|V|))$ steps by using a min-heap to efficiently find the smallest among all elements of the *dist* set. The update phase which is then executed examines every outgoing edge and therefore requires at most $|E|$ steps. Because of the fact that each edge is examined only once we get the amortized running time of $\mathcal{O}(|E| + |V| \cdot \log(|V|))$. This running time is possible due to Fibonacci heaps and is first presented by Fredman and Tarjan in 1984 [FT87].

3.2 Geometry and computation

The reader is expected to have basic knowledge of simple geometry objects (points, vectors, angles), coordinate systems and different computations and operations connected with them. In our course of work we will only use two dimensional coordinate systems. A point in such a coordinate system is defined by two coordinates: (x, y). They uniquely determine a point's position. A vector is a geometric object that represents a direction and is defined by a single point from our coordinate system: $\vec{v} = (x, y)$. It can be seen as a ray from the origin of the coordinate system to this point. Each vector has a length, which we will denote with $|\vec{v}|$. If you have two points $p = (x_1, y_1)$ and $q = (x_2, y_2)$, then the vector from p to q is computed as $\vec{pq} = (x_2 - x_1, y_2 - y_1)$. Assuming $\vec{a} = (x_a, y_a)$ and $\vec{b} = (x_b, y_b)$ are vectors, here are the basic operation on them:

- The length is given by $|\vec{a}| = \sqrt{x_a^2 + y_a^2}$
- Scalar product (also known as dot product):

  $$\vec{a} \cdot \vec{b} = x_a x_b + y_a y_b$$

  If the angle between the two vectors is donated by θ, then the following equation holds:

  $$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos(\theta)$$

- Normalization is the process of setting a vector length to 1 by scaling its coordinates. It is simply done by calculating the vector's current length and dividing each coordinate by this length.

  $$\vec{a} \xrightarrow{normalize} \left(\frac{x_a}{|\vec{a}|}, \frac{y_a}{|\vec{a}|}\right)$$

  The normalized vector of the vector $\vec{a}$ is expressed as $\hat{a}$.

3.2.1 Latitude and Longitude, Haversine Distance

The (Latitude, Longitude) pair uniquely defines a position on the earth's surface and allows us to effectively distinguish between different places around the globe in the context of coordinate systems. In our course of work this pair is mostly used to locate and display important nodes from the traffic network in our area of interest.
Another application for the (Lat, Lon) pair is the *Haversine distance* formula.

$$d_{Hav} := 2R \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Where:

- (ϕ_1, λ_1) and (ϕ_2, λ_2) are the (lat, lon) coordinates of the two points
- R is the radius of the sphere, in our case it is the earth's radius

The formula above gives the surface distance between two points located on the surface of a sphere. Since the earth is not a perfect sphere this formula is not completely accurate, but the error is almost negligible and the approximation delivered by the haversine distance formula is better than the one given by euclidean distance.

4. Modeling

Our main task is to show the best ways to get to a certain destination, so before actually stepping into solving this problem we needed information on the roads around our point of interest. This will allow us to extract and bring up the best routes leading to our destination. In this chapter you will learn where our road information comes from, how is it extracted and turned into a graph representation of the form $G = (V, E, \omega)$. This part serves as an introduction and basis for the algorithms and ideas we are going to use later on.

4.1 Data representation and internal constructs

The very first step our solution requires is importing road information about a certain region in the world. This region can vary in size and be as small as a single city or as large as several countries or continents. All the road information we need about this particular area we operate on is stored in raw data files and is imported at the very start, every time the application is run. The files mirror the internal graph representation and allow us to quickly build up our *road network*.

4.1.1 The Road Network

The road network can be viewed as a graph representation of our area of interest. The formal definition of this term is the same as the one we gave for a graph in the previous chapter - ($\textit{Road Network} := (V, E, \omega)$). It contains all road information we have on the region of interest. Crossroads, road twists and road turns are all represented by vertices, while streets are modeled as edges between those vertices. In our case it is important for the edges to be directed, because sometimes street allow movement only in one direction and we need to be able to reflect that. Here are the building blocks and special features of our road network:

- Vertices: they represent crossroads, junctions and help shape road twists. Our destination point, for example, is always a valid vertex in our road network.
    1. Latitude, Longitude - each vertex has a unique *(lat, lon)* pair, which allows it to be displayed on the world map using the Google Map API [1]. This pair is also used later on when calculating the picture coordinates needed for drawing the actual destination map.

[1] The Google Maps API is a programming interface for working with maps https://developers.google.com/maps/?hl=de

- Edges: these represent streets from our region of interest as connections between valid vertices. Here are the most important properties of an edge:

  1. Direction - just as we mentioned earlier the edges in our graph are directed, but they are also additionally classified as *forward* or *backward*. This means that if the forward edge $v \to w$ exist then our road network also contains the corresponding backward edge $w \leftarrow v$. This additional property of our graph representation is needed for the modified version of the Dijkstra's algorithm.

  2. Network Level - each edge is assigned an integer value from 1 to 8 which represents the road category (for full description of road category see Appendix**??**). From now on we will call this number *network level*. Smaller values like 1 or 2 state that the road is large and important (highway or motorway for example). As the network level gets larger the roads get less important. Very small residential or access roads have network level values of 7 or 8. This edge attribute just represents the road classification by size in our road network. The network level corresponds to the *highway* tag for streets in the Open Street Map Model [2].

  3. Weight - the cost to traverse an edge e is calculated by the following formula:

$$\omega(e) = \max\left(3.6\left(\frac{d_H(e)}{V_{max}}\right), 1\right)$$

     where:

     - $d_H(e)$ is the haversine distance between the two ends of the edge
     - V_{max} is the maximal velocity allowed to traverse this edge. This value is given and extracted from the OSM model as maximal speed for a Way.

     As you can see above, the minimal cost for traversing and edge in our road network is 1. We have done this to In the early stages of the project we used just the formula above. After testing and assessing the results, we realized that sharper distinction between the different network levels could lead to simpler and easier routes. We implemented this enhanced distinction by adjusting the cost of an edge to depend on its network level. This is the final version of the cost function our road network uses:

$$\omega_{netlvl}(e) = \omega(e)\left(1 + c_{netlvl}\frac{networkLevel(e)}{2}\right)$$

     where:

     - c_{netlvl} is a constant determining the magnitude of the distinction. It is given by the user and varies from 0 to 2, where 0 means $\omega_{netlvl}(e) = \omega(e)$
     - $networkLevel(e)$ is a function giving the network level value for a certain edge from the road network.

[2] highway tag of OpenStreetMap Way http://wiki.openstreetmap.org/wiki/Key:highway

4.2 Open Street Map parser

The raw data files we talked about in the previous paragraph contain all the information needed to build our road network and are the only thing our application needs as input in order to do its work. The only problem was they didn't exist at the start. We basically needed to read the road information from somewhere else and parse it to our file format. This step is done just one single time, but it's of great importance, because it determines the quality of the data we work with. Comparing some proprietary transport data and the one available at Open Street Map [3] we found out the second one was more up-to-date and contained better street classification. We decided on using the OSM data, because road categorization is a very important aspect when it comes to determining which routes to our destination are good and which should be avoided.
The next step was pretty clear - we had to build a parser that reads the formats provided by OSM (.pbf and .osm) and saves the road information we need into our file format. The parser itself is very simple and can only be operated via a command line interface. In order to read the files provided from OSM, we use the ReadOSM API [4]. It works by reading all elementary items composing the OSM model - Nodes, Ways, Relations and Info-tags. Then it stores these in a database using temporary tables and allows the user to access the stored information via specially defined functions.

[3] Open Street Map is an open source map of the world. http://www.openstreetmap.org
[4] ReadOSM is a C/C++ API for accessing Open Street Map data. https://www.gaia-gis.it/fossil/readosm/index

5. Road Selection

The problem we are going to solve in this chapter is the selection of the routes that are going to be included in our destination map. Formally, this can be expressed in extracting a much smaller sub-graph $G' = (V', E')$ from our road netowrk $G = (V, E)$. This new graph G' represents a road selection, that is focused on the destination point and shows various ways for reaching it. We compute the sub-graph G' by evaluating our road network carefully and selecting just a few streets we consider important and relevant to our destination. This selection should provide a subnetwork just big enough to contain the fastest and simplest way to reach the destination, while avoiding unnecessary details. Having too much details presented brings our destination map closer to an actual map can hurt its readability and effectiveness. We compute this selection using two major steps. First we calculate the subnetwork of optimal paths according to the edge cost function. Then we use this subnetwork and extract a finer road selection of routes by defining additional requirements and using filtering algorithms.

Before jumping to our actual solution we did some testing in order to help us better understand the shape of the subnetwork G' and help us define our requirements towards it. The testing itself involved contraction hierarchy [1] and Dijkstra algorithms and provided us with an idea what a good street subnetwork would look like. Having this as starting point we move forward by defining a set of requirements towards our selection and construct an algorithm that produces efficient subnetworks for a given destination by including just the right amount of detail (edges and vertices).

