SCALABLE EXACT VISUALIZATION OF ISOCONTOURS IN ROAD NETWORKS VIA MINIMUM-LINK PATHS[*]

Moritz Baum,[†] *Thomas Bläsius,*[‡] *Andreas Gemsa,*[†] *Ignaz Rutter,*[†§] *and Franziska Wegner*[†]

Abstract. Isocontours in road networks represent the area that is reachable from a source within a given resource limit. We study the problem of computing accurate isocontours in realistic, large-scale networks. We propose isocontours represented by polygons with minimum number of segments that separate reachable and unreachable components of the network. Since the resulting problem is not known to be solvable in polynomial time, we introduce several heuristics that run in (almost) linear time and are simple enough to be implemented in practice. A key ingredient is a new practical linear-time algorithm for minimum-link paths in simple polygons. Experiments in a challenging realistic setting show excellent performance of our algorithms in practice, computing near-optimal solutions in a few milliseconds on average, even for long ranges.

1 Introduction

How far can I drive my battery electric vehicle, given my position and the current state of charge? This question expresses range anxiety (the fear of getting stranded) caused by limited battery capacities and a sparse charging infrastructure. An answer in the form of a map that visualizes the reachable region helps to find charging stations in range and to overcome range anxiety. This reachable region is bounded by curves that represent points of constant energy consumption; such curves are usually called *isocontours* (or *isolines*). Isocontours are typically considered in the context of functions $f\colon \mathbb{R}^2 \to \mathbb{R}$, e. g., if f describes the altitude in a landscape, then the terrain can be visualized by showing several isocontours (each representing points of constant altitude). In our setting, f would describe the energy necessary to reach a certain point in the plane. However, f is actually defined only for a discrete set of points, namely for the vertices of the road network. Thus, we have to fill the gaps by deciding how the isocontour should pass through the regions between the roads. The fact that the quality of the resulting visualization heavily depends on these decisions makes computing isocontours in road networks an interesting algorithmic problem.

Formally, we consider the road network to be given as a directed graph $G = (V, E)$, along with vertex positions in the plane and two weight functions $\text{len}\colon E \to \mathbb{R}_{\geq 0}$ and

[*]This work was partially supported by DFG Research Grant WA 654/16–2 and the Helmholtz Program Storage and Cross-linked Infrastructures, Topic 6 Superconductivity, Networks and System Integration. An earlier version appeared in the Proceedings of the 24th Annual European Symposium on Algorithms [5].

[†]*Karlsruhe Institute of Technology, Germany*, firstname.lastname@kit.edu

[‡]*Hasso Plattner Institute, Germany*, thomas.blaesius@hpi.de

[§]*TU Eindhoven, The Netherlands*, i.rutter@tue.nl cons: $E \to \mathbb{R}_{\geq 0}$ representing length and resource consumption, respectively. For a source vertex $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$, a vertex $v \in V$ belongs to the *reachable subgraph* if the shortest path from s to v has a total resource consumption of at most r, i. e., shortest paths are computed according to the length, while reachability is determined by the consumption.

Coming back to our initial question concerning the electric vehicle, the source vertex is the initial position and the range is the current state of charge. There are different possible notions of the reachable part of the road network in this situation. If we assume the driver is traveling on fastest routes (as proposed by, e. g., an onboard navigation system), the length corresponds to travel time and energy is the resource consumed on an edge. Alternatively, one could be interested in vertices that are reachable on at least *some* route (typically, the shortest route is not the most energy efficient one). In this case, both length and resource consumption would correspond to the energy consumption on a road segment (i. e., len $\equiv$ cons). For simplicity, we assume that all edge weights are nonnegative. In fact, negative weights are commonly used to model energy recovery of electric vehicles, but any issues arising in this context can be resolved with known techniques [7, 36].

Note that our setting is sufficiently general to allow for other applications. By setting the length as well as the resource consumption of edges to the travel time, one obtains the special case of *isochrones*. There is a wide range of applications for isochrones, including reachability analyses [4, 24, 25, 48], geomarketing [19], and environmental and social sciences [35]. Known approaches focus on isochrones of small or medium range. But isochrones can be useful in more challenging scenarios, for example, to visualize the area reachable by a truck driver within a day of work. This motivates our work on fast isocontour visualization.

Our algorithms for computing isocontours in road networks are guided by three major objectives. The isocontours must be exact in the sense that they correctly separate the reachable subgraph from the remaining *unreachable subgraph*; the isocontours should be polygons of low *complexity* (i. e., consist of few segments, enabling efficient rendering and a clear, uncluttered visualization); and the algorithms should be sufficiently fast in practice for interactive applications, even on large inputs of continental scale.

Figure 1 shows an example of isocontours visualizing the range of an electric vehicle. Figure 1a depicts a polygon that closely resembles the output of isocontour algorithms considered state-of-the-art in recent works [18, 19, 25, 42]. However, the number of segments becomes quite large even in this medium-range example (more than 10 000 segments). Figure 1b shows the result of our approach presented in Section 5.3, which contains the same reachable subgraph but uses 416 segments in total.

Related Work. Several existing algorithms consider the problem of computing the subnetwork that is reachable within a given timespan. The MINE algorithm [24] runs a search based on Dijkstra's well-known algorithm [14] to compute isochrones in transportation networks. Improved variants exist to reduce space requirement [25, 40] and compute multiple isocontours with different ranges from a given source [41]. However, working on spatial databases, MINE and its extensions are too slow for interactive applications (with running times in the order of minutes for long ranges). To enable much faster shortest-path computation in

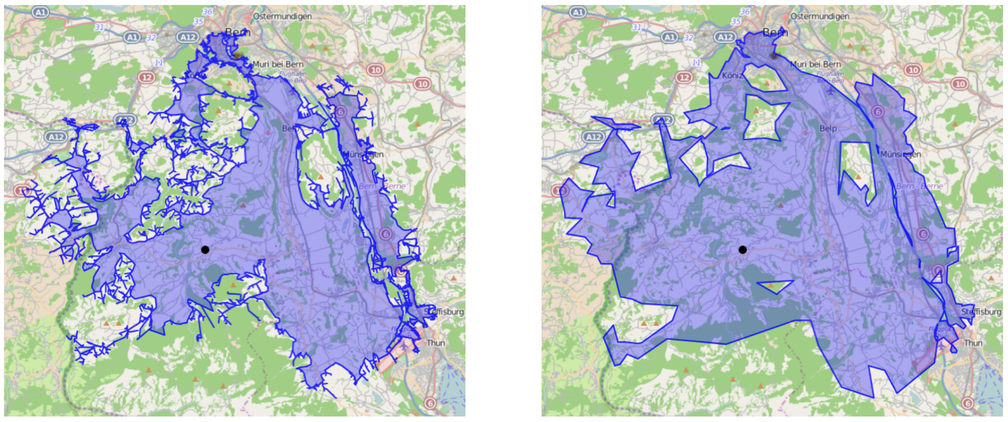

(a) (b)

Figure 1: Real-world example of isocontours in a mountainous area (near Bern, Switzerland). Both figures visualize the range of an electric vehicle positioned at the black disk with a state of charge of 2 kWh. Note that the polygons representing the isocontour contain holes, due to unreachable high-ground areas. (a) An isocontour with over 10 000 segments computed by the approach in Section 5.1, which resembles previous approaches [42]. (b) The result of one of our new approaches, presented in Section 5.3, using only 416 segments.

practice, *speedup techniques* [3] separate the workflow into an offline preprocessing phase and an online query phase. Recently, some speedup techniques were extended to the isochrone scenario [6, 20], enabling query times in the order of milliseconds. Yet, these approaches only deal with the computation of the reachable subgraph, rather than its visualization.

Regarding isocontour visualization, efficient approaches exist for shape characterization of point sets, such as α-shapes [17], χ-shapes [16], or Voronoi Filtering [1]. However, we are interested in separating subgraphs rather than point sets. A work by O'Sullivan et al. [48] introduces basic approaches for isocontour visualization based on merging shapes covering the reachable area. Alternatively, one can subdivide the plane into cells and highlight those that cover the reachable area [39]. Marciuska and Gamper [42] present two approaches to visualize isochrones in transportation networks. The first transforms a given reachable network into an isochrone area by simply drawing a buffer around all edges in range. The second one creates a polygon without holes, induced by the edges on the boundary of the embedded reachable subgraph. Resulting isocontours are similar to the one shown in Figure 1a, though omitting holes. Both approaches were implemented on top of databases, providing running times that are too slow for many applications (several seconds for small and medium ranges). Gandhi et al. [26] introduce an algorithm to approximate isocontours in sensor networks with provable error bounds. They preserve the topological structure of the given family of contours, forbidding intersections. Finally, there are different works presenting applications which make use of isocontours in the context of urban planning [4, 35, 42, 48], geomarketing with integrated traffic information [18, 19], and range visualization for electric vehicles [11, 29, 46, 47]. For isocontour visualization, recent works typically resort to the approach of Marciuska and Gamper [42] or less accurate solutions based on, e. g., the convex hull of all reachable points.

Contribution and Outline. We propose algorithms for efficiently computing polygons representing isocontours in road networks. All approaches compute isocontours that are exact, i. e., they contain exactly the subgraph reachable within the given resource limit, while having low descriptive complexity. Efficient performance of our techniques is both proven in theory and demonstrated in practice on realistic instances.

In Section 2, we formalize the notion of reachable and unreachable subgraphs. Moreover, we state the precise problem and outline our algorithmic approach to solve it. Section 3 attacks the first resulting subproblem of computing *border regions*, that is, polygons that represent the boundaries of the reachable and unreachable subgraph. An isocontour must separate these boundaries. In Section 4, we consider the special case of separating two hole-free polygons with a polygon with minimum number of segments. While this problem can be solved in $O(n \log n)$ time [52], we propose a simpler algorithm that uses at most two additional segments, runs in linear time, and requires a single run of a minimum-link path algorithm. We also propose a minimum-link path algorithm that is simpler than previous approaches [50]. Section 5 extends these results to the general case, where border regions may consist of more than two components. Since the complexity of the resulting problem is unknown, we focus on efficient heuristic approaches that work well in practice, but do not give guarantees on the complexity of the resulting range polygons. Section 6 contains our extensive experimental evaluation using a large, realistic input instance. It demonstrates that all approaches are fast enough even for use in interactive applications. We close with final remarks in Section 7.

2 Problem Statement and General Approach

Let $G = (V, E)$ be a graph representing a road network, which we consider as a geometric network where vertices have a fixed position in the plane and edges are represented by straight-line segments between their endpoints. We consider G to be *directed*, i. e., edges $E \subseteq V \times V$ have an orientation to capture constraints of real-world road networks (such as one way streets). Also, note that weights of two opposing edges (u, v) and (v, u) may differ (e. g., going uphill is more expensive than going downhill). For the sake of simplicity, examples in this section use undirected graphs with uniform edge weights (as in a symmetric directed graph). Finally, we assume that G is strongly connected, i. e., for every pair of vertices $s \in V$ and $t \in V$, there exists an s–t path in G.

A source $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$ together partition the network into two parts, one that is within range r from s, and the part that is not. An isocontour separates these two parts. We are interested in visualizing such isocontours efficiently. In the following, we give a precise definition of the (un)reachable parts of the network and formally define *range polygons*, which we use to represent isocontours. We also describe the problem setting, which, given some source vertex and a range, asks for a range polygon that fulfills certain optimization criteria.

Range Polygons. A path π in G starting at s is *passable* if the consumption of π, i. e., the sum of its edge consumption values, is at most r. A vertex $v \in V$ is *reachable* (with respect

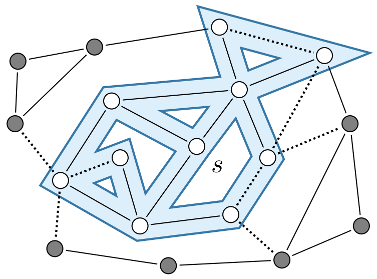


Figure 2: Graph with reachable (white) and unreachable (dark gray) vertices for a source s and a range of 2, assuming uniform weights of 1 (for both length and consumption). Edges drawn solid are passable (unreachable) if both endpoints are reachable (unreachable). Dashed edges are boundary edges if they have an unreachable endpoint and accessible otherwise. Note that the blue polygon (with shaded interior) is in fact a range polygon.

to the maximum range r) if the shortest s–v path is passable. A vertex that is not reachable is *unreachable*. For edges the situation is more complicated. We partition the edges into four types, namely unreachable edges, boundary edges, accessible edges, and passable edges. Figure 2 shows an example of the different edge types. If both endpoints of an edge (u, v) are unreachable, then also the edge (u, v) is *unreachable*. If exactly one endpoint is reachable, then (u, v) is not part of the reachable network, and we call it *boundary edge*. However, the fact that both u and v are reachable does not necessarily imply that (u, v) is part of the reachable network. Let π_u and π_v denote the shortest paths from s to u and v, respectively. If the resource consumptions of π_u and π_v do not allow the traversal of the edge (u, v) in either direction, i. e., $\text{cons}(\pi_u) + \text{cons}((u, v)) > r$ and $\text{cons}(\pi_v) + \text{cons}((v, u)) > r$, then we do not consider (u, v) as reachable. Since we can reach both endpoints of (u, v), we call it *accessible*. Otherwise, the edge can be traversed in at least one direction, so it is *passable*.

Let V_r be the set of reachable vertices and let $V_u = V \setminus V_r$ be the set of unreachable vertices. Similarly, let E_u, E_b, E_a, and E_r denote the set of unreachable edges, boundary edges, accessible edges, and passable edges, respectively. Note that for arbitrary pairs of edges (u, v) and (v, u), both edges belong to the same set. The reachable part of the network is $G_r = (V_r, E_r)$, and the unreachable part is $G_u = (V_u, E_u)$. A *range polygon* is a plane (not necessarily simple) polygon P separating G_r and G_u in the sense that its interior contains G_r and has empty intersection with G_u. Every range polygon P intersects each boundary edge an odd number of times and each accessible edge an even number of times. In particular, an accessible edge may be totally or partially contained in the interior of a range polygon.

If the input graph G is planar, one can construct a range polygon by first slightly *growing* the outer face of the subgraph induced by all reachable vertices. Then, *shrinking* the inner faces and making each shrunk face a hole results in a valid range polygon, though it may contain many holes; see the range polygon in Figure 2. However, if G is not planar, a range polygon may not even exist. If a passable edge intersects an unreachable edge, the requirements of including the passable and excluding the unreachable edge obviously contradict. To resolve this issue, we consider the planarization G_p of G, which is obtained from G by considering each intersection point p as a *dummy vertex* that subdivides all edges of G containing p. We transfer the above partition of G into reachable and unreachable

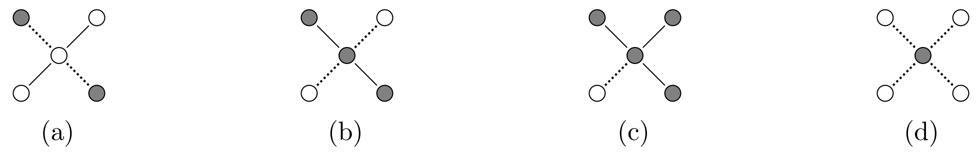


Figure 3: Different cases of crossing edges. (a) Intersection of a passable and an unreachable edge. The dummy vertex (center) is reachable. The unreachable edge is split into two boundary edges. (b) Intersection of an accessible and an unreachable edge. The dummy vertex is unreachable, dashed edges indicate new boundary edges. (c) Intersection of an unreachable edge and a boundary edge, creating unreachable edges and a new boundary edge. (d) An intersection of accessible edges creates an unreachable vertex and four boundary edges.

parts to G_p as follows. A dummy vertex is *reachable* if and only if it subdivides at least one passable edge of the original graph. As above, an edge of G_p is unreachable if both endpoints are unreachable, and it is a boundary edge if exactly one endpoint is reachable. If both endpoints are reachable, it is accessible (passable) if and only if the edge in G containing it is accessible (passable). Clearly, after the planarization, a range polygon always exists. Figure 3 shows different cases of crossing edges. Note that this way of handling crossings ensures that a range polygon for G_p contains the reachable vertices of G and excludes the unreachable vertices of G. However, unreachable edges of G may be partially contained in the range polygon if they cross passable edges. Finally, to avoid special cases, we add a bounding box of dummy vertices and edges to G_p, connecting each vertex in the bounding box to its respective closest vertex in G_p with an edge of infinite length. Thereby, we ensure that neither the reachable nor the unreachable subgraph is empty, as the reachable (unreachable) subgraph contains at least the source (bounding box).

Problem Setting. Given the input graph $G = (V, E)$ with edge lengths $\text{len}\colon E \to \mathbb{R}_{\geq 0}$ and consumption values $\text{cons}\colon E \to \mathbb{R}_{\geq 0}$, a source vertex $s \in V$, and a range $r \in \mathbb{R}_{\geq 0}$, we seek to compute a range polygon with respect to the planarized graph G_p that has the minimum number of holes, and among these we seek to minimize the complexity of the range polygon, i. e., its number of segments. Note that using G_p instead of G may increase the number of holes (see also the case depicted in Figure 3d), but guarantees the existence of a solution.

Practical approaches for fast retrieval of spatial data in road networks (e. g., shortest paths between given pairs of points) often follow the paradigm of *speedup techniques* [3]. Assuming that the input graph is static, such techniques use a relatively costly *offline* preprocessing stage to compute auxiliary data (typically, the resulting space overhead is required to be at most linear in the graph size). During the *online* phase, queries asking for the actual information are answered quickly by making use of the precomputed data. In our setting, such a query consists of a source $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$. Hence, we are allowed to preprocess data that depends solely on the graph, but not on the source or the range.

