Towards Realistic Pedestrian Route Planning*

Simeon Andreev[1], Julian Dibbelt[1], Martin Nöllenburg[1], Thomas Pajor[2], and Dorothea Wagner[1]

1 **Karlsruhe Institute of Technology, Germany**
`simeon.andreev@student.kit.edu`,
`{dibbelt,noellenburg,dorothea.wagner}@kit.edu`

2 **Microsoft Research, USA**
`tpajor@microsoft.com`

Abstract

Pedestrian routing has its specific set of challenges, which are often neglected by state-of-the-art route planners. For instance, the lack of detailed sidewalk data and the inability to traverse plazas and parks in a natural way often leads to unappealing and suboptimal routes. In this work, we first propose to augment the network by generating sidewalks based on the street geometry and adding edges for routing over plazas and squares. Using this and further information, our query algorithm seamlessly handles node-to-node queries and queries whose origin or destination is an arbitrary location on a plaza or inside a park. Our experiments show that we are able to compute appealing pedestrian routes at negligible overhead over standard routing algorithms.

1998 ACM Subject Classification G.2.2 Graph Theory, G.2.3 Applications, H.2.8 Database Applications, I.3.5 Computational Geometry and Object Modeling

Keywords and phrases pedestrian routing, realistic model, shortest paths, speed-up technique

1 Introduction

The computation of routes in street networks has received tremendous attention from the research community over the past decade, and for many applications efficient algorithms now exist; see [4] for a recent survey. The bulk of work, however, focuses on computing driving directions for cars. Other scenarios, such as computing routes for pedestrians, have been neglected or simply dismissed as a trivial matter of applying a different cost function.

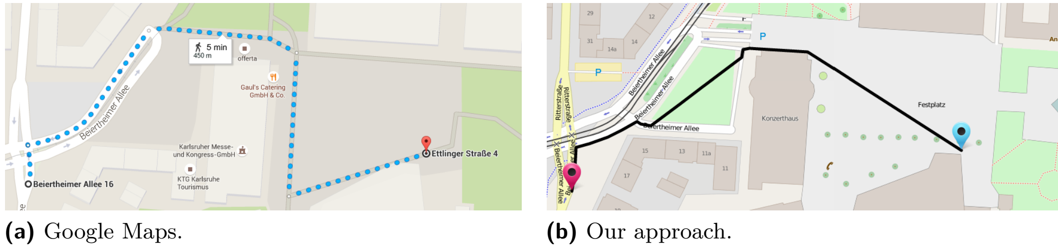
We argue that this naïve approach may lead to unnatural and suboptimal solutions. In fact, pedestrians utilize the street network quite differently from cars, which is often not captured by traditional approaches. For example to save distance, pedestrians are free to deviate from the streets, using the walkable area of public open spaces such as plazas and parks. On the other hand, crossing large avenues can be expensive (due to traffic), and it may be faster and safer to walk a small detour in order to use a nearby bridge or underpass.

In this work, we address the unique challenges that come with computing pedestrian routes. In order to obtain as realistic routes as possible, we propose to first augment the underlying street network model, and then to apply a tailored routing algorithm on top of it. After setting some basic definitions (Section 2), we propose geometric approaches for automatically adding sidewalks, calculating realistic crossing penalties for major roads, and preprocessing plazas and parks in order to traverse them in a natural way (Section 3). Our integrated routing algorithm seamlessly handles node-to-node queries and queries whose

* Partially supported by DFG grant WA654/23-1, EU grant 609026 (MOVESMART), and Google Focused Research Award. Most of the work done while Thomas Pajor was at Karlsruhe Institute of Technology.

© Simeon Andreev, Julian Dibbelt, Martin Nöllenburg, Thomas Pajor, and Dorothea Wagner;
licensed under Creative Commons License CC-BY
Conference/workshop/symposium title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–15

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Google Maps. **(b)** Our approach.

Figure 1 In contrast to current approaches, our route in this example makes use of sidewalks (avoiding unnecessary street crossings), begins on a plaza and traverses it in a natural way.

origin or destination is an arbitrary geographic location inside a plaza or park (Section 4). To efficiently support long-range queries, we also adapt the *Customizable Route Planning* (CRP) algorithm [9]—a well-known speed-up technique for computing driving directions in road networks—to our scenario. We evaluate our approach on OpenStreetMap data of Berlin and the state of Baden-Württemberg, Germany (Section 5). Our algorithm runs in the order of milliseconds, which is practical for interactive applications. We observe that we are able to compute pedestrian routes that are much more appealing than those by state-of-the-art route planners, such as shown in Figure 1. Section 5.2 shows further examples and an illustrated comparison of our method with three popular external services.

Related Work. We touch several subjects: sidewalk generation, traversal of open areas, and graph-based routing. See [12] for an overview and assessment of different sidewalk generation approaches. Many works consider extraction of street networks from satellite images, e. g., [15, 18]. While this approach is promising for roads, extracting sidewalks is problematic due to poor image resolution and occlusion (e. g., by trees). Moreover, satellite imagery is not as easily available as street data. In contrast, a street network analysis technique [17] generates sidewalk information directly from street layouts, but it does not handle multiple lanes and streets that are close to each other very well. An alternative technique [3] leverages building layouts to generate sidewalks, however, not all streets that have sidewalks are also adjacent to a building, resulting in incomplete output.

Traversing open areas is a classical problem in robotics and computational geometry, and numerous works exist on the subject [20]. Cell decomposition [13] yields paths that are offset from the obstacles and area boundary, and [14] combines several techniques—including Voronoi diagrams—to obtain robust collision-free robot motion paths. Visibility graphs [1] are specifically important to us, since they represent geometric shortest paths.

Given source and target nodes in a graph, Dijkstra's algorithm [11] computes shortest paths between them. A plethora [4] of work deals with accelerating Dijkstra's algorithm by using an additional offline preprocessing stage. In our work, we adapt the *Customizable Route Planning* (CRP) algorithm [9], which offers an excellent tradeoff between query performance and preprocessing effort. In essence, its preprocessing uses a nested multilevel partition of the graph to compute shortcuts between the boundary vertices in each cell. Traversing these shortcuts then enables the query to skip over large parts of the graph.

2 Preliminaries

We model the street network as a *undirected graph* $G = (V, E)$ with a set V of *nodes* and a set $E \subseteq \binom{V}{2}$ of *edges*. A node that is incident to exactly two edges is called a *2-node*.

