Line Segment Visibility in Simple Polygons: Exact, Robust, Scalable Computation and Applications

Sándor P. Fekete
Department of Computer Science, TU Braunschweig, Germany
L3S Research Center, Hannover, Germany

Prahlad Narasimhan Kasthurirangan
Department of Applied Mathematics and Statistics, Stony Brook University, NY, USA

Phillip Keldenich
Department of Computer Science, TU Braunschweig, Germany

Fabian Kollhoff
Department of Computer Science, TU Braunschweig, Germany

Chek-Manh Loi
Department of Computer Science, TU Braunschweig, Germany

Michael Perk
Department of Computer Science, TU Braunschweig, Germany

Abstract

The weak visibility polygon of a line segment s inside a simple polygon P, denoted by $V_P(s)$, is the region of the polygon that is visible from at least one point on s. Given its fundamental nature in computational geometry, several algorithms have been proposed to compute weak visibility polygons efficiently, each with different trade-offs in terms of preprocessing time, query time, and space complexity. Although there are many applications that require computing these polygons such as computer graphics, robot motion planning, and network communication systems, there is a lack of *any* implementations of these algorithms in the literature – not to mention one that is exact, robust, and scalable. Furthermore, weak segment visibility polygons are used as basic building blocks in several other algorithms, such as in minimum-link path computation.

In this work, we present an implementation of an optimal linear-time algorithm for computing the weak visibility polygon of a segment inside a triangulated simple polygon. Our implementation provides exact, robust geometric primitives and optimizations to handle large inputs with more than 18 000 000 vertices. We demonstrate two concrete applications: (1) construction of window partitions, a standard data structure in visibility algorithms, and (2) support for optimal minimum-link path queries between two points in a simple polygon, the latter serving as a direct use case of the former. Experimental results on a variety of polygon families confirm that the end-to-end running time scales linearly with the size of the polygon and is dominated by the cost of computing the triangulation, validating the practicality and scalability of the approach. The implementation is released as open source in the format of a CGAL package to support reproducibility and further research.

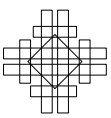
2012 ACM Subject Classification Theory of computation → Computational geometry; General and reference → Experimentation; Mathematics of computing → Arbitrary-precision arithmetic

Keywords and phrases Visibility, line segments, link distance, window partition, computation, implementation, robustness, scalability, exactness, CGAL

Digital Object Identifier 10.4230/LIPIcs.SoCG.2026.45

Supplementary Material *Software (and Data)*: `https://doi.org/10.5281/zenodo.19194126` [32]

Funding Supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) as part of project *Computational Geometry: Solving Hard Optimization Problems* (CG:SHOP) – 444569951.



Prahlad Narasimhan Kasthurirangan: Work was carried out during a research stay at TU Braunschweig. Partially supported by the US Office of Naval Research under grant no. N00014-26-1-2088.

Acknowledgements We thank the authors of [4] for sharing parts of their implementation with us.

1 Introduction

Developing efficient methods for geometric problems is at the core of computational geometry. The usual quality measure for such algorithms is their asymptotic worst-case complexity, making non-trivial, linear runtime an ultimate achievement, such as Chazelle's $\mathcal{O}(n)$ triangulation algorithm [11]. While this is undoubtedly one of the intellectual pinnacles of the field, and widely used as a subroutine for achieving fast asymptotic worst-case complexities for many other algorithms, its sophistication also highlights a serious issue: We are not aware of any practical implementation, leaving a serious question posed by none other than Chazelle himself, "Can computational geometry meet the algorithmic needs of practitioners?" [12].

Achieving this goal requires significant additional effort, often involving tough choices and scientific expertise. From the practical side, asymptotic worst-case complexity does not suffice; instead, we require actual *scalability*: what size instances can indeed be solved in reasonable time? This matters not only for hard problems, but also for algorithms for relatively "easy" tasks that are used as repetitive subroutines when dealing with more complex problems.

Another serious challenge is to achieve *robustness*, i.e., ensuring that an implementation does not fail catastrophically in the vicinity of geometric degeneracies? As demonstrated by Kettner, Mehlhorn, Pion, Schirra and Yap [33], even simple algorithms such as Graham's scan [23] can produce completely erroneous results when simply run with floating point arithmetic. An important tool to aid in achieving robustness is the use of *exact* computations, combined with correct handling of all degenerate special cases such as collinear points and overlaps, ensuring that numerical issues cannot lead to any errors. Achieving this with managable performance impact has been one of the goals in community efforts such as CGAL [55], which won the SoCG Test of Time Award in 2023; among many relevant publications, see Boissonat [5] for a practical paper on triangulations. Finally, an implementation can only be used if it is *available*, i.e., its code is published in a usable state and under a suitable license.

In this paper, we demonstrate practical progress along these lines. We describe (and provide) a practical solution for a fundamental problem of Computational Geometry: Compute the *weak visibility polygon of a segment* ℓ within a simple polygon P. This task occurs in a wide variety of problems, such as in robot navigation or optimal surveillance, e.g., in the classical Watchman Problem; see the surveys by Mitchell [45, 44]. In theory, the weak visibility polygon of a segment can be computed in linear time, combining Chazelle's $\mathcal{O}(n)$ triangulation algorithm with an $\mathcal{O}(n)$ algorithm by Guibas, Hershberger, Leven, Sharir and Tarjan [24]. However, to this date, there is no scalable, robust and exact implementation that solves this problem. Indeed, we are unaware of *any* open source implementation. Further relevance is highlighted by the relation to problems such as the Art Gallery Problem, which is not only NP-hard, but even known to be $\exists\mathbb{R}$-complete [1], illustrating the fragility of solutions and the need for robust solutions. Our implementation has all these properties.