General Approach. Consider the graph G' consisting of the union of the reachable graph G_r and the unreachable graph G_u. Clearly, all segments of the range polygon lie in faces of G'. A face of G' that is incident to both reachable and unreachable components is called

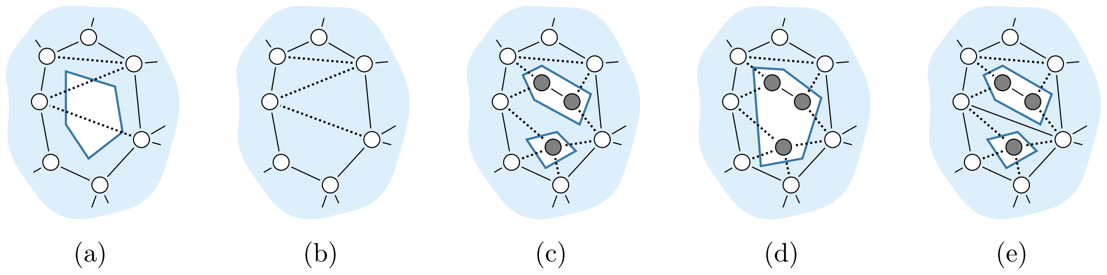


Figure 4: Removing unnecessary holes of a range polygon. (a) A hole that contains no unreachable vertices can always be removed. (b) The resulting interior (shaded area) of the range polygon. (c) Two holes that can be merged into one as they lie in the same border region. (d) The resulting range polygon. (e) These two holes cannot be merged, as they are separated by a passable edge.

a *border region*. Since a range polygon separates the reachable and unreachable parts, each border region contains at least one connected component of a range polygon. Therefore, the number of border regions is a lower bound on the number of holes. On the other hand, components in faces that are not border regions can be removed and multiple connected components in the same border region can always be merged, potentially at the cost of increasing the complexity; see Figure 4. Therefore, a range polygon with the minimum number of holes (with respect to G_p) can be computed as follows.

1. Compute the reachable and unreachable parts of G.
2. Planarize G, compute the reachable and unreachable parts of the planarization G_p.
3. Compute the border regions.
4. For each border region B, compute a simple polygon of minimum complexity that is contained in B and separates the unreachable components incident to B from the reachable component.

In the following sections, we discuss several alternative implementations for these steps. The first three steps are described together in Section 3. The main part of this work is concerned with Step 4. Each connected component of the boundary of a border region is a hole-free non-crossing polygon. Note that these polygons are not necessarily simple in the sense that they may contain the same segment twice in different directions; see Figure 5. Thus, each border region is defined by two sets R and U of hole-free non-crossing polygons, where R contains the boundaries of the reachable components and U contains the boundaries of the unreachable components. For our range polygon, we seek a simple polygon with the minimum number of links that separates U from R. This problem has been previously studied. Guibas et al. [32] showed that the problem is NP-complete in general. In our case, however, $|R| = 1$ holds since the reachable part of the network is, by definition, connected. Guibas et al. left this case as an open problem and, to the best of our knowledge, it has not been resolved.

In Section 4, we first consider border regions that are incident to only one unreachable component, i. e., $|R| = |U| = 1$. In this case, a polygon with the minimum number of segments

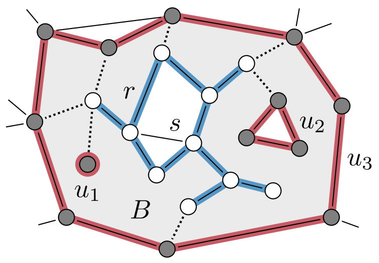


Figure 5: A border region B (shaded), with a reachable boundary $R = \{r\}$ (blue) and an unreachable boundary $U = \{u_1, u_2, u_3\}$ with three components (red). The reachable boundary corresponds to the part that is reachable from the indicated source s with a range of 2, assuming uniform edge costs of 1 (with respect to both length and consumption).

that separates R and U can be found in $O(n \log n)$ time (where n is the total number of segments in the border region) using the algorithm of Wang [52]. This algorithm is rather involved, but it contains a linear-time subroutine that computes an $\mathrm{OPT} + 1$ approximation and is based on the computation of two minimum-link paths. Instead, we propose a simpler algorithm that uses at most two more segments than the optimum, runs in linear time, and relies on a *single* run of a minimum-link path algorithm. Clearly, spending an additional segment to save about a factor of two in running time is a favorable tradeoff in practice. Furthermore, we give a new linear-time minimum-link path algorithm that is simpler than previous algorithms for this problem [50]. In Section 5, we consider the general case of our problem, where a border region may be incident to more than one unreachable component. We propose several algorithms for this problem. As mentioned above, the complexity of this problem is unknown, so we focus on efficient heuristic approaches that work well in practice but do not give provable guarantees on the number of segments.

3 Computing the Border Regions

We describe the computation of the reachable and unreachable part of the input graph $G = (V, E)$, the planarization of these parts, and extraction of all border regions. We modify Dijkstra's algorithm [14] to compute the reachable and unreachable parts in time $O(|V| \log |V|)$, presuming that $|E| \in O(|V|)$ for graphs representing road networks. Afterwards, we map this information to the planarized graph G_p in $O(|E|)$ time. Finally, we extract the border regions by traversing all faces of the planar graph that contain at least one boundary edge, which requires linear time in the size of all border regions. We also discuss efficient implementations of these three steps.

Given a source vertex $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$, we first use a variant of Dijkstra's algorithm to compute

- for each vertex $v \in V$, whether v is reachable from s;
- the set $E_x := E_b \cup E_a$ of all boundary edges and accessible edges;
- for every edge $(u, v) \in E \setminus E_x$, whether (u, v) is passable or unreachable.

For the sake of simplicity, we assume that shortest paths are unique (with respect to the length function). Thereby, we avoid the special case of two paths with equal distance but different consumption, which requires additional tie breaking. Then, we can use Dijkstra's algorithm to compute a shortest-path tree from s (based on edge lengths). Afterwards, we can traverse the tree starting at s to compute consumption values $c(v)$ for all $v \in V$ along shortest paths originating at s in linear time. We then know that a vertex $v \in V$ is reachable if and only if $c(v) \leq r$. The set E_x can be computed in a final linear scan over all edges $(u, v) \in E$, by checking for each the consumption $c(u)$ at its tail vertex and the consumption value $\mathrm{cons}(u, v)$ of the edge itself. Finally, an edge $(u, v) \in E \setminus E_x$ is passable if u (and thus, also v) is reachable, and it is unreachable otherwise.

Although running time of the procedure outlined above is almost linear in the graph size, it is impractical on large inputs [3]. In a recent work [6], faster techniques are presented that extend Dijkstra's algorithm to answer queries within milliseconds on large input graphs after some preprocessing. While these techniques consider a more restricted scenario (only boundary edges are computed, for the special case of $\mathrm{len} \equiv \mathrm{cons}$), they can easily be adapted to our scenario. Therefore, we focus on the remaining steps of our general approach.

Planarization. We planarize G during the preprocessing stage to obtain $G_p = (V_p, E_p)$. Since edge weights and directions are irrelevant for all subsequent steps of our algorithms, we consider G_p to be unweighted and undirected (this may also reduce space consumption of G_p in practice). Our implementation uses the well-known sweep line algorithm [8, 12] to compute G_p. It runs in $O((|V| + k) \log |V|)$ time, where k is the number of edge crossings. More involved algorithms with better asymptotic running times exist [2, 21]. However, the value of k is usually small for road networks and somewhat higher preprocessing effort is not an issue in our scenario. For each vertex in G_p, we store its original vertex in G (if it exists). In practice, where vertices are represented by indices $\{1, \ldots, |V|\}$, this mapping can be done implicitly, since V_p is a superset of V.

After computing the reachable subgraph of the original graph G as described above, we compute reachability of dummy vertices and the set E_x of boundary edges and accessible edges in G_p as follows. We add a flag to all edges in G during preprocessing to indicate whether an edge in G is also contained in G_p. Our search algorithm adds edges to E_x only if this flag is set. After the search terminates, we check for each dummy vertex $v \in V_p$ whether it is reachable, by checking passability of all original edges in E that contain v. To this end, we precompute an array of all original edges that were split during planarization, and for each split edge a list of dummy vertices it contains (an original edge may intersect several other edges). During a query, we scan this array of edges, and for each passable edge, we mark all its dummy vertices as reachable. Finally, we scan all edges in G_p that have at least one dummy vertex as endpoint, to determine any missing edges in E_x. To test whether some edge is accessible, we store pointers to the original edges containing it. Note that the linear scans produce limited overhead in practice, since the number of dummy vertices in graphs representing road networks is typically small. A vertex in V_p is reachable if it is a dummy vertex marked as reachable, or its corresponding original vertex is reachable (if it exists). For edges in $E_p \setminus E_x$, we check reachability of their endpoints to determine whether they are passable or unreachable.

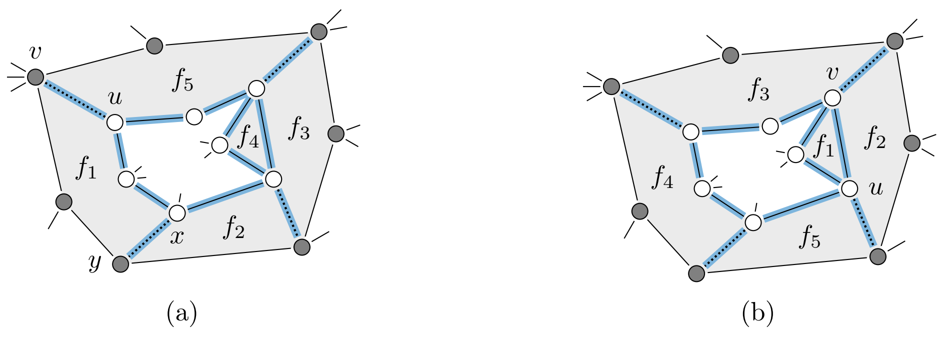


Figure 6: Visited edges (blue) when extracting a reachable boundary. (a) Starting at the boundary edge $\{u, v\}$, the face f_1 is traversed first until the boundary edge $\{x, y\}$ is encountered. Afterwards, the faces f_2, f_3, f_4, f_3, f_5 are visited in this order until $\{u, v\}$ is reached again. (b) Starting at the accessible edge $\{u, v\}$, the face f_1 is traversed until $\{u, v\}$ is reached again (on the same side). Then, the faces f_2, f_3, f_4, f_5, f_2 are processed before the other side of $\{u, v\}$ is reached.

An alternative approach modifies the search algorithm to work directly on a (weighted, directed) planar graph G_p to avoid the additional linear sweeps. However, this produces overhead during the search (e. g., case distinctions for dummy vertices). Consequently, such approaches did not provide significant speedup in preliminary experiments. Moreover, determining the reachable subgraph of G_p in a separate step simplifies the integration of speedup techniques for the more expensive search algorithm [6].

Extracting the Border Regions. Given the set E_x of boundary edges and accessible edges in G_p, we describe how to compute the actual border regions (i. e., the polygons describing R and U). The basic idea is to traverse all faces of G_p that contain edges in E_x, to collect the segments that form boundaries of the border regions. Clearly, all passable edges in these faces are part of some reachable boundary, while all unreachable edges belong to an unreachable boundary. Moreover, since G_p is (strongly) connected, all faces contained in a border region must contain an edge in E_x. Thus, traversing these faces is sufficient to obtain all border regions.

In somewhat more detail, we maintain two flags for every edge $\{u, v\}$ in E_x indicating whether u or v has been visited, respectively, each initially set to false. (We denote the edge $\{u, v\}$ because G_p is *undirected*.) Let $\{u, v\}$ be the first edge of E_x that is considered, and without loss of generality, let u be reachable. We compute the (unique) reachable component $R_{\{u,v\}}$ of the border region $B_{\{u,v\}}$ containing $\{u, v\}$; see Figure 6. We mark u as visited and traverse the face left of $\{u, v\}$, following the unique neighbor w of u in this face that is not v. Every edge that we traverse is added to $R_{\{u,v\}}$. As soon as we encounter an edge $\{x, y\} \in E_x$, we continue by traversing the *twin face* of $\{x, y\}$, i. e., the unique face of G_p that contains the other side of $\{x, y\}$. The edge $\{x, y\}$ itself is not added to $R_{\{u,v\}}$, but we mark the reachable endpoint x that was added to $R_{\{u,v\}}$ in the previous step as visited. The current extraction step is finished as soon as the other side of $\{u, v\}$ is reached; see Figure 6. If v is unreachable, $\{u, v\}$ is a boundary edge. Thus, we continue with the extraction of the unreachable component $U_{\{u,v\}}$ containing v in the same manner and assign it to $B_{\{u,v\}}$.

We loop over the remaining edges in E_x and extract boundaries corresponding to vertices not visited before. By extracting reachable components first, we ensure that the corresponding reachable boundary of some unreachable component is always known before extraction, namely, the boundary containing the reachable endpoint of the considered edge in E_x. Therefore, the unreachable component is assigned to the unique border region that contains this reachable boundary. To make sure that we only compute actual border regions (in contrast to regions containing only accessible edges but not boundary edges), we can either check for boundary edges explicitly during traversal of the faces or disallow the traversal to start from an accessible edge.

Implementation Details. To extract the components of all border regions, we have to traverse faces of the planar input graph. This can be done using established data structures, such as doubly connected edge lists [12]. Instead, we propose a more cache-friendly data structure to represent faces, which stores adjacent vertices of a face in contiguous memory. We use a single array that holds all faces of the graph. For each face, we store the sequence of vertices as they are found traversing the face in clockwise order starting at an arbitrary vertex. At the beginning and at the end of this sequence, we store sentinels that hold the index of the last and first entry of a vertex of the corresponding face, respectively. Traversing the face in either direction requires only a single scan along the array, jumping at most once to the beginning or end of the face. For consecutive vertices $u \in V_p$ and $v \in V_p$ in this array, we store at v its index in the corresponding twin face of the edge $\{u, v\}$. Finally, we store for every edge $\{u, v\}$ in the graph the two indices of the head vertex v in this data structure, i. e., its occurrence in the faces left and right of this edge.

To efficiently decide whether an edge is in E_x, or if an edge in E_x was marked as visited, we store this list as an array and sort it (e. g., by the index of the head vertex) before extracting the reachable components. Then we can quickly retrieve an edge in this array using binary search (we also tried using hash sets as an alternative approach, but this turned out to be slightly slower in preliminary experiments).

4 Range Polygons in Border Regions Without Holes

Given a border region B with reachable component R and a single unreachable component U, we present an algorithm for computing a polygon that separates R and U. In Section 5, we describe how to generalize our approach to the case $|U| > 1$.

The basic idea is to add an arbitrary boundary edge e to B, thereby connecting both components R and U. Since we presume that G is strongly connected, such a boundary edge always exists. In the resulting hole-free non-crossing polygon B', we compute a path with minimum number of segments that connects both sides of e. The algorithm of Suri [50] computes such a *minimum-link path* π' in linear time. We obtain a separating polygon S' by connecting the endpoints of π' along e. It is easy to see that this yields a polygon with at most two additional segments compared to an optimal solution.

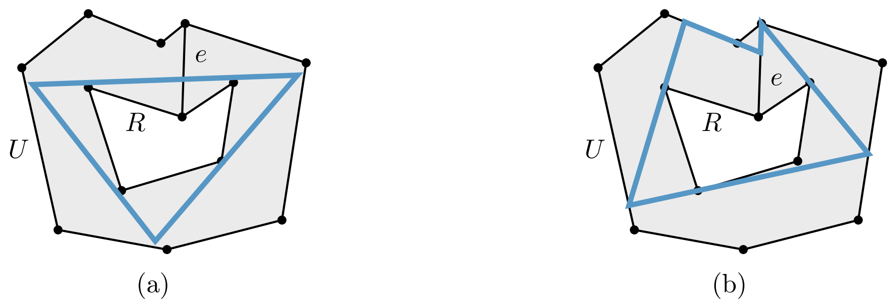


Figure 7: Separating polygons in a border region (shaded). (a) The polygon S (blue) separating R and U with $\mathrm{OPT} = 3$ links induces a path with four segments connecting both sides of e. (b) The polygon S' obtained by a minimum-link path from the depicted edge e has $\mathrm{OPT} + 2 = 5$ segments.

Lemma 1. *Let S be a polygon that separates R and U with minimum number of segments, and let* OPT *denote this number. Then S' has at most* $\mathrm{OPT} + 2$ *segments.*

Proof. We can split S at e into a path π connecting both sides of e. Clearly, π has at most $\mathrm{OPT} + 1$ links (if a segment of S crosses e, we split it into two segments with endpoints in e; see Figure 7). Since π' is a minimum-link path, we have $|\pi'| \leq |\pi| = \mathrm{OPT} + 1$. Moreover, S' is obtained by adding a single subsegment of e to π', so its complexity is bounded by $\mathrm{OPT} + 2$. □

We now address the subproblem of computing a minimum-link path between two edges of a simple polygon. The linear-time algorithm of Suri [50] starts by triangulating the input polygon. To save running time for queries in our scenario, we triangulate all faces of the planar graph G_p during preprocessing. Afterwards, in each step of Suri's algorithm, a *window* (which we formally define in a moment) is computed. To obtain the windows in linear time, it relies on several calls to a subroutine computing visibility polygons. While this is sufficient to prove linear running time, it seems wasteful from a practical point of view. In the following, we present an alternative algorithm for computing the windows that also results in linear running time, but is much simpler. It can be seen as a generalization of an algorithm by Imai and Iri [34] for approximating piecewise linear functions.