For a specific subset of edges $E' \subseteq E$ the *induced graph* $G[E'] = (V', E')$ contains E' and the nodes V', which are incident to the edges of E'. An *s–t path* in G is a node sequence $P_{s,t} = (s = v_1, \dots, v_k = t)$, with each $e_i = \{v_i, v_{i+1}\}$ contained in E. A graph G is *planar* if a crossing-free drawing of G in the plane exists. A specific *embedding* of G maps each node to a coordinate in the plane. The embedding of G subdivides the plane into disjoint polygonal regions called *faces* bounded by the edges of G. Note that in our street networks each node $v \in V$ corresponds to a physical location. Likewise, each edge $e \in E$ represents a street segment. The *cost* of e is given by $c\colon E \to \mathbb{R}_+$, where $c(e)$ is the time (in seconds) a pedestrian requires to traverse e. This value may, e.g., depend on the street category and the segment's physical length. For source and target nodes s and t, Dijkstra's algorithm [11] computes a *shortest s–t path* $P_{s,t}$, i.e., an s–t path whose cost $\sum_{i=1}^{k-1} c(e_i)$ is minimal.

Besides the street network, we consider the *walkable area* of public open spaces such as *plazas* and *parks*. We represent them by polygons, as follows. A *(simple) polygon* $Q \subset \mathbb{R}^2$ is defined as the interior of a sequence of *vertices* $Q = (p_1, \dots, p_n), p_i \in \mathbb{R}^2$ sorted clockwise and connected by non-self-intersecting *segments* $\overline{p_1p_2}, \overline{p_2p_3}, \dots, \overline{p_n, p_1}$. (We distinguish the *nodes* of a graph from the *vertices* of a polygon.) A *polygon with holes* Q is defined by a *boundary cycle* b_Q and *holes* $h_Q^1, \dots, h_Q^k$ in the interior of b_Q, where b_Q and h_Q^i again define simple polygons and their vertices are the vertices of Q. The *interior* of Q is $b_Q \setminus (\cup_i h_Q^i)$ and a point o or a segment s is *within* Q if o or s lie within this interior. The *visibility graph* $\mathrm{VG}(Q)$ of a polygon with holes Q is a geometric graph that consists of all vertices $p_1, \dots, p_n$ of Q and all segments $\overline{p_ip_j}$ which lie within Q. The *visibility polygon* $\mathrm{VP}(p, Q)$ of a point $p \in Q$ with respect to the containing polygon Q is the region within Q that is *visible* from p, i. e., for each $q \in \mathrm{VP}(p, Q)$ the segment $\overline{pq} \subseteq Q$. With $|Q|$ we denote the number of vertices of Q.

For many geometric computations, we use functionality of the computational geometry library CGAL [22], in particular for computing line segment intersections, polygon unions and differences, visibility graphs and polygons, range queries, and point-in-polygon queries; see [8] for descriptions of the algorithms. Furthermore, we implement a custom sweep line algorithm, which, given a set of disjoint polygons and a set of query points, determines for each point the polygon that contains it (if any); see Appendix A for details. To apply these geometric algorithms to our street data, we map geographic coordinates to points in $\mathbb{R}^2$ using the Mercator projection. We use Euclidean distances $||p - q||$ between points p and q.

3 Augmented Graph Model for Pedestrian Routing

We consider three key aspects where pedestrian routes differ from those of vehicles: (a) sidewalks are preferred over streets, if present; (b) plazas can be traversed freely; (c) in parks pedestrians may walk freely on the lawn, but park walkways are preferred. In this section, we present algorithms that process the street network in order to accommodate these differences. (We then discuss queries in Section 4.) Most of this preprocessing is independent of the edge costs in the network, hence, new costs can be integrated with little effort.

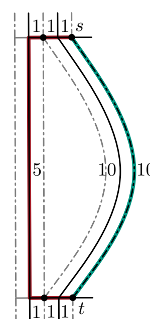
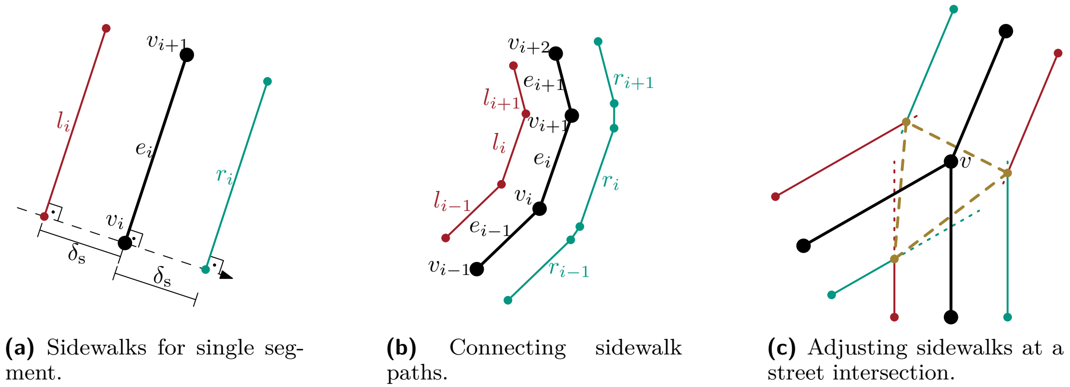


Figure 2 Shortest path when using streets (left) or sidewalks (right).

3.1 Sidewalks and Street Crossings

Unlike features, such as street direction, turn restrictions and separation into lanes, sidewalk data is often lacking (or inconsistently modeled) in popular street databases, such as OpenStreetMap (`openstreetmap.org`). As a result, state-of-the art pedestrian route planners mostly use the streets themselves and not their sidewalks. However, this may lead to unnecessary street crossings,



(a) Sidewalks for single segment.

(b) Connecting sidewalk paths.

(c) Adjusting sidewalks at a street intersection.

Figure 3 Generating sidewalks. Regular street segments are replaced by two sidewalk edges (a). Subsequent pairs of sidewalk edges are then connected along each 2-node path (b). Finally, the resulting sidewalks are adjusted at street intersections (dotted parts removed), and crossing edges (dashed) are added (c).

which can either be costly (due to traffic lights), or even be impossible. In contrast, when sidewalks are considered properly, a seeming detour may actually be the shorter path, see Figure 2. We therefore propose to replace some streets (as given by the input) with automatically generated sidewalks. We distinguishing between three street types: *Highways* represent streets that are inaccessible for pedestrians, hence, they have no sidewalks; *regular streets*, such as city streets, have sidewalks; and *walkways* are footpaths and streets small enough to require no sidewalks.