So our subnetwork should represents a wisely selected collection of streets (edges) from our road network (graph), which purpose is to simplify the process of finding a route to a given destination. The first and the most obvious requirement towards this subnetwork is having the streets follow connected paths that all lead to the final point. Having a series of streets leading away from the destination, ending nowhere does not actually help one get where he wants. We want our automatically generated destination map to be as efficient as possible. This leads us to our next requirement which is expecting our subnetwork to contain the fastest (in our case, with minimal edge cost) route possible from our standpoint to our destination. This translates into our selection consisting of only shortest paths. So for a certain location (vertex) in our generated map, the subnetwork displayed should contain the fastest possible way leading from this vertex to the destination point. Having

[1] As stated by Robert Geisberger, Peter Sanders et. al. [GSSV12], contraction hierarchies allow for extremely fast routing with shortest paths in large networks by doing a preprocessing that involves adding shortcut edges and building a hierarchy among the vertices of the network.

this requirement alone allows us to already extract a possible solution candidate - the subnetwork of shortest paths to our destination vertex. This subgraph is too vast and dense to allow for easy and quick readability, it is much smaller than the whole road network and presents a good basis for determining the optimal subnetwork of streets G'. Having just a destination vertex and a graph we now face the problem of determining the shortest path from each vertex to this chosen destination.

5.1 Shortest path subnetwork and our Dijkstra's algorithm

The only thing we have as a starting point is our destination vertex, which will serve as a source vertex in our version of the Dijkstra's algorithm. In its original form the algorithm computes the shortest paths *from* a given destination *towards* other vertices. In our case we have this part reverted and want to compute shortest paths from the vertices towards our destination. In order to achieve this we will make use of the addition edge property of forward and backward edges that our road network possesses. To compute the shortest "incoming" paths, we allow our version of the algorithm to consider only edges marked as *backward* when it updates the distance function *dist*. So our algorithm for determining the subnetwork of minimal cost is simply the Dijkstra's algorithm on the reversed edges in G. On each iteration when an update vertex is selected, the algorithm will update the *dist* and *pred* functions by assessing only the incoming edges at this update vertex. Following this computational model with the destination vertex as start vertex gradually expands the subnetwork of shortest paths with each iteration.

Algorithm 5.1: DESTINATION MAP DIJKSTRA

Input: The Road Network $G = (V, E, \omega)$, destination node s, maximal operational radius *maxRadius*, network level coefficient c_{netlvl}
Data: Priority queue Q
Output: Distances dist(v) and predecessors pred(v) for all $v \in V$, with $d_{Hav}(v) < maxRadius$

```
   // Initialization
1  forall v ∈ V do
2      dist(v) ← ∞
3      pred(v) ← null
4  Q.INSERT(s, 0)
5  dist(s) ← 0
   // Main loop
6  while Q is not empty do
7      u ← Q.DELETEMIN()
8      forall (u, v) ∈ E, where (u, v) backward edge do
           // Don't update if u is outside of our operational radius
9          if d_Hav(u) > radius then
10             continue
11         if dist(u) + ω(u, v) < dist(v) then
12             dist(v) ← dist(u) + ω(u, v)
13             pred(v) ← u
14             networkLevel(v) ← networkLevel((u,v))
               // Update queue respectively
```

After algorithm 5.1 completes computation the *dist* function stores the minimal distance from a certain vertex to our destination an the *pred* function allows for this path to be

reconstructed.
Until now we have only defined the network level for an edge, but on line 14 in the pseudo code we compute the network level of a vertex, based on the next edge we have to traverse on our route from this vertex to the destination point. We will later use the network level property of a vertex to further refine and improve our selection of important routes.
Because our destination map normally covers a few kilometers in diameter, we can safely stop expanding our subnetwork of shortest paths if we exceed some given limit (lines 9-10). We are going to call this limit *maximal operational radius*. It allows us to concentrate our focus only around the destination point and avoid unnecessary computation that will not be used in any of our next steps at all. This improves greatly the running time of our modified version of Dijkstra's algorithm.
An edge cost can be influenced by the parameter c_{netlvl} as described in 4. When this parameter is set to 0 an edge's cost does not depend on the road classification of the edge. Increasing this parameter punishes edges marked as parts of smaller streets by increasing their cost. This can be used to influence the behavior of our version of the Dijkstra's algorithm and make it prefer bigger, major roads. Both this parameters (the maximal radius and the c_{netlvl}) can be changed by the user in our application and provide additional flexibility. This algorithm alone offers a reduction of the road network into a subnetwork of streets important to our destination point. But as you can see on Figures 5.1 and 5.2 the level of detail presented and the density are too high to allow an efficient human reading on how to get there. The Dijkstra alone cannot produce an efficient subnetwork G', because it simply contains too much detail which will confuse the reader when trying to navigate to a destination.

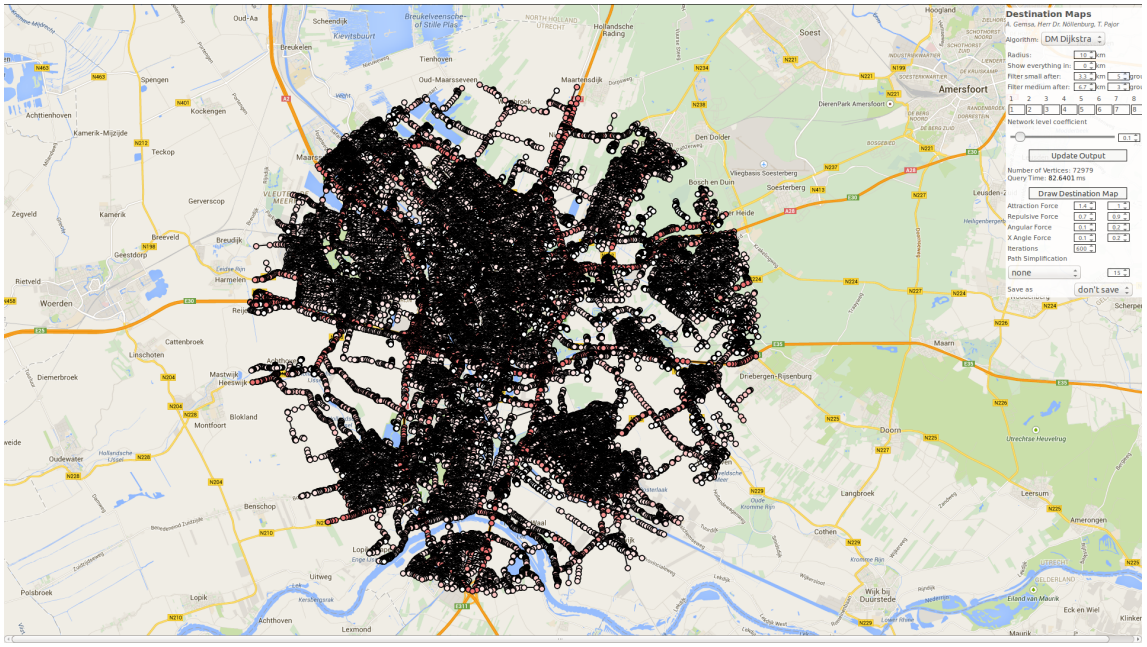


Figure 5.1: Subnetwork of shortest paths, radius 10 km

5.2 Vertex filtering and route reconstruction

We have already narrowed down our selection G' to the subnetwork of the minimal routes leading to the destination point. He we will refine this selection even more and narrow is down to just a few edges and vertices that can be processed by a human being efficiently. Our goal is to achieve just the right level of detail in our generated map. By displaying just enough routes leading to our destination we guarantee usefulness, independent of where