- It is *scalable*: We can solve instances with 18 000 000 points within seconds.
- It is *robust* and *exact*: We use exact arithmetic for all computations.
- It is also *available*: it comes as a CGAL package that can be used out of the box.

In this way, we literally follow Chazelle's Recommendation #1 in his 1996 task force report for the first practice track of SoCG, which is also central for SoCG's new practice track: "production and dissemination of geometric code" [12]. In addition, we showcase

two major use cases to demonstrate the practical applicability of our code for large-scale instances: we show how to compute a *window partition* of a polygon P, as well as computing the *minimum link* path between two points in P, for polygons with up to 2 000 000 vertices.

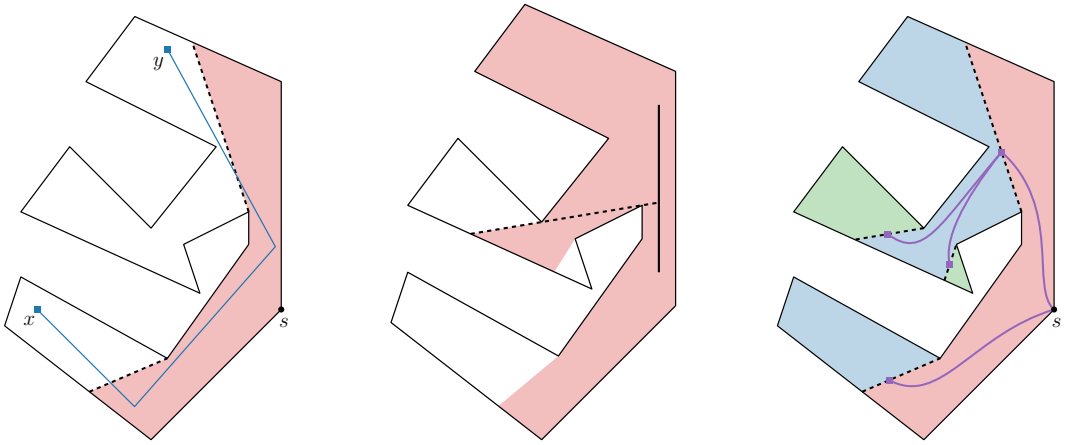


Figure 1 *Left:* $V_P(s)$ is given in red and the windows are marked using dashed lines. A minimum link path between x and y is marked in blue; $d_L(x, y) = 3$. Note that the shortest (Euclidean) path between x and y is not a minimum link path. *Middle:* A segment $\ell \in P$ is marked with a solid black line and $V_P(\ell)$ is marked in red. Note that windows of $V_P(\ell)$ can be defined by points in the interior of ℓ. *Right:* The window partition $\mathcal{W}(s)$ and the window tree T_s of s. Points with link distance one are colored red, with two are colored blue, and with green are colored green. T_s is marked in purple.

1.1 Preliminaries

In the following, P denotes a simple polygon and n its number of vertices. Two points x and y are *visible* to each other if the line segment $xy \subset P$. The *visibility polygon of a point* $s \in P$, denoted by $V_P(s)$, is the set of all points in P visible to s. Given a segment $\ell = pq$ inside of P, the *weak visibility polygon* $V_P(\ell)$ is the set of all points in P that are visible from some point on ℓ, i.e., $V_P(\ell) := \{p \in P \mid \exists\ s \in \ell : p \in V_P(s)\}$; see Figure 1.

For two points $p, q \in P$, the *link distance* $d_L(p, q)$ between them is the minimum number of edges in any polygonal path within P between x and y (see Figure 1, *Left*). Consider a point $s \in P$ and its visibility polygon $V_P(s)$. The vertices of $V_P(s)$ are either vertices of P or the shadows of reflex vertices of P seen by s. The edges of $V_P(s)$ between reflex vertices and their shadows are called *windows*. A window w is a chord of P and divides P into two parts; define $P(w, s)$ to be part of P which does not contain s. The *window partition* $\mathcal{W}_P(s)$ is the partition of P returned by Algorithm 1 described by Suri [50] given input s. We drop the subscript P from $\mathcal{W}_P(s)$ when it is clear from context which polygon we are referring to.

Algorithm 1 Window-Partition(x, P).

1: Compute $V_P(x)$;
2: Let $w_1, w_2, \ldots, w_k$ be the windows of $V_P(x)$;
3: **for** $i = 1$ to k **do**
4: Recursively call Window-Partition$(w_i, P(w_i, x))$;
5: **end for**
6: Output $R(x) := V_P(x)$;

Let $R(w)$ denote the region generated by a window w. The rooted dual tree T_s of $\mathcal{W}(s)$ has as vertices the windows defining $\mathcal{W}(s)$ and edges between two vertices if their regions share a common boundary. The root of T_s is s. We refer to T_s as the *window tree* of s. See Figure 1, *Right*. Let $g(u, v)$ denote the minimum distance of two nodes u, v in T_s and consider a point $p \in R(w)$. The link distance $d_L(s, p)$ is $g(s, w) + 1$ [50].

Constructing a triangulation $\mathcal{T}$ of a polygon P takes linear time [11]. Thus, computing $V_P(\ell)$ for a segment ℓ in P takes linear time – first triangulate P, then use the algorithm from [24]. As we are unaware of any implementation of [11], we use a constrained Delaunay triangulation of P provided by CGAL [5]. This has a worst-case complexity of $\mathcal{O}(n^2)$ [8] but works extremely well in practice, avoids degenerate triangles and thus simplifies our workflow. Our implementation runs in linear time given $\mathcal{T}$ as in [24]. To achieve this runtime, we implemented a specialized data structure called a *finger tree* [30, 29] (see Section 2.1). To the best of our knowledge, all previous implementations of finger trees are in functional programming languages like Haskell or Scala [29, 22]; we used C++ for their iterative versions. We implemented an improvement to [24] suggested in [50]; this allows us to compute $V_P(\ell)$ in time linear in the number of triangles in $\mathcal{T}$ it intersects rather than n. Apart from allowing us to reliably compute $V_P(\ell)$ even in extremely large input polygons, this is crucial to compute window trees in $\mathcal{O}(n)$. All our implementations use exact number types and handle degeneracies. In Section 2, we review the algorithms introduced in [24, 50] and discuss relevant implementation details. We recommend readers familiar with these results to skim through our implementation notes in Section 2 before skipping to Section 3.