Windows and Visibility. Let P be a simple polygon and let a and b be edges of P. We want to compute a minimum-link polygonal path starting at a and ending in b that lies in the interior of P. Let T be the graph obtained by arbitrarily triangulating P, i. e., the graph T consists of the vertices and edges of P together with any edges that connect two vertices according to the triangulation. Let t_a and t_b be the triangles in T incident to a and b, respectively. As T is a binary tree by construction, its (weak) dual graph has a unique path $t_a = t_1, t_2, \ldots, t_{k-1}, t_k = t_b$ from t_a to t_b; see Figure 8a. We call the triangles on this path *important* and their positions in the path their *indices.*

The *visibility polygon* $V(a)$ of the edge a in P is the polygon that contains all points that are visible from a. More formally, $V(a)$ contains a point q in its interior if and only

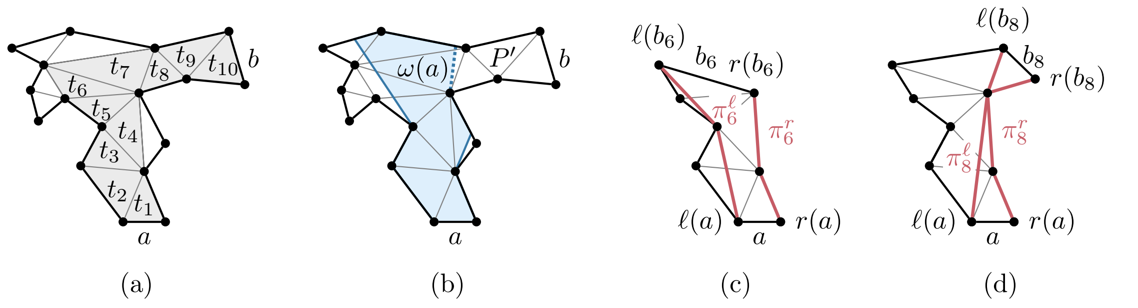


Figure 8: Important triangles, visibility, and shortest paths. (a) The (shaded) important triangles of a polygon with respect to indicated edges a and b. (b) The window $\omega(a)$ is the dotted edge of the (shaded) visibility polygon. (c) The left and right shortest path (red) do not intersect for $i = 6$. (d) The shortest paths (red) have an intersection for $i = 8$.

if there is a point p on a such that the line segment pq lies entirely inside P. Let i be the highest index such that the intersection of the triangle t_i with the visibility polygon $V(a)$ is not empty. The *window* $w(a)$ is the edge of $V(a)$ that intersects t_i closest (with respect to minimum Euclidean distance) to the edge between t_i and t_{i+1}; see Figure 8b. Note that $w(a)$ separates the polygon P into two parts. Let P' be the part containing the edge b that we want to reach. A minimum-link path from a to b in P can then be obtained by adding an edge from a to $w(a)$ to a minimum-link path from $w(a)$ to b in P'. Thus, the next window is computed in P' starting with the previous window $w(a)$. In the following, we first describe how to compute the first window and then discuss what has to be changed to compute the subsequent windows.

Let T_i be the subgraph of T induced by the triangles $t_1, \dots, t_i$ and let P_i be the polygon bounding the outer face of T_i. The polygon P_i has two special edges, namely a and the edge shared by t_i and t_{i+1}, which we call b_i. Let $\ell(a)$ and $r(a)$, and $\ell(b_i)$ and $r(b_i)$ be the endpoints of a and b_i, respectively, such that their clockwise order is $r(a)$, $\ell(a)$, $\ell(b_i)$, $r(b_i)$ (think of $\ell(\cdot)$ and $r(\cdot)$ being the left and right endpoints, respectively); see Figure 8c. We define the *left shortest path* π_i^ℓ to be the shortest polygonal path (shortest in terms of Euclidean length) that connects $\ell(a)$ with $\ell(b_i)$ and lies inside or on the boundary of P_i. The *right shortest path* π_i^r is defined analogously for $r(a)$ and $r(b_i)$; see Figure 8c.

Assume that the edge b_i is visible from a, i. e., there exists a line segment in the interior of P_i that starts at a and ends at b_i. Such a visibility line separates the polygon into a left and a right part. Observe that it follows from the triangle inequality that the left shortest path π_i^ℓ and the right shortest path π_i^r lie inside the left and right part, respectively. Thus, these two paths do not intersect. Moreover, the two shortest paths are *outward convex* in the sense that the left shortest path π_i^ℓ has only left bends when traversing it from $\ell(a)$ to $\ell(b_i)$ (the symmetric property holds for π_i^r); see the case $i = 6$ in Figure 8c. We note that the outward convex paths are sometimes also called "inward convex" and the polygon consisting of the two outward convex paths together with the edges a and b_i is also called *hourglass* [30]. The following lemma, which is similar to a statement shown by Guibas et al. [31, Lemma 3.1], summarizes the above observation. On the other hand, shortest paths that intersect are not outward convex in general; see Figure 8d.

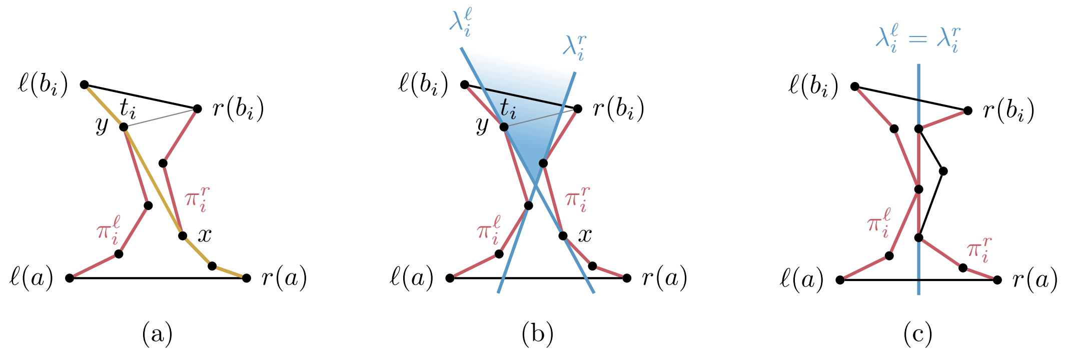


Figure 9: The visibility cone of an hourglass. (a) The shortest path (yellow) from $r(a)$ to $\ell(b_i)$ consists of a prefix of π_i^r, the segment xy, and a suffix of π_i^ℓ. (b) The two visibility lines λ_i^ℓ and λ_i^r (blue) spanning the (shaded) visibility cone. (c) A degenerated case, in which the visibility cone collapses to a line (as points are *not* in general position).

Lemma 2. *If the triangle t_i is visible from a (i. e., there exists a line segment in the interior of P_i that starts at a and ends at a point in t_i), then the left and right shortest path in P_{i-1} have empty intersection. Moreover, if these paths do not intersect, they are outward convex.*

Guibas et al. [31] argue that the converse of the first statement is also true, i. e., if the two paths have empty intersection, then the triangle t_{i+1} is visible from a. Their main arguments go as follows. The shortest path (with respect to Euclidean length) in the hourglass that connects $r(a)$ with $\ell(b_i)$ is the concatenation of a prefix of π_i^r, a line segment from a vertex x of π_i^r to a vertex y of π_i^ℓ, and a suffix of π_i^ℓ; see Figure 9a. We call the straight line through x and y the *left visibility line* and denote it by λ_i^ℓ. We assume λ_i^ℓ to be oriented from x to y and call x and y the *source* and *target* of λ_i^ℓ. Analogously, one can define the *right visibility line* λ_i^r; see Figure 9b. We call the intersection of the half-plane to the right of λ_i^ℓ with the half-plane to the left of λ_i^r the *visibility cone*. It follows that the intersection of the visibility cone with the edge b_i is not empty and a point on the edge b_i is visible from a if and only if it lies in this intersection [31]. This directly extends to the following lemma.

Lemma 3. *If the left and right shortest path in P_{i-1} have empty intersection, t_i is visible from a. Moreover, a point in t_i is visible from a if and only if it lies in the visibility cone.*

Note that when π_i^r and π_i^ℓ are both outward convex and intersect, the visibility cone degenerates to a line; see Figure 9c. For the sake of simplicity, we presume that all points are in general position, which prevents this special case. (In practice, it is handled implicitly by the implementation of Line 4 in Algorithm 1.) The above observations then justify the following approach for computing the window. We iteratively increase i until the left and the right shortest path of the polygon P_i intersect. We then know that the triangle t_{i+1} is no longer visible; see Lemma 2. Moreover, as the left and the right shortest path did not intersect in P_{i-1}, the triangle t_i is visible from a; see Lemma 3. To find the window, it remains to find the edge of the visibility polygon $V(a)$ that intersects t_i closest to the edge between t_i and t_{i+1}. Thus, by the second statement of Lemma 3, the window must be a

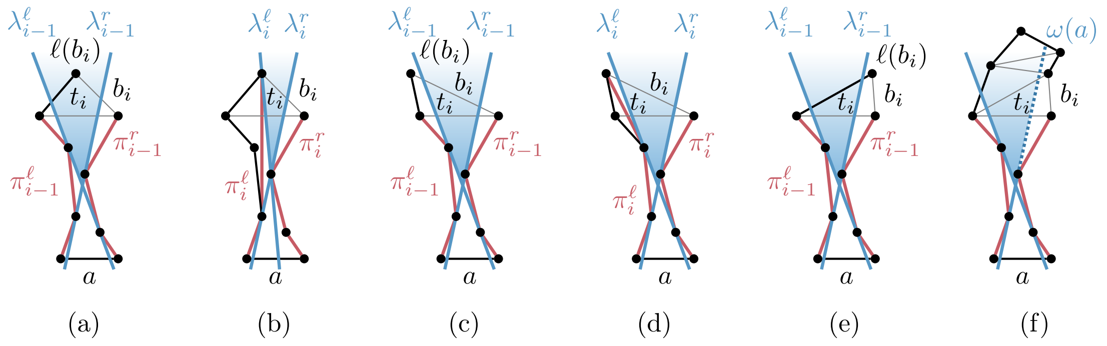


Figure 10: Visiting a new triangle. (a) The new vertex $\ell(b_i)$ lies in the visibility cone. (b) The updated left shortest path π^ℓ_i and left visibility line λ^ℓ_i. (c) The vertex $\ell(b_i)$ lies to the left of λ^ℓ_{i-1}. (d) The left shortest path has to be updated, the left visibility line remains unchanged. (e) The vertex $\ell(b_i)$ lies to the right of λ^r_{i-1}, i. e., t_{i+1} is not visible from a. (f) The dotted window $\omega(a)$ is a segment of λ^r_{i-1}.

segment of one of the two visibility lines. It remains to fill out the details of this algorithm, argue that it runs in overall linear time, and describe what has to be done in later steps, when we start at a window instead of an edge.

Computing the First Window. We start with the details of the algorithm starting from an edge; see also Algorithm 1. Assume the triangle t_i is still visible from a, i. e., π^ℓ_{i-1} and π^r_{i-1} do not intersect. Assume further that we have computed the left and right shortest path π^ℓ_{i-1} and π^r_{i-1} as well as the corresponding visibility lines λ^ℓ_{i-1} and λ^r_{i-1} in a previous step. Assume without loss of generality that the three corners of the triangle t_i are $\ell(b_{i-1})$, $\ell(b_i)$, and $r(b_i) = r(b_{i-1})$. There are three possibilities shown in Figure 10, i. e., the new vertex $\ell(b_i)$ lies either in the visibility cone spanned by λ^ℓ_{i-1} and λ^r_{i-1} (Figure 10a), to the left of the left visibility line λ^ℓ_{i-1} (Figure 10c), or to the right of the right visibility line λ^r_{i-1} (Figure 10e).

By Lemma 3, a point in t_i is visible from a if and only if it lies inside the visibility cone. Thus, the edge b_i between t_i and t_{i+1} is no longer visible if and only if the new vertex $\ell(b_i)$ lies to the right of λ^r_{i-1}; see Figure 10e. In this case, we can stop and the desired window $w(a)$ is the segment of λ^r_{i-1} starting at its touching point with π^r_{i-1} and ending at its first intersection with an edge of P; see Figure 10f and Lines 4–6 of Algorithm 1.

In the other two cases (Figure 10a and Figure 10c), we have to compute the new left and right shortest path π^ℓ_i and π^r_i and the new visibility lines λ^ℓ_i and λ^r_i (Figure 10b and Figure 10d). Note that the old and new right shortest path π^r_{i-1} and π^r_i connect the same endpoints $r(a)$ and $r(b_{i-1}) = r(b_i)$. As the path cannot become shorter by going through the new triangle t_i, we have $\pi^r_i = \pi^r_{i-1}$. The same argument shows that $\lambda^r_i = \lambda^r_{i-1}$ (recall that the visibility lines were defined using a shortest path from $\ell(a)$ to $r(b_{i-1}) = r(b_i)$).

We compute the new left shortest path π^ℓ_i as follows; see Lines 7–9 of Algorithm 1. Let x be the latest vertex on π^ℓ_{i-1} such that the prefix of π^ℓ_{i-1} ending at x concatenated with the segment from x to $\ell(b_i)$ is outward convex. We claim that π^ℓ_i is the path obtained by this concatenation, i. e., this path lies inside P_i and there is no shorter path lying inside P_i.

Algorithm 1: Computes the first window $w(a)$.

```
   // initial paths (one-vertex sequences) and visibility lines
1  $\pi^\ell = [\ell(a)]$;  $\pi^r = [r(a)]$;  $\lambda^\ell = \text{line}(r(a), \ell(a))$;  $\lambda^r = \text{line}(\ell(a), r(a))$
2  for $i = 1$ to $k$ do
3      if $r(b_i) = r(b_{i-1})$ then
           // $b_i$ not visible $\Rightarrow$ return window
4          if $\ell(b_i)$ lies to the right of $\lambda^r$ then
5              $x$ = first intersection of $\lambda^r$ with $P$ after target($\lambda^r$)
6              return segment(target($\lambda^r$), $x$)
           // extend left path $\pi^\ell$ like in Graham's scan
7          append $\ell(b_i)$ to $\pi^\ell$
8          while last bend of $\pi^\ell$ is a right bend do
9              remove second to last element from $\pi^\ell$
           // $\ell(b_i)$ in visibility cone $\Rightarrow$ update left visibility line $\lambda^\ell$
10         if $\ell(b_i)$ lies to the right of $\lambda^\ell$ then
11             target($\lambda^\ell$) = $\ell(b_i)$
12             while $\lambda^\ell$ is not a tangent of $\pi^r$ at source($\lambda^\ell$) do
13                 source($\lambda^\ell$) = successor of source($\lambda^\ell$) in $\pi^r$
14     else
           // case $\ell(b_i) = \ell(b_{i-1})$ is symmetric to above case $r(b_i) = r(b_{i-1})$
```

It follows by the outward convexity, that there cannot be a shorter path inside P_i from $\ell(a)$ to $\ell(b_i)$. Moreover, by the assumption that π^ℓ_{i-1} was the correct left shortest path in P_{i-1}, the subpath from $\ell(a)$ to x lies inside P_i. Assume for contradiction that the new segment from x to $\ell(b_i)$ does not lie entirely inside P_i. Then it has to intersect the right shortest path and it follows that the right shortest path and the correct left shortest path have non-empty intersection, which is not true by Lemma 2.

To get the new left visibility line λ^ℓ_i, we have to consider the shortest path in P_i that connects $r(a)$ with $\ell(b_i)$. Let x and y be the source and target of λ^ℓ_{i-1}, respectively, i. e., the shortest path from $r(a)$ to $\ell(b_{i-1})$ is as shown in Figure 11a. If the new vertex $\ell(b_i)$ lies to the left of λ^ℓ_{i-1} (Figure 11b), then the shortest path from $r(a)$ to $\ell(b_i)$ also includes the segment from x to y. Thus, $\lambda^\ell_i = \lambda^\ell_{i-1}$ holds in this case. Assume the new vertex $\ell(b_i)$ lies to the right of λ^ℓ_{i-1} (Figure 11c). Let x' be the latest vertex on the path π^r_i such that the concatenation of the subpath from $r(a)$ to x' with the segment from x' to the new vertex $\ell(b_i)$ is outward convex in the sense that it has only right bends; see Figure 11c. We claim that this path lies inside P_i and that there is no shorter path inside P_i. Moreover, we claim that x' is either a successor of x in π^r_{i-1} or $x' = x$. Clearly, the concatenation of the path from $r(a)$ to x with the segment from x to $\ell(b_i)$ is outward convex, thus the latter claim follows. It follows that the segment from x' to $\ell(b_i)$ lies to the right of the old visibility line λ^ℓ_{i-1}. Thus, it cannot intersect the path π^ℓ_i (except in its endpoint $\ell(b_i)$), as π^ℓ_{i-1} lies to the left of λ^ℓ_{i-1}. Moreover, as we chose x' to be the last vertex on π^r_{i-1} with the above

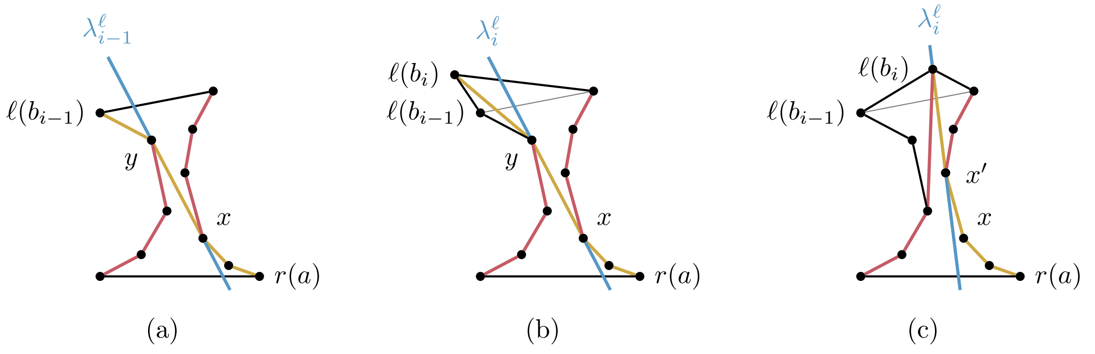


Figure 11: Updating the visibility line. (a) The shortest path from $r(a)$ to $\ell(b_{i-1})$ (yellow) defining the left visibility line λ^ℓ_{i-1}. (b) The visibility line does not change if $\ell(b_i)$ lies to the left of λ^ℓ_{i-1}. (c) Illustration of how the visibility line changes if $\ell(b_i)$ lies to the right of λ^ℓ_{i-1}.

property, this new segment does not intersect π^r_i (except in x'). Hence, the segment from x' to $\ell(b_i)$ lies inside P_i. As before, it follows from the convexity that there is no shorter path inside P_i. Thus, λ^ℓ_i is the line through x' and $\ell(b_i)$ (x' is the new source and $\ell(b_i)$ is the new target). It follows that Lines 10–13 correctly compute the new left visibility line.