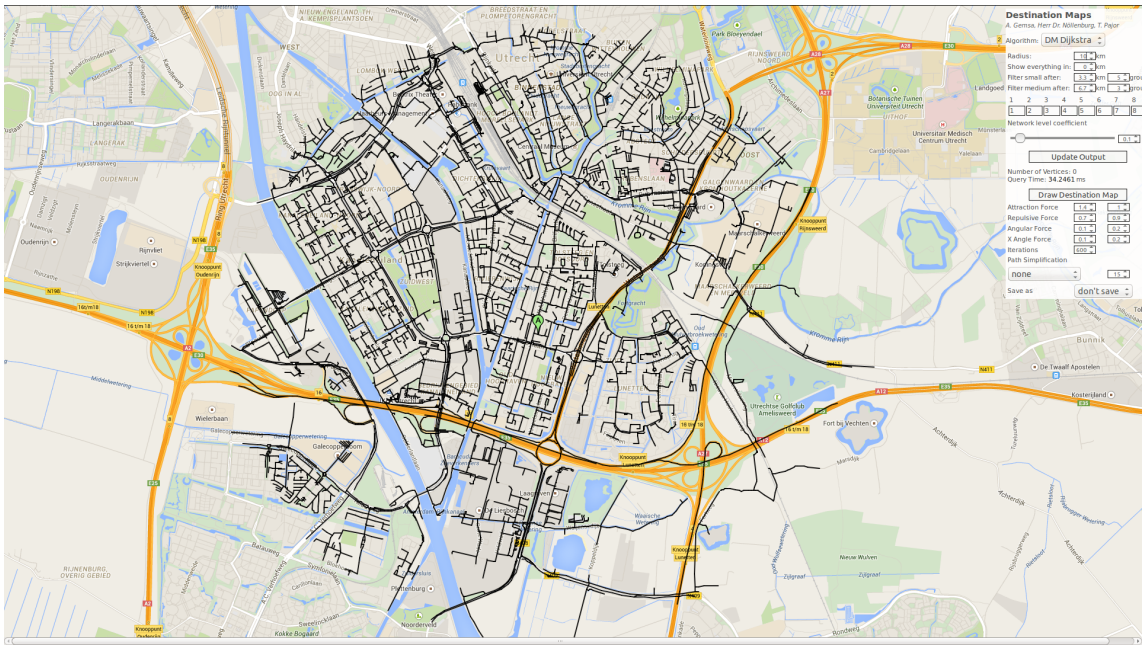


Figure 5.2: Same subnetwork, only edges displayed

the reader is and allow for a quick read and navigation at the same time. We are going to transform the subnetwork produced by algorithm 5.1 into a human-readable map by applying filters and defining additional constraints.
The main purpose of a destination map is to help navigating to a particular location as you draw near it. Taking this property into consideration already gives us the idea that the level of detail presented on the map should increase as we close on our destination point. This means that the outer regions of the sketch should contain only major roads that lead to the area around our destination. As we approach this destination the network level of the streets should become more diverse and allow for smaller roads to participate in the navigation. To better illustrate this imagine a destination located in a neighborhood of a city. A driver is most likely to enter the city or area using a major road and does not need the unnecessary distraction of dealing with smaller roads when still away from the destination. As he enter the city interior however he will be forced to use smaller and smaller roads until he finally arrives at the destination point. We want this exact navigational approach to be resembled on our sketch by presenting just the right level of detail according to how close to the end point we are. Our idea is to select start vertices based on their haversine distance to the destination point and the network level they posses. Each of this start vertices will serve as a something similar to a sample point and will help determine important routes leading to the destination from different directions. After marking a vertex as a start vertex we add the route from this vertex to the destination to G'. We use a model involving circular filter that cover different areas around our destination. Using these filters we are able to choose just a few vertices that are very likely to be start points for a person willing to get to the destination point. The circular filters are defined as follows:

- *No-filter radius* (N_{radius}) - this defines a small circle area around our destination where no filter will be applied and all vertices will be selected.

- *Medium filter radius* (M_{radius}) - this defines the distance after which we select a certain vertex if its network level is smaller than a given constant. We will name this constant M_{netlvl}.

Algorithm 5.2: SUBNETWORK FINE SELECTION

Input: The Road Network $G = (V, E, \omega)$, destination vertex s, maximal operational radius *maxRadius*, filter parameters N_{radius}, M_{radius}, M_{netlvl}, B_{radius}, B_{netlvl}, the functions *dist*, *pred* and *networkLevel* computed by the Dijkstra's algorithm 5.1

Data: Set of edges E_{added}

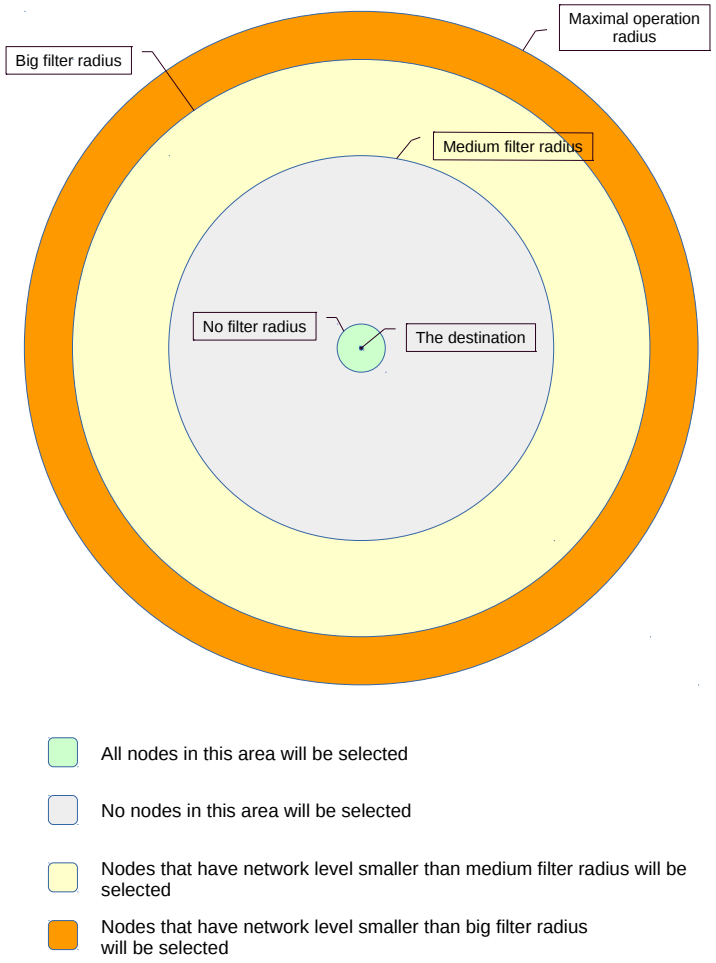
Data: Set of vertices V_{added}

Output: Lists of vertices $V_{displayed}$ and list of edges $E_{displayed}$ going to be part of the destination map.

```
// Main loop
forall v ∈ V do
    if dist(v) < ∞ then
        // Apply filters
        if d_Hav(s, v) > N_radius then
            if d_Hav(s, v) < M_radius then
                continue
            if d_Hav(s, v) >= M_radius and d_Hav(s, v) < B_radius and
            networkLevel(v) > M_netlvl then
                continue
            if d_Hav(s, v) >= B_radius and netlvlv > B_netlvl then
                continue

        // Here the vertex v is marked as selected
        // We add the route from v to s
        currentVertex ← v
        while pred(currentVertex) ≠ −1 do
            if E_added.CONTAINS((currentVertex, pred(currentVertex))) then
                break
            E_added.INSERT((currentVertex, pred(currentVertex)))
            V_added.INSERT(currentVertex)
            currentVertex ← pred(currentVertex)

        // Calculate some interesting vertex information
        info ← calculateVertexData(V_added)

        // Now we fill the V_displayed and E_displayed lists accordingly
        forall u ∈ V_added do
            if in degree of shown edges of u ≠ 1 then
                V_displayed.ADDLIST(u, shown = true, info(u))
            else
                V_displayed.ADDLIST(u, shown = false, info(u))

        forall e ∈ E_added do
            E_displayed.ADDLIST(e)
```

- *Big filter radius* (B_{radius}) - this defines a distance even bigger than the previous medium radius and also comes with a network level constraint constant B_{netlvl} that is even stronger (smaller) than the previous M_{netlvl}.

Figure 5.3: Circular filters



Having this constants defined we end up with donut-like areas around our destination from which we select vertices based on the values of the network level constraints M_{netlvl} and B_{netlvl}. Figure 5.3 on page 20 illustrates in detail how this model works. The circular filter and the start vertex selection are implemented in lines 3-9 in algorithm 5.2.
The application user can change the parameters used in the model above and adjust the subnetwork G'. A change in this parameters does not require running the Dijkstra's algorithm again and presents a cheap and efficient way to shape and refine the output. This feature is important, because the landscapes and the road networks differ greatly in size, structure and density. For example if you find the extracted subnetwork too overcrowded and hard to read, you can set stricter values for the two constants M_{netlvl} and B_{netlvl} and reduce the number of selected vertices. Another good example would be if we need more detail close to the destination. By default the no-filter radius N_{radius} is always zero and by increasing it a little we are able to add this additional information just around our goal.
At this point we have done two things:

1. We have executed Dijkstra's algorithm on our road network and computed the shortest routes to our destination for every vertex in a given radius.

2. We have selected a few vertices in this given radius using the three circular filters described above.

Our next and final step towards selecting what should be on the map is constructing the actual routes that lead to the endpoint. We do this by including the shortest path from each of the selected vertices to our destination. This is a very simple task, since we have already computed the values of the *pred* function. We just have to follow the chain of vertices and edges provided by it and add each vertex and edge to our final result (lines 10-16 in algorithm 5.2). Many of these paths overlap or completely contain other paths in them. If we reach such point we don't actually need to follow and add a path that has already been processed (lines 12-13). This is possible because of the shortest path property that says that *sub-paths of shortest paths are also shortest paths.*

6. Drawing and refinement

At this moment we have the selection of important roads that will be displayed on our destination map. The vertices are represented as latitude-longitude pair and the streets as edges between two vertices. The current task is to translate this extracted information into a routing sketch. We do this in three major steps: coordinates calculation, simplification and layout improvement. Since we have to draw the destination map somewhere we need something like a virtual sheet of paper. We will call it *drawing surface* and it just represents a rectangular shaped window where the destination maps will be shown. Our first step toward drawing the sketch is to transform the lat-lon coordinate pairs of each vertex into picture coordinates for our drawing surface. Then we move forward by simplifying the shape of the roads by applying an area-based algorithm to each polyline on our drawing surface. The final step is to adjust the position of each vertex, so that the destination map is easier to read. This is done by defining forces between the vertices and running an iterative spring embedder that changes the layout of the drawn graph.