Street Polygons. Naïvely, one could add sidewalks to the left and to the right of every regular street in the input [17]. Unfortunately, this results in sidewalks being placed in the middle of multi-lane streets or in median strips, which is clearly unwanted. We therefore propose to avoid areas enclosed by regular or highway streets that are too small or thin to hold sidewalks.

To achieve this, we first compute a set $\mathcal{S}$ of *street polygons*, representing such areas without sidewalks. Consider the embedded graph G_{hr} induced by the set of highway and regular street edges. We obtain the planarization G'_{hr} of G_{hr} using a standard sweep-line algorithm for line segment intersections [8]. Let f be a face in G'_{hr} and let a_f and p_f denote its area and perimeter, respectively. Then f is considered a street polygon, if $a_f/p_f \leq \beta_{\mathrm{r}}$ (too thin) or $a_f \leq \beta_{\mathrm{a}}$ (too small) for suitably chosen thresholds $\beta_{\mathrm{r}}, \beta_{\mathrm{a}}$.

Sidewalks. Our goal is to place sidewalks to the left and to the right of each street edge at some offset, unless they would be placed inside a street polygon. They should also follow curves and handle street intersections correctly, see Figure 3. To do so, we consider the embedded graph G_{r}, induced by the regular street edges. Recall that in G_{r} street intersections are modeled by nodes v of degree $\deg(v) \geq 3$, while the street's curvature is modeled as paths of 2-nodes. For each maximal 2-node path $(v_1, \ldots, v_k)$ and its adjacent intersection nodes v_0 and v_{k+1} (where we treat dead ends as intersection nodes, too), we consider the edge sequence $(e_0, \ldots, e_k)$, where $e_i = \{v_i, v_{i+1}\}$. For each edge e_i, we create two sidewalk edges l_i and r_i, and offset them (from e_i) by a distance δ_{s}; see Figure 3a. In order to form correct paths along bends, these edges need to be trimmed or linked via auxiliary edges, depending on the bend angles; see Figure 3b.

At each street intersection $v \in G_{\mathrm{r}}$ with $\deg(v) \geq 3$, we sort the incident edges in cyclic order. This order yields adjacent sidewalks, which we again trim at their respective intersection points or link by an auxiliary edge; see Figure 3c. For each street edge e incident to v, we also add an edge between the two sidewalks at v associated with e, which allows to cross e at v; see again Figure 3c.

Next, we remove all sidewalk portions contained in street polygons of $\mathcal{S}$. Using a standard line segment intersection algorithm [8], we first subdivide sidewalks at the boundaries of street polygons. Then, we use our point-in-polygon algorithm (see Appendix A) to remove all sidewalk segments with both endpoints inside a polygon of $\mathcal{S}$. This results in (at most) two sidewalks per street, as opposed to two sidewalks per lane.

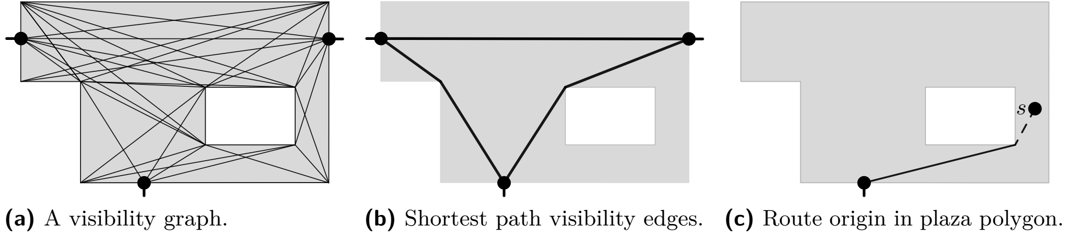
Finally, we assemble the *routing graph* G induced by sidewalk, crossing and walkway edges (but not highway and regular street edges). For connectivity, we add nodes at the intersections of sidewalks with walkways, subdividing the intersecting edges, again by running a line segment intersection algorithm [8].

Crossing Penalties. We may further utilize the street polygons $\mathcal{S}$ in order to penalize certain street crossings where waiting times can be expected. As the area covered by parallel street lanes is represented in $\mathcal{S}$, an edge e of G which passes through a multi-lane street also has a portion within $\mathcal{S}$, and we may penalize this portion in our cost function. We use two types of penalties. The "one-time" penalty α_{e} models a general waiting time, either for a pedestrian light or for traffic to clear. We add α_{e} to the cost of each edge that *enters* $\mathcal{S}$. More precisely, an edge $e = \{u, v\}$ in G enters $\mathcal{S}$ if u is outside $\mathcal{S}$ and the segment of e has common points with $\mathcal{S}$. The second penalty, denoted α_{w}, is a *penalty per unit of length* spent within $\mathcal{S}$. It reflects that wider streets generally require longer waiting times to cross. We find the edge portions of G within $\mathcal{S}$ while we remove sidewalks within $\mathcal{S}$. We use our sweep line algorithm (cf. Appendix A) to find edges starting outside $\mathcal{S}$. Such edges with portions within $\mathcal{S}$ also enter the street polygons.

3.2 Plazas

Pedestrians may traverse plazas freely. However, somewhat surprisingly, most state-of-the-art pedestrian navigation services route around such walkable areas, not through them. We propose to utilize visibility graphs to remedy this shortcoming. We assume that the street network database provides traversable plazas as a set $\mathcal{P}$ of *plaza polygons*, possibly with holes due to obstacles. Given $\mathcal{P}$ and the previously obtained routing graph G, we compute the *entry nodes* of each plaza: These lie on the intersection of a plaza polygon's boundary and the routing graph and are obtained by a line segment intersection algorithm [8]. We add each entry node both to the plaza polygon (as a vertex) and to the routing graph. For each polygon $Q \in \mathcal{P}$, we then compute the *visibility graph* $\mathrm{VG}(Q)$. If Q has no holes, we require quadratic time [16, 22], otherwise cubic time. (Since we encounter only very few polygons with holes in practice, we did not implement a more efficient algorithm, such as [1].) Let $E_{\mathrm{vis}}(Q)$ be the visibility edges of $\mathrm{VG}(Q)$, and $E_{\mathrm{vis}} = \cup_{Q \in \mathcal{P}} E_{\mathrm{vis}}(Q)$. We add E_{vis} as further pedestrian edges to the routing graph G.