1.2 Related work

Computing visibility polygons is central to computational geometry. We focus on weak visibility polygons of segments inside simple polygons; for a more general overview of visibility problems, see [56, 46, 48, 21, 15]. Given a triangulation of a polygon, Guibas, Hershberger, Leven, Sharir and Tarjan [24] presented an $\mathcal{O}(n)$ algorithm that computes the visibility polygon. A number of algorithms have been proposed for data structures that can efficiently compute these for a query segment ℓ, each with its own trade-offs in terms of preprocessing time, query time, and space complexity. We summarize the most relevant ones in Table 1.

Table 1 A summary of theoretical data structures to compute the visibility polygon of a line segment in a triangulated simple polygon. The value k is the size of $V_P(\ell)$ for a query segment ℓ.

Data Structure	Preprocessing Time	Size	Query Time
Bose, Lubiw and Munro [6]	$\mathcal{O}(n^3 \log n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(k \log n)$
Bygi and Ghodsi [9]	$\mathcal{O}(n^3 \log n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(k + \log n)$
Aronov et al. [3]	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(k \log^2 n)$
Chen and Wang [13]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(k \log n)$
Chen and Wang [13]	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(k + \log n)$
Guibas et al. [24]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
With finger tree	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
With CGAL's `Multiset`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$

Our results are given in bold. We remark that implementing the more recent results with improved output sensitive query time [3, 6, 9, 13] with exact predicates seems extremely challenging. It is also unclear if these will lead to improvements in runtime in practice (see Section 4 for experiments).

Computing segment visibility is a crucial subroutine in several important geometric data structures; perhaps the most well-known one being window partitions and window trees [50]. Other than computing minimum link paths, window partition is also used in convex decomposition algorithms [7, 38] as well as in pursuit evasion problems [52].

Minimum link paths have numerous applications across domains: in motion planning, where turns are expensive and their number should be minimized [47]; in communication systems design under line-of-sight constraints, where a min-link path minimizes the number of necessary repeaters; in placement of telescoping manipulators with flexible telescopic links [34]; in curve simplification by approximating curves with minimum link paths [25, 31, 43]; and in VLSI layouts, where wires typically run along orthogonal axes, motivating the study of link path problems in rectilinear settings [54, 14]. For a comprehensive overview, we refer the reader to [39, Chapter 12] and [40]. There also exist several works on implementing minimum link paths: [53] is restricted to rectilinear paths and is the only work with a publicly available implementation we could find, [20, 57] also consider rectilinear paths, and [17, 58] are restricted to rectilinear environments. Baum, Bläsius, Gemsa, Rutter, and Wegner use minimum link paths in [4] to simplify the boundary of a polygon in a corridor; this is the only other experimental paper on (non-rectilinear) minimum link paths we are aware of. While the authors provided parts of their non-public implementation to us and the paper provides valuable insights into optimizing minimum link path computations for a specific source-target pair, their implementation is tailored to their specific use case and does not offer a method to compute optimal paths between two arbitrary points in a simple polygon.

2 Weak visibility of a segment

In this section, we give an overview of the algorithm by Guibas, Hershberger, Leven, Sharir and Tarjan [24] for computing the visibility polygon of a segment inside a simple polygon P. For a more detailed explanation we refer the reader to the original paper. Consider a triangulated simple polygon P and line segment $\ell = ab$ in P. The algorithm has two steps: first, we compute the *shortest path tree* from both a and b (see Section 2.1 for a definition). Simple data structures can compute this in $\mathcal{O}(n \log n)$ time, reduced to $\mathcal{O}(n)$ using a *finger tree*. In the second step, we compute $V_P(\ell)$ in linear time by iterating over all edges of the polygon and testing if any part of them is visible from ℓ. We only require (amortized) constant time to determine this for each edge of P (and to return the part of this edge that is seen) when we have the shortest path trees of a and b. We discuss details in Section 2.2.

2.1 Shortest path tree

Let P be a simple polygon with n vertices and let s be a given *source vertex* of P. For two points $x, y \in P$, let $\pi(x, y)$ denote the Euclidean shortest path between x and y inside P. Lee and Preparata [37] showed that for all vertices v of P, $\pi(s, v)$ is a polygonal path whose corners are vertices of P and that $\bigcup_v \pi(s, v)$ over all vertices v forms a tree rooted at s. We use $Q_P(s)$ to denote this *shortest-path tree* rooted at s – the vertices of $Q_P(s)$ are vertices of P and (u, v) is an edge of the tree if it is an edge of either $\pi(s, u)$ or $\pi(s, v)$.

Let $\mathcal{T}$ be a triangulation of P and G the dual graph of $\mathcal{T}$. Let $d = uw$ be a diagonal or and edge of P and let a be the deepest common ancestor of u and w in $Q_P(s)$, see Figure 2 *Left*. Then the path $\pi(a, u)$ and $\pi(a, w)$ are two outward convex chains [37], i.e., the convex hull of each subpath lies outside of the region bounded by $\pi(a, u)$, $\pi(a, w)$, and uw. Following [37], we call $F = F_{uw} = \pi(a, u) \cup uw \cup \pi(a, w)$ the *funnel* of d with *cusp* a.

