Karlsruhe Institute of Technology

Efficient Calculation and Visualisation of Range Polygons

Bachelor Thesis of

Sven Zühlsdorf

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Moritz Baum, M.Sc.
	Dipl.-Inform. Andreas Gemsa

Time Period: 1st February 2013 – 31st May 2013

Abstract

This bachelor thesis focuses on determining and visualizing the reachable vertices of a graph. A vertex is considered reachable if its distance from a given source vertex is smaller than or equal to given a maximal range. To formalize the problem we introduce the RANGEPOLYGON problem, which is solved by a set of polygons delimiting the area containing reachable vertices. To enforce an accurate representation we also define the *boundary* of the reachable area. We solve the problem in two steps by first determining the boundary from the given inputs and then using the boundary to create a visualization. For the routing part of the problem we adapt Dijkstra's algorithm to return the boundary and introduce Customizable Route Planning [DGPW11] as a speedup technique. For visualization we develop and present exact solutions and a conservative heuristic to generate range polygons from the boundary. We then show that our approaches are faster and generate less complex polygons than Alpha Shapes [EKS83], an algorithm used for surface reconstruction.

Deutsche Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Bestimmung und visuellen Darstellung von erreichbaren Knoten eines Graphens. Ein Knoten ist erreichbar wenn seine Entfernung zu einem bestimmen Startknoten eine bestimmte Distanz nicht überschreitet. Um das Problem zu formalisieren führen wir das RANGEPOLYGON Problem ein. Die Lösung für deses Problem ist eine Menge an Polygonen, die die erreichbaren von den unerreichbaren Knoten trennt. Wir definieren zusätzlich die *Grenze* des erreichbaren Bereiches um eine genaue Darstellung der Reichweitenpolygone zu erzwingen. Wir lösen das Problem in zwei Schritten: Zuerst bestimmen wir die Grenze aus den Eingabedaten, und anschließend eine visuelle Darstellung mit Hilfe der Grenze. Zur Bestimmung der Grenze passen wir Dijkstras Algorithmus an unsere Problemstellung an und stellen Customizable Route Planning [DGPW11] als eine mögliche Beschleunigungstechnik vor. Für die Darstellung entwerfen wir Algorithmen zur exakten Wiedergabe der Grenze, sowie eine Heuristik es erlaubt erreichbare Fläche als unerreichbar zu makieren, aber nicht unerreichbare Fläche als erreichbar. Anschließend vergleichen wir unsere Algorithmen mit Alpha Shapes [EKS83], einem Algorithmus zur Oberflächenrekonstruktion, und kommen zu dem Schluss, dass unsere Algorithmen sowohl schneller sind, als auch weniger komplexe Polygone berechnen.

Contents

1 Introduction **1**

2 Preliminaries **3**
- 2.1 Computational Geometry 3
- 2.2 Graphs 3
- 2.3 Dijkstra's Algorithm 5

3 Problem Statement **7**

4 Determining the Boundary **11**
- 4.1 Precalculations 11
- 4.2 Basic Routing Algorithm 11
- 4.3 Acceleration 13
  - 4.3.1 Basic CRP 13
  - 4.3.2 Our Adaptations 14

5 Visualization **17**
- 5.1 Basic Approach 17
- 5.2 Stabbing 18
- 5.3 Merging 19
- 5.4 Alpha Shapes 22

6 Experiments **23**

7 Conclusion **29**

Bibliography **31**

1. Introduction

During route planning we might not always be interested in the shortest path to a given target, but instead want to know which area can be reached within a certain range. For example, drivers of electric cars must plan their routes according to available charging stations along the way, due to the limited range of electric cars. Displaying the area that can be reached with the current battery charge on a map would visually aid the driver while planning a trip. While driving, displaying the remaining range on a navigation system's map could also increase the feeling of security as the driver knows where they will run out of fuel and thus can plan accordingly. In theory, running out of fuel is also a problem for other cars, however, gas stations are frequent enough to not require additional planning.

We introduce the RangePolygon problem that, given a source vertex and a maximum range, is solved by a set of polygons enclosing the reachable area. These polygons can then be displayed on a map and might be interpreted as isolines consisting of points with a distance to the source vertex that is equal to the maximum range. Usually isoline are used in geographic contexts as [VK96] shows.

For finding a shortest path between Dijkstra's algorithm [Dij59] is often used. Several speedup techniques for Dijkstra's algorithm exist: A*, Landmarks, Contraction Hierarchies to name a few. Short descriptions and comparisons between them can be found in [DSSW09]. However, most of them were designed for one-to-one queries and adaptation to one-to-all queries is difficult. Customizable Route Planning (CRP) [DGPW11] is a speedup technique which not only can be adapted to one-to-all queries but also supports multiple metrics. Customization to a specific metric only requires few seconds for precalculations. Another speedup technique for one-to-all queries is PHAST [DGNW12]. However PHAST is based on Contraction Hierarchies and therefore can not be adapted to arbitrary metrics quickly [DSSW09].

Our basic visualization approach is based on the same idea as marching cubes [LC87]. Given the eight points of a cube and a marking indicating whether the points are inside or outside of a volume, marching cubes visualize the course of the border of that volume within the cube. The line stabbing problem requires a set of lines to be intersected by a chain of line segments in a specified order. Several different versions of this problem are presented in [GHMS93]. They also show that minimizing the number of line segments in the chain is NP-hard. Alpha Shapes [EKS83] reconstruct a surface or shape from a given point set, which can be used for various purposes like simply visualizing or determining groups in the point set.

We now give the outline of this work. In Chapter 2 we introduce the basic notation and algorithms used in this work. In Chapter 3 we give a formal definition of the problems discussed in this work. We note that the problems can be easily split into two parts and introduce and adapt algorithms to solve the two subproblems in the following two chapters. Chapter 4 covers the routing part of the problems. We first adapt Dijkstra's algorithm to our problem and then introduce Customizable Route Planning as speedup technique. While other speedup techniques adaptable to our use case, like PHAST, exist, we prefer CRP because it works well with any metric. In Chapter 5 we use the results of the routing part to generate a visualization. Our basic visualization approach is based on the same idea as marching cubes. For an improved version of our approach we try to reduce the number of line segments closing the reachable area, while staying in a specific boundary. In Chapter 6 we then compare the algorithms developed in the previous chapters with each other. The visualization algorithms are additionally compared to Alpha Shapes and the convex hull. Finally, in Chapter 7, we summarize our results and give an outlook posing open questions for further work.

2. Preliminaries

In this chapter we establish basic terms used in this work with definitions similar to those found in literature. Derivations from these definitions in the following chapters will be motivated and explained when first needed.

2.1 Computational Geometry

A *polygon* is a sequence of points $(p_1, \ldots, p_n)$ in the plane, which form a sequence of straight line segments $((p_1, p_2), (p_2, p_3), \cdots, (p_n, p_1))$. We call a polygon *simple* if and only if no two of its line segments intersect.

The *convex hull* $\mathrm{CH}(S)$ of a set S of points in the plane is the smallest simple polygon for which all line straight segments $(u, v), u, v \in S$ lie inside or on the boundary of $\mathrm{CH}(S)$. The convex hull can be computed in $O(|S| \log h)$ time, where h is the number of points $|\mathrm{CH}(S)|$ of the convex hull, as shown by [KS86].

Given a set S of points in the plane and a parameter α, *Alpha Shapes* [EKS83] create polygons representing the outline or shape of the point set. In the 2D plane, the area outside of every circle with radius $\sqrt{\alpha}$ not enclosing any points of S is considered to be inside the Alpha Shape. Circular arcs in the resulting outline are then replaced by straight line segments. An informal description for the three dimensional case is found at [CGA]: "Imagine a huge mass of ice-cream making up the space $\mathbb{R}^3$ and containing the points as "hard" chocolate pieces. Using one of these sphere-formed ice-cream spoons we carve out all parts of the ice-cream block we can reach without bumping into chocolate pieces, thereby even carving out holes in the inside (e.g. parts not reachable by simply moving the spoon from the outside). We will eventually end up with a (not necessarily convex) object bounded by caps, arcs and points. If we now straighten all "round" faces to triangles and line segments, we have an intuitive description of what is called the α-shape of S." Depending on the value of α the result ranges from the initial point set ($\alpha \to 0$) to the convex hull of S ($\alpha \to \infty$).