6.1 Preprocessing - coordinates transformation

The first step we take is to calculate the coordinates for our drawing surface. It represents a rectangle with width and height, which we denote with F_{width} and F_{height}. Our task now is to find a projection (function) that, for each vertex, takes its lat-lon pair and calculates new coordinate for this drawing surface. We are going to call the newly calculated coordinates *original picture coordinates* and the projection calculating them will be represented as a function $\pi\colon\ (lat,\ lon) \rightarrow (x, y)$ with $0 \geq x < F_{width} and 0 \geq y < F_{height}$. In order to describe how the coordinate transformation works we will need the values of the maximal and minimal latitude and longitude among our vertices. We name this values $lat_{min}, lat_{max}, lon_{min}, lon_{max}$ and together they define a bounding box, which contains all vertices. The work of our projection function is described by:

$$\pi(lat, lon) = ((lon, lat) + \vec{t})r$$

where: the vector $\vec{t} = (-lon_{min}, -lat_{max})$ defines a translation and r is a rescale ratio calculated as:

$$r = min\left\{\frac{F_{width}}{lon_{max} - lon_{min}}, \frac{F_{height}}{lat_{max} - lat_{min}}\right\}$$

The vector $\vec{t}$ sets the bottom-left corner of our bounding box to $(0, 0)$. After that we multiply the translated coordinates with the parameter r, which simply rescales the bounding box

(and therefore the vertex coordinates) to fit in the rectangular drawing surface.
This method of transforming lat-lon coordinate pairs to a standard Cartesian coordinate system comes with a small visual error because the shape of the earth resembles a sphere and the coordinate system is just a rectangle. For our purposes however and given the fact that our bounding box does not cover vast areas (normally just few kilometers in diameter) this method is sufficient enough and does not harm the visual outcome.
In the previous chapter we worked on a road network represented by a weighted, directed graph. After calculating the picture coordinates and projecting this graph onto the drawing surface, we start to look at it as a geometrical structure, rather than a road network. In this chapter the vertices are simply represented by point on the drawing surface and an edge is a just a line between two points and possesses no direction.

6.2 Path simplification

The roads represented on the drawing surface are ofter complex and require a substantial amount if vertices that describe them. Having too many vertices slows down the force based embedding presented later in this Chapter. Such detailed representation of the road is usually not needed for the map to be useful and could be replaced by interpolation of the road curve. We even consider simplifying the street shapes helpful, as long as it does not harm navigation too much. This leads us to the problem of simplifying the routes displayed on the drawing surface. This step is *optional* in our application and the user can skip it and move directly to drawing the map. In addition to the polyline itself, the simplification algorithm requires one more parameter: t_{area}. This parameter represents an area threshold upon which we decide whether a vertex is kept or not. The algorithm itself processes just polylines and therefore requires us to split our collection of edges (routes) into polyline segments and feed them to it one by one.

Algorithm 6.1: PATH SIMPLIFICATION

Input: List of vertices P that represents the polyline, the parameter t_{area}
Output: List of vertices result that remain after the simplification

```
   // Initialization
1  result.addVertex(P[0])
2  current ← 1
3  prev ← 0;
   // Main loop
4  while current smaller than P.SIZE- 1 do
5      area ← calculateArea(P[prev], P[current], P[current + 1])
6      if area < t_area then
7          current ← current + 1
8      else
9          result.addVertex(P[current])
10         current ← current + 1
11         prev ← prev + 1
12 result.addVertex(P[P.SIZE − 1])
13 return result
```

The algorithm simplifies each polyline as follows:

1. The first and the last vertex of the polyline are always marked as included.
2. We iterate over the rest intermediate vertices, starting from the first and moving to the last. For each inner vertex we evaluate the area of the triangle formed by this

vertex and its left and right neighbors (they always exist, because its an inner vertex in our polyline; line 5, algorithm 6.1). If this area is bigger than t_{area} (lines 8-11) we keep the inner vertex and mark it as included. If not (lines 6-7), we simply delete the vertex. Already deleted vertexes do not count as neighbors when we move forward and calculate the next iterations.

After successful execution the list *result* contains the vertices that remain and therefore defines the new simplified polyline. The algorithm has linear runtime and needs to be applied to every polyline in our graph. This means we need a total of $O(|E|)$ time units in order to simplify the whole graph with the parameter t_{area}.

6.3 Force based drawing

We have reached the last part in our destination map creation process: *layout improvments*. Our goal is to achieve something similar to a *lens effect* around the destination, where we change the graph layout and aim to bring forward the roads near the destination. We rescaling the length of the roads (edges) depending on their distance to the destination point. This rescaling is done using a spring embedder. We implement this embedder by constructing an algorithm that step by step changes the position of the vertices and thus allows for better visual representation. On each step each vertex in our graph is evaluated and moved according to the forces applied to it. We also maintain picture coordinates for each vertex, which at the start are the same as the original picture coordinates calculated in section 6.1, but change during the course of the layout improvement algorithm. After applying a given number steps the forces move the vertices and we end up with a slightly changed layout, which is shown as an end result.

6.3.1 The idea and some useful constants

The most important aspect of making this spring embedder work is defining the forces that guide it. In order to do so we set our mind toward what the destination map represents and thus define what a better graph layout.

1. Firstly, it is very important that our layout improvement does not harm the usefulness and the functionality of our map. This mean that we can't allow ourselves to change the map to such an extend that it no longer resembles the original routes. The reader needs to be able to make the connection between what is presented on the map and reality. We achieve this by defining forces that help the vertices position themselves in such a way that the original angles between the edges are kept. This force also strives to keep the angle between each edge and the x-axis (the bottom of our drawing surface). This prevents the whole layout from rotating itself too much and still being classified as improved.

2. Secondly, since the main focus is on our destination we want the layout to resemble this by being more detailed in the area around the destination point and compact in the outer regions. So our idea is to have something like a lens effect around the destination. This should emphasize the routes near the destination and attract the attention towards them and at the same time make streets located further away smaller up to a state where they are still visible and can be used for navigation, but take less space. This effect is achieved by defining an ideal edge length function $I_{length} : E \to \mathbb{R}_{>0}$ which determines how long would be the ideal length for each edge according to its location. Kamada and Kawai [KK89] do something similar in their "An algorithm for drawing general undirected graphs", where they use a spring embedder that tries to maintain predefined distances between vertices. We achieve this change in the length of each edge by reorganizing the vertices's location using a force of attraction between those of them connected by an edge.

3. We also define a repulsion force. This force keeps vertices from coming too close to one another. This is important because it prevents clustering and allows the vertices to use the space of the drawing surface.

The attraction and repulsion force are classical parts of a force based drawing model and are present in many spring embedders [Ead84] [FR91] [KK89] and as Kobourov [Kob12] says "in these methods, there are repulsive forces between all nodes, but also attractive forces between nodes which are adjacent."

6.3.2 Scaling factor

We move forward by presenting our idea of the lens effect and explaining how exactly the function I_{length} is calculated. In order to do this we define and make use of the following parameters (all calculations are done in picture coordinates):

- We will refer to the destination vertex as s.
- A function $D_{euclid} : (V \times V) \to \mathbb{R}_{\geq 0}$ that computes the euclidean distance between two vertices. This function is also used for calculating and edge's length since an edge is simply a pair of vertices.
- A parameter $D_{max} = max\{D_{euclid}(v,\, s)\}$ where $v \in V$
- A parameter $c_{min} \in (0,\, 1]$ that represents the minimal reduction factor for an edge.
- A parameter $c_{max} \in [1,\, \infty)$ that represents the maximal magnification factor for an edge.
- A parameter r_1 that determines where exactly the edges will remain unscaled. The distance at which this happens defines a circle around the destination and is calculated as $D_1 = D_{max} r_1$.

Now we will use this parameters to develop a model that calculates the function I_{length} and thus determine the length we would want each edge to have. We do this by defining a scaling factor $F_{scale} : V \to [c_{min},\, c_{max}]$ for each vertex. This function is calculated for each $v \in V$ as:

$$F_{scale}(v) = \begin{cases} c_{max} + \frac{D_{euclid}(v,\, s)}{D_1}(c_{max} - 1) & \text{if } D_{euclid}(v, s) \leq D_1 \\ 1 + \frac{D_{euclid}(v,\, s) - D_1}{D_{max} - D_1}(1 - c_{min}) & \text{if } D_{euclid}(v, s) > D_1 \end{cases}$$

This function calculates a scaling factor for each vertex depending on the distance of a vertex v to the destination. We use this scaling factor values for a vertex to define a scaling factor of an edge, which determines the edge's ideal length related to its length. So for every edge we define the function $F_{scale} : E \to [c_{min},\, c_{max}]$ which for $\{u,\, v\} \in E$ is calculated as: $F_{scale}((v,\, u)) = 1/2(F_{scale}(v) + 1/2(F_{scale}(u))$. Having this scale factor defined for an edge, we can simply define the function calculating the ideal length of an edge (v, u) as $I_{length}((v,\, u)) = F_{scale}((v,\, u)) D_{euclid}(v,\, u)$.