Since the number of visibility edges $E_{\mathrm{vis}}(Q)$ of a plaza polygon $Q \in \mathcal{P}$ is generally quite high (see Figure 4a), routing through plazas can become expensive. We therefore mark the subset $E_{\mathrm{vis}}^{\mathrm{sp}} \subset E_{\mathrm{vis}}$ of visibility edges that are part of shortest paths between any pair of entry nodes (the query may then ignore unmarked edges); see Figure 4b. We do so by running Dijkstra's algorithm from each entry node, only relaxing visibility edges of the node's plaza. Note that $E_{\mathrm{vis}}^{\mathrm{sp}}$ suffices to route *across* plazas, but queries that begin or end on a plaza may



(a) A visibility graph. **(b)** Shortest path visibility edges. **(c)** Route origin in plaza polygon.

Figure 4 Small polygon Q with a hole and all visibility edges (a) and the ones that are also on shortest paths (b). Routing from within Q requires all visibility edges (c).

still require all edges in E_{vis}; see Figure 4c and Section 4. Also note that computing $E^{\text{sp}}_{\text{vis}}$ requires knowledge of the routing cost function (all other preprocessing does not). However, since the necessary shortest path queries are restricted to each plaza and the number of entry nodes is typically small, this step is not costly compared to the total preprocessing effort.

3.3 Parks

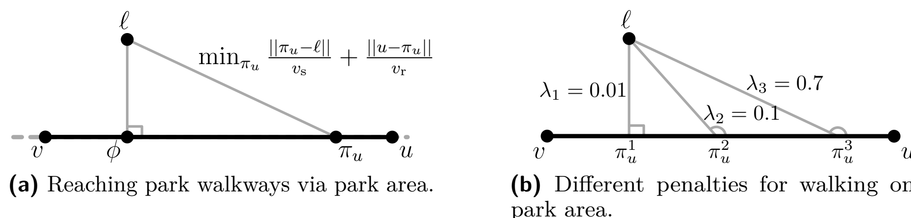
Unlike plazas, parks have designated walkways, which we favor by routing on walkable park areas (such as lawn) only at the beginning or end of a route. In order to quickly locate nearby walkways during queries, we precompute the faces of a park induced by its walkways.

Similarly to plazas, we assume that the walkable area of parks is given as the set $\mathcal{L}$ of *park polygons* (possibly with holes) by the street network database. We compute the entry nodes the same way we do for plazas. We then use our algorithm from Appendix A to compute the set E_L of edges in G contained in each $L \in \mathcal{L}$ (in a single sweep). Thus, $G_L = G[E_L]$ contains exactly the park walkways within L. We add the boundary of L to G_L (as nodes and edges) and planarize G_L. We define the set of *park faces* F_L to be the faces of G_L, and $\mathcal{F} = \bigcup_{L \in \mathcal{L}} F_L$. During queries, we will use $\mathcal{F}$ for locating park walkways and routing to/from them (see Section 4).

4 Computing Routes

We now discuss how we leverage our model from Section 3 to compute realistic pedestrian routes. We are generally interested in queries between arbitrary locations ℓ_o (origin) and ℓ_d (destination). Usually, one handles such *location-to-location queries* by first mapping the locations to their nearest nodes (or edges) of the network, and then invoking a shortest path algorithm between those. However, for locations inside plazas and parks this method would result in inaccurate routes. Instead, we propose the following approach. First, we test whether $\ell \in \{\ell_o, \ell_d\}$ is located inside a plaza or a park. In either case, we first connect ℓ to G with sensible edges and then run Dijkstra's algorithm between ℓ_o and ℓ_d on this augmented graph. If neither is the case, we just find the nearest nodes in G using a k-d tree [5], as in the classic scenario. We discuss more details next.

Plazas. To test whether the origin or destination location ℓ is on a plaza, we simply perform a point-in-polygon test [8]. Now, assume that $Q \in \mathcal{P}$ is the polygon, which contains ℓ. We compute the visibility polygon $\text{VP}(\ell, Q)$ of ℓ with respect to Q by applying the recent algorithm of Bungiu et al. [7]. We also use our sweep line algorithm from Appendix A



(a) Reaching park walkways via park area.

(b) Different penalties for walking on park area.

Figure 5 Using the park area and walkway. We minimize the walking time on the park area plus that on the walkway (left). The manner in which the park area is utilized varies with v_s (right).

to obtain the nodes V_Q^ℓ in $\mathrm{VG}(Q) \subset G$ that are located within $\mathrm{VP}(\ell, Q)$. We then simply connect ℓ to each node $p \in V_Q^\ell$ by adding edges $\{\ell, p\}$ to the graph G.

Recall from Section 3.2 that to route across plazas, the visibility edges in $E_{\mathrm{vis}}^{\mathrm{sp}}$ suffice. Hence, we ignore edges $e \in E_{\mathrm{vis}} \setminus E_{\mathrm{vis}}^{\mathrm{sp}}$ during the query, unless $e \in \mathrm{VG}(Q)$ for the polygon Q containing ℓ, in which case it is required for correctness.

Parks. For the case that ℓ is contained in a park, we first obtain the enclosing park face f (similarly to the plaza case). We now consider two different walking speeds: the regular walking speed v_r, and another (slower) one v_s for park faces (e. g., lawn). We set $\lambda = v_s/v_r$ (with $\lambda \in (0, 1]$) as a query time parameter; values $\lambda < 1$ penalize walking on the lawn, with smaller λ values leading to higher penalization.

Taking this into account, our goal is to connect ℓ to the walkways of f, such that the total walking duration is minimized. We thereby compute the optimal path toward each edge $e = \{u, v\} \in f$ separately, as follows. Consider a point π_u on e. To reach u from ℓ via π_u, one requires total walking time $w = \frac{||\pi_u - \ell||}{v_s} + \frac{||u - \pi_u||}{v_r}$; see Figure 5a. For a given λ, the minimum walking time w^* is achieved by the *projection point* $\pi_u^* = \phi + \frac{\lambda}{1-\lambda^2} \cdot \frac{||\ell - \phi||}{||u - \phi||} \cdot (u - \phi)$, where ϕ is the perpendicular projection of ℓ on the line through e; see [2] for a derivation of this formula. As seen in Figure 5b, a small value of λ causes a perpendicular projection: walking on the lawn is costly and therefore minimized. A larger value of λ allows for a more direct, target-aimed projection, saving distance but using more of the walkable park area.