We compute $Q_P(s)$ in linear time by maintaining funnels of diagonals in $\mathcal{T}$ as we process its vertices. We maintain the current funnel F_{uw} as a sorted list $[u_l, u_{l-1}, \ldots, u_1, a, w_1, \ldots, w_k]$ where $a = u_0 = w_0$ is the cusp of F, $\pi(a, u) = [u_0, u_1, \ldots, u_l]$ and $\pi(a, w) = [w_0, w_1, \ldots, w_k]$ with $u_l = u$ and $w_k = w$. We call Algorithm 2 [24] with s and an adjacent vertex v_1 as the initial funnel F; for an interior source point s, we start by locating the triangle containing s and create three funnels, one for each edge, with cusp s.

Algorithm 2 ComputePath(F).

1: $u \leftarrow$ first element of F and $w \leftarrow$ last element of F;
2: $a \leftarrow \text{CUSP}(F)$;
3: $\triangle uwx \leftarrow$ unique unprocessed triangle incident to $d = uw$;
4: Compute $v \in F$ such that vx is a tangent of F at v;
5: Split $F \cup \{x\}$ into $F_1 = [u, \ldots, v, x]$ and $F_2 = [x, v, \ldots, w]$;
6: **if** $v \in \pi(a, u)$ **then**
7: $\quad \text{CUSP}(F_1) \leftarrow v$ and $\text{CUSP}(F_2) \leftarrow a$;
8: **else if** $v \in \pi(a, w)$ **then**
9: $\quad \text{CUSP}(F_1) \leftarrow a$ and $\text{CUSP}(F_2) \leftarrow v$;
10: **end if**
11: $\pi(s, x) \leftarrow \pi(s, x) \cup vx$;
12: Call ComputePath(F_1) and ComputePath(F_2) if ux or wx is a diagonal of P;

We explain Lines 4 and 11 of Algorithm 2 in more detail. As $\pi(a, u)$ and $\pi(a, w)$ are outward convex, the slopes of the segments defining them are monotonic. Thus, we can do a binary search on these slopes to find the point of tangency for x to F in Line 4; see Figure 2, *Left*. We do not store shortest paths explicitly as suggested by Line 11, we simply store back pointers from x to v; $\pi(s, x)$ is computed by following these pointers until we reach s.

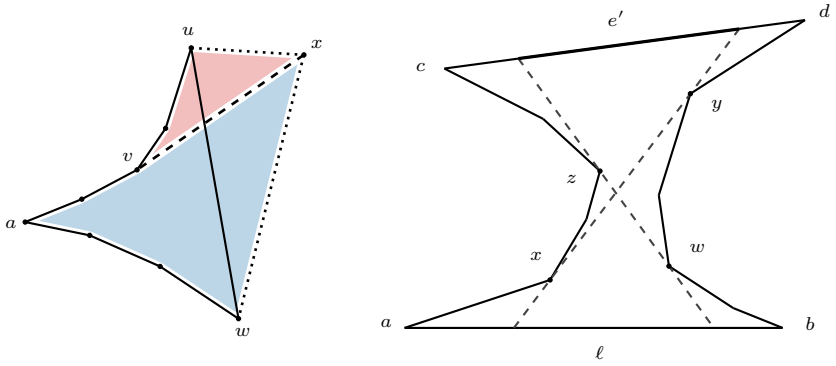


Figure 2 *Left:* Splitting the funnel $F = [u, \ldots, w]$ when a new vertex x is processed. F is split into two sub-funnels $F_1 = [u, \ldots, v, x]$ (red) and $F_2 = [x, v, \ldots, w]$ (blue). *Right:* Hourglass for visibility polygon calculation [24].

Although using binary search as above is straightforward with a binary search tree, it requires $\log n$ time, as $\pi(s, u)$ and $\pi(s, v)$ (and thus our list representing F) might contain $\mathcal{O}(n)$ elements. For an overall linear runtime, we need to find v in (amortized) constant time. This is achieved by storing the list F as a so-called *finger tree* [26]. Our finger trees are modified 2-3 search trees which include pointers (called *fingers*) to the last and first element

of F. They allow for access to v, as well as the splitting of F at v in $\mathcal{O}(\log \delta)$ time, where δ is the distance from v to the nearest finger. See Guibas, McReight, Plass and Roberts [26] for details on properties of finger trees. Gibiansky [22] also provides valuable insights.

Implementation notes. To the best of our knowledge, all implementations of finger trees are in functional programming languages; we implemented an iterative version in C++. As suggested in [24], we can use simpler data structures such as linked lists to represent funnels, resulting in a runtime of $\mathcal{O}(n \log n)$ instead. To compare with our finger tree implementation, we use CGAL's built-in `Multiset` class, as it supports search and split natively. In Section 4, we evaluate the performance of our finger tree implementation against CGAL's `Multiset` class. Collinear points pose a challenge for balanced trees, as orientation checks alone cannot establish a strict ordering. By carefully distinguishing cases, we ensure that the funnel remains free of collinearities at all times.

2.2 Segment visibility polygon

Given a polygon P and a segment $\ell = ab$ inside of P, recall that the weak visibility polygon $V_P(\ell)$ contains all points in P that are visible from a point on ℓ, i.e., $V_P(\ell) = \{s \in P \mid \exists t \in \ell : s \text{ is visible from } t\}$. The following structural lemma is crucial.