6.3.3 Attraction Force

With the information from the previous section we are now ready to define our first force. The idea for this force came from the spring embedder of Fruchterman and Reingold developed in 1991 [FR91]. Their attractive force is defined as $f_a(d) = d^2/k$, where d is the distance between two adjacent vertices and k is defined as $k = \sqrt{area/number\ of\ vertices}$. They apply this force f_a by multiplying it with the vector defined by the two adjacent

vertices. Fruchterman and Reingold aim to minimize the force between every adjacent pair of vertices by making them spread and use the space provided. So their force is allowed to move a vertex along the line defined by this vertex and an adjacent one and aims to spread the layout in such a way that the available space is better utilized. The approach Fruchterman and Reingold used is applicable only for unweighted graphs that are expected to have equal edge length. This is why we expand this idea similarly to Kamada and Kawai [KK89] and change it to work for weighted graphs and mimic our lens effect. We achieve this by defining a force f_a, which moves a vertex along the line defined by an outgoing edge with the goal of getting the edge to expand (or shrink) to its ideal length. So for every adjacent pair of vertices v and u we define the force of attraction as follows:

$$f_a(v,\, u) = c_{att} \log\left(\frac{D_{euclid}(v,\, u)}{I_{length}((v,\, u))}\right)$$

The parameter c_{att} is non-negative and determines how big of an influence this force has. This parameter has a start and end value and gradually changes with each iteration. This allows for additional flexibility and allows the map creator to change the influence of this force throughout the whole process of changing the graph layout. The log function allows this force to be negative. A negative value means that the edge between v and u should shrink, while a positive value means it should be magnified. This force can be seen as attraction from the vertex u to the vertex v and its impact is expressed as a direction vector $f_a(v,\, u)\hat{k}$, where $\vec{k}$ is the vector from v to u. So the attraction force that is applied to a certain vertex v, can be expressed as sum of vectors:

$$f_a(v) = \sum_{u \text{ neighbor of } v} f_a((v,\, u))\hat{k}$$

6.3.4 Repulsion Force

In our model the repulsion force is defined between vertex pairs that are not connected with an edge and it prevents vertices from getting too close to each other. There is no need to calculate a repulsion force between connected vertices, because the attraction force already makes the two vertices repulse each other if the attraction force f_a is positive. The repulsive force between two vertices v and u, that are not adjacent is defined as:

$$f_r(v, u) = \frac{c_{rep}}{D_{euclid}(v,\, u)^2}$$

We again apply this force with vector multiplication, but this time we multiply it with the vector $\vec{t}$ from u to v. Analogously to the attraction force sum, we define

$$f_r(v) = \sum_{u \text{ not neighbor of } v} f_r((v,\, u))\hat{t}$$

At this point we have calculated the magnitude and direction for both the attraction and the repulsion forces in the form of two vectors. We denote the sum of this vectors as a translation for the vertex v and calculate it as follows: $\vec{m_v} = f_a(v) + f_r(v)$. Here we change the position of the vertex v. We move it very little in the direction $\vec{m_v}$ by by updating its picture coordinates accordingly. Again the parameter c_{rep} has a start and end value that gradually changes with each iteration.
Applying just these two forces will change the relative position of the vertices drastically and will most likely change the angles between the edges to such extend that the newly computed layout completely differs from the original one. In order to prevent this we use an additional angle correction force, which is calculated and applied every time after a vertex is moved.

6.3.5 Angle Correction Force

This force keeps the angle between adjacent edges unchanged. Every time we move a vertex v using the attraction and repulsion forces we will adjust the vertices adjacent to v in order to keep their relative positions and the angles between the outgoing from v edges. This force assumes that the vertex v is where it should be and will not attempt to change its position. Having this as a starting point we fix the angles between for each edge $\{v, u\}$ by change the position of the vertex u if needed. Knowing the current position of v and the original position of u, we can calculate where u should be, relative to v. We calculate the translation of v since the start in the form of a vector $\vec{t_v} = v.picCoord - v.originalPicCoord$. We apply this translation vector to u and determine the vector on which the vertex u should lie, so that the initial angle for this edge with the coordinate axes is kept. We name this vector $\overrightarrow{vu'}$ and it is calculated as $u_{new} = \vec{t_v} + u.originalPicCoord - v.picCoord$. We must also respect the edge length between v and u. This is why we calculate the position of u' by changing the length of the vector $\overrightarrow{vu'}$ to be exactly $|\overrightarrow{vu}|$. This is done in two steps. First we normalize the vector $\overrightarrow{vu'}$ and then we multiply it by $|\overrightarrow{vu}|$. Now having the position of u', we examine the vector $\overrightarrow{uu'}$ If the length if this vector is less than 0.2, we don't take any action and leave u where it is. If this is not the case, we move the vertex u towards the position u' by a tiny bit and make a step towards restoring the correct position for the edge (v, u). Similarly to the previous two force, here we also have a non-negative constant that controls this force's influence: c_{ang}. It again has start and end value, which determine its change during the iteration process.

7. Examples and Experiments

In this chapter we will show the functionality and will demonstrate some of the capabilities of our application. We have included some data about the running time of our algorithms and will show how the whole process of creating a destination map looks like. The prototype application is mainly written in C++ and uses technologies like HTML, CSS, JavaScript and Ajax. For the test data and examples below we compiled with -O3 optimization and -std=c++11 flag for the new C++ standard from 2011.

7.1 Time and complexity

As described in the previous chapters, our solution to drawing a destination map consists of several major steps: Dijkstra, selection, simplification and drawing. Here we have measured the time needed for each of these steps in different situations. The application makes use of just one core and the test were made on a machine running Ubuntu 14.04 with AMD FX 8150 processor, clocked at 4.0 Ghz. The results have been obtained by random sampling (200 samples) throughout the territory of Netherlands and represent an average approximation.

Table 7.1: Time needed for computing the subnetwork of shortest paths and selecting relevant routes.

Radius	20 km (12km, 16km)	100 km (12km, 16km)
Vertices (total)	103029	1424683
Vertices (shown)	1488	14729
Dijkstra	29 ms	64 ms
Selection	117 ms	997 ms
Total Time	142 ms	1061 ms

The data presented here shows how long it takes for our application to finish the algorithms described in Chapter 5. The numbers in the brackets on the first line represent the medium and the big filter radiuses used respectively. Normally destination maps don't cover a span greater than few kilometers. Even when operating on a large region (200 km in diameter) the road selection takes little more than 1s time.
Similarly we evaluate the time cost for the force based draw procedure.

Table 7.2: Time required for simplifying and drawing the layout.

Vertices	1355	1355	1355	1355
Simplification	no	yes, area 40	no	yes, area 40
Vertices left	1355	106	1355	106
Iterations	200	200	1000	1000
Forces	363 ms	1.7 ms	1678 ms	2.26 ms
Drawing (x1)	210 ms	33 ms	210 ms	33 ms
Total Time	573 ms	35 ms	1888 ms	35 ms

These results represents the time needed to draw a destination map that has a radius of 8 km using different parameters. The prototype application opens an Xorg[1] windows and uses cairo [2] to draw the graph layout in this window. Currently the drawing is called every few steps in order to show how the forces influence the graph layout throughout the iterations. There is also the option to export the end result in .pdf or .png format.

7.2 Examples

These examples illustrate the process of creating a destination map for a chosen location by providing picture material in the form of screen shots from our prototype application.

7.2.1 Amsterdam, Holland

In this example we have selected a point in Amsterdam with Lat-Lon (52.312166, 4.890048). When the user starts the application, it loads the road network data and a map of the area can be seen on the screen. At this stage no destination point has been selected. The user can select one by simply clicking somewhere on the map. After registering the click, the application finds the closest vertex to the location of the mouse cursor and marks it as the destination. A road selection according to the default parameter values is displayed and can be seen on Figure 7.1. At this point we can tweak and change parameter values in order to reshape this road subnetwork. After parameter change the user has to click the "Update output" button for a new call to Dijkstra and the road selection algorithms.

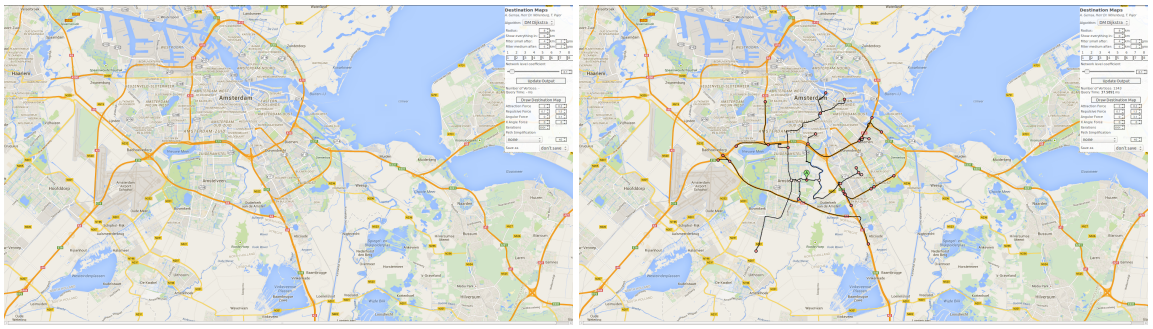


Figure 7.1: The initial empty map and the generated routes after selecting destination.

The transition from Figure 7.2 to Figure 7.3 shows how routes with higher network level (smaller streets) can be removed by using stricter values in the filters. When the user is satisfied with the road subnetwork displayed on the map, he can move forward to drawing it. Here we will use the drawing algorithm on Figure 7.3.

[1]Xorg is a open source window system. More details here: http://www.x.org/wiki/
[2]Cairo is a 2D drawing library: http://cairographics.org/.