2.2 Graphs

We define a *graph* $G = (V, E)$ as a tuple of a set of *vertices* V and a set of *edges* E. We call the number of vertices $n = |V|$ and the number of edges $m = |E|$. An edge is a pair of two different vertices $e = (u, v); u \neq v; u, v \in V$. The vertex u is the *source* of the edge, while

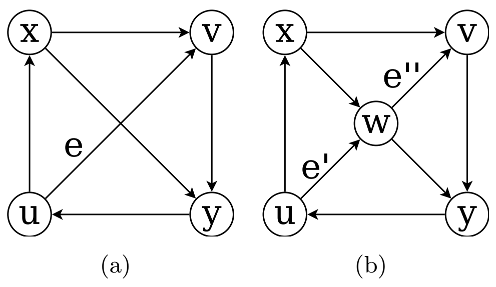


Figure 2.1: Planarization of intersecting edges by inserting vertices. (a): original graph, (b): its planarization

v is the *target*. If the order of vertices in an edge does not matter, i.e., (u, v) and (v, u) are indistinguishable from each other, we call this edge *undirected* and write it as $\{u, v\}$. A graph containing only undirected edges is a *undirected graph*, otherwise it is *directed*. A function $\omega(e)\colon E \to \mathbb{R}_0^+$ associating a non-negative weight to each edge $e \in E$ is called a *weight function*. A sequence of vertices $(v_1, v_2, \cdots, v_{n-1}, v_n)$ with $(v_1, v_2), \ldots, (v_{n-1}, v_n)$ being edges of G is called a *path* from v_1 to v_n. If a weight functions exists, the sum of all edge weights $\sum_{i=1}^{n-1} \omega(v_i, v_{i+1})$ of the path is its *length*. A *shortest path* from s to t is a path with minimal length among all paths from s to t. The length of a shortest path from s to t is the *distance* $\mathrm{d}(s, t)$ from s to t. If no path from s to t exists the distance $\mathrm{d}(s, t)$ is infinity. A graph is *strongly connected*, if and only if for each vertex there exists a path to every other vertex of the graph.

A directed graph is *acyclic* if and only if for each vertex u the only path from u to itself is the trivial path (u). A *(rooted, directed) tree* is an acyclic graph in which exactly one vertex r, the *root*, has exactly one path to all other vertices. The *shortest path tree* T of a graph G from s is a tree with root s. It includes all vertices to which a path from s exists. All paths in T, especially those starting from s, are shortest paths in G.

A *planar graph* is a graph which can be embedded in the plane without crossing edges. Graphs with a non-planar embedding can be *planarized* by adding additional vertices. For each point at which edges intersect, an additional vertex w is created and edges $e = (u, v)$ intersecting w are split into $e' = (u, w)$ and $e'' = (w, v)$, see Figure 2.1 for an example. This results in a planar graph $G_{planar} = (V_{planar}, E_{planar})$ with V_{planar} being a superset of V. Note that planarization introduces additional paths: In Figure 2.1(b) the path (u, w, y) has no equivalent in the original graph depicted in Figure 2.1(a). The areas enclosed by edges of a planar graph are called *faces*. A *triangulated graph* is an embedded planar graph in which all faces are triangles. In our work this includes the outer face. We assume the existence of an *infinite vertex* which is connected to all vertices of the convex hull. The *Delaunay triangulation* is a special triangulation in which no vertices lie inside the circumcircle of any face. For a given point set V the Delaunay triangulation can be computed in $O(|V| \log |V|)$ time, as shown by [LS80]. The *constrained Delaunay triangulation* is an adaption of the Delaunay triangulation in which a given set E of edges must be present. It behaves like the Delaunay triangulation but since E can be chosen in a way causing the Delaunay triangulation to always violate the circumcircle property, the property is relaxed: Vertices inside the circumcircle of each face are allowed if the line between the vertex and the center of the circumcircle intersects an edge $e \in E$. The constrained Delaunay triangulation can also be constructed in $O(|V| \log |V|)$ time as shown in [Che89]. A function translating a path (u, v, w) into a boolean, indicating whether a turn from (u, v) to (v, w) at vertex v is allowed, is called a *turning restrictions function*.

A *subgraph* of a graph G is a graph $G' = (V', E')$ containing a subset $V' \subseteq V$ of V and a subset $E' \subseteq E$ of E which only contains edges connecting vertices in V'. A subgraph induced by a set of vertices $V' \subseteq V$ is a graph $G' = (V', E')$ containing all vertices of V' and all edges $e = (u, v) \in E$ with $u, v \in V'$ connecting vertices of V'.

A *partition* of a graph G is a division of its vertices V among a given number k of sets $V_1, \dots, V_k$ called *cells*. These cells include all vertices of G ($\bigcup_{i=1}^{k} V_i = V$) and each vertex is only assigned to one cell ($\bigcap_{i=1}^{k} V_i = \emptyset$). Edges (u, v) connecting vertices of different cells ($u \in V_i, v \in V_j, i \neq j$) are called *cut edges*. Vertices adjacent to cut edges are *boundary vertices* of a cell. By recursively partitioning the cells of a partition, a *multi level partition* can be obtained: Given a graph G, the number of levels ℓ and the number of cells $k_1, \dots, k_\ell$ for each level, G is partitioned into k_ℓ cells $V_1^\ell, \cdots, V_{k_\ell}^\ell$. This partition is the level-ℓ partition, the original graph is assumed to have level 0. The partitions for the levels between are obtained by partitioning the subgraphs G_j^{i+1} induced by V_j^{i+1} into a total of k_i cells $V_1^i, \cdots, V_{k_i}^i$. The cell $V_g^i, 0 < g < k_i$ is a *subcell* of $V_j^{i+1}, 0 < j < k_{i+1}$ if and only if V_g^i is a subset of v_j^{i+1}.

2.3 Dijkstra's Algorithm

Routing on a graph can be done using *Dijkstra's algorithm* [Dij59] (Algorithm 2.1). Given a graph G, a start vertex s and a weight function ω, the algorithm returns a shortest path tree rooted at s. For each vertex v the algorithm stores the tentative distance $\mathrm{dist}(v)$ from s to v and its predecessor $\mathrm{pred}(v)$ in the shortest path tree. The tentative distance is the length of the shortest path from s to v Dijkstra's algorithm has found yet and may not be equal to the actual distance $d(s, v)$. The algorithm also manages a priority queue Q in which discovered vertices for which their tentative distance may not yet equal the actual distance are stored with the key being their tentative distance. Initially the tentative distances of all vertices are set to infinity and their predecessor is set to `null`. The distance of s is set to 0 and s is inserted into Q with key 0. Then, as long as Q is not empty, the vertex u with the smallest key is *settled*. Settling a vertex v means extracting it from Q. Its tentative distance $\mathrm{dist}(u)$ from s will no longer change and is equal to the distance $\mathrm{d}(s, u)$ from s to u. Also all adjacent edges (u, v) are *relaxed*. This means that if the tentative distance $\mathrm{dist}(v)$ is greater than $\mathrm{dist}(u) + \omega(u, v)$ it is set to $\mathrm{dist}(u) + \omega(u, v)$ and the predecessor $\mathrm{pred}(v)$ is set to u. If v is already in Q its key is decreased to $\mathrm{dist}(v)$, otherwise v is inserted into Q with key $\mathrm{dist}(v)$ and v is now considered *discovered*. Dijkstra's algorithms runs in $O(|V| \log |V| + |E|)$ time if using Fibonacci heaps as priority queue [CLRS01]. By inserting edges into Q and keeping distances and predecessors for edges instead of vertices the algorithm can be adapted to *edge based Dijkstra* which runs in $O(|E| \log |E| + |P_3(G)|)$ time, where $P_3(G)$ is set of all paths containing three vertices. The runtime can be trivially deduced from [Vol08] and [Win02].

Algorithm 2.1: Dijkstra

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

// Initialization
forall $v \in V$ do
    $\mathsf{d}(v) \leftarrow \infty$
    $\mathsf{pred}(v) \leftarrow$ null
$\mathsf{Q}$.insert$(s, 0)$
$\mathsf{d}(s) \leftarrow 0$