Lemma 4. *Let t_h be the triangle with the highest index h that is visible from a. Then Algorithm 1 computes the first window $w(a)$ in $O(h)$ time.*

Proof. We already argued that Algorithm 1 correctly computes the first window. To show that it runs in $O(h)$ time, first note that the polygon P_h has linear size in h. Thus, it suffices to argue that the running time is linear in the size of P_h. In each step i, we first check whether the next triangle is still visible by testing whether the new vertex $\ell(b_i)$ (or $r(b_i)$) lies to the right of the visibility line λ^r_{i-1} (or to the left of λ^ℓ_{i-1}). This takes only constant time. When updating the left and right shortest path, we have to iteratively remove the last vertex of the previous path until the resulting path is outward convex. This takes time linear in the number of vertices we remove. However, a vertex removed in this way will never be part of a left or right shortest path again. Thus, the number of these removal operations over all h steps is bounded by the size of P_h. When updating the visibility lines, the only operation potentially consuming more than constant time is finding the new source x'. As x' is a successor of the previous source x (or $x' = x$), we never visit a vertex of P_h twice in this type of operation. Thus, the total running time of finding these successors over all h steps is again linear in the size of P_h. □

Initialization for Subsequent Windows. As mentioned before, the first window $w(a)$ we compute separates P into two smaller polygons. Let P' be the part including the edge b (and not a). In the following, we denote $w(a)$ by a'. To get the next window $w(a')$, we have to apply the above procedure to P' starting with a'. However, this would require to partially retriangulate the polygon P'. More precisely, let t_h be the triangle with the highest index that is visible from a and let b_h be the edge between t_h and t_{h+1}; see Figure 12a. Then b_h separates P' into an initial part P'_0 (the shaded part in Figure 12a) and the rest (having b on

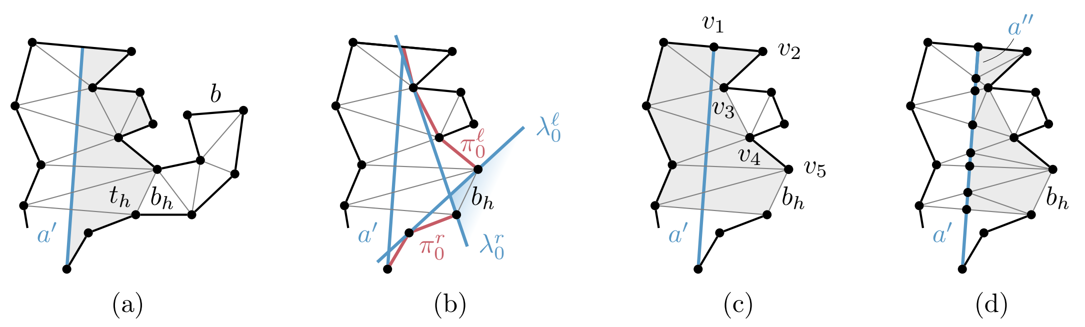


Figure 12: Initializing computation of the second window. (a) The polygon P' we are interested in after computing the first window a'. The initial part P'_0 is shaded. (b) Initial left and right shortest path π_0^ℓ and π_0^r (red) with corresponding visibility lines λ_0^ℓ and λ_0^r (blue). (c) The sequence $v_1, \ldots, v_6$ we use for Graham's scan. Triangles of P intersected by a' are shaded. (d) Computing the shortest path from $\ell(a'') = \ell(a')$ to $\ell(b_h)$ in the subdivided polygon (shaded) using Algorithm 1 actually applies Graham's scan to $v_1, \ldots, v_6$.

its boundary). The latter part is properly triangulated, however, the initial part P'_0 is not. The conceptually simplest solution is to retriangulate P'_0. However, this would require an efficient subroutine for triangulation (and dynamic data structures that allow us to update P and T, which produces overhead in practice). Instead, we propose a much simpler method for computing the next window.

The general idea is to compute the shortest paths in P'_0 from $\ell(a')$ to $\ell(b_h)$ and from $r(a')$ to $r(b_h)$; see Figure 12b. We denote these paths by π_0^ℓ and π_0^r, respectively. Moreover, we want to compute the corresponding visibility lines λ_0^ℓ and λ_0^r. Afterwards, we can continue with the correctly triangulated part as in Algorithm 1.

Concerning the shortest paths, first note that the right shortest path π_0^r is a suffix of the previous right shortest path, which we already know. For the left shortest path π_0^ℓ, consider the polygon induced by the triangles that are intersected by a'; see Figure 12c. Let $v_1, \ldots, v_g$ be the path on the outer face of this polygon (in clockwise direction) from $\ell(a') = v_1$ to $\ell(b_h) = v_g$. We obtain π_0^ℓ using *Graham's scan* [28] on the sequence $v_1, \ldots, v_g$, i. e., starting with an empty path, we iteratively append the next vertex of the sequence $v_1, \ldots, v_g$ while maintaining the path's outward convexity by successively removing the second to last vertex if necessary; see Algorithm 2. We note that applying Graham's scan to arbitrary sequences of vertices may result in self-intersecting paths [10]. However, we will see that this does not happen in our case.

It remains to compute the visibility lines λ_0^ℓ and λ_0^r corresponding to the hourglass consisting of a', b_h, and the shortest paths π_0^ℓ and π_0^r. Note that the whole edge b_h is visible from a', since a' intersects the triangle t_h. Thus, the visibility lines go through the endpoints of b_h. It follows that λ_0^ℓ is the line that goes through $\ell(b_h)$ and the unique vertex on π_0^r such that it is tangent to π_0^r; see Figure 12b. This can be clearly found in linear time in the length of π_0^r. The same holds for the right visibility line.

Lemma 5. *Algorithm 2 computes the initial left and right shortest paths π_0^ℓ and π_0^r as well as the corresponding visibility lines in $O(|P'_0|)$ time.*

Algorithm 2: Computes the initial left and right shortest paths with corresponding visibility lines.

```
1  $a' =$ last window
2  $\pi^r =$ right shortest path computed in the previous step
3  $\pi^r_0 =$ suffix of $\pi^r$ starting with $r(a')$
   // Graham's scan on the sequence $v_1, \dots, v_g$
4  $\pi^\ell_0 =$ empty path
5  for $i = 1$ to $g$ do
6      append $v_i$ to $\pi^\ell_0$
7      while last bend of $\pi^\ell_0$ is a right bend do
8          remove second to last element from $\pi^\ell_0$
9  $\lambda^r_0 =$ tangent of $\pi^\ell_0$ through $r(b_h)$
10 $\lambda^\ell_0 =$ tangent of $\pi^r_0$ through $\ell(b_h)$
11 return $(\pi^r_0, \pi^\ell_0, \lambda^\ell_0, \lambda^r_0)$
```

Proof. We mainly have to prove that the path π^ℓ_0 obtained by applying Graham's scan on the sequence $v_1, \dots, v_g$ actually is the shortest path from $\ell(a)$ to $\ell(b_h)$ in P'_0 (which includes that it is not self-intersecting). This can be seen by reusing arguments we made for computing the first window. To this end, we reuse the triangulation we have for P by placing new vertices where a' crosses triangulation edges; see Figure 12d. Note that the resulting polygon, which we denote by P''_0, is almost triangulated, i. e., each face is a triangle or a quadrangle. We can thus triangulate P''_0 by adding one new edge in each quadrangle as in Figure 12d. Note that a' is separated into several edges in P''_0; let a'' be the topmost of these edges (i. e., the last one in clockwise order). Assume we want to compute the minimum-link path from a'' to b_h in P''_0. First note that the triangle t_h is visible from a''. Thus, our algorithm for computing the first window computes the shortest path from $\ell(a') = \ell(a'')$ to $\ell(b_h)$. Note further that the vertices visited in Lines 7–9 of Algorithm 1 are the vertices $v_1, \dots, v_g$ in this order. Thus, Algorithm 1 actually constructs the left shortest path by using Graham's scan on the sequence $v_1, \dots, v_g$. It follows that directly applying Graham's scan to the sequence $v_1, \dots, v_g$ correctly computes the left shortest path in P'_0 from $\ell(a')$ to $\ell(b_h)$. Clearly, the running time of Algorithm 2 is linear in the size of P'_0. □

We compute subsequent windows as shown before, until the last edge b is found. The actual minimum-link path π is obtained by connecting each window $w(a)$ to its corresponding first edge a with a straight line [50]. Linear running time of the algorithm follows immediately from Lemma 4 and Lemma 5. Theorem 1 summarizes our findings.

Theorem 1. *Given two edges a and b of a simple polygon P, our algorithm computes a minimum-link path from a to b contained in P in linear time.*

Implementation Details. To obtain the desired polygon that separates R and U, we can connect the first and last segment of π along the boundary edge e, as described above. However, to potentially save a segment and for aesthetic reasons, we first test whether

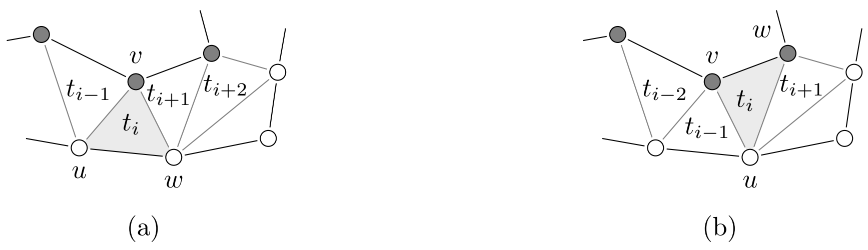


Figure 13: Identifying the next important triangle. (a) The next important triangle t_{i+1} shares with t_i the unique edge vw that is not uv and has exactly one reachable endpoint. (b) The next important triangle t_{i+1} contains the edge uw.

the last window can be extended to the first segment of the path without intersecting the boundary of neither R nor U. This can be done by continuing the computation of the last window from t_0 after b was found.

We do not construct P and its triangulation T explicitly, but work directly on the triangulated input graph. The next important triangle is then computed on-the-fly as follows. Consider an important triangle $t_i = uvw$, and let uv be the edge shared by the current and the previous important triangle; see Figure 13. Clearly, exactly one endpoint of uv is part of the reachable boundary, so without loss of generality let u be this endpoint. Then the next important triangle is the triangle sharing vw with t_i if w is reachable, and the triangle sharing uw with t_i otherwise. In other words, the next triangle is determined by the unique edge that has exactly one reachable endpoint. Triangles are stored in a single array, similar to the data structure to store faces of the planar graph described in Section 3 (though we do not need sentinels, as triangular faces have constant size). The minimum-link path algorithm operates on this data structure to obtain the sequence of important triangles on-the-fly.

5 Heuristic Approaches for General Border Regions

A border region $B = R \cup U$ may consist of several unreachable components, i. e., $|U| > 1$, while $|R| = 1$ always holds. In the general case, it is not clear whether one can compute a (non-intersecting) range polygon of minimum complexity that separates R and U in polynomial time [32]. Even for the simpler subproblem of computing a minimum-link path in a polygon with several components (without assigning them to the reachable or unreachable boundary), the fastest known algorithm has quadratic running time [37, 45]. This is impractical for large instances. In fact, the problem was recently shown to be 3SUM-hard [44], so algorithms with subquadratic running time may not even exist [23]. Therefore, we propose four heuristic approaches with (almost) linear running time (in the size of B) that are simple and fast in practice. Figure 14 shows example outputs of the different heuristics.

Given a border region B, the approach presented in Section 5.1 simply returns the reachable boundary R; see Figure 14a. Results are similar to previous algorithms for isochrones [42]. Since the complexity of the range polygons can become quite high, we propose more sophisticated heuristics. The basic idea of the approach introduced in Section 5.2 is to use the graph triangulation to separate B along edges for which either both endpoints

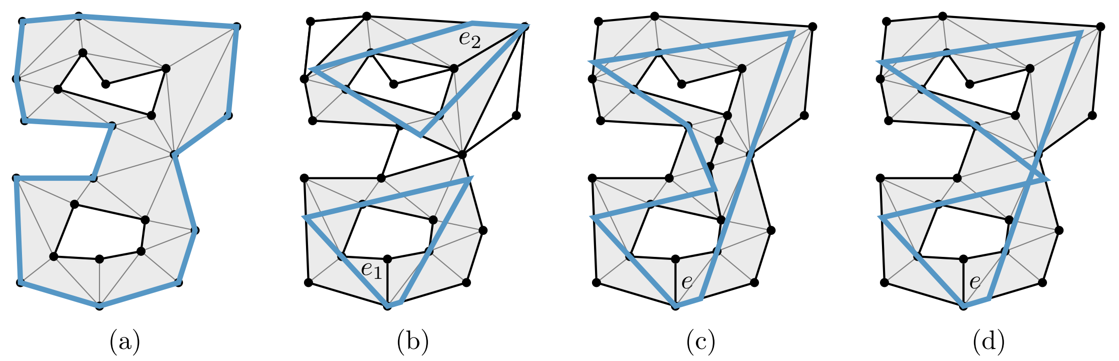


Figure 14: Example output of different heuristic approaches, for a (shaded) border region with two unreachable components. Minimum-link paths are computed from indicated boundary edges. (a) Output of the approach presented in Section 5.1, which returns the reachable boundary. (b) The approach in Section 5.2 separates border regions into smaller subinstances. (c) The approach in Section 5.3 connects unreachable components to obtain a simpler instance. (d) Our last approach in Section 5.4 computes a polygon whose edges may cross each other.

are in R or both endpoints are in U. The modified instances consist of single unreachable components that are separated from the reachable component by the algorithm in Section 4; see Figure 14b. In Section 5.3, we propose to insert new edges that connect the components of U to create an instance with $|U| = 1$. Then, we compute a minimum-link path in the resulting border region as in Section 4; see Figure 14c. Finally, Section 5.4 details a heuristic that modifies the approach of Section 4 to compute a possibly self-intersecting minimum-link path separating R and U with minimum number of segments; see Figure 14d. Consequently, the resulting polygon has at most two more segments than an optimal solution. We rearrange this polygon at crossings to obtain a range polygon without self-intersections.

5.1 Extracting the Reachable Component

Given a border region B, the first approach returns the reachable boundary R. The resulting range polygon closely resembles the results of known approaches, which essentially consist of extracting the reachable subgraph [25, 42]. Note that this approach does not have to compute the unreachable boundary explicitly. Thus, we can improve performance by modifying the extraction of border regions described in Section 3, such that only the reachable part of the boundary is traversed. In a sense, our modified extraction algorithm can be seen as an efficient implementation of previoues approaches [25, 42]. Its linear running time (in the size of B) follows from the fact that we traverse every edge of R once, and every boundary edge or accessible edge of the planar graph G_p contained in B a constant number of times. Clearly, the number of these edges is linear in the size of B.

5.2 Separating Border Regions Along their Triangulation

The idea of this approach is as follows. For each border region B, we consider its triangulation. We add all edges of the triangulation that either connect two reachable vertices or two unreachable vertices of G_p to B, possibly splitting B into multiple regions $B' = R' \cup U'$ (see edges separating the border region in Figure 14b). For each region B', we obtain $|U'| \leq 1$, since two components of U must be connected by an edge of the triangulation or separated by an edge with two endpoints in R. Then, we run the algorithm presented in Section 4 on each instance B' with $|U'| = 1$ to get the range polygon. Linear running time follows, as we run the linear-time algorithm of Section 4 on disjoint subregions of B.

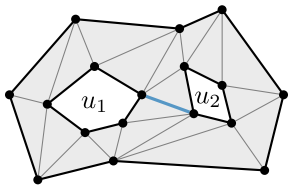
Clearly, the number of edges we add to B is not minimal, i. e., in general we could omit some of them and still obtain $|U'| \leq 1$ for each region B'. On the other hand, computing the set of separating edges described above is trivial, making our approach very simple. In what follows, we describe how it is implemented without explicitly computing the border region B. Instead, we use the set E_x to identify border regions that need to be handled. Recall that E_x contains all boundary edges and accessible edges of the border regions; see Section 3. We loop over all edges in this set and check for each boundary edge $\{u, v\} \in E_x$ whether it was already visited. If this is not the case, we start a minimum-link path computation from the triangle to the left of this edge (accessible edges in E_x are skipped by this loop, since they connect two reachable vertices and are thus considered part of the boundary). The sequence of important triangles is then computed on-the-fly as described in Section 4. Whenever the algorithm passes a boundary edge in E_x, it is marked as visited.

This heuristic can be seen as a simple but effective way of producing border regions with a single unreachable component. It is very easy to implement and even simplifies aspects of the minimum-link path algorithm described in Section 4, because modified border regions contain only important triangles (all other triangles are removed from the modified border regions). Thus, computational effort for finding the second endpoint of a new window and the initial visibility lines is restricted to a single triangle, enabling the use of simpler data structures. On the other hand, the result of the algorithm heavily depends on the triangulation of the input graph. In addition to that, the number of modified regions B' can become quite large; see Section 6. Our next approach therefore proposes a more sophisticated way to obtain regions with a single unreachable component.

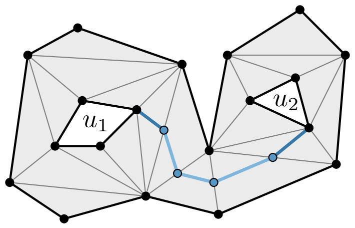
5.3 Connecting Unreachable Components

This approach adds new edges to border regions with more than one unreachable component, such that they connect all unreachable components without intersecting the reachable boundary; see Figure 14c. We obtain a modified instance B' with a single unreachable component and apply the algorithm from Section 4. Note that in general, unreachable components cannot be connected by straight lines; see Figure 15b. For a similar (more general) scenario, Guibas et al. [32] propose an approach to compute a subdivision that requires $O(h)$ more segments than an optimal solution (where h is the number of components in the input region). We propose a heuristic without any nontrivial worst-case guarantee. However, it is easy to implement and produces good results in practice (see Section 6).

Given a border region B whose unreachable boundaries U consist of multiple components, our algorithm runs a breadth first search (BFS) on the dual graph of the triangulation of B to find paths that connect the unreachable components. Then, we add new edges that connect the components and retriangulate the modified border region.