We now use the aforementioned formula to compute for each edge $e = \{u, v\} \in f$ the projection points π_u^* and π_v^*. To check whether a segments $s_u = \overline{\pi_u^* \ell}$ is walkable within the park, we test whether the point π_u^* lies within the visibility polygon $\mathrm{VP}(\ell, Q)$. If so, we add the edge $\{\ell, u\}$ with cost $\frac{||\pi_u^* - \ell||}{v_s} + \frac{||u - \pi_u^*||}{v_r}$ to G. Node v is handled analogously.

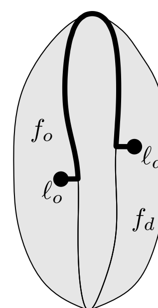


Figure 6 Detour due to long thin face.

Note that since we directly connect the origin ℓ_o and destination ℓ_d to the edges of their enclosing faces, we are unable to route around obstacles in parks. Moreover, we are unable to walk across other park faces (except the ones containing the origin and destination locations). However, this may result in unnatural routes, if origin and destination are in the same park separated by a thin face; see Figure 6. We solve this issue by introducing a radius parameter ϵ, and additionally compute edges to the boundaries of all faces (of the same park) that have vertices within distance ϵ of ℓ. We use range queries [22] to obtain those faces. If, both, origin ℓ_o and destination ℓ_d are in the same park and within distance ϵ, we additionally consider the direct route $\overline{\ell_o \ell_d}$ with cost $\frac{||\ell_d - \ell_o||}{v_s}$ explicitly.

Customizable Route Planning. Typical pedestrian routes are very short, thus, one might argue that Dijkstra's algorithm computes them sufficiently fast. Still, a practical routing engine should be robust against long-distance queries as well. We therefore propose to make use of the *Customizable Route Planning* (CRP) algorithm [9]. It is a state-of-the-art speedup technique, developed for computing driving directions in road networks. CRP employs three phases: The *preprocessing phase* uses a nested multilevel partition to compute (for each level) a metric-independent overlay graph over the boundary nodes of the partition. The *customization phase* takes a cost function as input and computes the actual edge weights of the overlay graph. Finally, the *query phase* runs bidirectional Dijkstra's algorithm, using the overlay graph to the effect of "skipping" over large parts of the network. See [9] for details.

Adapting CRP to our scenario requires little effort. We use the routing graph G for computing both the multilevel partition and the overlay graph. To easily support queries beginning or ending within parks or plazas, we enforce that nodes within the same park or plaza are never put into different cells of the partition. (We do this by running the partitioner on a slightly modified graph, in which we contract all nodes associated with the same park or plaza.) To see why this is correct, recall that the temporary edges added by the query only point to nodes within the park or plaza which contains the origin (or destination) location. By construction these nodes are all part of the same cell (on every level of the partition), therefore, the distances in the overlay graph are unaffected and still correct.

Note that in our CRP query we do not bother ignoring visibility edges in G that are not on shortest paths: They are only present on the bottom level, therefore, the query skips over them automatically in most cases.

5 Experiments

We implemented all algorithms in C++ using g++ 4.8.3 (flag -O3) and CGAL 4.6. We conducted our experiments on a single core of a 4-core Intel Xeon E5-1630v3 CPU clocked at 3.7 GHz with 128 GiB of DDR4-2133 RAM. Our data set was extracted from OpenStreetMap (OSM) on May 15, 2015, and includes roads, plazas and parks.[1] We use two instances: Berlin (BE) and the state of Baden-Württemberg (BW), both in Germany. While BE is an eclectic city with plenty of large streets, parks and plazas (making it interesting for evaluating pedestrian routes), we use BW to demonstrate the scalability of our approach.

This section first presents a quantitative evaluation of our approach before it compares the quality of our routes to the state of the art in a case study.

5.1 Quantitative Evaluation

We determined sensible values for the parameters of our preprocessing (cf. Section 3) by running preliminary experiments. We set the sidewalk offset to $\delta_\mathrm{s} = 3\,\mathrm{m}$, and set values for sidewalks suppression of small and thin street polygons to $\beta_\mathrm{a} = 1000\,\mathrm{m}^2$ and $\beta_\mathrm{r} = 3.17\,\mathrm{m}$. For queries we assume a regular walking speed of $v_\mathrm{r} = 1.4\,\frac{\mathrm{m}}{\mathrm{s}}$ [6], and we set $v_\mathrm{s} = 0.9\,\frac{\mathrm{m}}{\mathrm{s}}$ for walkable park areas, i. e., $\lambda \approx 0.6$. We also set the park face expansion value to $\epsilon = 20\,\mathrm{m}$. Regarding intersections, we set the crossing penalties to $\alpha_\mathrm{e} = 10\,s$ and $\alpha_\mathrm{w} = 1\,\frac{\mathrm{s}}{\mathrm{m}}$, which leads to about 30 s of expected waiting time for typically-sized intersections.

[1] Note that OSM offers a tag for indicating availability of sidewalks at streets, however, it has not been widely adopted as of now, cf. `http://taginfo.openstreetmap.org/keys/?key=sidewalk`.

Table 1 Size figures before and after preprocessing. Besides graph size, we report the total number of vertices for plaza, park, and obstacle polygons. Preprocessing time is given in [m:s].

	OSM Input					Pedestrian Output				
	Nodes	Edges	Plaza	Park	Obst.	Nodes	Edges	Plaza	Park	Time
BE	378 298	890 682	9 727	33 072	1 116	452 586	1 132 928	7 276	19 903	1:26
BW	8 235 762	17 740 940	74 547	86 380	4 439	10 209 641	22 750 644	63 300	43 632	32:45

Note that though we set these parameter values uniformly for our experiments, the approach would easily allow setting specific values per intersection or park face, if such detailed data was available. Also note that in our instances we do not add crossing edges within street polygons, i. e., at large multi-lane intersections (cf. Section 3). In fact, OpenStreetMap provides these already, and adding further crossings may result in dangerous paths, forcing the pedestrian to cross several lanes without the aid of traffic regulations.