// Main loop
while $\mathsf{Q}$ is not empty do
    $u \leftarrow \mathsf{Q}$.deleteMin$()$
    forall $(u, v) \in E$ do
        if $\mathsf{d}(u) + \omega(u, v) < \mathsf{d}(v)$ then
            $\mathsf{d}(v) \leftarrow \mathsf{d}(u) + \omega(u, v)$
            $\mathsf{pred}(v) \leftarrow u$
            if $\mathsf{Q}$.contains$(v)$ then
                $\mathsf{Q}$.decreaseKey$(v, \mathsf{d}(v))$
            else
                $\mathsf{Q}$.insert$(v, \mathsf{d}(v))$
```

3. Problem Statement

The goal of our work is, given a start point and a *range*, a maximal distance from a starting point, to provide a visual indication on a map which area can be reached without exceeding the range. The boundary of this area typically is simple polygon, but due to geographical or traffic-related circumstances the area might enclose other, unreachable areas. In this case multiple polygons are required to draw the boundary of the reachable area. We require the graph to be strongly connected as vertices in other components than the starting vertex will always be unreachable. A formal definition of the problem is as follows.

Problem 1. RangePolygon*: Given a strongly connected graph $G = (V, E)$, an embedding of G into the plane, a start vertex $s \in V$, a weight function ω and a range $r \in \mathbb{R}^+$. Compute a connected area A, delimited by a set of polygons P, that contains exactly those vertices $v \in V$ that are* reachable*, i.e., whose distance* $\mathrm{d}(s, v)$ *from s is smaller or equal to r.*

A simple solution to this problem is to have a large polygon enclosing all vertices of V and several small polygons surrounding vertices v with a distance $\mathrm{d}(s, v)$ greater than r, as illustrated in Figure 3.1. Since the goal of this work is visual representation of the reachable area, such a solution is irritating at best. To force the area to more closely resemble the area that is actually reachable, we now introduce the boundary of the reachable area.

Given a triangulated graph $G = (V, E)$ and a subset $S \subseteq V$ of reachable vertices, an edge $(u, v) \in E$ is called *boundary edge* if and only if it connects a reachable vertex $u \in S$ with an unreachable vertex $v \in V \setminus S$. All boundary edges intersect the *cut* between S and $V \setminus S$. Faces adjacent to a boundary edge are *boundary faces.* Edges of a boundary face that are not boundary edges are part of the *outer border* if they connect two unreachable vertices, or part of the *inner border* if they connect two reachable vertices. We call the subgraph containing all boundary faces and their adjacent vertices and edges the *boundary.*

Lemma 3.1. *Each boundary face is delimited by exactly two boundary edges and exactly one edge that is part of either the inner or the outer border.*

Proof. Since every face of a triangulation has three adjacent vertices, there are four cases based on the number of vertices reached: If none or all three of the vertices are reached this face is not part of the boundary. If one vertex is reachable the edges connecting it with the other two vertices of this face are boundary edges while the edge connecting

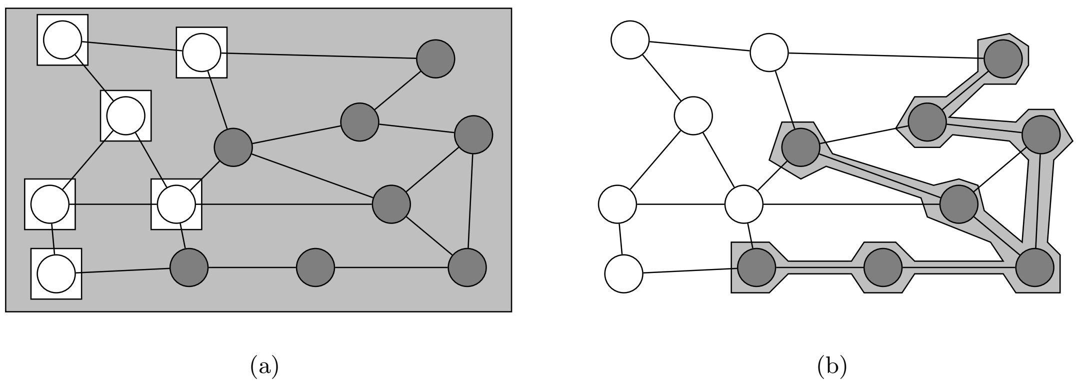


Figure 3.1: Trivial solutions for RangePolygon: Filled vertices are reachable, unfilled ones not. The gray area represents the reachable area A of RangePolygon. (a): Everything is marked as reachable except small areas surrounding unreachable vertices. (b): Only a small shape connecting all reached vertices.

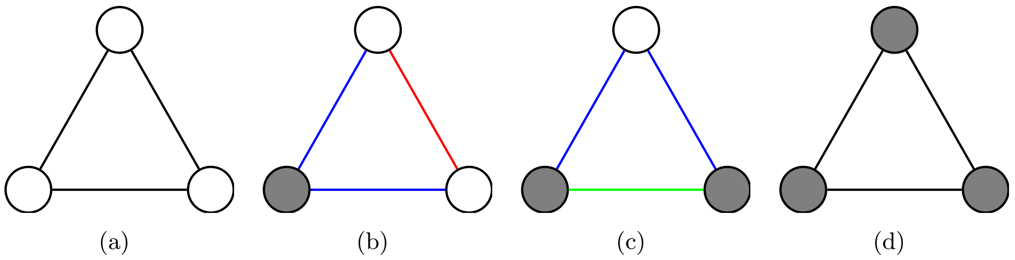


Figure 3.2: Classification of edges adjacent to boundary faces based on the number of reachable vertices. Filled vertices are reachable. (a): No reachable vertex, face outside the boundary. (b), (c): One or two reachable vertices, face is a boundary face; blue edges are boundary edges, green edges belong to the inner border and red edges to the outer border. (d): All Vertices reachable, face inside the boundary.

the unreachable vertices is part of the outer border. If two vertices are reachable the edge between them is part of the inner border and the edges connecting them with the unreachable vertex of this face are boundary edges. This is illustrated in Figure 3.2. □

Marching cubes share a similar idea [LC87]. The main differences are that we operate in 2D instead of 3D, our vertices are not arranged in a grid-like pattern, and we have no ambiguity about the course of the boundary since our faces are triangles.

Recall that the reachable area may enclose unreachable vertices. Due to these ”holes” the boundary may not be a single shape but form multiple ring-like shapes and some border edges may be shared by two boundary faces, as seen in Figure 3.3.

Using the definition of the boundary we can refine our problem to forbid solutions like the ones shown in Figure 3.1.

Problem 2. RefinedRangePolygon*: Given a triangulated graph $G = (V, E)$, a start vertex $s \in V$, a weight function ω, and a range $r \in \mathbb{R}^+$, compute a connected area A, delimited by a set of polygons P, that has the following properties. The area A includes a vertex $v \in V$ if and only if its distance $\mathrm{d}(s, v)$ from s is smaller or equal to r. Additionally, each boundary edge must intersect exactly one line segment of one polygon of P, while no polygon may intersect edges of the inner or outer border.*

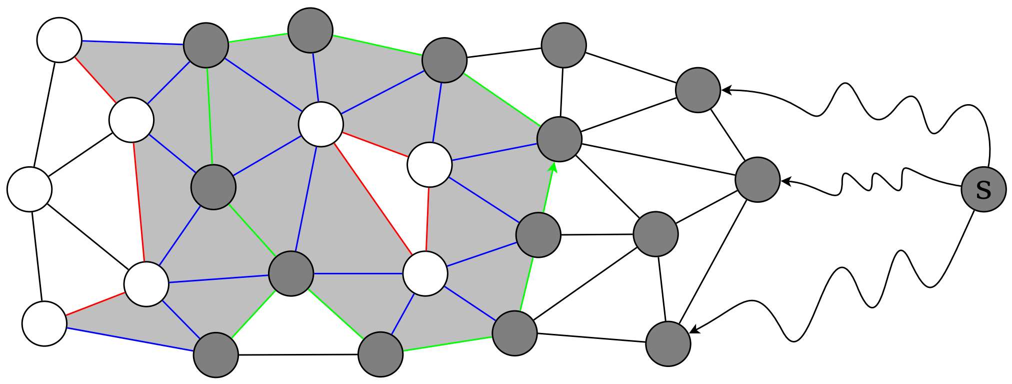


Figure 3.3: Reachable area enclosing a hole and boundary faces sharing the same border edge. Assume the source node is somewhere to the right of the shown part of the graph. Filled vertices are reachable, filled faces are boundary faces. Edge colors are the same as in 3.2(b): Blue edges are boundary edges, green edges are part of the inner border and red edges belong to the outer border.

We consider a solution for RefinedRangePolygon which uses less line segments of higher quality In cases where exact representation of the boundary is not required it may be permissible to allow errors in the visualization to reduce the number of line segments needed to draw P. Thus we formulate a weaker variant of RefinedRangePolygon allowing errors to achieve a more appealing visualization:

Problem 3. RelaxedRangePolygon*: Given a triangulated graph $G = (V, E)$, a start vertex $s \in V$, a weight function ω, and a range $r \in \mathbb{R}^+$, compute a connected area A delimited by a set of polygons P. The number of line segments needed to draw P, the amount of unreachable area included in A and the amount of reachable area not included in A must be minimized.*

Weighting these criteria against each other depends on the circumstances of a query, e.g., a query like ”Where can I get in about one hour?” may allow moderate errors on both sides of the boundary. However if the range polygons are for electric cars marking a charging station as reachable that actually is too far away might cause the car to run out of fuel.

The RefinedRangePolygon problem can be split into two smaller problems: First determining the boundary, which is covered in Chapter 4 and second using the boundary to create the range polygons, which is covered in Chapter 5.

4. Determining the Boundary

In this chapter we address the first subproblem of RefinedRangePolygon, determining the boundary.

4.1 Precalculations

In Chapter 3 we defined the boundary as a subgraph containing all boundary faces with their adjacent vertices and edges. Since the boundary is defined on a triangulation of the original graph, we first requires a planarization. For road networks the geographical coordinates of the vertices are an obvious choice for a (possibly non-planar) embedding. Because the planarization introduces additional paths, it seems preferable to route on the original graph and then determine the boundary on the triangulated graph. Translating between the vertices of the original and the triangulated graph poses problems as the reachability of vertices v added by the planarization cannot be trivially tested. A vertex v is only reachable if for at least one edge (u, v) the distance $\mathrm{d}(u) + \omega(u, v)$ is smaller than the range r. Instead we express the additional paths as turning restrictions, which are a natural occurrence in road networks anyway.

4.2 Basic Routing Algorithm

Since determining the boundary is a routing based problem, our first approach is using Dijkstra's algorithm. A vertex v is *reachable* if and only if the distance $\mathrm{d}(s, v)$ from the start vertex s to v is less or equal to the maximum range r. However, Dijkstra's algorithm does not handle turning restrictions. For example, on an intersection with left turns forbidden an alternative is going straight followed by three right turns and finally going straight again. This crosses the intersection twice which would translate to settling a vertex twice while running Dijkstra's algorithm. The edge based version of Dijkstra's algorithm avoids this problem while also making implementation of turning restrictions trivial [GV11].

We can reduce the size of the data structures kept by our version since we only care whether vertices are reachable or not. Compared to Dijkstra's algorithm we neither need the distance $\mathrm{dist}(s, \cdot)$ nor the predecessor $\mathrm{pred}(\cdot)$ of a vertex. Note that by not storing the predecessor we can no longer construct a shortest path tree. The result is the following algorithm. For a pseudo-code description see Algorithm 4.1.

For each vertex we keep information whether we found the vertex to be reachable, for each edge we store if we already relaxed it and if it is currently part of the boundary.

Algorithm 4.1: Adapted Dijkstra