(a)



(b)

Figure 15: A border region with two unreachable components u_1 and u_2. (a) The indicated components u_1 and u_2 are connected by a single edge (blue) of the triangulation. (b) The unreachable components u_1 and u_2 are connected by two connecting edges (dark blue) and several bridging edges (light blue), which use the inserted dummy vertices.

We distinguish two cases for connecting two unreachable components u_1 and u_2 in U. First, u_1 and u_2 may be connected by a single edge of the triangulation, i. e., an edge that has an endpoint in each component. Then, we can add this edge to B to connect u_1 and u_2; see Figure 15a. Second, we have to deal with components that are not connected by such an edge. In this case, there exists at least one edge e in the triangulation of B, such that both endpoints of e are on the reachable boundary, and e separates B into two subregions containing u_1 and u_2, respectively. Hence, any path connecting u_1 and u_2 in B crosses e. Our goal is to find a short path in the dual graph of the triangulation that connects u_1 and u_2. Then, we add new edges to the corresponding sequence of triangles to connect u_1 and u_2 in B; see Figure 15b. Afterwards, we locally retriangulate the modified part of B.

Connecting Components. Our algorithm starts by checking for each pair of components, whether they are connected by a single edge in the triangulation of B. This can be done in a sweep over the vertices of each unreachable component, scanning for each vertex its outgoing edges in the triangulation. Whenever an edge is found that connects two unreachable components, we merge these components and consider them as the same component in the further course of the algorithm (making use of a union-find data structure).

To connect all remaining unreachable components after this first step, we proceed as follows. Consider the (weak) dual graph of the triangulation of B. Since no pair of remaining unreachable components can be connected by a single edge in the primal graph, each triangle intersects at most one unreachable component. We assign a component to each vertex in the dual graph, namely, the reachable component if the corresponding triangle contains only reachable vertices, otherwise the unique unreachable component this triangle intersects. For each unreachable component, we add a super source vertex to the dual graph that is connected to all vertices assigned to this component; see Figure 16. Now, our goal is to find a tree of minimum total length in this graph that connects all super sources, i. e., a minimum Steiner tree. Since this poses an NP-hard problem in general [27], we use the approach of Kou et al. [38], which achieves an approximation ratio better than 2. Its basic idea is to iteratively add shortest paths between two sources that are not connected yet in a greedy fashion. A search proposed by Wu et al. [54] computes these paths in a given weighted graph. (Faster algorithms exist for this weighted case [43, 53].) Since the given (dual) graph

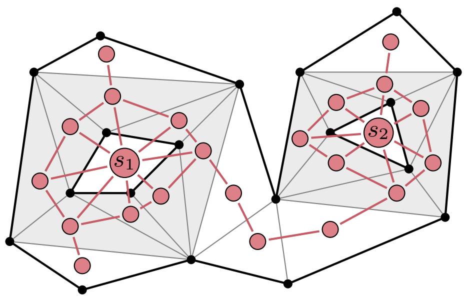


Figure 16: The dual graph of a border region (red), with indicated super sources s_1 and s_2. Shaded triangles are assigned to one of the two unreachable components.

is unweighted in our case, the search algorithm by Wu et al. boils down to a multi-source variant of a BFS, which we now describe in detail.

Given an undirected graph G and k source vertices, it keeps a list of vertex labels $\ell(\cdot)$ to mark visited vertices, store their parents in the search, and the corresponding source vertex of the path that reached this vertex. We use a union-find data structure (with k elements) to maintain connectivity of sources. Initially, we mark all source vertices as visited and set each as its own source. Moreover, all source vertices are inserted into a queue. Then, the main loop is run until all source vertices are connected. In each step, the search extracts the next vertex u from the queue and checks all incident edges $\{u, v\}$. If v was not visited, it marks its label as visited, and sets its source as the source of the label of u. Additionally, v is inserted into the queue. On the other hand, if v was already visited, we found a path that connects two sources. We check whether the sources of the labels $\ell(u)$ and $\ell(v)$ are not connected yet. If this is the case, we found the first path that connects them, so we unify the sources (i. e., they are considered equal in the further course of the algorithm). The actual path can be retrieved by backtracking from u and v, respectively, following the parent pointers until the source is reached. The concatenation of both paths yields a path that connects two source vertices. The algorithm stops when all sources are connected.

After the search terminates, we *split* some triangles by adding new vertices and edges to B in the following manner; see Figure 15b. Consider a path in the dual graph connecting two components u_1 and u_2. First, we remove the first and last vertex of this path (since these are previously added super sources). For the remaining path, consider the corresponding sequence of triangles in B. Clearly, all but the first and last triangle of this path only have endpoints in the reachable component (otherwise, backtracking would have started or stopped earlier). For every edge shared by two triangles in this path, we add a *bridging vertex* at the center of this edge. Thus, a bridging vertex is always contained in an edge with two reachable endpoints. Between any pair of bridging vertices contained in the same triangle, we add a *bridging edge* connecting them. Finally, at the first and last triangle of the path, we add a *connecting edge* from the unique endpoint that belongs to an unreachable component to the bridging vertex. Assigning all added vertices and edges to the unreachable boundary, the resulting border region B' contains a connected unreachable component $U' \supseteq \{u_1, u_2\}$.

Finally, we add new edges (if necessary) to any created quadrangles to maintain the triangulation. Thus, the resulting modified border region B' is triangular and its unreachable boundary consists of a single component. We run the algorithm described in Section 4 to obtain the desired range polygon.

For correctness, we need to show that the union of all edges added according to the computed Steiner tree creates no crossings. First, note that bridging edges in a triangle (corresponding to different subpaths of the Steiner tree) never cross each other. Second, we claim that if a connecting edge is inserted in some triangle, no other edge is added to that triangle. Since a connecting edge has an endpoint in some unreachable component, at most one edge of the triangle has two reachable endpoints. Hence, it contains at most one bridging vertex, and therefore no bridging edge. Moreover, the triangle contains at least one bridging vertex, which is the other endpoint of the connecting edge. Thus, it has exactly two reachable endpoints and does not contain more than one connecting edge.

The BFS described above visits each vertex of the dual graph at most once. In each step, a constant number of calls to the union-find data structure is made to check whether sources of two given labels are connected and unify them if necessary (vertex degree is constant except at super sources, where no checks are performed). All other operations require constant time. Using path compression for the union-find data structure [51], this yields a running time of $O(n\alpha(n))$ of the BFS, where n is the number of vertices in the dual graph (which is linear in the size of B) and α the inverse Ackermann function. Since the remaining steps of the heuristic (adding vertices and edges to triangles, computing a minimum-link path) require linear time, the overall running time is almost linear.

Improvements. In practice, the performance of the BFS is dominated by the number of visited vertices. We propose tuning options that reduce this number significantly, without affecting correctness of the approach (although results may change slightly).

One crucial observation is that realistic instances of border regions often have an unreachable boundary consisting of one large component (the major part of the unreachable subgraph), and many tiny components (e. g., unreachable dead ends in the road network), similar to Figure 5. Then, the search from the large component dominates the running time. Instead, we can run the BFS starting from all but the largest component. This requires only negligible overhead (we identify the largest component in an additional linear scan), but searches from small components are likely to quickly converge to the large component. In preliminary experiments, this reduced running time significantly. Further, after extracting the next vertex from the queue, we first check whether its source was connected to the largest component in the meantime. If this is the case, we prune the search at this vertex (i. e., we discard the vertex and do not check its incident edges), because it now represents the search from the largest component. Similarly, before running the BFS we omit vertices of the largest component when checking for edges in the triangulation that connect two components.

Going even further, we always expand the search from the component that is currently the smallest. In its basic variant, the BFS uses a queue to process vertices in first-in-first-out order. For better (practical) performance, we replace it by a priority queue whose elements are components (represented by source vertices) instead of vertices. Additionally, we maintain a queue for each component that stores vertices and extracts them in first-in-first-out order. In the priority queue, each component uses its complexity (i. e., its number of edges in the border region) as key. In each step of the BFS, we check for the component with the smallest key in the priority queue, and extract the next vertex from the queue of this component. If it has run empty, we remove the component from the priority queue. New vertices are always added to the queue that corresponds to the component of their source label. Whenever two components are unified, we also update them in the priority queue by removing one of the two involved components, attaching its queue to the other component, and updating the key accordingly. If components in the priority queue are implemented as *lists* of queues, new queues can be removed and reattached in constant time. In total, the use of a priority queue then increases the asymptotic running time of the BFS by a logarithmic factor, but we observe a significant speedup in practice.

Data Structures and Implementation Details. When running a BFS on the dual graph of a border region B, we implicitly represent the search graph using the triangulation of the graph G_p. To determine incident edges of a vertex in the dual graph, we check the edges of its triangle in the primal graph. If any of these edges is not contained in G_p (i. e., it was added during triangulation), or if it is contained in E_x, there exists an edge in the dual graph connecting the triangle to the twin triangle of this edge.

In the improved variant that makes use of a priority queue, our implementation actually keeps a single queue per component, rather than a list of queues. Whenever some components are unified, the keys of affected components in the priority queue are updated in a scan over all elements it contains. This increases asymptotic running time of the BFS by another linear factor (in the number of components, which can be linear in the size of the border region). However, the number of components is usually small in practice and we avoid overhead for maintaining dynamic lists of queues.

Recall that we consider a scenario where many subsequent queries for different range polygons are made to the same graph. To avoid costly reinitialization of the vertex labels between such queries, we make use of timestamps, implicitly encoded within the component indices to save space. After every search, the global timestamp is increased by the number of unreachable components in the border region. When storing a component index in a label, it is increased by the global timestamp. A label is invalid if its index is below the global timestamp. To retrieve the actual index of a valid label, we subtract the global timestamp.

Backtracking runs on-the-fly during the BFS and stops whenever we reach a previously split triangle, since this means we have reached a previously computed path. We maintain flags at each triangle, to determine whether a bridging edge or a connecting edge should be inserted (and if so, between which pair of endpoints). We also build a list of all split triangles, for fast (sequential) access to all triangles that were split, in order to add the corresponding vertices and edges in the triangulation after the BFS has terminated. These additional edges and vertices are stored as temporary modifications in the triangulation of G_p. We make use of the following data structures. Edges of the triangulation that are added to U' (to connect two unreachable components) are explicitly stored in a list. To quickly check whether some edge of the triangulation was added to U', we sort this list (e. g., by head vertex index) after the BFS terminated to enable binary search. In our setting (some 1 000 inserted edges for the hardest queries), this was slightly faster than using hash sets. To store bridging edges and connecting edges, we temporarily modify the triangulation. To this end, we add an invalidation flag and a temporary index to the vertices of every triangle in the triangulation of G_p. Moreover, we maintain a list of temporary triangles. Each vertex in a split triangle is marked as invalid and its temporary index is set to the corresponding entry in this list. Before retrieving a triangle vertex, we first check whether it is invalid and redirect to the temporary vertex if this is the case. For faster reinitialization, we replace invalidation flags by timestamps, similar to component timestamps described above.

When splitting triangles, we have to set the twins of all new edges. If the twin triangle was not created yet, we store the pending edge in a list. This list is searched for existing twins whenever a new triangle is added. If a twin is found, we set twins for both affected edges, and remove them from the set.

5.4 Computing Self-Intersecting Minimum-Link Paths

Our last approach computes a minimum-link path in B that separates the reachable boundary from the unreachable boundaries. While the resulting polygon has at most $\mathrm{OPT}+2$ segments, it may intersect itself; see Figure 14d. To obtain a range polygon from a self-intersecting polygon, we rearrange it accordingly at intersections.

The remainder of this section focuses on computing minimum-link paths in border regions with several unreachable components. To achieve this, we have to make some modifications to the algorithm described in Section 4. First, note that the (weak) dual graph of the triangulation of B is not outerplanar if $|U| > 1$. Consequently, paths between vertices (in the dual graph) are no longer unique. In fact, some vertices may occur multiple times in a path that separates reachable and unreachable boundaries; see the corresponding sequence of triangles crossed by the polygon in Figure 14d. In what follows, we first show how to obtain the sequence of important triangles in this general case. Then, we describe modifications that are necessary to retain correctness of the algorithm described in Section 4 running on this sequence of important triangles.

Computing the Important Triangles. Given a boundary edge e of the border region B, we are interested in a minimum-link path that connects both sides of e and separates the reachable boundary from all unreachable boundaries. We compute a sequence $t_1, \dots, t_k$ of triangles such that any minimum-link path with the above property must pass this sequence in this order. Our approach runs in two phases. The first phase traverses the reachable boundary of B and lists all encountered boundary edges in the triangulation, i. e., all edges with one endpoint in each R and U, even if they are not present in the input graph G_p. Clearly, the minimum-link path must intersect exactly these boundary edges in the same order to ensure that all unreachable vertices are on the same side of the path. By construction, any pair of consecutive boundary edges a and b in this list is connected by a path π in the dual graph that contains no boundary edge besides a and b (where an edge in the dual graph is called boundary edge if it corresponds to a boundary edge of the primal graph). In fact, observe that π is actually unique, since any cycle in the dual graph contains at least one boundary edge. The second phase computes this unique path π for each pair of consecutive

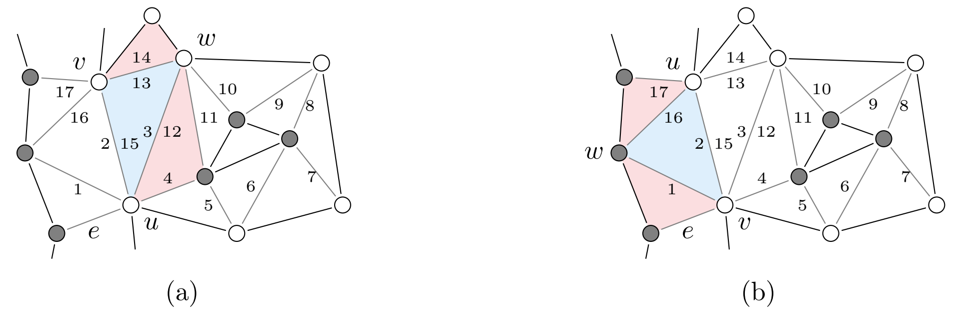


Figure 17: Border region with edge indices after starting traversal at the indicated edge e. Note that edges have two indices (one for each direction of traversal). Indices are ∞ if not specified. Indices of boundary edges are also stored in an array $[1, 4, 5, 6, 7, 8, 9, 10, 11, 16, 17]$. (a) The third visited triangle in the second phase (shaded blue) has two possible next triangles t_{uw} and t_{vw} (shaded red). The next triangle is t_{uw}, because the index of the edge vw with greater index (13) exceeds the next boundary edge index (4). (b) The next triangle in this example is t_{uw}. The index of the next boundary edge is updated from 16 to 17.

boundary edges. The concatenation of all these paths yields the actual sequence of important triangles. It serves as input for the algorithm that computes a minimum-link path connecting both sides of e in B.

During the first phase, we exploit the fact that the reachable boundary of B is always connected. We assign *indices* to all edges in the triangulation that are contained in the border region and intersect the reachable boundary, according to the order in which they are traversed starting from e. In doing so, we distinguish both sides of edges; see Figure 17. For consistency, sides of edges that are not traversed get the index ∞. Clearly, this information can be retrieved in a single traversal of the reachable boundary, similar to the procedure described in Section 5.1, but running on the triangulation of the border region. Moreover, during this traversal we collect an ordered list of indices corresponding to boundary edges. Observe that every boundary edge in B is traversed exactly once.

The second phase runs on the dual graph of the triangulation and retrieves the desired sequence of triangles. A key observation is that this sequence must pass all boundary edges exactly once and in increasing order of their indices. Therefore, we can compute the sequence of important triangles as follows. We maintain the index of the next boundary edge that was not traversed yet, initialized to the first element of the list. Starting at the triangle t_1 containing the first boundary edge e, we add triangles to the sequence of important triangles until e is reached again. Let $t_i = uvw$ denote the previous triangle that was appended to this sequence. Then we determine the next triangle t_{i+1} as follows; see Figure 17. Let uv be the unique edge shared by t_i and t_{i-1} (in the case of $i = 1$, we have $uv = e$). To determine the next triangle, we consider the two possible triangles t_{uw} containing the edge uw and t_{vw} containing vw. Without loss of generality, let the index of uw be smaller than the index of vw and thus, finite. This implies that uw is not contained in the boundary of B (otherwise, it would have index ∞). If both u and w are part of the reachable boundary, we know that uw separates B into two subregions; see Figure 17a. Thus, t_{uw} is the next triangle if and only if the subregion containing t_{uw} contains a boundary edge that was not passed yet. Therefore,

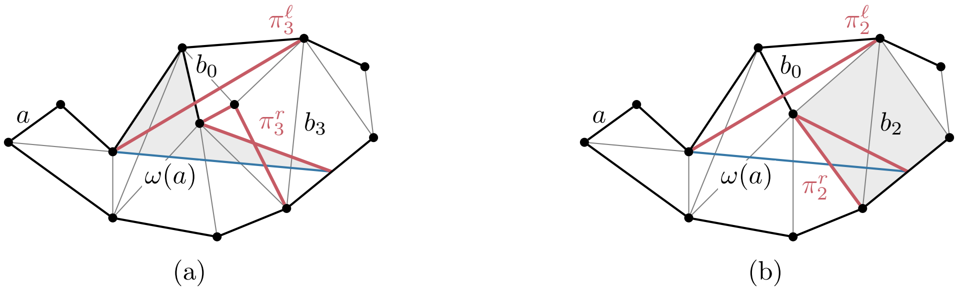


Figure 18: Shortest paths in general border regions. Assume that vertices in the center of the polygon are not connected to the remaining unreachable boundary (bottom). (a) The shortest path π_3^r from the right endpoint of the previous window $\omega(a)$ to the right endpoint of b_3 intersects itself. (b) The shortest path π_2^r from the right endpoint of $\omega(a)$ to the right endpoint of b_2 contains a left bend after passing an unreachable vertex that is not connected to the unreachable boundary.

we continue with t_{uw} if and only if the index of the other edge vw is greater than of the next boundary edge. If either u or w is part of an unreachable boundary, uw is the next boundary edge; see Figure 17b. We update the index of the next boundary edge to the next element in the according list.