Preprocessing. Table 1 presents size figures for the input and output of our preprocessing. Note that BW is significantly larger than BE (factor of 20 in graph size and factor of 9 in plaza polygons). This is reflected by the preprocessing effort, which takes about 23 times longer on BW. However, the graph size increases by less than 30 % (nodes and edges) by our preprocessing. Unfortunately, polygons representing walkable areas (parks and plazas) in OSM may overlap and, moreover, polygons with holes are not supported. Instead, obstacles are represented as an additional type of polygon. We therefore first compute the union of overlapping polygons and then subtract potential obstacles from it [22]. This explains the (somewhat peculiar) drop of 50 % in the number of park polygon vertices in our output. Note that only less than 3 % of the resulting plaza polygons have holes in them (not reported in the table).

Table 2 presents more detailed figures. We observe that each part of our preprocessing requires a similar amount of time. Regarding sidewalks, only a small subset (12 %) of the roads is actually substituted. (Recall that we replace neither highways nor walkways.) However, the number of sidewalk edges per substituted road segment is more than two on average, due to complex intersections and other effects (cf. Section 3). Regarding plazas, we observe that the number of visibility edges is only a small fraction of the graph (less than 10 %), with less than 10 % of those actually being on shortest paths. The necessary shortest path computations take less than 3 seconds on BW (not reported in the table). For

Table 2 Detailed preprocessing figures. Besides running time, we report the number of added sidewalks, substituted streets, avg. vertices per plaza polygon (Plaza avg.), visibility edges (Vis.) and the fraction of them on shortest paths (SP. [%]), avg. vertices per park (Park avg.), avg. faces per park (Faces/park), avg. vertices per park face (Face avg.), and the vertices of all park faces (Faces total).

	Sidewalks			Plazas				Parks				
	Added sidewalks	Subst. streets	Time [s]	Plaza avg.	Vis. total	SP. [%]	Time [s]	Park avg.	Faces/ park	Face avg.	Faces total	Time [s]
BE	266 336	105 146	32.6	15.38	86 912	9.5	23.0	22.4	10.4	10.68	98 108	31.6
BW	5 580 842	1 824 185	743.7	17.45	772 416	7.7	563.6	20.8	6.6	11.26	155 840	657.4

Table 3 Evaluating the query performance of our approach. We distinguish each combination of the origin/destination being on a street node (s), plaza polygon (p), or park face (f). We report the time in milliseconds to check for each of these cases (Localization), the time for our initialization stage (Initialization) or not applicable (—), and the time for running Dijkstra's algorithm (Dij.).

	BE						BW					
	Localization			Initialization		Dij.	Localization			Initialization		Dij.
Query	Plaza	Park	Street	Plaza	Park	[ms]	Plaza	Park	Street	Plaza	Park	[ms]
s–s	0.021	0.027	0.004	—	—	31.8	0.033	0.040	0.005	—	—	808.0
s–p	0.021	0.016	0.002	0.165	—	30.4	0.032	0.020	0.003	0.173	—	871.6
p–p	0.016	—	—	0.264	—	27.9	0.027	—	—	0.351	—	889.4
s–f	0.021	0.022	0.002	—	0.359	28.3	0.029	0.026	0.002	—	0.310	758.7
p–f	0.017	0.011	—	0.145	0.362	30.6	0.027	0.014	—	0.178	0.303	810.1
f–f	0.020	0.021	—	—	0.733	27.6	0.029	0.027	—	—	0.622	733.6

parks, we observe that including walkways (to compute park faces) increases the number of park vertices by a factor of 5 ("Faces total" in the table). While this results in a high average number of vertices per entire park (111 for BE), the number of vertices per park face remains small, which is the influential performance figure for queries that begin or end in a park.

Queries. We now evaluate the query performance. Recall that our query algorithm takes as input two arbitrary locations, which may be inside a plaza or park, and in which case the query will route from the precise location to the vertices of its surrounding polygon. Table 3 separately evaluates our algorithm for each scenario of placing the origin or destination on a street node (s), inside a plaza (p), or a park face (f). Per scenario, we generated 1,000 queries, choosing origin and destination (i.e., node, plaza polygon or park face) uniformly at random. For the plaza or park case, we further chose an interior point at random.

The query is oblivious to the specific scenario, i.e., we only pass geographic locations as input, and it needs to perform the necessary checks to figure out the right scenario itself. However, at below 80 µs these checks (including the determination of the specific street node or enclosing polygon) take negligible time. The initialization stage for plazas (computing additional visibility edges) or parks (computing and testing projections) is considerably more expensive, but still runs well below a millisecond, orders of magnitude faster than the subsequent run of Dijkstra's algorithm. Note that in our implementation we never add any edges explicitly (cf. Section 4), but rather simply initialize Dijkstra's algorithm with all vertices (and their respective distances) to which these temporary edges would point.

Customizable Route Planning. We finally evaluate the combination of our query algorithm with the Customizable Route Planning (CRP) approach [9] on our larger BW network. We use PUNCH [10] for partitioning. Our partition has five nested levels with at most $[2^8, 2^{11}, 2^{14}, 2^{17}, 2^{20}]$ vertices per cell. (This is the same configuration as in [9].) We compute the partition on the routing graph (that is output by our preprocessing), however, we temporarily replace nodes of the same plaza or park by a single supernode. This keeps polygons from spreading over cell boundaries and simplifies the CRP query. Computing the metric-independent partition takes several minutes and the subsequent customization phase takes about five seconds. Note that to integrate a new cost function, e.g., due to different crossing penalties, only the customization phase has to be rerun, which is very fast.