▶ **Lemma 1** ([24]). *If a point interior to an edge $e' = cd$ is visible from a segment $\ell = ab$ then (up to swapping c and d) the two paths $\pi(a, c)$ and $\pi(b, d)$ are outward convex.*

Consider such an edge $e' = cd$. The structure formed by ℓ, e', $\pi(a, c)$ and $\pi(b, d)$ is called an *hourglass*, see Figure 2, *Right*. When running the shortest path tree algorithm with a as the source, the funnel $F_{e'}$ has x as its cusp and y as an adjacent vertex. Similarly, when running the algorithm with b as the source, the funnel $F_{e'}$ has w as its cusp and z as an adjacent vertex. These two tangents define the visible section of e'.

During the computation of the $Q_P(s)$, we check, for each vertex v, in constant time if $\pi(s, v)$ is outward convex or not. To extract the visibility polygon, we iterate over the vertices of the polygon and check whether an edge is (partially) visible as explained above. Then we connect continuous visible portions of the boundary, skipping over invisible edges, by connecting visible vertices with their shadow vertices (defined by the tangents). While the algorithm defined in [24] is not output sensitive (taking $\mathcal{O}(n)$ time regardless of the size of $V_P(\ell)$), we explain in Section 2.3 modifications to this algorithm to have this characteristic.

Implementation notes. Although point visibility is available in CGAL [28], our implementation can also compute the visibility polygon of a point s by essentially treating it as degenerate segment, simplifying window extraction. Figure 3 shows the visibility polygon $V_P(s)$ of a query point s. All labeled points are collinear, so the point c is visible from s. We call these low-dimensional features caused by degeneracies *needles* and the polygons that have them *non-regularized*. Often needles are undesired; however, for the implementation of the window partition tree in Section 2.4, we require such visibility polygons. Thus, the visibility polygons we generate are non-regularized; we provide a simple linear-time post-processing function that regularizes visibility polygons for use cases where such polygons are preferred.

2.3 Output sensitivity

A naive implementation of Algorithm 1 using [24] would, for each window, compute a complete shortest path tree of P, resulting in a worst case runtime of $\Omega(n^2)$. Suri [50] suggested modifications to [24] to make the computation of segment visibility output sensitive. The

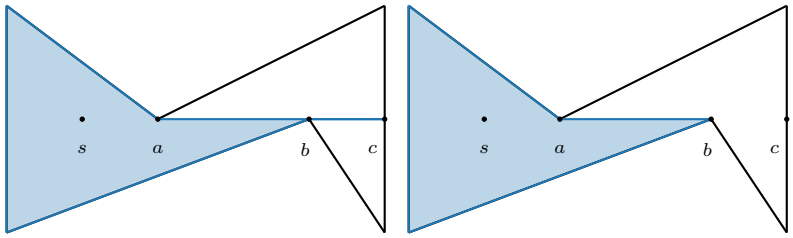


Figure 3 Non-regularized and regularized visibility polygon of s. All labeled points are collinear.

main idea is to synchronize the shortest path tree computation of both endpoints of a segment ℓ and abort the computation of the visibility polygon if an edge is no longer visible from ℓ.

Let P be a simple polygon and $\mathcal{T}$ be a triangulation of the interior of P. For a given segment $\ell = ab$, we start by computing the shortest path tree rooted in a and b along the triangles intersected by ℓ. When processing the next point x in a triangle $\triangle uwx$, we only continue with a diagonal ux if either $\pi(a, u)$ and $\pi(b, x)$, or $\pi(a, x)$ and $\pi(b, u)$ are both outward convex. Symmetrically, we only continue with diagonal wx if either $\pi(a, w)$ and $\pi(b, x)$, or $\pi(a, x)$ and $\pi(b, w)$ are both outward convex. The resulting runtime to compute a visibility polygon $V_P(\ell)$, is $\mathcal{O}(k_\ell)$, where k_ℓ is the number of triangles intersected by $V_P(\ell)$. To ensure that the implementation achieves this runtime, we carefully adjust the traversal of the polygon boundary to only consider vertices that are part of the considered triangles.

Note that k_ℓ can be arbitrarily large compared to the size of the visibility polygon $V_P(\ell)$, e.g., for a triangular visibility polygon that intersects many triangles of $\mathcal{T}$. However, exploiting the fact that each triangle is intersected by at most three partitions [50], we obtain a linear-time algorithm to compute the window partition $\mathcal{W}(s)$ of a point s in P. Note that to start the shortest-path tree computation, we have to locate a triangle intersecting ℓ. We do this using CGAL's built-in `locate` function, which in the worst case has a linear runtime.

2.4 Window partition

To compute the window partition $\mathcal{W}(s)$, we combine Algorithm 1 with our output sensitive visibility polygon computation. The algorithm initially computes a single triangulation $\mathcal{T}$ of P, which we use to compute the non-regularized visibility polygon of s, giving us the initial set of windows. Then, for each window w, we can compute the visibility polygon $R(w) := V_{P(w,s)}(w)$ on the remaining polygon $P(w, s)$. We reuse $\mathcal{T}$ by checking if the next vertex is in $P(w, s)$. This window partition forms the basis for our implementation of the minimum link path algorithms in Section 3.

3 Minimum link path

For a simple polygon P and two interior points s and t, the *link distance* $d_L(s, t)$ between s and t is the minimum number of line segments (links) required to form a polygonal path from s to t that is completely contained in P. The link metric has received considerable attention in computational geometry [49, 50, 42, 2, 41, 35, 27], typically focusing on answering distance queries and not constructing actual minimum link paths. Using window partition trees and other observations, Arkin, Mitchell and Suri [2] proposed an algorithm that can answer link distance queries in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n^3)$ preprocessing time and space. Although the authors mention that the path can be constructed by backtracking through the windows, no details are provided. We implemented two linear time algorithms [50] for answering minimum link path queries based on window partitions: an exact algorithm and an (additive) approximation algorithm that is at most four links longer than the optimal solution.