We continue until the first edge e is reached again. Note that the second phase (traversing the dual graph) can be performed on-the-fly during minimum-link path computation (i. e., the sequence of triangles does not have to be built explicitly).

Computing Minimum-Link Paths. Given a sequence $t_1, \ldots, t_k$ of important triangles in a border region B computed as described above, we discuss how to compute a minimum-link path between both sides of e. In particular, we show which modifications to the approach presented in Section 4 are necessary to preserve correctness.

Consider the computation of a window from an arbitrary *initial edge* shared by two triangles in $t_1, \ldots, t_k$ as described in Section 4. Clearly, the subsequence of triangles that is visited until a window is found does not contain several occurrences of the same triangle, since this would imply that a straight visibility line intersects it at least twice. Consequently, the fact that triangles may appear multiple times in $t_1, \ldots, t_k$ does not affect window computation starting at an edge. A similar argument applies when initializing the computation of a subsequent window. Recall that in this step, the visibility cone from the last window to the next initial edge is computed. All triangles considered in this step are intersected by the last window in a certain order (see Section 4) and because windows are straight lines, we cannot encounter the same triangle twice.

However, the computation of the next window after this initialization step requires some modification, since triangles visited during initialization may reoccur when computing the window from the next initial edge. As a result, the subpaths computed during initialization and when starting from this edge may cross each other; see Figure 18a. In this example, the subpath of the right shortest path π^r starting at the initial edge b_0 intersects the segment from the right endpoint of the previous window $w(a)$ to the right endpoint of b_0. Self-intersections would not pose a problem per se if we generalized the definition of shortest paths to polygons with self-intersections. However, without modifications, Algorithm 1 in Section 4 may produce wrong results in certain special cases. Figure 18b shows such an example. Although the shortest path from the right endpoint of $w(a)$ to the right endpoint of b_2 does not intersect itself in this case, its second segment lies in the half plane to the left of the first segment. Hence, Algorithm 1 will falsely remove the last bend. The resulting incorrect path consists of the single segment from the right endpoint of $w(a)$ to $r(b_0)$. Clearly, this leads to the construction of an incorrect visibility cone. We say that the last segments of the right shortest paths shown in Figure 18 are *visibility-intersecting*, as they reach into the area that is visible from $w(a)$. Formally, a segment is visibility-intersecting if it intersects the interior of the hourglass H_0 bounded by the previous window $a' := w(a)$, the next initial edge b_0, and the initial left and right shortest paths π_0^ℓ and π_0^r; see the shaded area in Figure 18a. Note that visibility-intersecting segments can only occur in the shortest path that corresponds to the unreachable boundary, since the reachable boundary consists of a single component.

In what follows, we show how we can avoid visibility-intersecting segments that may spoil the algorithm presented in Section 4. As argued above, visibility-intersecting segments only occur if a triangle that was visited during the initialization phase is visited again when computing the next window from a boundary edge. We could resolve this issue conceptionally easy by retriangulating parts of the border region, namely, the part called P_0' in Section 4. Below, we present an approach that avoids retriangulation by making use of few simple checks instead. (In a sense, it simulates the situation after such a retriangulation.) To this end, we show how we can easily detect visibility-intersecting segments. Then, we show that we can simply omit such segments from the corresponding shortest path. As a result, our adaptation is very easy to implement and produces negligible overhead in practice.

We claim that a segment that is appended to a shortest path π^r is visibility-intersecting if and only if this segment intersects the previous window and is not an endpoint of this window. For the sake of simplicity, we assume general position. Thus, the window a' and the path π^r share no common segment. In practice, such a segment can easily be detected and removed from both the path and the window during initialization of π_0^r.

Lemma 6. *Given the previous window a', let b be the next initial edge in B. Let t_i ($i \geq 1$) be an important triangle such that the shortest path π_{i-1}^r contains no visibility-intersecting segments and the edge b_i shared by t_i and t_{i+1} is (partially) visible from a'. The next segment s appended to π_{i-1}^r is visibility-intersecting if and only if s intersects the open line segment a'.*

Proof. Note that we consider the previous window a' to be an open line segment, since its right endpoint coincides with an endpoint of the first segment of π_{i-1}^r, which clearly is not visibility-intersecting.

First, assume the segment $s = pq$ is visibility-intersecting and assume for contradiction that s does not intersect a'. Since s is visibility-intersecting, it intersects the interior of the hourglass H_0 enclosed by a', b_0, π_0^r and π_0^ℓ. Since the interior of H_0 contains no vertices (in particular, neither p nor q), the edge s of t_i must intersect the boundary of H_0 at least twice. However, s does not intersect the interior of the edge b_0, since both s and b_0 are edges of triangles. As s does not intersect a' by assumption, it intersects π_0^r or π_0^ℓ. Moreover, since both paths are concave in H_0, s must intersect both paths. (If it intersects any path twice at two points p' and q', the subsegment that connects p' and q' does not intersect the interior of H_0, so s has at least one additional intersection with the boundary of H_0.) But s does not intersect π_0^ℓ, because this would imply that the paths π_i^r and π_0^ℓ have non-empty intersection, contradicting the fact that b_i is visible from a'.

Second, assume that s intersects the open segment a'. Since a' contains no endpoint of s, we know that s intersects the interior of H_0. Hence, s is visibility-intersecting. □

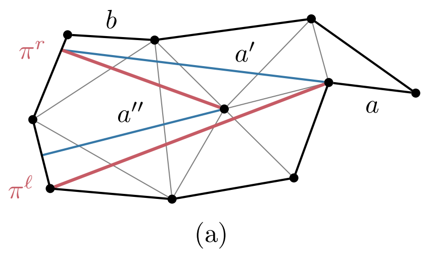
Next, we show that visibility-intersecting segments can safely be omitted from the shortest path computed by the algorithm. Let $t_i, \dots, t_j$ be a subsequence of important triangles, such that the edge of t_i appended to π_{i-1}^r by the algorithm of Section 4 is visibility-intersecting, and t_j is the first triangle (i. e., with lowest index $j > i$) such that the edge b_j shared by t_j and t_{j+1} does not intersect the open segment a'; see the sequence of shaded triangles in Figure 18b. Thus, all edges $b_i, \dots, b_{j-1}$ intersect a'. Moreover, π_{i-1}^r and b_j lie on the same side of a' (otherwise, the window a' would cross the reachable boundary of B). We distinguish two cases, depending on whether b_j is visible from a'. We show that in both cases, we can skip all right endpoints of the edges $b_i, \dots, b_{j-1}$ when updating the path π^r to obtain the correct next window.

First, assume that b_j is (partially) visible from a'. We claim that no right endpoint of an edge $b_i, \dots, b_{j-1}$ is contained in the shortest path π_j^r. To see this, let u denote the last vertex of π_{i-1}^r and w the right endpoint of b_j. Clearly, a' separates u and w from all right endpoints of the edges $b_i, \dots, b_{j-1}$. Since b_j is visible, this implies that the segment uw crosses all edges $b_i, \dots, b_{j-1}$. Therefore, it does not intersect the boundary of B and there can be no shorter path from u to w passing any right endpoint of these edges.

Second, assume that b_j is not visible from a'. We claim that no right endpoint v of an edge $b_i, \dots, b_{j-1}$ is contained in the visibility cone from a'. Assume for contradiction that such an endpoint v is visible from a'. We know that v and the edge b_{i-1} lie on opposite sides of a'. Since v is visible from a', there exists a straight line that crosses a', b_{i-1} and v in this order. Consequently, it must cross a' twice, contradicting the fact that it is a straight line. Since v is not part of the visibility cone from a', it is not relevant for the computation of the next window $w(a')$.

In both cases, we can safely ignore right endpoints of all edges $b_i, \dots, b_{j-1}$. We adapt our algorithm as follows. Before adding a segment to the shortest path that corresponds to the unreachable boundary, we check whether it intersects the previous window a'. If this is the case, we do not add it to the path.

Finally, applying these modifications, we have to clear one last issue to enable correct initialization of the computation of the next window. Recall that during this initialization, one shortest path becomes the suffix of a previous path; see Section 4. Figure 19 shows an example where this suffix is not available in the modified algorithm. Assume we want to compute a minimum-link path between the indicated edges a and b. Figure 19a depicts the first window $w(a) = a'$. Starting from a', the shortest paths π^r and π^ℓ are computed to obtain the next window $w(a')$ (note that the right endpoint of $w(a')$ is an unreachable component consisting of a single vertex). To compute the next window $w(a'')$ from $w(a') = a''$, we



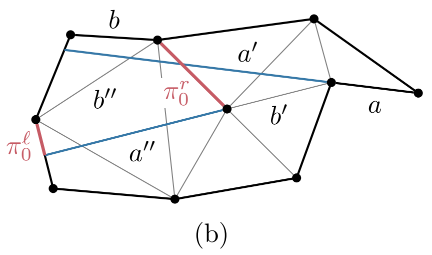


Figure 19: Computing initial shortest paths. Assume we want to compute a minimum-link path from a to b, such that the (unreachable) vertex in the center is to the right of this path. (a) The window $a'' = \omega(a')$ computed after the first window a', with final shortest paths (red). (b) The initial shortest paths π_0^ℓ and π_0^r connecting the endpoints of a'' and the next initial edge b'' when computing the next window $\omega(a'')$. To resolve the issue, we compute another shortest path from the right endpoint of b', which equals π_0^r when a'' is found.

first have to compute the initial paths π_0^ℓ and π_0^r, as shown in Figure 19b. However, π_0^r consists of a segment that is not present in the previous right shortest path, because it is visibility-intersecting for this path; see π^r in Figure 19a.

To resolve this problem, we maintain another shortest path π' starting at the corresponding endpoint of the initial edge b', i. e., segments are added as in the original algorithm in Section 4. As argued before, this path does not contain self-intersections. Then, the initial shortest path is a suffix of π', since the only segments omitted from π' are on the shortest path from the previous window $w(a) = a'$ to the previous initial edge b'. Clearly, these segments cannot be part of the next initial shortest path, since b' is fully visible from a'. Hence, the first endpoint of the window $w(a')$ must be a point in π'.

Final Remarks. In summary, our algorithm consists of two steps. The first step traverses the reachable boundary in the triangulation of B. The second step runs a modified version of the algorithm described in Section 4, maintaining the next boundary edge index to compute the sequence of important triangles on-the-fly. Both steps are modifications of previous algorithms that clearly maintain their linear running time. While the resulting polygon P may intersect itself, it has at most $\mathrm{OPT}+2$ segments. To obtain a range polygon P' without self-intersections, we can split P into several non-crossing polygons at intersections. From the resulting smaller polygons, we discard those that contain no vertices of G_p or are fully contained in another polygon. To ensure that P' consists of a single component, according to our primary optimization criterion in Section 2, we can reuse (partial) segments of P to connect the non-crossing components of P'. The number of additional segments clearly is linear in the number of self-intersections, which is small in practice (see Section 6).

6 Experiments

We implemented all approaches in C++, using g++ 4.8.3 (-O3) as compiler. Experiments were conducted on a single core of a 4-core Intel Xeon E5-1630v3 clocked at 3.7 GHz, with 128 GiB of DDR4-2133 RAM, 10 MiB of L3, and 256 KiB of L2 cache.

Table 1: Results of our algorithms when computing the range of an electric vehicle, for different ranges (using travel time as edge length function). We report, for each algorithm and scenario, the number of components of the resulting range polygon (Cp.), the complexity of the range polygon (Seg.), the number of self-intersections (Int.), and the running time of the algorithm in milliseconds (T.). Figures are average values of 1 000 random queries.

	EV, 16 kWh				EV, 85 kWh			
Algorithm	Cp.	Seg.	Int.	T. [ms]	Cp.	Seg.	Int.	T. [ms]
`RP-RC`	41	19 396	—	4.50	131	92 554	—	9.46
`RP-TS`	69	610	—	4.30	219	1 973	—	7.78
`RP-CU`	41	561	—	10.15	131	1 820	—	25.11
`RP-SI`	41	549	4.79	7.52	131	1 781	15.06	22.25

Input Data. Our main benchmark instance is a graph representing the road network of Europe, kindly provided by PTV AG (`ptvgroup.com`). We extract travel times on road segments from the provided data. This enables us to generate edge weights for computing isochrones. Energy consumption data for electric vehicles is derived from a detailed micro-scale emission model [33], calibrated to a real Peugeot iOn. It has a battery capacity of 16 kWh, but we also consider batteries with 85 kWh, as in high-end Tesla models. Edges for which no reasonable energy consumption can be derived (e. g., due to missing elevation data) are removed from the graph [7]. To ensure that our graph is strongly connected, we extract the largest strongly connected component of the remaining subgraph. This results in a graph with 22 198 628 vertices and 51 088 095 edges. To improve spatial locality of the input data, we reorder these vertices according to a vertex partition of the graph [7]. This slightly improves query performance. We construct a planar graph, which is directed but unweighted in our implementation. As mentioned in Section 2, we add four bounding box vertices in each corner of the embedding, along with eight (directed) edges connecting each vertex to the closest vertex of the input graph and the two closest bounding box vertices. During planarization, 293 741 vertices are added and 654 765 edges are split. Note that a dummy vertex may intersect more than two original edges, which explains why the number of split edges is more than twice the number of dummy vertices. They are replaced by 1 591 914 dummy edges (creating 6 294 multi-edges due to overlapping original edges in the given embedding). After planarization, the resulting graph has 22 492 373 vertices and 52 025 261 edges. After triangulating all faces, it has 131 977 245 edges. In total, data structures used by our algorithms require about 20 GiB of space for storing the graph, the faces of the planar graph, its triangulation, and any additional data structures used by the different algorithms. In a typical server-based scenario, this is not an issue, since the whole dataset fits into RAM.

Evaluating Queries. We evaluate query scenarios for range visualization of an electric vehicle, as well as isochrones. For electric vehicles, we consider two scenarios, one for medium (16 kWh) and one for large (85 kWh) battery capacity, corresponding to a range of roughly 100 and 500 km, respectively. We compare the results provided by the algorithms proposed in Section 5. Each algorithm was tested on the same set of 1 000 queries from source

Table 2: Overview of the results of our algorithms when computing isochrones for different ranges. Reported results are similar to Table 1. They were obtained by running 1 000 random queries from the same set of source vertices.

	Isochrones, 60 min				Isochrones, 500 min			
Algorithm	Cp.	Seg.	Int.	T. [ms]	Cp.	Seg.	Int.	T. [ms]
RP-RC	53	22 458	—	4.75	231	238 123	—	20.25
RP-TS	151	1 076	—	4.65	694	4 981	—	14.96
RP-CU	53	913	—	12.11	231	4 208	—	65.09
RP-SI	53	881	9.95	8.70	231	4 055	45.80	51.94

vertices picked uniformly at random. We denote by `RP-RC` (*range polygon*, extracted *reachable components*) the approach presented in Section 5.1, by `RP-TS` (*triangular separators*) the algorithm from Section 5.2, by `RP-CU` (*connecting unreachable* components) the approach from Section 5.3, and by `RP-SI` (*self-intersecting* polygons) the algorithm from Section 5.4. Below, we only evaluate Steps 2 to 4 of the algorithm outlined in Section 2. The first step, i. e., the computation of the reachable and unreachable parts of the graph, was covered by previous work [6]. It is briefly discussed at the end of this section.

Table 1 shows an overview of the results of all heuristics when visualizing the range of an electric vehicle, organized in two blocks. The first considers the medium-range scenario, while the second shows results for long ranges. For each, we report the average number of components, complexity, and the number of self-intersection of the computed range polygons. We also report the average running time in each case. For `RP-SI`, the number of components and the complexity are reported as-is after running the modified minimum-link path algorithm described in Section 5.4 (i. e., resulting polygons have the number of self-intersections reported in the table). Thus, figures slightly change after resolving the intersections (both the number of components and the complexity may increase slightly).

All algorithms perform very well in practice, with timings of 25 ms and below even for large battery capacities. The simpler algorithms, `RP-RC` and `RP-TS` are faster by a factor of 2 to 3, compared to the more sophisticated approaches. On the other hand, we see that range polygons generated by `RP-RC` have a much higher complexity, exceeding the optimum by more than an order of magnitude. The heuristic `RP-TS` provides much better results in terms of complexity, but is still outperformed by the other two approaches. Moreover, the triangular separation increases the number of components by almost a factor of 1.7 (all other approaches in fact compute the minimum number of holes). Regarding the two more involved approaches, `RP-CU` and `RP-SI`, we see that the additional effort pays off. Both approaches compute range polygons with the optimal number of components, while keeping the complexity close to the optimum. In fact, we know that each component in the possibly self-intersecting polygon computed by `RP-SI` requires at most two additional segments compared to an optimal solution. Taking into account that many small components are triangles (which have optimal complexity), we derive lower bounds on the optimal average complexity of 529 (16 kWh) and 1 720 (85 kWh) for a range polygon with minimum number of components. Hence, our experiments indicate that the average relative error of both `RP-CU`

Table 3: Running times of different phases of the algorithms (where applicable) for isochrones with a range of 500 minutes. For each algorithm, we report the total running time (Total) together with the running time for transferring the input to the planar graph (TP), extracting the border regions (BE), connecting components (CC), the range polygon computation with minimum-link paths (RP), and the test for self-intersections (SI), given in milliseconds.

Algorithm	TP	BE	CC	RP	SI	Total
`RP-RC`	8.21	12.01	—	—	—	20.25
`RP-TS`	8.22	—	—	6.45	—	14.96
`RP-CU`	8.23	26.66	22.99	7.81	—	65.09
`RP-SI`	8.20	31.79	—	9.53	2.34	51.94

(upper bounded by 6 %) and `RP-SI` (upper bounded by 4 %) is negligible in practice. The number of intersections produced by `RP-SI` is also rather low, but the majority of computed range polygons contains at least one intersection (97.2 % for a range of 85 kWh; not reported in the table).