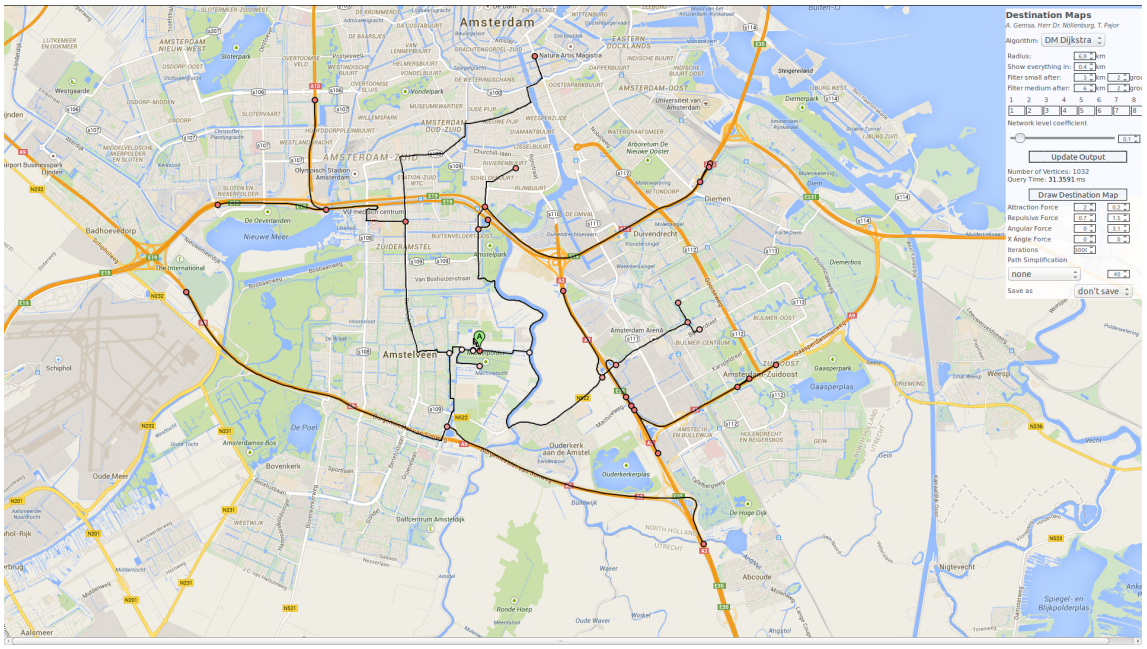


Figure 7.2: Changing the radius parameters changes the current road selection.

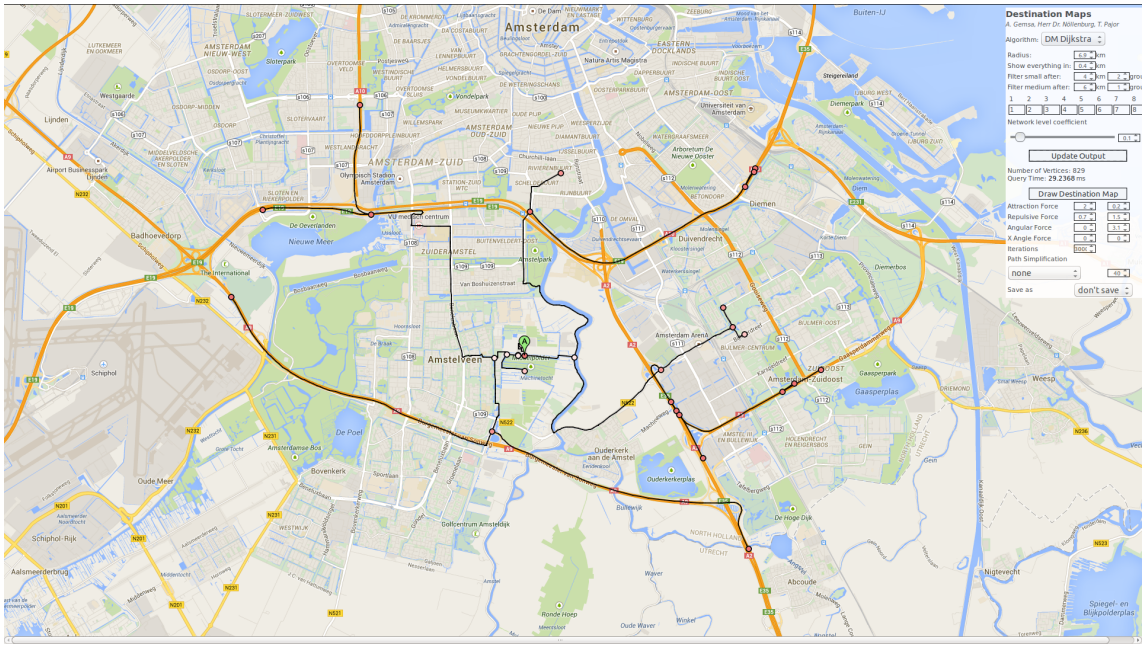


Figure 7.3: Changing the network level in the filters can add/remove route in our destination map.

Smaller streets that have bigger network levels are painted yellow, while motorways have been represented with bigger black lines. The destination point is highlighted with a green circle.

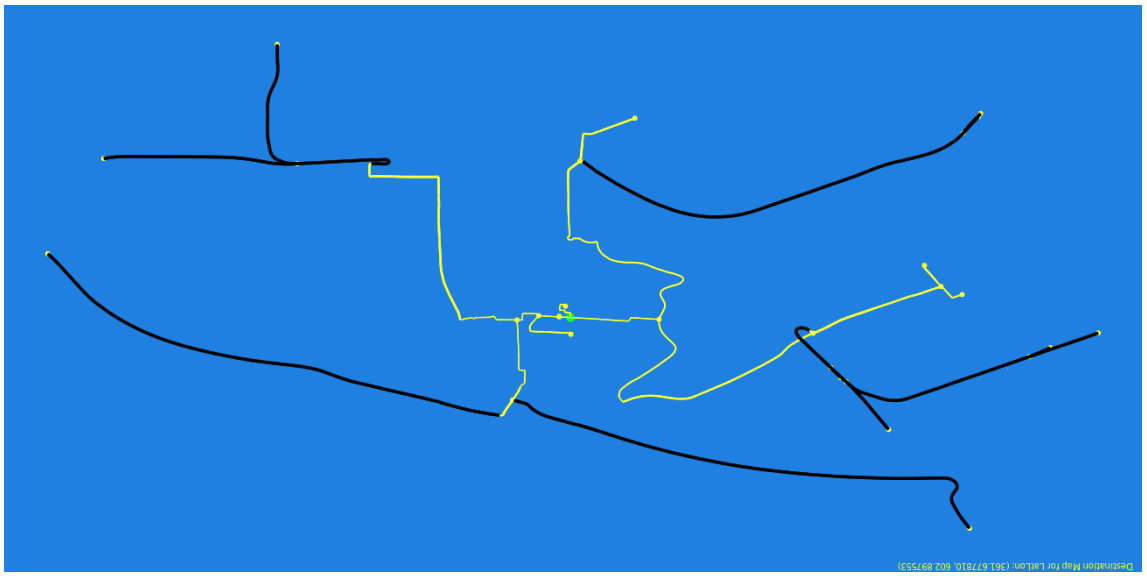


Figure 7.4: This is without path simplification and without force based layout improvement. Th graph is drawn directly after the picture coordinates have been calculated with no further changes.

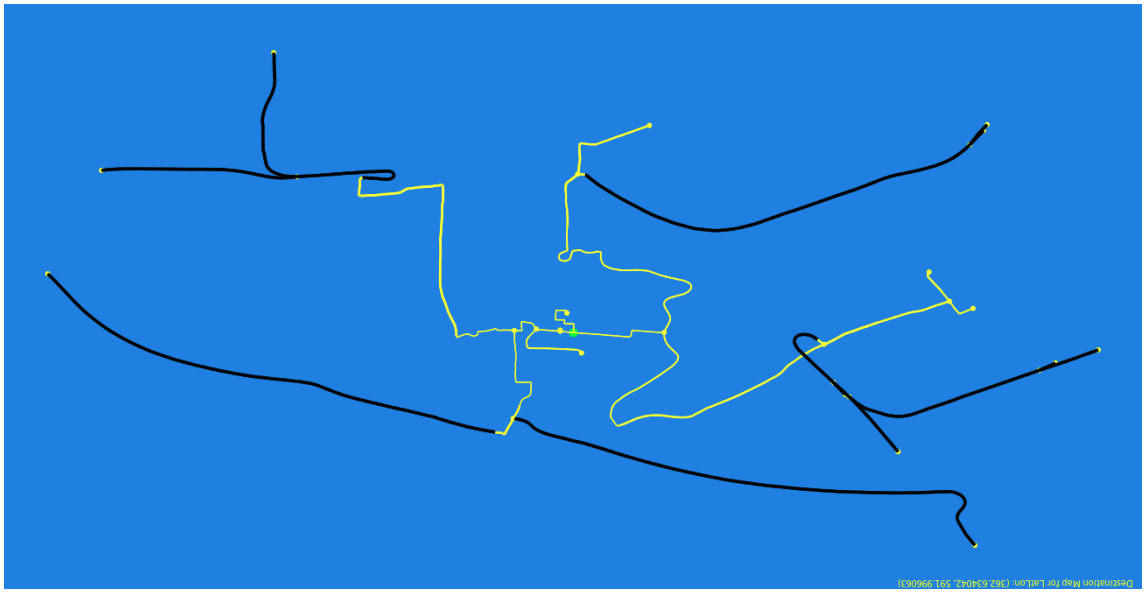


Figure 7.5: Still not using the path simplification algorithm, but running 800 iterations using the forces defined in Chapter 6