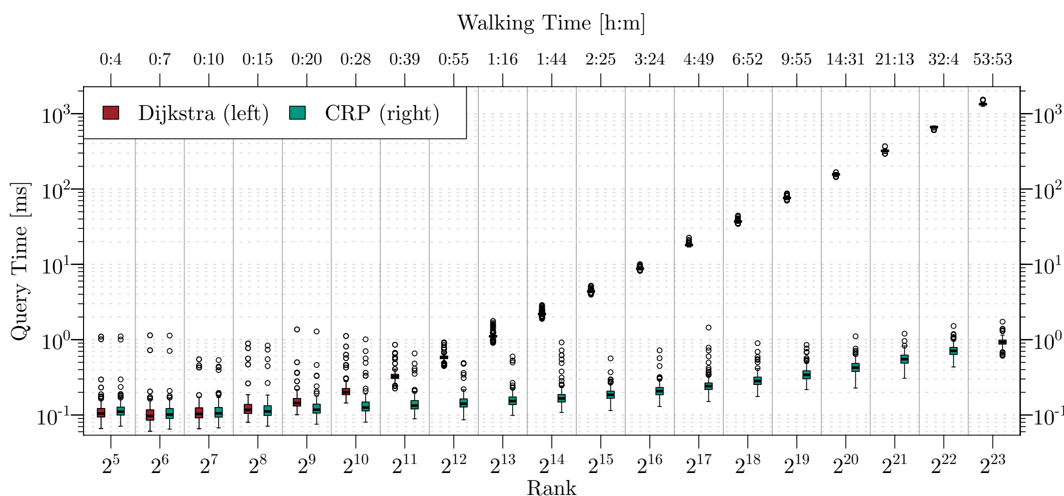


Figure 7 Dijkstra rank plot on our BW instance, comparing the performance of Dijkstra's algorithm with CRP. The top axis shows the average walking time for the queries in each bucket.

Figure 7 compares the performance of CRP with Dijkstra's algorithm using the *Dijkstra rank* methodology [19]: When running Dijkstra's algorithm from node s, node u has *rank* x, if it is the x-th node taken from the priority queue. By selecting random origin and destination pairs according to ranks $2^1, 2^2, \ldots, 2^{\lfloor \log |V_r| \rfloor}$ (we select 1,000 queries per bucket), the plot simultaneously captures short- mid- and long-range queries. We observe that for short-range queries the performance of Dijkstra's algorithm is very similar to that of CRP (below 200 µs on average). However, from rank 2^{10} onward, Dijkstra's algorithm becomes significantly slower (rising to more than a second), while the average running time of CRP remains below 1 ms at any rank. Note that while most pedestrian queries are likely of short range, a production system must nevertheless be robust against any type of query.

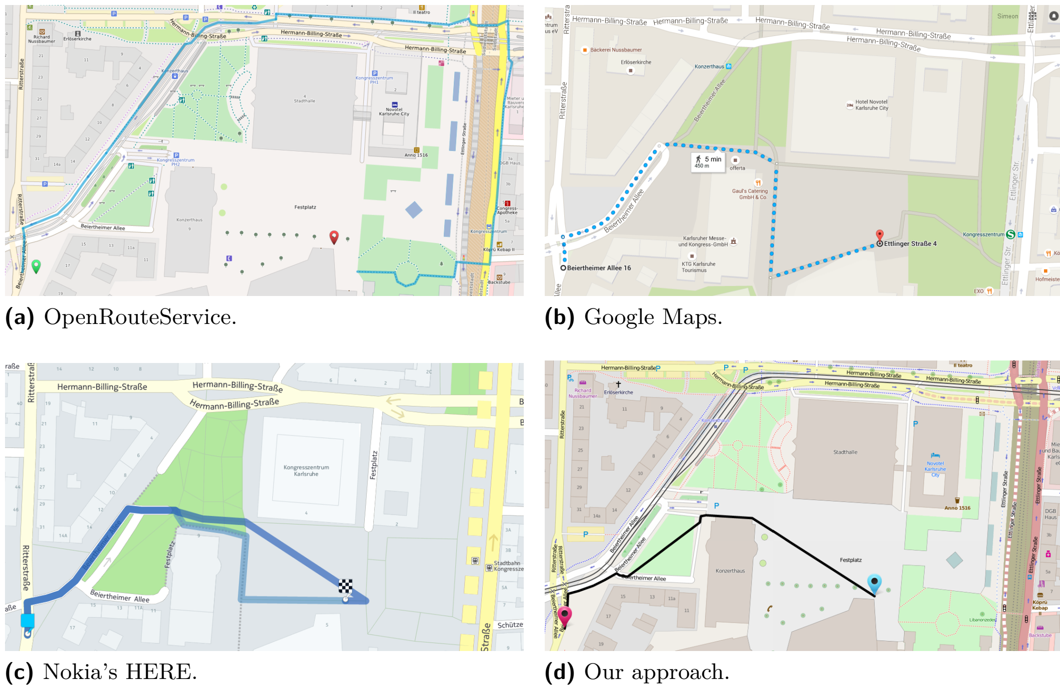
5.2 Case Study

We now present a case study, which compares the output of our approach to OpenRouteService[2], Google Maps[3] and Nokia HERE[4].

Figure 8 shows an example in the city of Karlsruhe, Germany. It highlights the importance of, both, the presence of sidewalks and being able to route across walkable areas. Clearly, OpenRouteService has the worst result, as it does not consider the boundary of the plaza (Festplatz) for routing, which results in a large detour. Because of improper sidewalk data, the routes of Google Maps and HERE suggest to go across the same street (Beiertheimer Allee) twice, which is unnatural and unnecessary. While Nokia HERE is the only competing approach that has some additional edges for walking across open areas (thus yielding a more realistic route), the utilization of these edges seems to be heuristic, still yielding an (unnatural) detour. In contrast, our route has no unnecessary street crossings (because of our generated sidewalk data), and the plaza is traversed in a natural way.

Figure 9 shows an example of a route starting on a plaza between buildings and ending

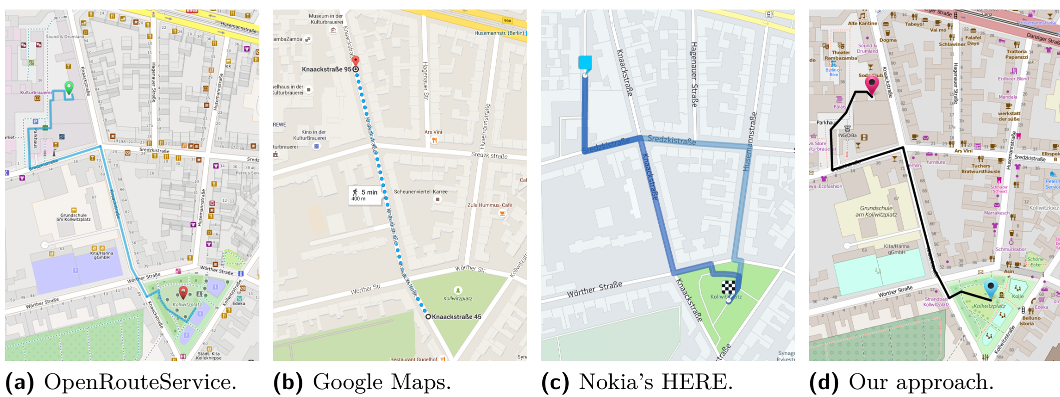
[2] http://www.openrouteservice.org/
[3] https://maps.google.de/
[4] https://www.here.com/



(a) OpenRouteService. **(b)** Google Maps.