```
Input: Graph $G = (V, E)$, source node $s$, weight function $\omega$, turning restrictions
       function $R$, maximum range $r$
Data: Priority queue Q
Output: Reachability reachable($v$) for all $v \in V$, Boundary onBoundary($e$) for all
        $e \in E$

// Initialization
1  forall $v \in V$ do
2      reachable($v$) $\leftarrow$ false
3  forall $(u, v) \in E$ do
4      relaxed($(u, v)$) $\leftarrow$ false
5      onBoundary($(u, v)$) $\leftarrow$ false
6  reachable($s$) $\leftarrow$ true
7  forall $(s, v) \in E$ do
8      Q.INSERT($(s, v), \omega(s, v)$)
9      relaxed($(s, v)$) $\leftarrow$ true
10     onBoundary($(s, v)$) $\leftarrow$ true

// Main loop
11 while Q is not empty do
12     $k \leftarrow$ Q.MINKEY()
13     if $k > r$ then
14         break
15     $(u, v) \leftarrow$ Q.DELETEMIN()
16     reachable($v$) $\leftarrow$ true
17     forall $(v, w) \in E$ do
18         if relaxed($(v, w)$) = false and R.ISRESTRICTED($u, v, w$) = false then
19             Q.INSERT($(v, w), k + \omega(v, w)$)
20             relaxed($(v, w)$) $\leftarrow$ true
21             onBoundary($(w, v)$) $\leftarrow$ false
22             if reachable($w$) = false then
23                 onBoundary($(v, w)$) $\leftarrow$ true
24 return reachable, onBoundary
```

We also manage a priority queue Q containing edges (u, v) with the tentative distance $\text{dist}(s, u) + \omega(u, v)$ at the end of the edge as key. The edges inside Q were relaxed by the algorithm, but are not yet settled. Initially, we mark every vertex besides the start vertex s as not reachable and every edge as not relaxed.

All edges (s, v) are inserted into the priority queue Q with their weight $\omega(s, v)$ as key and marked as relaxed. They are also marked to be part of the boundary. Now, as long as Q is not empty, the edge (u, v) with the smallest key k is removed from Q. If k is greater than the maximum range r, we end the algorithm and return the vertices marked as reachable and the edges marked as being on the boundary. Otherwise the edge is settled: The vertex v is marked as reachable and all edges (v, w) that are not marked as relaxed yet and for which the turn (u, v, w) is allowed are inserted into Q with $k + \omega(v, w)$ as key and marked as relaxed. Also, all edges (v, w) for which w is not marked as reachable are marked to be on the boundary, while for all edges (w, v) this mark is cleared.

Lemma 4.1. *Once an edge $(u,v) \in E$ is settled, all edges $(v,w) \in E$ which were not relaxed yet and for which the turn (u,v,w) is allowed are preceded by (u,v) in a shortest path from s containing (v,w).*

Proof. We show this by induction. The first edge settled is the edge (s,v) with the lowest weight. Since there is no shorter outgoing edge from s, (s,v) is a shortest path from s to v. Assuming there is a path $(s,\dots,v,w)$ which is shorter than (s,v,w), there must be a path from s to v shorter than (s,v). This is a direct contradiction to (s,v) being a shortest path.

It remains to show that when settling (u,v) and the above conditions are met, (u,v) is the first valid predecessor for (v,w). If v was not yet reached (u,v) must be on a shortest path from s to v. Assume there is a path $(s,\dots,w,x,\dots,v)$ that is shorter than $(s,\dots,u,v)$. One edge of $(s,\dots,w,x,\dots,v)$ must be in the priority queue, since initially all edges $(s,\cdot)$ are inserted into the queue and v was not reached yet. We call this edge (w,x). Since (u,v) is removed earlier from Q than (w,x), the path $(s,\dots,u,v)$ is not longer than $(s,\dots,w,x)$. Combining both inequalities leads to $(s,\dots,w,x,\dots,v)$ being shorter than $(s,\dots,w,x)$ or $(x,\dots,v)$ being shorter than the empty path. This contradicts the assumption Thus (u,v) is part of a shortest path from s to v, v is now marked as reachable, boundary markings of incident edges are updated and (u,v) is the predecessor for any edge (v,w) that was not relaxed yet and for which the turn (u,v,w) is allowed. Otherwise, if v was already reached, for all edges (v,w) which are not yet relaxed and for which the turn (u,v,w) is allowed, the turn (u',v,w) is forbidden for all edges (u',v) reaching v earlier, since otherwise one of those edges would have relaxed (v,w). Thus if there is a shortest path from s to another vertex t containing (v,w), there is a path from s to t of equal length that contains (u,v) immediately before (v,w). □

Theorem 4.2. *After the algorithm terminates all vertices and edges are correctly marked as reachable or being on the boundary.*

Proof. Directly follows from Lemma 4.1. □

4.3 Acceleration

As running our version of Dijkstra's algorithm turned out to be significantly slower than the visualization (see Chapter 6), we decided to use *Customizable Route Planning* (CRP) [DGPW11] as speedup technique. While other speedup techniques adaptable to our use case, like PHAST, exist, we prefer CRP because it works well with any metric. CRP was designed with three goals: It should allow fast customization to previously unknown metrics, the space required to store metric dependent information must be small and it must still allow real-time queries. All three goals must hold true for any metric. This is achieved by splitting the work into three steps: First, metric-independent precalculations, which may require some days, second, metric-dependent precalculations, which may take a few seconds, and third, answering the actual queries, which must be fast enough for interactive programs.

4.3.1 Basic CRP

The metric-independent precalculations consists of creating a multi level partition of the graph. This partition should minimize the number of cut edges, while keeping the number of vertices in each cell balanced. The time required for the later steps heavily depends on the number of cut edges created by the partition.

During the metric dependent precalculations, for each cell of the partition the distances between all vertices having edges leaving the cell, called *cell boundary vertices*, are calculated. Starting with lowest level of the multi level partition, for each cell we calculate the distance between all cell boundary vertices by running many-to-many queries on the subgraph induced by the vertices of the cell using Dijkstra's algorithm. Between each pair of cell boundary vertices we then insert *shortcut edges* using their distance as weight. For a specific level the *overlay graph* is the graph containing all boundary edges of all cells of that level and the cut and shortcut edges connecting them. For cells of higher levels the many-to-many queries are run using only the overlay graph of the level below.

One-to-one queries, from source vertex s to target vertex t, are answered by using a bidirectional variant of Dijkstra's Algorithm. Once boundary vertices are encountered the algorithm increases the level it is operating on to the highest level this vertex is a boundary vertex for, but not beyond the lowest level for which start and target vertex reside in the same cell. It then continues on the overlay graph of that level. When the first vertex v is settled by both, the forward and the backward search, the query is finished and the distance $\mathrm{d}(s,t)$ between s and t is $\mathrm{d}(s,v) + \mathrm{d}(v,t)$.

4.3.2 Our Adaptations

For our range queries we need to make some adaptations: During the metric-independent precalculations edges added by the triangulation are ignored since they do not help routing, but significantly increase the number of cut edges. As stated above, keeping the number of cut edges low is required for fast queries.

Since our range queries have no target vertex, we cannot use a bidirectional version of Dijkstra's algorithm and instead use the original one. Whenever we encounter a boundary vertex v while running Dijkstra's algorithm, we always try to continue using the overlay graph of the highest level v is a cell boundary vertex for. However, as long as we find at least one shortcut edge for the current level adjacent to v that cannot be traversed with the remaining range, we descend to the overlay graph of the next lower level and retry. If all adjacent shortcut edges of the cell are traversable we mark this cell as completed. This serves two purposes: First descending from completed cells is unnecessary as we already know all vertices within to be reachable. Second during visualization we might encounter vertices that are marked unreachable because CRP found a cell they are in to be completely reachable. This happens if edges added by the triangulation connect vertices of different cells of which at least one is not a boundary vertex. Since we are ignoring these edges when creating the partition this is a common occurrence.

In Chapter 3 we noted that the reachable area may enclose unreachable vertices. If for a cell boundary vertex of a cell containing such "hole" all shortcut edges are traversable, our adaptation of CRP would incorrectly mark the cell as completed and not find that hole. To compensate we compute the diameter of each cell during the metric-dependent precalculations and revise the condition causing us to descend: Instead of descending if some shortcut edges are not traversable, we now descend if the cells diameter is greater than the remaining range.

Marking the edges of the boundary in a similar way to our adaption of Dijkstra's algorithm does not work, because it only marks settled vertices as reachable. CRP however tries not to explore all vertices and thus only boundary vertices and regions where we descended on the original graph would be marked reachable. Instead we mark an edge (u, v) to be part of the boundary only if the following three conditions are met:

- The edge (u, v) must be part of the original graph (i.e. not a shortcut edge),
- the edge can not be traversed, and

- the vertex v is neither marked as reachable nor one of the cells v is in is completed.

This might not find all boundary edges, but at least one per unreachable area. Due to the original graph being strongly connected for each unreachable area there must be at least one edge e not added by the triangulation connecting a reachable vertex u with a vertex v of the unreachable area. If e is not a cut edge, it will force our adaptation of CRP to descend to the original graph since the diameter of the cells containing u (or v) are greater than the remaining range. In either case we will settle u and while doing so relax e, which will mark e as being on the boundary since it cannot be traversed as its weight is greater than the remaining range. In Chapter 5 we will show that finding a single boundary edge per unreachable area suffices to determine a correct solution.

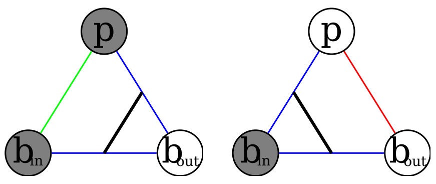
5. Visualization

In this chapter we address the second subproblem of RefinedRangePolygon. Given the boundary create a visual representation of it.

The goal for the visualization is to quickly convert the boundary into a set of polygons solving the RefinedRangePolygon problem. First we split the boundary into several *"tubes"*. Each tube contains the information required to create a single polygon for the final solution. Starting from a face adjacent to an arbitrary boundary edge, we store the coordinates of the vertices of the boundary edge we picked and whether they are reachable into the tube. We then traverse the triangulation crossing only boundary edges. For each face encountered we store the coordinates of the vertex not shared with the previous face and whether it is reachable into the tube. All boundary edges encountered while doing this are marked as not being on the boundary to prevent converting the same part of the boundary multiple times. From this we can deduce that it suffices to only mark one edge as being on the boundary to find the corresponding tube. The time required to gather the tubes depends on the time required to find an arbitrary marked edge and to clear that marking on random edges and lies in $O(t \cdot O(\textsc{FindMarkedEdge}) + b \cdot O(\textsc{ClearMarking}))$, where t is the number of tubes and b the number of boundary edges.

5.1 Basic Approach

Our basic approach is similar to marching cubes: For each boundary face we use the coordinates used for embedding to calculate the middle points of the two boundary edges (the blue edges in Figure 3.2) and draw a line between them. While iterating over the points of a tube we save the last point we encountered on each side of the boundary as b_{in} and b_{out}. Initially these are the first two points in the tube, which are guaranteed to lie on different sides of the boundary. We then iterate over the points in the tube, starting with the third. We will call the point we are currently considering p. Assuming p is reachable, we draw a line segment between the middle points of $(b_{\text{in}}, b_{\text{out}})$ and (p, b_{out}), and then replace b_{in} by p. If, when calculating middle points, one of the points happens to be the infinite vertex, which has no defined coordinates, we use the point that is not the infinite vertex as middle point. If p is not reachable we do the same, but replacing occurrences of b_{in} with b_{out} and vice versa. A demonstration for both cases can be seen in Figure 5.1. We then continue with the next point until we reach the first two points of the tube again and close the polygon. An example is shown in Figure 5.2(a). This basic approach



(a) Vertex p is reachable (b) Vertex p is not reachable

Figure 5.1: Demonstration showing the line segment drawn (thick line), depending on whether the vertex p is reachable. Filled vertices are reachable. Blue edges are boundary edges, green edges are part of the inner border, red ones belong to the outer border.

generates one line segment per boundary edge and takes $O(b)$ time where b is the number of boundary edges.

5.2 Stabbing

The basic approach tends to create zigzag lines due to always using the middle points of boundary edges, as shown in Figure 5.2(a). Most of these zigzag lines could be replaced by fewer straight lines intersecting multiple boundary edges, which reduces drawing complexity. This caused us to improve our basic approach: Instead of simply connecting middle points, we now try to minimize the number of line segments of the polygon. We try to greedily expand the current line segment until we can no longer do so without violating RefinedRangePolygon.

Besides b_{in} and b_{out}, we now also save the start point s of the current line segment and a cone c originating from s. Initially b_{in} and b_{out} are the first two points in the tube and s is their middle point. The cone c contains the semicircle originating from s and delimited by the lines $c_{\text{in}} = (s, b_{\text{in}})$ and $c_{\text{out}} = (s, b_{\text{out}})$ that includes the third point of the tube. We iterate over the points in the tube, starting with the third, and call the the currently considered point p. From now on we will assume p is reachable, if p is not reachable the same algorithm applies, but with occurrences of the indices "in" and "out" swapped. First we check the position of p compared to c. There are three cases, which are illustrated in Figure 5.3. If p is inside the cone c, c_{in} is updated to (s, p). This makes the cone smaller, which is done because the line segments excluded by updating c_{in} do not intersect the current boundary edge (p, b_{out}). If p lies outside of c and both lie on different sides of c_{in}, we do nothing. Expanding the cone would allow line segments which do not intersect all previously encountered boundary edges. If p lies outside of c and both lie on the same side of c_{in}, updating c_{in} would cause the order of the lines delimiting c to swap. This means we cannot draw a straight line from s to any point of the boundary edge (p, b_{out}) without violating RefinedRangePolygon. Therefore we draw a line from s to the last boundary edge and set up a new starting point and cone. This is done by calculating the intersection point q of c_{out} and $(b_{\text{in}}, b_{\text{out}})$, drawing a line from s to q and setting s to q. Drawing the first line for each tube is skipped to allow removing or optimizing the placement of the first starting point. The cone c is updated to the new starting point by setting its delimiting lines to $c_{\text{in}} = (s, p)$ and $c_{\text{out}} = (s, b_{\text{out}})$. The boundary edge (p, b_{out}) is considered to be on the inside of c. In the degenerate case, i.e., when s equals b_{out}, c_{out} is set to (s, b_{in}) and the edge (b_{in}, p) is considered to be outside of c. In any case, independent of the position of p, b_{in} is replaced by p and the iteration continues with the next point. We stop iterating over the tube when we reach the starting point of the first drawn line segment again. Algorithm 5.1 is a pseudo-code representation of the described algorithm.

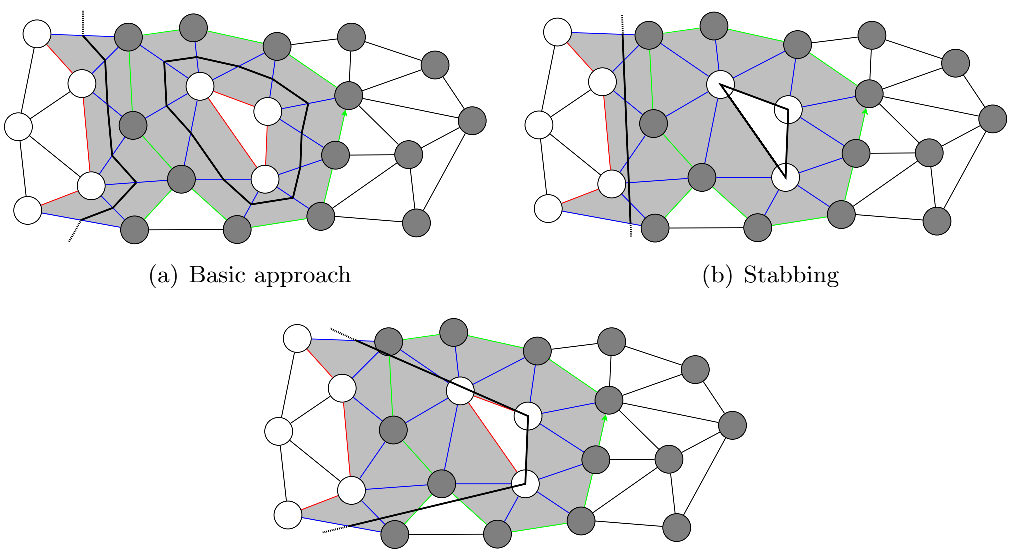

(a) Basic approach (b) Stabbing (c) Merging

Figure 5.2: Example outputs of different versions of our approach. The source vertex for the query is assumed to be on the right of the shown part of the graph. Filled vertices are reachable, filled faces are boundary faces. Colored edges are also part of the boundary: Blue edges are boundary edges, green edges are part of the inner border, red ones belong to the outer border. Thick lines represent line segments generated by the different versions.

The stabbing approach also requires $O(b)$ time, but has a higher constant compared to the basic approach due to geometric computations. An example is shown in Figure 5.2(b).

5.3 Merging

While using the stabbing approach is a huge improvement over the basic one, it still is an exact representation of the boundary, and as such replicates every feature of it. Motorway intersections that are only partially reachable typically cause the stabbing approach produces many short lines. Roads causing an outward ”spike” are another example where the visualization can be improved by allowing errors. Therefore we allowed errors, and as result no longer solve RefinedRangePolygon but RelaxedRangePolygon. We first generate an exact solution using the stabbing approach. Then we iterate over all generated polygons recursively merging adjacent lines if their uncommon endpoints are closer than a given distance d and we do not mark unreachable area as reachable. For distance based queries we choose $d = r/10$. As we forbid including unreachable area this is a conservative solution for RelaxedRangePolygon.

While this improves the visualization in the mentioned cases, we noticed the outer polygon intersecting polygons created for holes near the outer polygon, which definitely is undesirable. To amend this we also decided to merge hole polygons into the outer polygon if they are close enough to possibly intersect. Before merging line segments as described above, we find pairs (p, q) of points of the outer polygon and a hole polygon whose distance is less than $d/2$. The hole polygon is then merged into the outer one by splitting both polygons at p and q, and inserting two lines (p, q) and (q, p) connecting both polygons. As the hole polygon is now part of the outer polygon we check the remaining hole polygons for points near the new part of the outer polygon and recursively merge them if required. We chose the distance $d/2$ because any point lying on a line segment between two points, which are

Algorithm 5.1: Stabbing