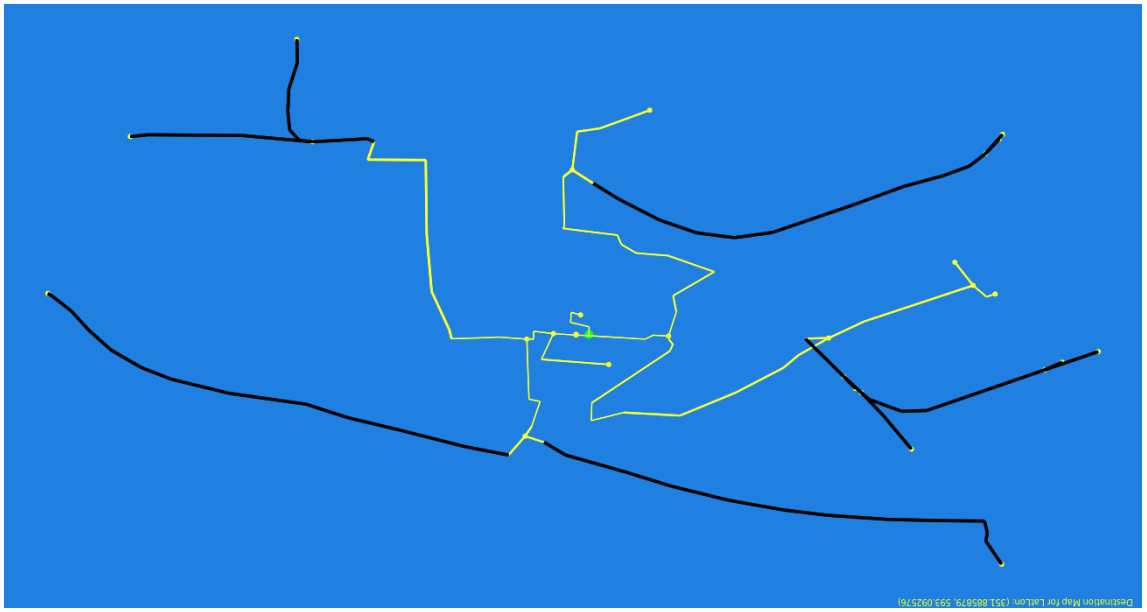


Figure 7.6: Using path simplification with an area threshold of 40. The number of iterations has been increased to 7000.

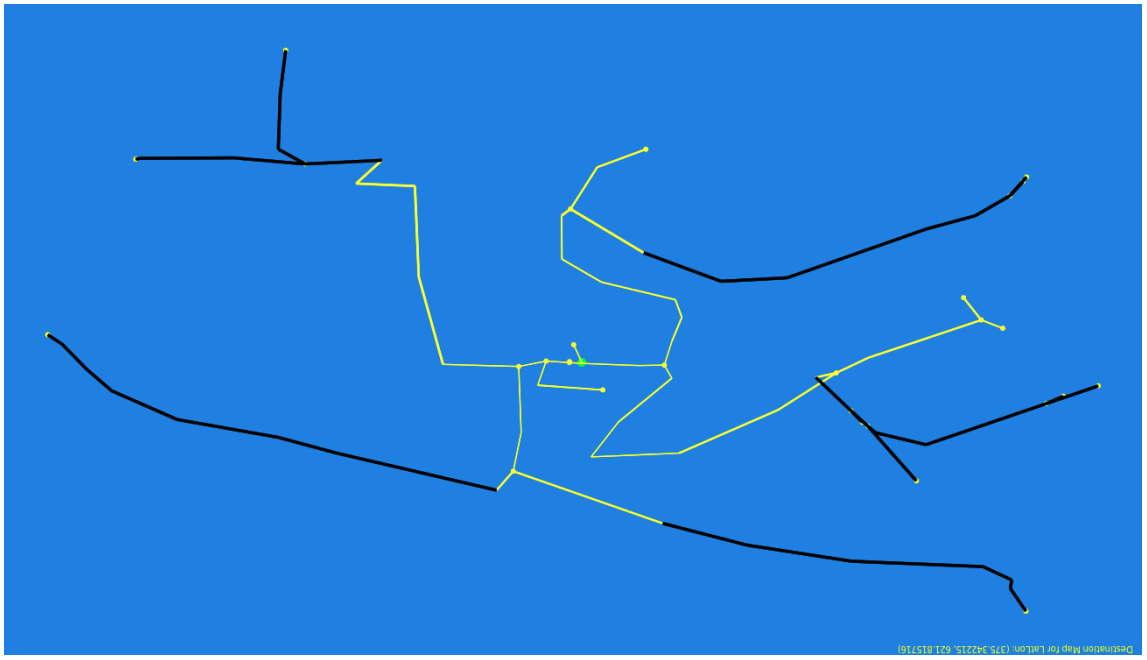


Figure 7.7: Using path simplification with an area threshold of 50 and already losing some important detail. The number of iterations here is again 7000.

7.2.2 Karlsruhe, Germany

In this example we have chosen the main library at Karlsruhe Institute of Technology (KIT) as a destination point.

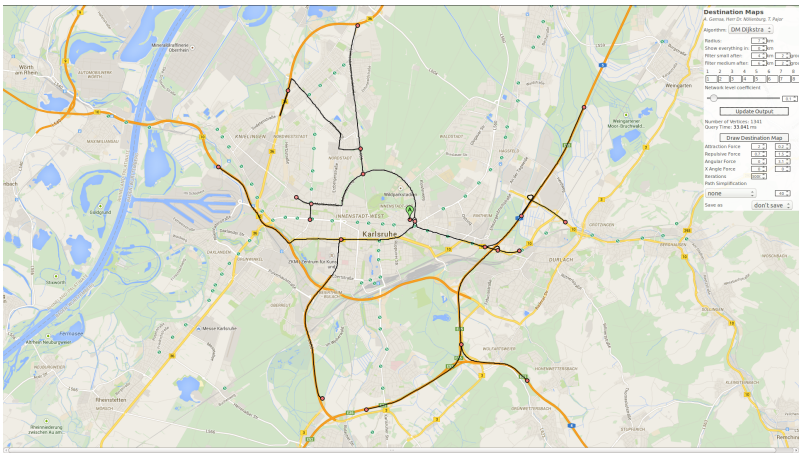


Figure 7.8: This is how the road selection looks like just after pin-pointing the destination.

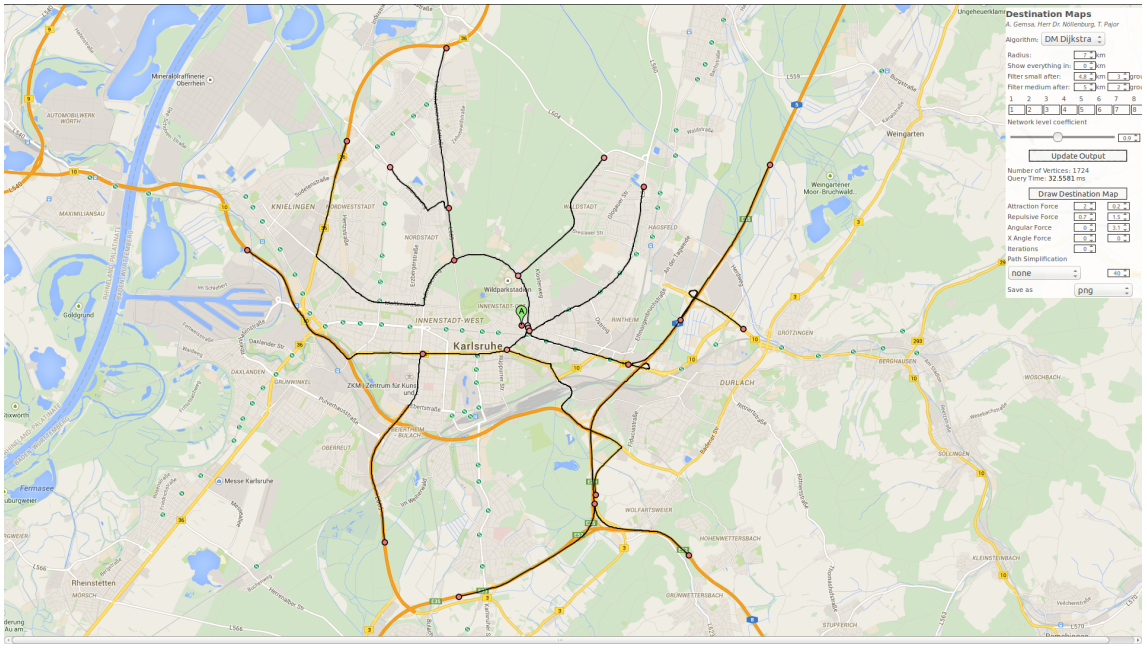


Figure 7.9: Changing filter radiuses and forcing the Dijkstra to prefer larger roads gives us different subnetwork of roads.

The examples presented here show that parameters have a big saying in which routes will be included in our destination map and how these routes are displayed. The results presented here do not require more than a few seconds to be calculated. The prototype application also provides a real time view of the graph layout as its being changed by the forces. While the forces can improve the graph layout, they could also change it in a negative way in some bizarre cases or if the parameters are set too aggressively. One prevention technique against chaotic movement of vertices could be to identify such situations and stop the forces from moving vertices in a way that will harm the graph layout. For example crossings that occur or disappear due to vertices movement would result in confusion and inaccuracy and should be avoided.

Figure 7.10: Drawing of the subnetwork of roads as it is. Zero iterations for our force based drawing.