3.1 Exact algorithm

To construct exact minimum link paths from a source point s to any target point t in a simple polygon P, we use the window partition $\mathcal{W}(s)$ rooted at s. First, we locate the region $R(w)$ that contains t and then traverse the unique path $(w, w_1 \dots s)$ from w to s in T_s. Our construction of $\mathcal{W}(s)$ informs us of a minimum link path from s to t in P that has a vertex on each of the windows $w, w_1, \dots, s$ in that order. Constructing T_s takes $\mathcal{O}(n)$ time and space [50] and locating the target node takes $\mathcal{O}(n)$. Point location could be reduced to $\mathcal{O}(\log n)$ time with preprocessing, but the preprocessing takes super-linear time [15, Chapter 6]. In Section 3.3, we explain how we reconstruct the minimum link path in P in linear time.

3.2 Approximation algorithm

To compute an additive $(\mathrm{OPT}+4)$ approximative minimum link path from p to q in a simple polygon P, we can use the window partition $\mathcal{W}(s)$ of an arbitrary point $s \in P$. For two vertices u and v of T_s, let $g(u, v)$ denote the distance in T_s between them.

▶ **Lemma 2** ([50]). *Let w_1, w_2 be two nodes of T_s, and let $p \in R(w_1), q \in R(w_2)$ be two points of P. Then $g(w_1, w_2) - 1 \leq d_L(p, q) \leq g(w_1, w_2) + 3$.*

The idea is to first identify the two regions $R(w_p)$ and $R(w_q)$ that contain p and q in $\mathcal{W}(x)$ respectively. Then, we identify the lowest common ancestor w_0 of w_p and w_q in T_s. This allows us to construct two paths: one from w_p to w_0 and one from w_q to w_0. Finally, we have to connect the two paths with an additional link on w_0.

Note that, even if $w_0 = w_p$ or $w_0 = w_q$, i.e., the path is directed in T_s, we may still require an additional link using the above algorithm: let $w_0 = w_p$ and $w_0, w_1, \dots, w_q$ be a directed path in T_s. As in Section 3.1, we construct a path from q to p with one vertex v_i on each window w_i for $0 \leq i \leq t$. Unlike the exact algorithm, we cannot guarantee that p is visible from v_0, we can only guarantee the existence of a point v_0' on w_0 that is visible from p. Therefore, we may need to add an additional link from v_0 to v_0' and then to p.

We precompute window partitions rooted at a set of seed vertices and answer queries by probing a small subset of them. At query time, a small set of candidate seed vertices is chosen, e.g., the k nearest to the endpoints. Additionally, we post-process the candidates by removing any introduced loops. The algorithm returns a path with a minimum number of links among all candidates and runs in time proportional to the number of candidates and the cost of locating nodes and reconstructing paths, i.e., $\mathcal{O}(k \cdot n)$ in the worst case for k candidates. Note that Lemma 2 also provides a lower bound on the link distance between p and q that can be used to evaluate the quality of the approximated path.

3.3 Path reconstruction

Now we reconstruct the path from s to t for the exact algorithm detailed in the prior sections. Specifically, we look to construct $(s, p_0, p_1, \dots, p_j, t)$, where p_i belongs to the window w_i. This is done backwards from t. For the step from w_0 to s, any point on w_0 works because s sees all points on w_0. For going from a window w_{i+1} to w_i, this can easily be done in $O(1)$ by extending w_{i+1} to a line and determining its intersection with w_i. However, especially if t is not fixed when constructing the window partition, e.g., because we are answering multiple queries with the same source s, we need to find a point on the last window w_j that sees t. There is a straightforward algorithm that can do this in linear time: compute $V_{R(w_j)}(t)$ and intersect this with w_j to obtain the maximal connected subsegment of w_j that sees t. In practice, this approach is not preferable due to the lack of efficient implementations of

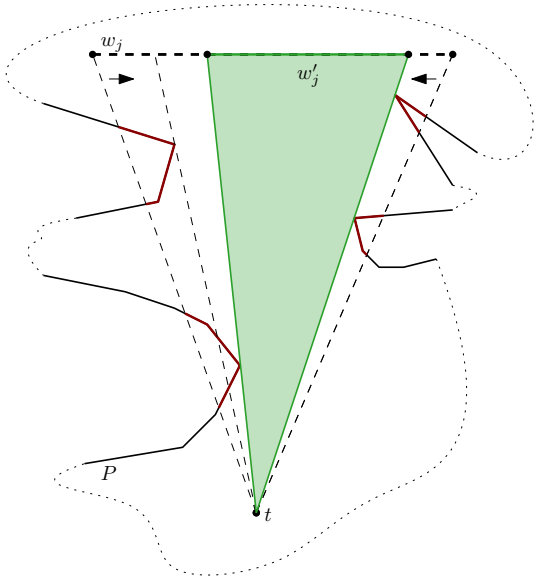


Figure 4 Illustration of the routine described in Section 3.3. The interior point t emits two initial rays defining a cone through the endpoints of window w_j. Several boundary chains (in red) intersect these rays; walking these chains tightens the cone whenever an occluding vertex yields a stricter supporting direction. The intersection of the final cone with w_j is the visible subsegment w'_j. The arrows indicate the direction in which the cone tightens.

visibility algorithms for non-regularized polygons that often appear in the generated window partitions, see Figure 3. The linear-time algorithm available in CGAL [28] is not directly applicable, as it requires the polygon to be simple. Additionally, the triangular expansion approach implemented in CGAL, which is often preferred in practical applications, requires a triangulation of each window, which cannot be directly derived from our triangulation of P, as windows are not necessarily edges of the triangulation; even if we had such a triangulation, the interface of CGAL's visibility package does not allow specifying a triangulation as input. Even without these issues, triangulating each region separately would result in a (practical) superlinear worst-case runtime.