```
    Input: Tube $T$
    Data: Cone $c$, delimited by lines $c_{\text{in}}$ and $c_{\text{out}}$, current start point $s$, current point
          $p$, last encountered points on each side $b_{\text{in}}$ and $b_{\text{out}}$
    Output: Range polygon $\mathsf{P}$ for tube $\mathsf{T}$
    // Initialization
 1  $\mathsf{P} \leftarrow ()$
 2  if $\mathsf{T}(1)$ is reachable then
 3      $b_{\text{in}} \leftarrow \mathsf{T}(1)$
 4      $b_{\text{out}} \leftarrow \mathsf{T}(2)$
 5  else
 6      $b_{\text{in}} \leftarrow \mathsf{T}(2)$
 7      $b_{\text{out}} \leftarrow \mathsf{T}(1)$
 8  $\mathsf{s} \leftarrow$ MIDPOINT($b_{\text{in}}$, $b_{\text{out}}$)
 9  $c_{\text{in}} \leftarrow (s, b_{\text{in}})$
10  $c_{\text{out}} \leftarrow (s, b_{\text{out}})$
11  firstLine $\leftarrow$ null
    // Main loop
12  for $i \leftarrow 3$, true, $i \leftarrow i + 1 \mod |\mathsf{T}|$ do
13      $p \leftarrow \mathsf{T}(i)$
14      if firstLine $= (p, b_{\text{out}})$ then
15          $\mathsf{P} \leftarrow \mathsf{P} \cup (\mathsf{P}(1))$
16          break
17      if $p$ is reachable then
18          if $p$ inside $c$ then
19              $c_{\text{in}} \leftarrow (s, p)$
20          else if $p$ and $c$ same side of $c_{\text{in}}$ then
21              $q \leftarrow$ INTERSECT($c_{\text{out}}$, ($b_{\text{in}}$, $b_{\text{out}}$))
22              if firstLine $=$ null then
23                  firstLine $\leftarrow (b_{\text{in}}, b_{\text{out}})$
24              $\mathsf{P} \leftarrow \mathsf{P} \cup (q)$
25              $s \leftarrow q$
26              $c_{\text{in}} \leftarrow (q, p)$
27              $c_{\text{out}} \leftarrow (q, b_{\text{out}})$
                // $(p, b_{\text{out}})$ is inside $c$
28              if $q = b_{\text{out}}$ then
29                  $c_{\text{out}} \leftarrow (q, b_{\text{in}})$
                    // $(p, b_{\text{in}})$ is outside $c$
30      else
            // Same as the if part, but with occurrences of the indices
               "in" and "out" swapped.
31  return $\mathsf{P}$
```

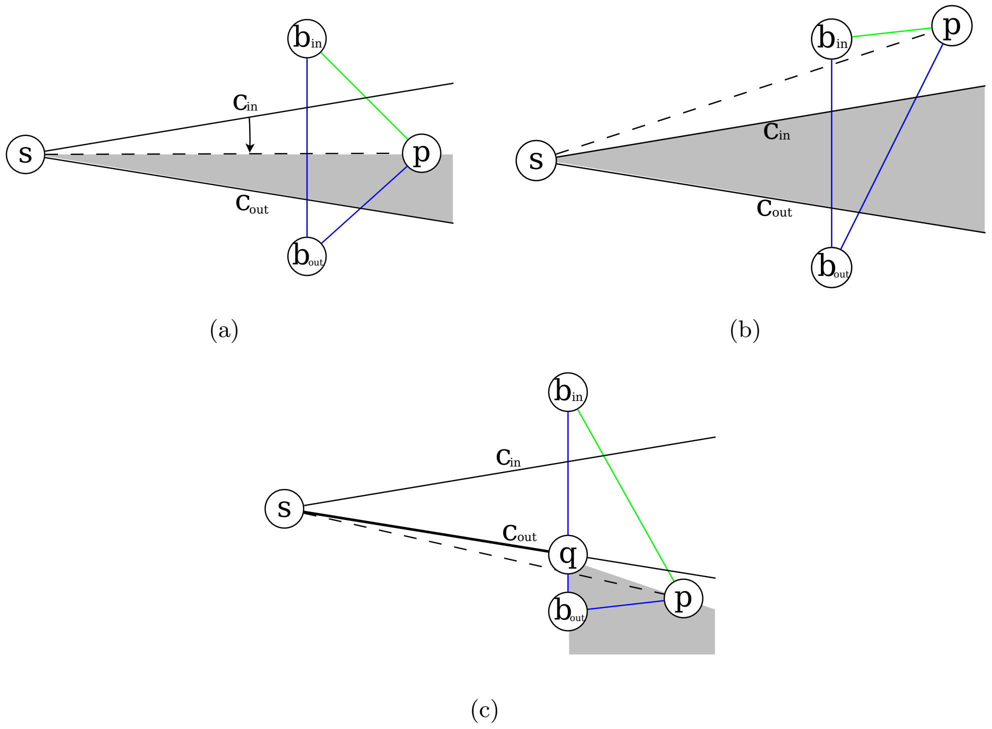


Figure 5.3: Updating the cone dependent on the position of p, assuming p is reachable. The non-dashed lines originating from s represent the lines c_{in} and c_{out} delimiting the current cone, the dashed line (s, p) is the currently considered update to the cone. The filled area represents the area inside the updated cone. The colored lines carry the same meaning as in earlier figures: Blue lines represent boundary edges, while green lines are edges of the inner border.
(a): The point p is inside the cone and the delimiting line c_{in} is updated to (s, p).
(b): The point p is outside of the cone, but p and the cone lie on different sides of c_{in}. Updating the cone would make it larger, which would allow lines that do not intersect all previously encountered boundary edges. Therefore the cone is not updated.
(c): The point p is outside of the cone and p and the cone lie on the same side of c_{in}. Note that the boundary edge (p, b_{out}) has no point inside the cone. Enlarging the cone is not an option for the same reason as in (b). Therefore we calculate the intersection point q of c_{out} and $(b_{\text{in}}, b_{\text{out}})$, draw the (thick) line segment (s, q) and update the cone to the new starting point.

a distance d apart, cannot be further away from the closest endpoint.As we are calculating the distance between all points of the outer polygon and all points of all hole polygons, the time required is quadratic in the number of line segments generated by stabbing. In the worst case (when generating one line segment for each boundary edge) the total time required is $O(b^2)$. An example can be seen in Figure 5.2(c).

5.4 Alpha Shapes

We will compare our approach with the Alpha Shape [EKS83] of all reached vertices. Alpha Shapes do not use the boundary but reconstruct the surface of a point set only using the parameter α. The parameter α is chosen to be the smallest value that results in a single shape (possibly with holes). Having multiple shapes would be interpreted as vertices in one shape not being reachable from the other. On the other hand a small α is desired to keep the inclusion of unreached vertices low.

6. Experiments

All experiments were run on a server with two Intel Xeon E5-2670 for a total of 16 cores clocked at 2.6GHz. The experiments only used a single core. The server has 64GiB of RAM and runs openSUSE 12.2. Our programs were compiled using GCC version 4.7.1 with full optimizations and disabled assertions (i.e. `-O3 -DNDEBUG`). The road network used was the German road network which was provided by the PTV AG. It has 4692092 vertices and 10806191 edges. After planarization the graph has 4745133 vertices and 11423669 edges. 17047117 additional edges were added by the triangulation.

For the precalculations we used a slower machine with four AMD Opteron 6172, a total of 48 cores clocked at 2.1GHz. This machine has 256GiB of RAM and also uses openSUSE 12.2 as operating system. The precalculations were split into two parts. Parsing and triangulating the map data only used a single core and took about 12 minutes for the German road network. For constructing a partition for CRP we used Buffoon [SS12], which took 28.5 hours and used 16 cores. The created partition has four levels. The highest level has 32 cells and the lower levels have 16 times the number of cells of the level directly above. The metric-dependent precalculations for CRP took less than a second.

Using the Euclidean distance in meters as metric, we ran queries for ranges ranging from 50km to 500km in 50km increments. For each range we chose 1000 start vertices uniformly at random and measured the time required to determine the boundary with our adaptations of Dijkstra's algorithm and CRP. Due to a bug in our adaptation of CRP, CRP sometimes incorrectly marked a few additional vertices as reachable, resulting in a slightly larger boundary compared to the one produced by Dijkstra's algorithm. Since these errors were small and we do not expect the runtime of a correct implementation to significantly change, we still present the time CRP used. We also measured the time needed to calculate and visualize the range polygons and the number of line segments produced by each version of our approach. For comparison we include Alpha Shapes and the convex hull. The results for the visualization are based on the boundary calculated by our adaptation of Dijkstra's algorithm.