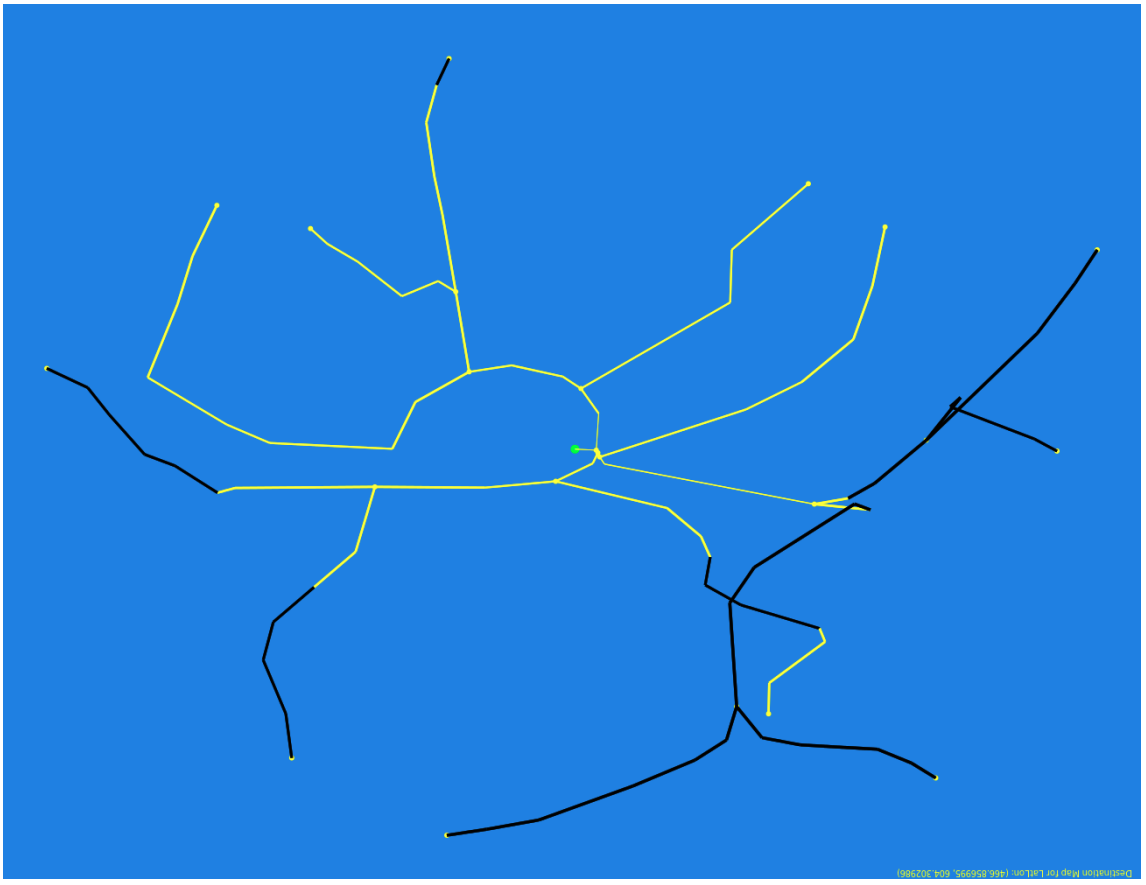


Figure 7.11: Using path simplification with area threshold 25. Number of iterations is 5000.

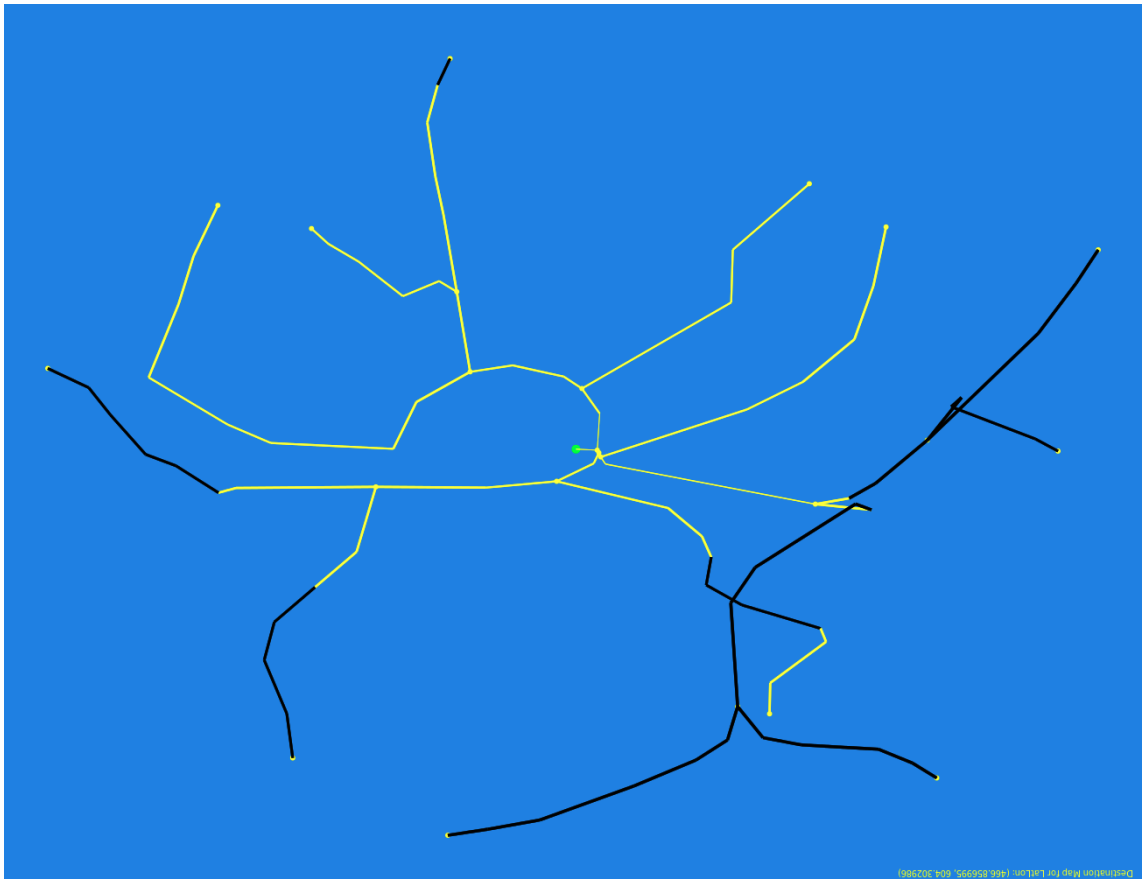


Figure 7.12: Using path simplification with area threshold 40. Number of iterations is 10000.

8. Conclusion

Our goals here is creating an automated tool that can create destination maps that are comparable to human made ones. The quality of such a map depends mainly on the streets included in it and the way these streets are represented. This leads to the idea that a tool for automated generation of destination maps should implement these two steps and be able to select a relevant collection of routes and then present this collection in a human friendly sketch. In this paper we have presented our idea of such tool.
In order to solve the road selection problem we considered two approaches. Before building our solution on the subgraph produced from Dijkstra's algorithm, we experimented using contraction hierarchies. Our idea was to use the in order to determine important vertices that stand higher in the hierarchy and build our destination map around them. We gave up on the contraction hierarchies because they are too vast and often include unwanted/irrelevant detail far away from the destination. We believe that with some modifications they can be further developed to help determine the relevant selection of roads that is presented in the destination map. Contraction hierarchies can be of great help if the area of interest is large and as suggested by Peter Sanders et al. [GSSV12] can improve the running time of Dijkstra and find shortest paths between vertices very fast utilizing precomputed shortcuts. Shortest paths and Dijkstra's algorithm are one possible approach when it comes to road selection leading to a certain destination. Our solution uses this approach and also takes into consideration the road classification in the face of a network level property of each street. This helps narrow down the road selection to a handful of important routes and helps eliminate unwanted detail. While in many cases this is enough, there are some examples where extending the roads and including more detail could be beneficial. Currently our approach does not have a build-in mechanism for determining how and which roads can be extended to provide better visual navigation. Implementing this additional step would bring completeness in the selected roads and allow for easier read on the generated map.
Moving forward with drawing the map our application focuses on presenting the selected routes in a better way. It includes information about the streets and the destination in the drawing in the form of colors and makes sure the map reader can easily distinguish between the elements included by improving the graph layout. A good step towards improving the map even more would be having some of the streets named on the map. This will allow map readers to navigate much easier when knowing the exact address or street name, which very ofter is the case with destination maps. Another helpful addition would be adding terrain detail to the map such as water or forests. One may also consider representing some of the bigger buildings in the area with symbols. For example, crosses for churches and graveyards or fork and knife for restaurants.

We think, that destination maps can be viewed in many different ways, which can lead to many different requirements and solutions. There is no fixed set of criteria that defines a good destination map. Our approach combines different techniques with the common goal of creating a basic destination map tat can be used as a pocket navigational note. By adding some of the unimplemented features such as road extensions, terrain detail and street names this tool might become a replacement for hand-made destination maps.

Bibliography

[DGN+14] Daniel Delling, Andreas Gemsa, Martin Nöllenburg, Thomas Pajor, and Ignaz Rutter. On d-regular schematization of embedded paths. *Comput. Geom.*, 47(3):381–406, 2014.

[Dij59] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DP73] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973. doi:10.3138/FM57-6770-U75U-7727.

[Ead84] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, November 1991.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[KAS+10] Johannes Kopf, Maneesh Agrawala, David Salesin, David Bargeron, and Michael F. Cohen. Automatic generation of destination maps. *Siggraph Asia 2010*, 2010.

[KK89] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, April 1989.

[Kob12] Stephen G. Kobourov. Spring embedders and force directed graph drawing algorithms. *CoRR*, abs/1201.3011, 2012.