To avoid these problems, we compute the maximal connected subsegment w'_j of w_j that is visible from t in linear (in the complexity of $R(w_j)$) time as follows. Consider the ray between t and the left endpoint of w_j; this possibly intersects $\partial R(w_j)$. Consecutive hitting points define chains of edges on one side of the ray that occlude parts of w. March through the vertices in these chains and choose the vertex defining the rightmost ray from t through it. We find the supporting ray for the right endpoint symmetrically. If these supporting rays cross, w_j is not visible from t (an impossibility). Otherwise, they define the maximal subsegment on w_j visible from t; see Figure 4 for an illustration. Because each other step of path reconstruction takes time $O(1)$, this allows path reconstruction for source s in time $O(|R(w_j)| + j)$ once $\mathcal{W}(s)$ is computed and the window containing t is identified.

For the approximation algorithm, we apply the above procedure for both the windows containing s and t; apart from this and an added segment on the window defining their deepest common ancestor, the situation is analogous.

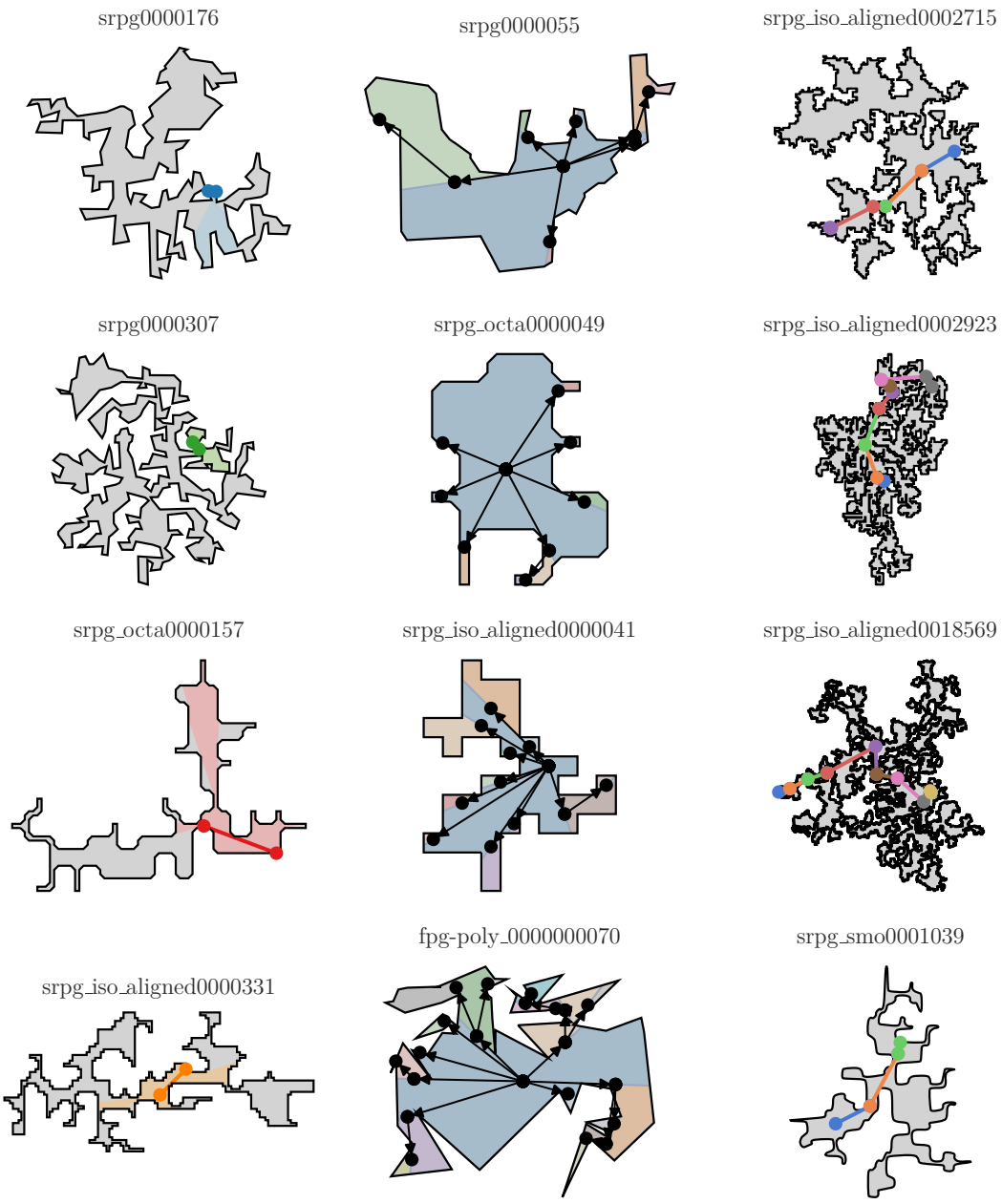


Figure 5 Outputs for the algorithms described in our paper. *Left Column:* Weak visibility polygon of a segment in a simple polygon. *Middle Column:* Window partition of a simple polygon from a source point s. *Right Column:* Minimum link path between two points s and t in a simple polygon.

4 Experimental evaluation

In this section, we present the experimental evaluation of our implementations, see Figure 5 for some example outputs. All experiments were run on otherwise idle workstations equipped with AMD Ryzen 7 5800X CPUs and 128 GiB of DDR4-3200 memory, using Ubuntu Linux 24.04.3 LTS as operating system. The core routines were implemented in C++17, compiled against CGAL 6.0.1 and Boost 1.83.0 using g++ 13.3.0. Our instances and code are publicly available[1]. To guard against programming errors, our code contains large amounts of optional checks and assertions. We also ran a suite of tests with these checks enabled, using instrumentation with tools such as the undefined behavior and address sanitizers of our compiler, which should catch essentially all memory errors. We validated that all paths and visibility polygons found in our experiment stay within the given polygon, and any approximate path has at least the same number of links as the corresponding exact link path.