In Table 2, we provide the according figures for isochrones, considering a harder scenario with a range of 500 minutes (roughly eight hours). Resulting isocontours are among the most complex on average in our setting (for longer ranges, the border of the network is reached by many queries). Despite the increase in running time and solution size, we make similar observations as before. All approaches show great performance, with average running times of 65 ms and below in all cases. Again, the average complexity of range polygons computed by `RP-RC` is larger compared to other heuristics by about a factor of 50, with range polygons consisting of more than 200 000 segments on average for the long range. This clearly justifies the use of our proposed algorithms, since a significant decrease of this number is beneficial when efficient rendering or transmission over mobile networks is an issue. Moreover, a lower number of segments leads to a more appealing visualization for ranges of this order. For `RP-TS`, the number of components now exceeds the optimum by about a factor of 3. Again, the two more sophisticated approaches `RP-CU` and `RP-SI` yield best results with average relative errors bounded by 7 % and 3 %, respectively.

For the hardest scenario (isochrones, 500 min.), we report more detailed information on running times of the different phases of all algorithms in Table 3. Note that the total running time slightly differs from the sum of all subphases, since it was determined independently. The planarization phase (TP) consists of the linear sweeps described in Section 3. Since the same work needs to be done for all approaches, the running time is identical in all cases (bar measurement noise). Of course, the relative amount spent in this step differs per algorithm. In case of `RP-TS` it requires more than half of the total running time. The time to extract the border regions (BE) applies to all algorithms except `RP-TS`, where this is done implicitly by checking the reachability of vertices. Since `RP-RC` extracts only the reachable boundary, this phase takes less than half the time compared to `RP-CU` (the unreachable boundary is typically larger). Finally, `RP-SI` spends most time in this step, since it runs the extraction on the triangulated graph, which is significantly more dense. Recall that `RP-SI` in fact only extracts the reachable border, similar to `RP-RC`, so this phase is slower by more than a factor

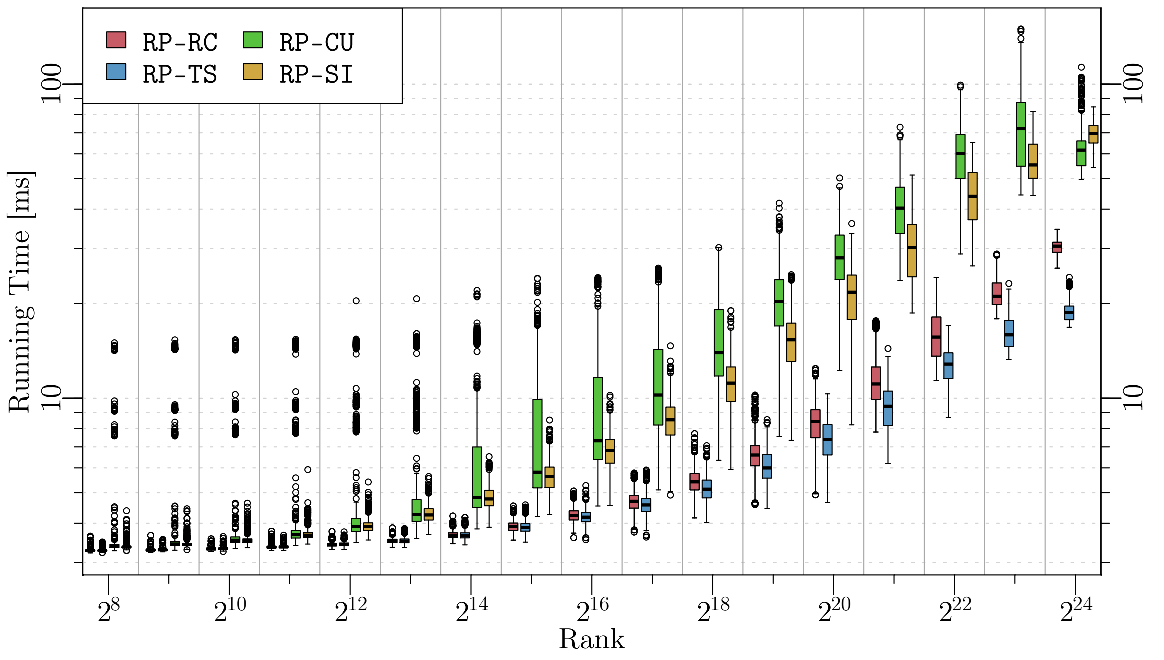


Figure 20: Running times of all approaches subject to Dijkstra rank. Smaller ranks indicate queries of shorter range. For each rank, we report results of 1 000 random queries.

of 2.5. Connecting unreachable components (CC) is only necessary in `RP-CU` and takes less time than the extraction of border regions. Computing the actual range polygon takes a similar amount of time for all approaches that run this phase (6 to 10 ms). For `RP-TS`, it is slightly faster, since the algorithm works only on important triangles, reducing the number of visited triangles and simplifying the algorithm. On the other hand, `RP-SI` is the slowest approach in this phase. This can be explained by the overhead caused by the modifications described in Section 5.4. Moreover, in contrast to `RP-TS` and `RP-CU`, there are no artificial edges in the border regions. Hence, windows computed by `RP-SI` are longer on average, increasing the number of triangles visited by the algorithm.

In summary, we see that extracting the border region takes a major fraction of the total effort for all approaches that compute B explicitly. Despite the algorithmic simplicity of this phase, this can be explained by the size of the border region. In our experiments, it consisted of more than 500 000 segments on average. On the other hand, only a fraction of the triangles in the border regions are visited by the minimum-link path algorithm.

Evaluating Scalability. Figure 20 analyzes the scalability of our algorithms, following the methodology of Dijkstra ranks [3, 49]. The Dijkstra rank of a shortest-path query is the number of queue extractions performed by Dijkstra's algorithm, presuming that the algorithm stops once the target is found. Thus, higher ranks reflect harder queries. To obtain queries of different ranks, we run Dijkstra's algorithm as described in Section 3 to obtain consumption

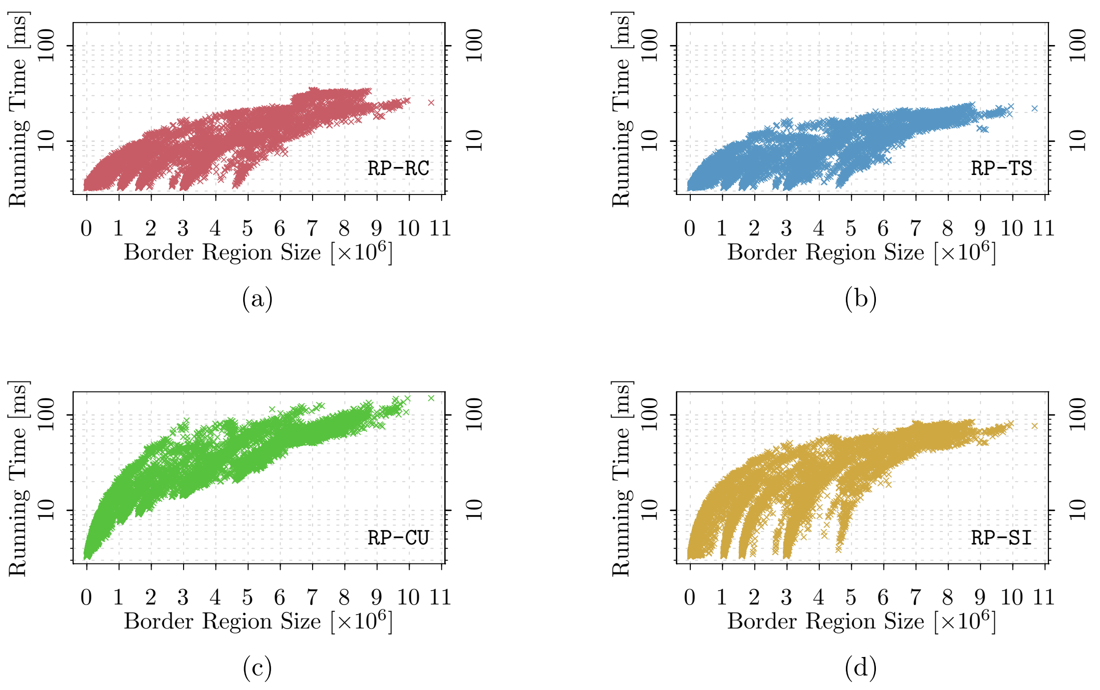


Figure 21: Running times of the same queries as in Figure 20, subject to border region size, for (a) RP-RC, (b) RP-TS, (c) RP-CU, and (d) RP-SI, respectively.

values along shortest paths from 1 000 source vertices chosen uniformly at random. For a search from some source s, consider the resource consumption c of the vertex that is extracted from the queue in step 2^i of Dijkstra's algorithm (the maximum rank is bounded by the graph size). We consider a query from s with range c as a query of rank 2^i. For each rank in $\{2^1, \ldots, 2^{\log |V|}\}$, we evaluate the 1 000 queries generated this way.

Query times of all approaches increase with the Dijkstra rank, which correlates well with the complexity of the border region. Moreover, scaling behavior is similar for all approaches. In accordance with our theoretical findings, our experiment suggests that it increases linearly in the size of the border region for queries beyond a rank of 2^{12}. For queries of lower rank, transferring the input to the planar graph dominates running time, as it is linear in the graph size and thus, independent of the rank. The approach RP-TS is consistently the fastest approach on average for all ranks beyond 2^{16}. Except for a few outliers, our algorithms answer all queries in well below 100 ms. For more local queries (i. e., smaller ranges), query times are much faster (20 ms and below if the rank is at most 2^{20}, corresponding to about a million vertices visited by Dijkstra's algorithm). The more expensive approaches have a higher variance and produce more outliers, which is explained by their more complex phases. For example, the performance of the BFS used by RP-CU heavily depends on how close unreachable components of the border region are in the dual graph.

For each approach, Figure 21 shows query times depending on the border region size for all queries whose rank is at least 8 (i. e., the same set of queries as in Figure 20).

Table 4: Performance of different minimum-link path algorithms. For each scenario, we report the number of segments in the input polygon ($|P|$) and the minimum number of links of resulting paths (Seg.). For Suri's algorithm [50], we show the number of visibility polygon computations (V. Pol.), the total number of segments in the input for these computations (Pol. Seg.), and the total number of visible triangles in these inputs (Trng.). For FMLP, the table provides the number of triangles visited by the algorithm (Trng.) and the running time in milliseconds. Figures are average values for 1 000 queries. Running times exclude the time for triangulating the input polygon, which is part of preprocessing.

			Suri [50]			FMLP	
Scenario	$\|P\|$	Seg.	V. Pol.	Pol. Seg.	Trng.	Trng.	T. [ms]
EV, 16 kWh	134 049	415	2 010	307 583	48 762	8 901	0.74
Iso, 60 min	135 112	700	3 413	320 244	57 549	11 250	1.05
EV, 85 kWh	357 335	1 328	6 442	850 293	178 574	31 657	3.17
Iso, 500 min	637 224	3 203	15 655	1 547 962	359 969	66 163	6.67

Query times clearly increase with border region size. However, we also see clusters of similar running times at certain region sizes. These patterns can be explained by a small number of huge faces in the graph corresponding to, e. g., long coastlines in the network. Including such a face in the border region greatly increases its size, but has limited affect on running time if it is not part of the reachable subgraph.

Evaluating the Computation of Minimum-Link Paths. We take a closer look at the performance of our algorithm for computing minimum-link paths introduced in Section 4, which we refer to as FMLP (*fast minimum-link paths*). To properly evaluate the algorithm in the context of our experimental setting, we proceed as follows. For each query, we consider the largest corresponding border region (with respect to number of segments). To obtain a polygon without holes, as required by the algorithm, we first run our heuristic to connect all unreachable components described in Section 5.3. Then, we add an arbitrary boundary edge to the modified border region, and compute a minimum-link path that connects both sides of this edge. Results are shown in Table 4. Each scenario is based on the respective sets of random queries used in Table 1 and Table 2.

We also compare the performance of FMLP to Suri's algorithm [50], which finds the next window starting from an arbitrary window (or edge) a by computing several visibility polygons as follows. It starts by computing the visibility polygon of a in the polygon bounding all important triangles (as defined in Section 4) intersected by a. Then, it iteratively computes new visibility polygons, each time doubling the number of important triangles in the input polygon until there is an important triangle that is (partially) invisible from a. To obtain the actual window, a final visibility polygon is computed for a polygon bounding the same set of important triangles together with all non-important triangles whose closest important triangle (with respect to distance in the dual graph) belongs to this set. Then, the next window is an edge of this visibility polygon. Clearly, a practical implementation of this algorithm requires a fast subroutine to compute visibility polygons. Moreover, it needs to fill certain degrees of freedom, e. g., generating the input for this subroutine, or determining the window from the resulting visibility polygon. A fair experimental comparison of running times requires a tuned implementation of Suri's algorithm that efficiently fills these degrees of freedom, which is clearly beyond the scope of this work. Instead, Table 4 provides measures that are independent of both the machine and the implementation, such as the number of calls to the subroutine for computing visibility polygons and the total number of segments in the generated input polygons. A recent experimental study on visibility polygon computation [9] proposes a linear-time algorithm on hole-free polygons that is based on a triangulation of the input. It outperforms other approaches in their evaluation because it processes only *visible* triangles and thus, only a fraction of the input. For our purposes, this approach would have to be generalized to compute visibility from windows (not just single points). Still, it is a good candidate for a practical implementation of Suri's algorithm. Therefore, we also report the total number of *visible* triangles in all polygons constructed by Suri's algorithm.

For all considered scenarios, Suri's algorithm requires several thousand calls to the subroutine for visibility polygons. The total number of segments in all polygons built by Suri's algorithm is over 1.5 million for the hardest scenario (500 min isochrones), which even rules out explicit construction of these polygons for practical applications. In addition to that, the total number of triangles visited by Suri's algorithm exceeds the number of triangles visited by FMLP (using the practical visibility polygon algorithm proposed above [9]) by about a factor of 5 to 6. For FMLP, the workload per visited triangle is very small (updating visibility lines and shortest paths). On the other hand, the proposed visibility polygon algorithm used by Suri's algorithm is recursive [9] and therefore possibly less cache efficient. Given that Suri's algorithm requires additional overhead for constructing input polygons and determining the actual windows from visibility polygons, we conclude that the algorithm introduced in Section 4 is much more suitable for practical use.

Comparing the different scenarios evaluated in Table 4, each represents a certain level of difficulty, with the average complexity of the input polygon ranging from some 100 000 to 600 000 segments. Clearly, the number of visited triangles, the number of segments of the resulting path, and the running time increase with the complexity of the input. However, we also see in Table 4 that the algorithm performs excellently in practice, computing minimum-link paths in less than 7 milliseconds, even for input polygons with more than half a million vertices. Somewhat surprisingly, the isochrone scenarios (60 min) is slightly harder then the range scenario (16 kWh), despite a similar input complexity. This can be explained by the different shapes of the respective border regions. Isochrones in road networks reach further on highways and other fast roads, leading to spike-like shapes in the resulting border regions; see also Figure 22. On the other hand, isocontours representing the range of an electric vehicle typically have a more circular shape as highways allow to move faster, but consume more energy. Consequently, range polygons for isochrones require more segments and yield the more challenging scenario.

Other Instances. We also present experiments for instances extracted from openly available OpenStreetMap data. We consider the road network of Switzerland with 3 259 674 (3 269 666) vertices and 6 488 514 (6 518 469) edges before (after) planarization and the road network of Germany with 23 913 390 (23 966 527) vertices and 48 239 355 (48 398 283) edges before

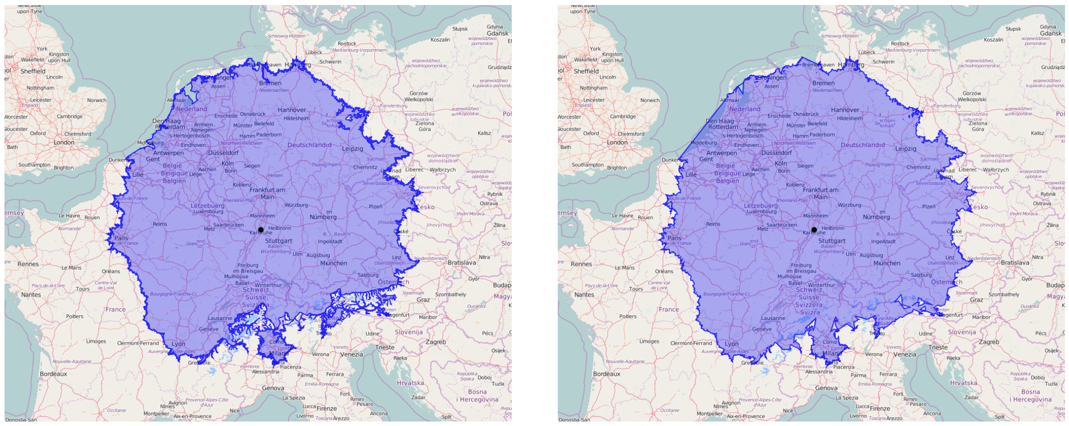
Table 5: Results of our algorithms when computing isochrones for a range of 60 minutes on two different instances. Reported results follow Table 1. They were obtained by running 1 000 random queries, as before.

	Switzerland				Germany			
Algorithm	Cp.	Seg.	Int.	T. [ms]	Cp.	Seg.	Int.	T. [ms]
`RP-RC`	142	258 816	—	15.15	332	223 039	—	16.85
`RP-TS`	422	2 419	—	8.05	924	5 073	—	13.39
`RP-CU`	142	1 957	—	65.10	332	4 070	—	68.76
`RP-SI`	142	1 832	47.05	59.55	332	3 833	95.16	68.53

(after) planarization. Note that these instances have many degree-2 vertices for detailed representation of road curvatures. This explains the relatively large size of the instances (e. g., the graph of Germany contains more vertices than our main benchmark instance).