Table 6.1 shows the average time used by our adaptations of Dijkstra's algorithm and CRP for each range. The speedups of CRP compared to Dijkstra's algorithm are significantly smaller than those achieved by one-to-one queries. This can be explained by CRP having to explore more vertices for range queries compared to one-to-one queries, where start and target are a given distance apart, whereas the work done by Dijkstra's algorithm is the same in both cases. The speedup of CRP increases as the range increases, since higher

Range	Dijkstra	CRP
50 km	118 ms	71.7 ms
100 km	331 ms	110 ms
150 km	680 ms	167 ms
200 km	1.11 s	256 ms
250 km	1.65 s	357 ms
300 km	2.07 s	450 ms
350 km	2.84 s	550 ms
400 km	3.16 s	618 ms
450 km	3.72 s	675 ms
500 km	4.21 s	711 ms

Table 6.1: Average time required to determine the boundary.

levels of the multi-level partition can be used and bigger parts of the graph can be "skipped over". However a maximal speedup of 6 is very slow compared to speedups of "more than 3000" [DGPW11] for one-to-one queries.

Table 6.2 shows average time required to run the visualization algorithms and Table 6.3 the average number of lines they generate. While the basic approach is by far the fastest visualization algorithm, it also produces one line per boundary edge. The stabbing approach reduces the number of line segments by about 90% but takes significantly longer. The time increase is caused by using CGAL's exact construction kernel, which uses an exact number representation for which arithmetic operations are slower. The inaccuracies of the inexact construction kernel, which uses doubles as internal number representation, would be acceptable for our work, but the stabbing approach tends to place polygon points on or very close to vertices of the boundary, causing geometric tests to frequently fail due to floating point errors. Using the merging approach reduces the number of line segments by another 66%. For small ranges the time increase caused by merging polygons is insignificant, for larger ranges the quadratic behavior becomes more apparent. For ranges above 400km the time required and (excluding the merging approach) the average number of line segments decreases. This happens because the border of the map is more frequently encountered, which causes edges adjacent to the infinite vertex of the triangulation to become boundary edges. As there is only one such edge per vertex in the convex hull, the number of edges on a section of outside of the convex hull is typically less than on an equally long section in the interior of the graph. With ranges of 450km and above, picking a vertex in the middle of Germany as start vertex range queries can cover the whole map. In that case only the edges adjacent to the infinite vertex are boundary edges causing the range polygon to be similar to the convex hull.

The construction of Alpha Shapes is slow because they require a triangulation of the point set. Reusing the triangulation calculated for the original graph is impractical as copying the triangulation and then locating and removing the points not in the alpha shape point set would take longer than just recalculating the triangulation unless almost all vertices are reachable.

A comparison between all versions of our approach, Alpha Shapes and the convex hull can be seen in Figure 6.1. While the area enclosed by all algorithms are similar, the created polygons are very different. The zigzag lines created by our basic approach are clearly visible 6.1(a). The part covering France however uses only a few, relatively long lines. This happens because, since we used the German road network, the graph contains no vertices outside of Germany and we can only encounter edges added by the triangulation, resulting in a sparse graph with large faces. Remember that the basic approach creates

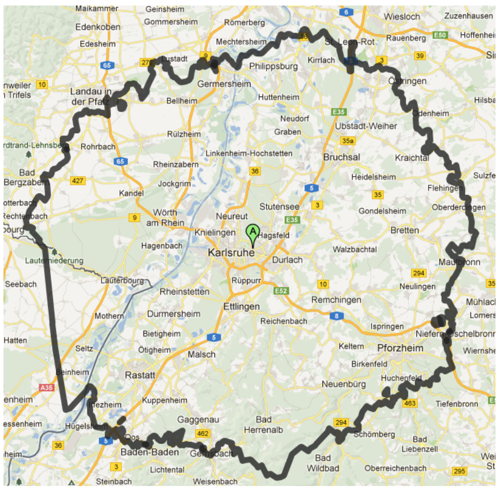
Range	Basic	Stabbing	Merging	Alpha Shapes
50 km	357 μs	14.1 ms	15.5 ms	1.83 s
100 km	677 μs	26.9 ms	33.5 ms	8.71 s
150 km	862 μs	34.4 ms	45.3 ms	19.5 s
200 km	1.02 ms	39.0 ms	54.1 ms	34.7 s
250 km	1.04 ms	43.7 ms	64.3 ms	-
300 km	1.16 ms	46.7 ms	77.3 ms	-
350 km	1.50 ms	52.1 ms	98.6 ms	-
400 km	1.40 ms	59.6 ms	122 ms	-
450 km	1.37 ms	58.1 ms	116 ms	-
500 km	1.33 ms	52.9 ms	98.3 ms	-

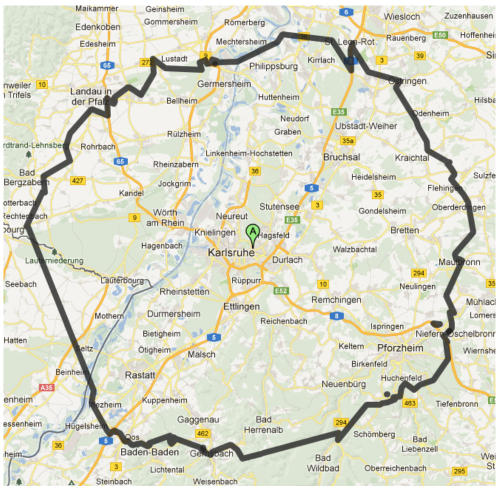
Table 6.2: Average time required for the visualization algorithms. The time required to calculate the convex hull is not included because Alpha Shapes were used to generate the convex hull. Both were not measured beyond the 200km range due to time constraints.

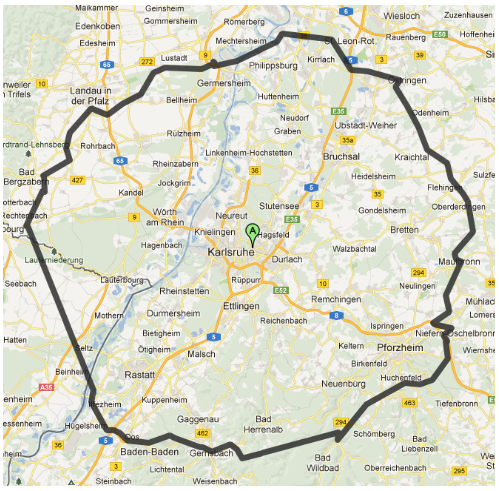
Range	Basic	Stabbing	Merging	Alpha Shapes	Convex hull
50 km	2642	241	106	498	33
100 km	5084	464	131	735	40
150 km	6530	598	143	926	43
200 km	7415	675	158	1160	44
250 km	8301	761	172	-	-
300 km	8686	803	210	-	-
350 km	9526	874	272	-	-
400 km	10276	988	390	-	-
450 km	9973	961	411	-	-
500 km	9006	859	402	-	-

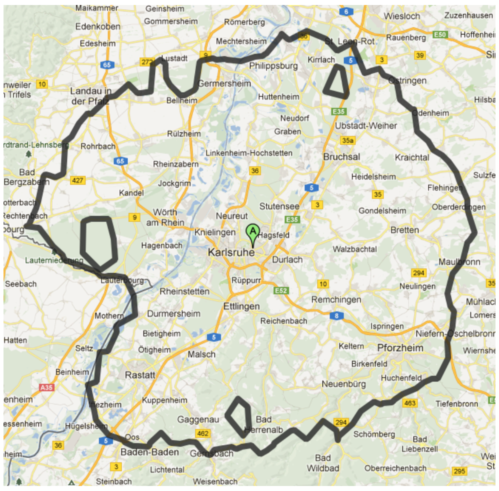
Table 6.3: Average number of lines generated for the visualization algorithms. For ranges greater than 200km range Alpha shapes and convex hull were not generated due to time constraints.