(c) Nokia's HERE. **(d)** Our approach.

Figure 8 Comparison with several readily available pedestrian route planning services. The origin of the route is a street address, and its destination is inside a plaza.



(a) OpenRouteService. **(b)** Google Maps. **(c)** Nokia's HERE. **(d)** Our approach.

Figure 9 Comparison with several readily available pedestrian route planning services. The route begins in a plaza and ends in a park.

in a park (Berlin, Germany). Unlike the previous example, OpenRouteService is able to route around (but not across) the plaza, because the plaza's boundary has been tagged as walkable. On the other hand, GoogleMaps seems to lack information in that region and so maps the query locations to the nearest street network node (which is actually blocked by the building structure). HERE has walkways on the plaza, but also uses a shortcut which passes through a cinema; it yields a shorter path but is obscure and unlikely: guiding the pedestrian to a door is puzzling, the building may be closed, etc. As before, the route of our approach traverses the plaza without detours.

Towards the destination, the routes of OpenRouteService, GoogleMaps and HERE are all incomplete: they find the nearest node and simply use it as the query target. In contrast, our approach allows walking directly across the lawn and avoids the small detours introduced by the other approaches.

6 Conclusion

In this paper, we presented an approach for quickly computing realistic pedestrian routes. We proposed geometric algorithms to automatically augment the street network with sensible sidewalks and edges in plazas, making it possible to walk across them in a natural way. Our query algorithm extends classic node-to-node queries by allowing the origin or destination to be an arbitrary location inside a park or plaza. We also combined our algorithm with the well-known Customizable Route Planning technique, which enabled us to compute appealing pedestrian routes within milliseconds, fast enough for interactive applications.

Future work includes more realistic models (e.g., for traffic lights or more precise human walking behavior); leveraging of building layouts [3]; and additional optimization criteria like elevation and stairs, which have been used in the context of bicycle routing [21]. We would also be interested in using our sidewalk generation algorithm in a semi-automatic tool for adding sidewalk data back to OpenStreetMap.

References

1. H. Alt and E. Welzl. Visibility Graphs and Obstacle-avoiding Shortest Paths. *Zeitschrift für Operations Research*, 32(3-4):145–164, 1988.
2. Simeon Danailov Andreev. Realistic Pedestrian Routing. Bachelor thesis, Karlsruhe Institute of Technology, November 2012.
3. Miquel Ginard Ballester, Maurici Ruiz Pérez, and John Stuiver. Automatic Pedestrian Network Generation. In *Proceedings 14th AGILE International Conference on GIS*, pages 1–13, 2011.
4. 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. Technical Report abs/1504.05140, ArXiv e-prints, 2015.
5. Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975.
6. Raymond C. Browning, Emily A. Baker, Jessica A. Herron, and Rodger Kram. Effects of Obesity and Sex on the Energetic Cost and Preferred Speed of Walking. *Journal of Applied Physiology*, 100(2):390–398, 2006.
7. Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller. Efficient Computation of Visibility Polygons. *CoRR*, abs/1403.3905, 2014.
8. Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

9 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks. *Transportation Science*, 2015.
10 Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011.
11 Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
12 Hassan A. Karimi and Piyawan Kasemsuppakorn. Pedestrian Network Map Generation Approaches and Recommendation. *International Journal of Geographical Information Science*, 27(5):947–962, 2013.
13 Jean-Claude Latombe. *Robot Motion Planning*, volume 124 of *Springer International Series in Engineering and Computer Science*. Springer, 1991.
14 Ellips Masehian and M. R. Amin-Naseri. A Voronoi Diagram-visibility Graph-potential Field Compound Algorithm for Robot Path Planning. *J. Robotic Systems*, 21(6):275–300, 2004.
15 M. Mokhtarzade and M.J. Valadan Zoej. Road Detection from High-Resolution Satellite Images Using Artificial Neural Networks. *International Journal of Applied Earth Observation and Geoinformation*, 9(1):32–40, 2007.
16 M. H. Overmars and Emo Welzl. New Methods for Computing Visibility Graphs. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 164–171, 1988.
17 Scott Parker and Ellen Vanderslice. Pedestrian Network Analysis. In *Walk 21 IV*, Portland, OR, 2003.
18 Ting Peng, Ian H. Jermyn, Veronique Prinet, and Josiane Zerubia. Extended Phase Field Higher-Order Active Contour Models for Networks. *International Journal of Computer Vision*, 88(1):111–128, 2010.
19 Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.
20 J.T. Schwartz and M. Sharir. A Survey of Motion Planning and Related Geometric Algorithms. *Artificial Intelligence*, 37(1—3):157–169, 1988.
21 Sabine Storandt. Route Planning for Bicycles – Exact Constrained Shortest Paths Made Practical Via Contraction Hierarchy . In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 234–242, 2012.
22 The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015.

A Batched Point in Polygon Tests

While we could use the computational geometry library CGAL [22] for most of the required geometric computations, we implemented our own sweep line algorithm for a batched point in polygon test. Given a set of disjoint polygons (without holes) and a set of query points, our algorithm determines for each point the polygon that contains it (if any). The algorithm works by sweeping the query points and the end points of the polygon segments, which define our event points, from left to right, maintaining a self-balancing binary tree of the segments which intersect the current (vertical) sweep line. Whenever the current event point is a query point $o = (x, y)$, we find the two segments s_a and s_b, which lie vertically directly above and below o. If s_a and s_b belong to a polygon $Q = (p_1, \dots, p_n)$ with leftmost vertex p_1, we check whether $s_a = \overline{p_i p_{i+1}}$, $s_b = \overline{p_j p_{j+1}}$, and $j > i$. If so, o lies within Q, otherwise it is not contained in any polygon. If m is the number of polygon edges and n is the number of points

and polygon vertices then this algorithm requires $O(n \log m)$ time. For a set of polygons with holes we may use the same approach once for the boundaries and once for the holes.