Table 5 shows results for 1 000 queries on each instance for a medium range of 60 minutes and source vertices picked uniformly at random. (We omit longer ranges, where particularly for Switzerland major parts of the graph become reachable in every query.) The larger graph sizes are reflected in average solution size and running time. However, compared to previous experiments, the decrease in the number of segments of our sophisticated approaches is even more significant. For Switzerland, the average result size drops by a factor of more than 100 when using any of the approaches based on minimum-link paths. This is explained by both the large number of degree-2 vertices and the fact that the Switzerland instance contains many large faces (representing lakes or mountains). This allows the heuristic to produce many long segments; see also the example shown in Figure 22. Lower bounds on the optimal complexity yield similar error bounds as before (at most 6 % and 7 % for `RP-SI` on Switzerland and Germany, respectively). For longer ranges (not reported in the table), running times increase because due to the graph size, border region extraction becomes rather expensive. However, even for the Germany instance running times are always below 200 milliseconds on average.

Computation of the Reachable Subgraph. In all scenarios considered above, we ignored the computational effort to obtain the reachable subgraph, i. e., Step 1 of the general approach in Section 2. We proposed a variant of Dijkstra's algorithm to achieve this in Section 3. However, it would become the bottleneck of the overall running time in all scenarios, as it takes several seconds on average for long ranges. Similar observations were made for variants of Dijkstra's algorithm running on large-scale road networks in the context of speedup techniques for shortest-path algorithms [3]. Recently, it was shown that algorithms producing output similar to Step 1 can be made practical using preprocessing-based techniques [6]. As a result, running times below 50 milliseconds can be achieved for this step (even less when parallelized). Since these techniques can easily be adapted to produce the actual output of Step 1 required by our algorithm, our approaches enable the visualization of isocontours in road networks in less than 100 milliseconds in total on our benchmark instance. Thereby, we enable interactive applications even on road networks of continental scale.



(a) (b)

Figure 22: The area reachable within five hours from KIT in Karlsruhe, Germany (black disk). (a) Result of `RP-RC` from Section 5.1 with 295 015 segments. (b) Result of `RP-CU` from Section 5.3 with 6 606 segments.

Case Study. To briefly discuss visualization quality, we present some examples of isocontour visualization on our main instance. Figure 22 shows an isochrone for a range of five hours, using the simple approach `RP-RC` and one of our more sophisticated methods, `RP-CU`. Comparing the visualization of both approaches, we see that major parts of the isocontours look very similar. However, the number of segments in the result of `RP-RC` shown in Figure 22a is significantly larger, making the isocontour appear more cluttered. At certain points, though, the isocontours differ: The result of `RP-CU` contains a long segment at the top left border (covering the coast of Belgium and the Netherlands), while the result of `RP-RC` stays closer to the shore. This difference is explained by the fact that the sea corresponds to a single huge face in the planar embedding, allowing the minimum link path to cover long distances with a single segment. Similar observations can be made at the southern boundary of the reachable area, where many mountains and lakes correspond to huge faces in the embedding. Besides long segments, huge faces can produce other undesirable artifacts, such as spikes. To prevent this, one could add further constraints, e. g., forcing the range polygon to stay close to the reachable boundary. Such constraints can be implemented by slightly shrinking faces (during preprocessing) if their area exceeds a certain boundary. These shrunk dummy faces could then be added to the unreachable boundary.

Figure 23 compares results of all four approaches in a small-scale example. All three algorithms based on minimum-link paths produce very similar results, though the resulting polygon contains more holes when using `RP-TS`. Note that the result of `RP-SI` contains self-intersections on its left side.

7 Conclusion

In this work, we proposed several approaches for computing isocontours in large-scale road networks. Given the subgraph that is reachable within the given resource limit, this problem

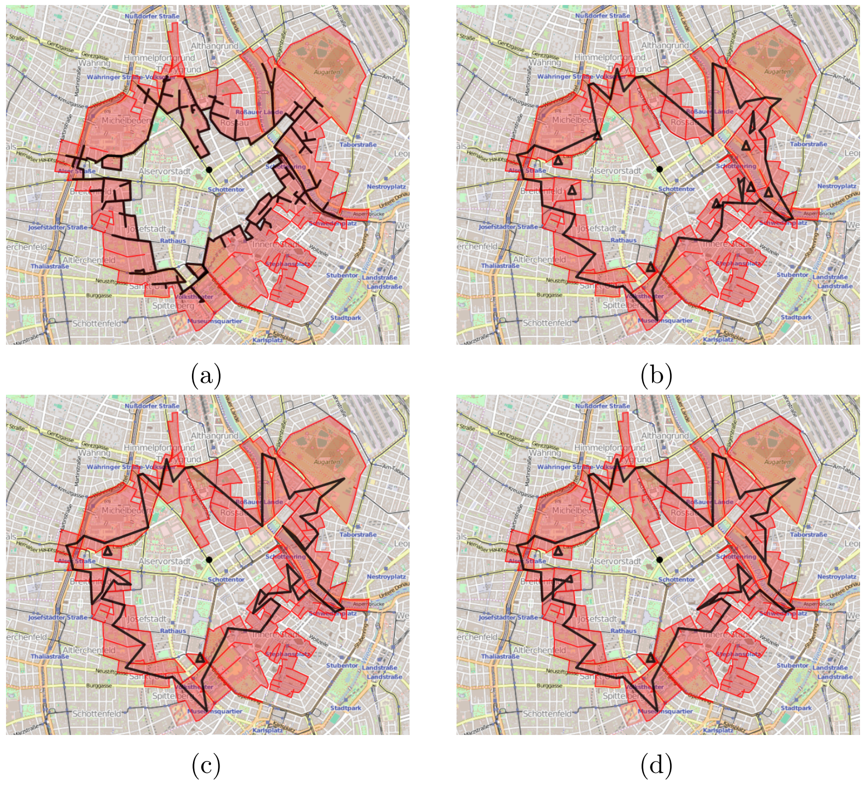


Figure 23: Small-scale example showing different representations of a short-range isocontour from the black disk in the city of Vienna, Austria. Polygons represent isocontours, the corresponding border regions are shaded red. We show the results of (a) `RP-RC`, (b) `RP-TS`, (c) `RP-CU`, and (d) `RP-SI`, respectively.

boils down to computing a geometric representation of this subgraph. We introduced range polygons, following the three major objectives of exact results (reachable and unreachable parts are correctly separated), low complexity (range polygons consist of few segments), and practical performance. We presented three novel algorithms to compute near-optimal solutions in (almost) linear time. Their key ingredient is a new linear-time algorithm for minimum-link paths in simple polygons, making it the first practical approach to a problem well-studied in theory [37, 44, 45, 50]. Our experimental evaluation reveals that all approaches are fast enough for interactive applications on inputs of continental scale.

Regarding future work, we discuss interesting open issues and possible further improvements. First, our techniques exploit the fact that the reachable boundary of a border region is always connected, i. e., $|R| = 1$. This might not be the case in related scenarios, such as multi-source isocontours or in multimodal networks, where one can disembark from public transit vehicles only at certain points [24]. Thus, it would be interesting to know whether our approaches can be extended to the case of $|R| > 1$. Moreover, Gamper et al. [24] consider the extent to which reachable edges can be passed in their definition of isochrones. This is relevant particularly for short ranges, or if the graph consists of edges with very long distance. Hence, we could modify our approaches to handle this slightly different model. For aesthetic reasons, one could aim at avoiding long straight segments or spikes in the range polygon, which are likely to occur in faces encompassing large areas corresponding to, e. g., big lakes or mountains. As discussed in Section 6, such constraints could be integrated by adding (during preprocessing) artificial boundaries to faces whose area exceeds a certain threshold. On the other hand, one could also aim at further line simplification, at the cost of inexact results. However, such methods should avoid intersections between different components of the range polygon (i. e., maintain its topology), and error measures should consider the graph-based distance from the source to parts of the network that are incorrectly classified by the range polygon (vertices that are close with respect to Euclidean distance may in fact have a long distance in the graph). To this end, reusing ideas from known line simplification algorithms [13, 15, 22] could be a promising approach. One could also aim at exact approaches based on line simplification, by implementing additional constraints. Finally, another interesting open problem is the consideration of continuous range visualization for a moving vehicle. Instead of computing the isocontours from scratch, one could try to reuse previously computed information.

Acknowledgements. We would like to thank Roman Prutkin for interesting discussions and our anonymous reviewers for many helpful comments and suggestions.

References

[1] Nina Amenta and Marshall Bern. Surface Reconstruction by Voronoi Filtering. *Discrete & Computational Geometry*, 22(4):481–504, 1999.

[2] Ivan J. Balaban. An Optimal Algorithm for Finding Segments Intersections. In *Proceedings of the 11th Annual Symposium on Computational Geometry (SoCG'95)*, pages 211–219. ACM, 1995.

[3] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

[4] Veronika Bauer, Johann Gamper, Roberto Loperfido, Sylvia Profanter, Stefan Putzer, and Igor Timko. Computing Isochrones in Multi-Modal, Schedule-Based Transport Networks. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'08)*, pages 78:1–78:2. ACM, 2008.

[5] Moritz Baum, Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner. Scalable Exact Visualization of Isocontours in Road Networks via Minimum-Link Paths. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA'16)*, volume 57 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 44:1–44:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[6] Moritz Baum, Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner. Fast Exact Computation of Isochrones in Road Networks. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*, volume 9685 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2016.

[7] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-Optimal Routes for Electric Vehicles. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'13)*, pages 54–63. ACM, 2013.

[8] John L. Bentley and Thomas A. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

[9] Francisc Bungiu, Michael Hemmer, John E. Hershberger, Kan Huang, and Alexander Kröller. Efficient Computation of Visibility Polygons. In *Proceedings of the 30th European Workshop on Computational Geometry (EuroCG'14)*, 2014.

[10] Alex Bykat. Convex Hull of a Finite Set of Points in Two Dimensions. *Information Processing Letters*, 7(6):296–298, 1978.

[11] Peter Conradi, Philipp Bouteiller, and Sascha Hanßen. Dynamic Cruising Range Prediction for Electric Vehicles. In *Advanced Microsystems for Automotive Applications 2011: Smart Systems for Electric, Safe and Networked Mobility*, VDI Buch, pages 269–277. Springer, 2011.

[12] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, third edition, 2008.

[13] Mark de Berg, Marc van Kreveld, and Stefan Schirra. Topologically Correct Subdivision Simplification Using the Bandwidth Criterion. *Cartography and Geographic Information Systems*, 25(4):243–257, 1998.

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

[15] 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. *The Canadian Cartographer*, 10(2):112–122, 1973.

[16] Matt Duckham, Lars Kulik, Mike Worboys, and Antony Galton. Efficient Generation of Simple Polygons for Characterizing the Shape of a Set of Points in the Plane. *Pattern Recognition*, 41(10):3224–3236, 2008.

[17] Herbert Edelsbrunner, David G. Kirkpatrick, and Raimund Seidel. On the Shape of a Set of Points in the Plane. *IEEE Transactions on Information Theory*, 29(4):551–559, 1983.

[18] Alexandros Efentakis, Sotiris Brakatsoulas, Nikos Grivas, Giorgos Lamprianidis, Kostas Patroumpas, and Dieter Pfoser. Towards a Flexible and Scalable Fleet Management

Service. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWTCS'13)*, pages 79–84. ACM, 2013.

[19] Alexandros Efentakis, Nikos Grivas, George Lamprianidis, Georg Magenschab, and Dieter Pfoser. Isochrones, Traffic and DEMOgraphics. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'13)*, pages 548–551. ACM, 2013.

[20] Alexandros Efentakis and Dieter Pfoser. GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, volume 8737 of *Lecture Notes in Computer Science*, pages 358–370. Springer, 2014.

[21] David Eppstein, Michael T. Goodrich, and Darren Strash. Linear-Time Algorithms for Geometric Graphs with Sublinearly Many Edge Crossings. *SIAM Journal on Computing*, 39(8):3814–3829, 2010.

[22] Stefan Funke, Thomas Mendel, Alexander Miller, Sabine Storandt, and Maria Wiebe. Map Simplification with Topology Constraints: Exactly and in Practice. In *Proceedings of the 19th Meeting on Algorithm Engineering & Experiments (ALENEX'16)*, pages 185–196. SIAM, 2017.

[23] Anka Gajentaan and Mark H. Overmars. On a Class of $O(n^2)$ Problems in Computational Geometry. *Computational Geometry*, 5(3):165–185, 1995.

[24] Johann Gamper, Michael Böhlen, Willi Cometti, and Markus Innerebner. Defining Isochrones in Multimodal Spatial Networks. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM'11)*, pages 2381–2384. ACM, 2011.

[25] Johann Gamper, Michael Böhlen, and Markus Innerebner. Scalable Computation of Isochrones with Network Expiration. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management (SSDBM'12)*, volume 7338 of *Lecture Notes in Computer Science*, pages 526–543. Springer, 2012.

[26] Sorabh Gandhi, John E. Hershberger, and Subhash Suri. Approximate Isocontours and Spatial Summaries for Sensor Networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN'07)*, pages 400–409. ACM, 2007.

[27] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[28] Ronald L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1(4):132–133, 1972.

[29] Stefan Grubwinkler, Tobias Brunner, and Markus Lienkamp. Range Prediction for EVs via Crowd-Sourcing. In *Proceedings of the 10th IEEE International Vehicle Power and Propulsion Conference (VPPC'14)*. IEEE, 2014.

[30] Leonidas J. Guibas and John E. Hershberger. Optimal Shortest Path Queries in a Simple Polygon. *Journal of Computer and System Sciences*, 39(2):126–152, 1989.

[31] Leonidas J. Guibas, John E. Hershberger, Daniel Leven, Micha Sharir, and Robert E. Tarjan. Linear-Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons. *Algorithmica*, 2(1):209–233, 1987.

[32] Leonidas J. Guibas, John E. Hershberger, Joseph S. B. Mitchell, and J. S. Snoeyink. Approximating Polygons and Subdivisions with Minimum-Link Paths. *International Journal of Computational Geometry & Applications*, 3(4):383–415, 1993.

[33] Stefan Hausberger, Martin Rexeis, Michael Zallinger, and Raphael Luz. Emission Factors from the Model PHEM for the HBEFA Version 3. Technical report I-20/2009, University of Technology, Graz, 2009.

[34] Hiroshi Imai and Masao Iri. An Optimal Algorithm for Approximating a Piecewise Linear Function. *Journal of Information Processing*, 9(3):159–162, 1987.

[35] Markus Innerebner, Michael Böhlen, and Johann Gamper. ISOGA: A System for Geographical Reachability Analysis. In *Proceedings of the 12th International Conference on Web and Wireless Geographical Information Systems (W2GIS'13)*, volume 7820 of *Lecture Notes in Computer Science*, pages 180–189. Springer, 2013.

[36] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13, 1977.

[37] Irina Kostitsyna, Maarten Löffler, Valentin Polishchuk, and Frank Staals. On the Complexity of Minimum-Link Path Problems. In *Proceedings of the 32nd Annual Symposium on Computational Geometry (SoCG'16)*, volume 51 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49:1–49:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[38] Lawrence T. Kou, George Markowsky, and Leonard C. Berman. A Fast Algorithm for Steiner Trees. *Acta Informatica*, 15(2):141–145, 1981.

[39] Daniel Krajzewicz and Dirk Heinrichs. UrMo Accessibility Computer – A Tool for Computing Contour Accessibility Measures. In *Proceedings of the 8th International Conference on Advances in System Simulation (SIMUL'16)*, pages 56–60. IARIA, 2016.

[40] Nikolaus Krismer, Doris Silbernagl, Günther Specht, and Johann Gamper. Computing Isochrones in Multimodal Spatial Networks Using Tile Regions. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM'17)*, pages 33:1–33:6. ACM, 2017.

[41] Nikolaus Krismer, Günter Specht, and Johann Gamper. Incremental Calculation of Isochrones Regarding Duration. In *Proceedings of the 26th GI-Workshop on Foundations of Databases (GvDB'14)*, volume 1313 of *CEUR Workshop Proceedings*, pages 41–46. CEUR-WS.org, 2014.

[42] Sarunas Marciuska and Johann Gamper. Determining Objects within Isochrones in Spatial Network Databases. In *Proceedings of the 14th East European Conference on Advances in Databases and Information Systems (ADBIS'10)*, volume 6295 of *Lecture Notes in Computer Science*, pages 392–405. Springer, 2010.

[43] Kurt Mehlhorn. A Faster Approximation Algorithm for the Steiner Problem in Graphs. *Information Processing Letters*, 27(3):125–128, 1988.

[44] Joseph S. B. Mitchell, Valentin Polishchuk, and Mikko Sysikaski. Minimum-Link Paths Revisited. *Computational Geometry*, 47(6):651–667, 2014.

[45] Joseph S. B. Mitchell, Günter Rote, and Gerhard Woeginger. Minimum-Link Paths Among Obstacles in the Plane. *Algorithmica*, 8(1):431–459, 1992.

[46] Peter Ondrúška and Ingmar Posner. Probabilistic Attainability Maps: Efficiently Predicting Driver-Specific Electric Vehicle Range. In *Proceedings of the 10th IEEE Intelligent Vehicles Symposium (IV'14)*, pages 1169–1174. IEEE, 2014.

[47] Peter Ondrúška and Ingmar Posner. The Route Not Taken: Driver-Centric Estimation of Electric Vehicle Range. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*, pages 413–420. AAAI Press, 2014.

[48] David O'Sullivan, Alastair Morrison, and John Shearer. Using Desktop GIS for the Investigation of Accessibility by Public Transport: An Isochrone Approach. *International Journal of Geographical Information Science*, 14(1):85–104, 2000.

[49] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Conference on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[50] Subhash Suri. A Linear Time Algorithm for Minimum Link Paths Inside a Simple Polygon. *Computer Vision, Graphics, and Image Processing*, 35(1):99–110, 1986.

[51] Robert E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[52] Cao An Wang. Finding Minimal Nested Polygons. *BIT Numerical Mathematics*, 31(2):230–236, 1991.

[53] Peter Widmayer. On Approximation Algorithms for Steiner's Problem in Graphs. In *Proceedings of the 12th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'86)*, volume 246 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 1987.

[54] Ying Fung Wu, Peter Widmayer, and Chak Kuen Wong Wong. A Faster Approximation Algorithm for the Steiner Problem in Graphs. *Acta Informatica*, 23(2):223–229, 1986.