one line segment per boundary face encountered. The stabbing approach 6.1(b) is able to replace most zigzag lines with straight lines, significantly reducing the complexity of the polygons, resulting in a more appealing visualization. The merging approach 6.1(c) looks similar to the stabbing approach, but removes some ”spikes” (e.g., at the northern end of the A5, near the top right of the images) at the cost of marking some reachable areas as unreachable. Using the convex hull 6.1(e) in this case creates a smooth looking, but still accurate result. It marks unreachable areas as reachable however. Alpha Shapes produce the most different result 6.1(d), since it, like the convex hull, only uses the set of reachable vertices, but not the boundary. The outer polygon has some inward ”dents” and there are some holes, which can not be found in the other visualizations. These dents and holes are created because there are no reachable vertices within them, but Alpha Shapes depend on vertices being close to each other (less than $2\sqrt{\alpha}$ apart) to not create a hole between them. As result, sparse regions of the graph tend to be marked as unreachable by Alpha Shapes. In Figure 6.2 the merging approach marks a relativly large area as unreachable due to merging the hole in the lower part of the image. The query causing this result has a range of about 100km and the excluded area has a width of about 4km. In Figure 6.3 the convex hull create a large error, since it is not aware of the geographic setting of the query. The algorithms not shown in the last two figures behave in the same way as shown in Figure 6.1 and show no special proterties, the stabbing algorithm is shown as a correct solution to compare to.


(a) Basic approach


(b) Stabbing


(c) Merging


(d) Alpha Shapes

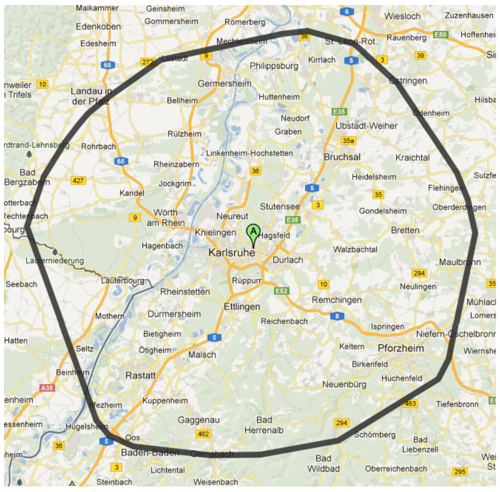
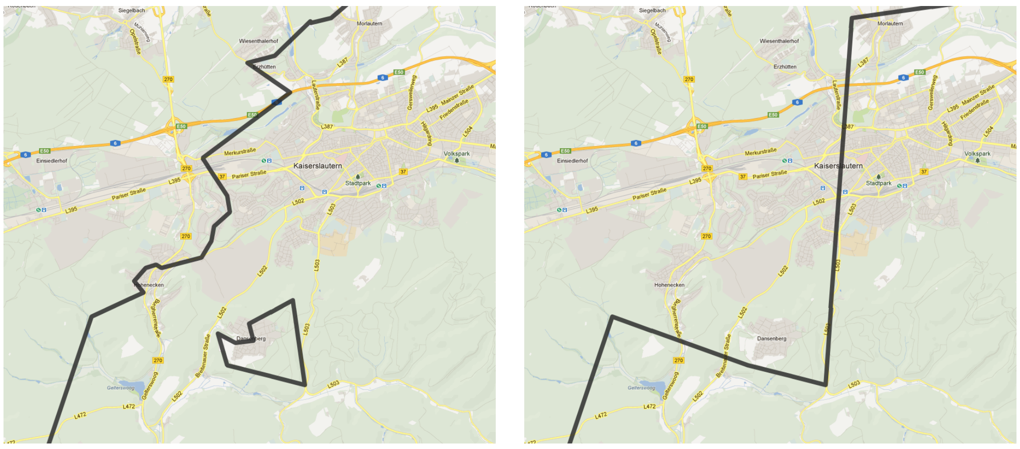
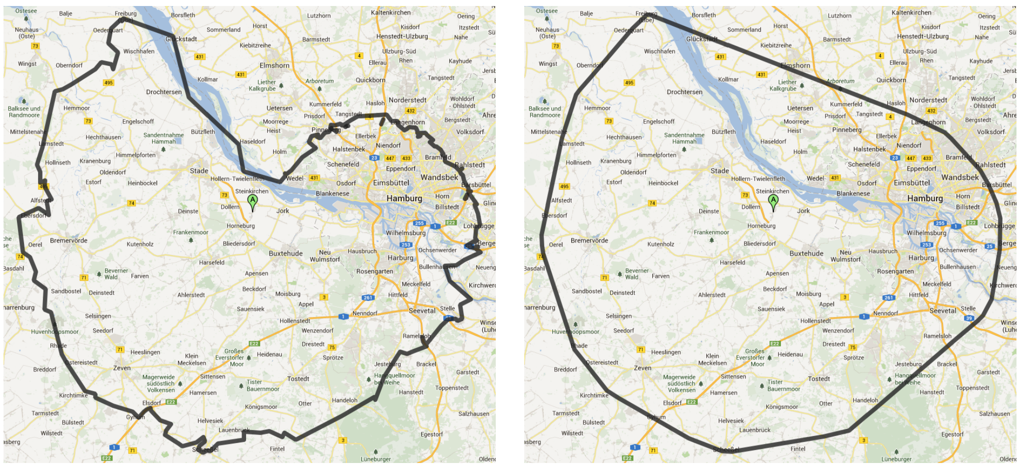

(e) Convex Hull

Figure 6.1: A comparison between all versions of our approach, Alpha Shapes and the convex hull. The A marker indicates the starting point.



(a) Stabbing (b) Merging

Figure 6.2: A comparison demonstrating an error created by the merging approach.



(a) Stabbing (b) Convex Hull

Figure 6.3: A comparison demonstrating an error created by the convex hull.

7. Conclusion

In this work, we first introduced the RangePolygon problem and defined the *boundary*. We then developed a solution for both subproblems. To determine the boundary of the reachable vertices within a given range of a given source vertex, we adapted Dijkstra's algorithm to this problem and used CRP as speedup technique. To create a visual representation of the boundary, we developed an algorithm related to marching cubes. To reduce the number of line segments generated we then replaced it by a line stabbing approach. We also introduced a relaxed version of the problem, RelaxedRangePolygon, and created a conservative visualization algorithm for it, which is based on the stabbing approach. Finally we compared the different algorithms with Alpha Shapes and the convex hull. As result we find, depending on whether exactness is required, the stabbing or the merging approach to create the best result. Both out-perform Alpha Shapes in terms of computation time required and number of line segments generated. The basic approach is much faster, but generates too many line segments.

Finally we raise some questions, defining areas for future work. We observed the speedup of using CRP instead of Dijkstra's algorithm for range queries to be very small compared the speedup during one-to-one queries. This raises the question if there are speedup techniques which make use of the fact that only the reachability of a vertex must be determined. Remember that neither the actual distance of a vertex from the start vertex nor the shortest path tree is needed.

Our implementation of the stabbing approach currently requires computations to be performed exact, due to errors introduced by normal floating point arithmetic. Identifying and avoiding the cases where standard floating point arithmetic fails could improve the runtime of the stabbing algorithm significantly. In [GHMS93] it is shown that minimizing the number of line segments for similar stabbing algorithms is NP-hard. We expect our version to be NP-hard as well. Can it be guaranteed that our version is never worse than a constant factor compared to the optimal solution, i.e., the solution with the minimal number of line segments? Are there other stabbing algorithms producing less line segments?

Can the RefinedRangePolygon problem be efficiently solved in dynamic use cases? Given the inputs for the RefinedRangePolygon and a path whose first vertex is the source vertex, are there algorithms to calculate the range polygons for all vertices along the path that can reuse information gained from an earlier vertices? What if the path is not known a-priori?

Bibliography

[CGA] CGAL Documentation, 2D Alpha Shapes. `http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Alpha_shapes_2/Chapter_main.html`. Accessed: 2013-05-30.

[Che89] L Paul Chew. Constrained delaunay triangulations. *Algorithmica*, 4(1-4):97–108, 1989.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[DGNW12] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 2012.

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In Pardalos and Rebennack [PR11], pages 376–387.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[EKS83] Herbert Edelsbrunner, David Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *Information Theory, IEEE Transactions on*, 29(4):551–559, 1983.

[GHMS93] Leonidas J Guibas, John E Hershberger, Joseph SB Mitchell, and Jack Scott Snoeyink. Approximating polygons and subdivisions with minimum-link paths. *International Journal of Computational Geometry & Applications*, 3(04):383–415, 1993.

[GV11] Robert Geisberger and Christian Vetter. Efficient Routing in Road Networks with Turn Costs. In Pardalos and Rebennack [PR11], pages 100–111.

[KS86] David G Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM journal on computing*, 15(1):287–299, 1986.

[LC87] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM Siggraph Computer Graphics*, volume 21, pages 163–169. ACM, 1987.

[LS80] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[PR11] Panos M. Pardalos and Steffen Rebennack, editors. *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*. Springer, 2011.

[SS12] Peter Sanders and Christian Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 16–29. SIAM, 2012.

[VK96] Marc Van Kreveld. Efficient methods for isoline extraction from a tin. *International Journal of Geographical Information Systems*, 10(5):523–540, 1996.

[Vol08] Lars Volker. Route Planning in Road Networks with Turn Costs, 2008. Student Research Project. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/volker_sa.pdf`.

[Win02] Stephan Winter. Modeling Costs of Turns in Route Planning. *GeoInformatica*, 6(4):345–361, 2002.