4.1 Weak line segment visibility

For evaluating our implementation, we consider the following research questions.

RQ1 How does our implementation scale to large inputs, and how is the total runtime distributed onto the individual subroutines?

RQ2 Is there a measurable difference between using finger trees and CGAL's multiset? Which of the two approaches is faster?

RQ3 How does the performance of our implementation depend on the size of the output?

To address these questions, we randomly generate segments for simple polygons from the Salzburg Database of Geometric Inputs[2] [19], the largest of which has 18 556 888 vertices. These polygons are of various types and include polygons with many collinear points or other degeneracies. To generate segments, we sample a source point uniformly at random within the given polygon, compute the visibility polygon of that point, and then either sample a target point in the visibility polygon or pick a random vertex of it. The former approach corresponds to sampling segments inside polygons uniformly at random, whereas the latter focuses on placing one of the endpoints on the boundary of the polygon, exercising edge cases. Using each polygon and each approach twice results in a total of 24 096 instances. For each such instance, and each of the two shortest path tree data structures, we compute the exact visibility polygon five times, measuring the runtime (both total and of various subroutines). Figure 6 depicts the resulting runtimes. We can see that the overall runtime is dominated by computing a triangulation of the input polygon, which we do via CGAL's constrained Delaunay triangulation (CDT). Even CGAL's `mark_in_domain` routine, which simply flags the CDT's triangles that constitute the polygon, is typically more expensive than the actual visibility computation. The fact that the runtime for the visibility computation seems to be almost unaffected by the size of the input polygon can be explained by the fact that it – except for a few small operations such as finding the segment endpoints in the triangulation – mostly depends on the size of the output polygon; see Figure 7, *Left*. On our inputs, the implementation using finger trees is slightly faster than the one using CGAL's `Multiset`, see Figure 7 *Right*. In a preliminary microbenchmark based on building funnels by left insertion, our finger trees became significantly faster already at around 15 funnel vertices; this makes a difference even for the typically small funnels we encounter in our algorithm. We therefore used our finger trees in our other experiments.

[1] `https://doi.org/10.5281/zenodo.19194126`

[2] We used all valid simple polygons from `https://zenodo.org/records/3784789` not labeled as contrived.

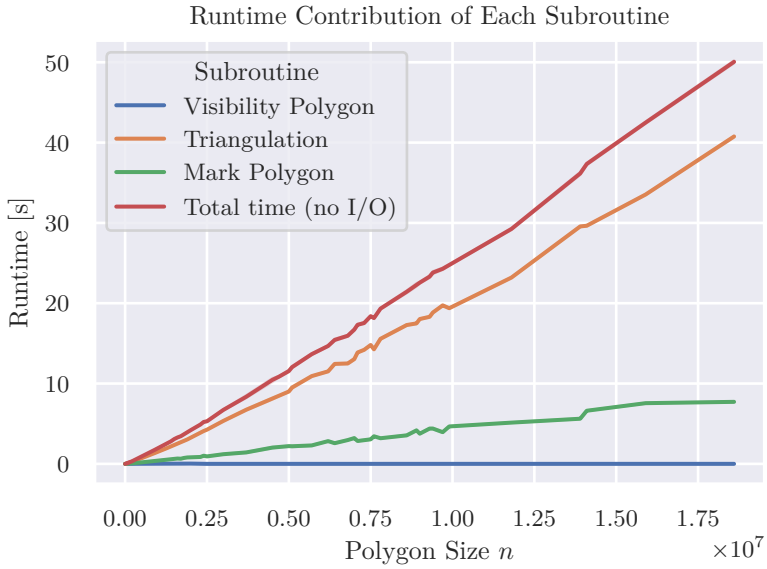


Figure 6 The runtime of different subroutines (triangulating the polygon, marking faces in the polygon, and computing the visibility polygon itself).

4.2 Minimum link paths

Recall that computing a window partition is a basic routine that is useful in different contexts. Furthermore, we only need one window partition from a source point s to compute shortest link-distance paths to any target t, and only a constant number to approximate shortest link-distance paths for any source-target pair inside our polygon. We therefore study the performance of our window partition on its own, as well as in the context of minimum link paths, addressing the following research questions.

RQ4 How does our window partition implementation scale to large polygons?

RQ5 How does it compare to the other parts of our exact minimum link path implementation?

RQ6 How does the approximate minimum link path algorithm perform w.r.t. runtime and solution quality?

To answer these questions, we randomly selected a sample of 1000 simple polygons with at most 2 000 000 vertices from [19]. For each such polygon P, we generate 10 source points $s \in P$ uniformly at random and computed a window partition for each source. Furthermore, for each source point s, we generate a random target point $t \in P$ outside of the visibility polygon of s and computed an exact minimum link s-t-path, as well as an approximate minimum link path using $k = 7$ randomly chosen source points for the window partitions in our approximation. The resulting runtimes are depicted in Figure 8. For each path computed using the approximation algorithm, Figure 9 shows the absolute, additive error depending on the number of window partitions k; the quality of the approximation notably improves with higher k. One should, however, note that increasing k makes both preprocessing and path reconstruction more expensive, both scaling linearly in k.

Overall, we find that both our window partition and minimum link path implementation scale well to large polygons; in fact, in our experiments, time spent inputting, verifying and outputting polygons was typically orders of magnitude higher than pure computation time. If we want to query minimum link paths from a single source s, each subsequent query helps

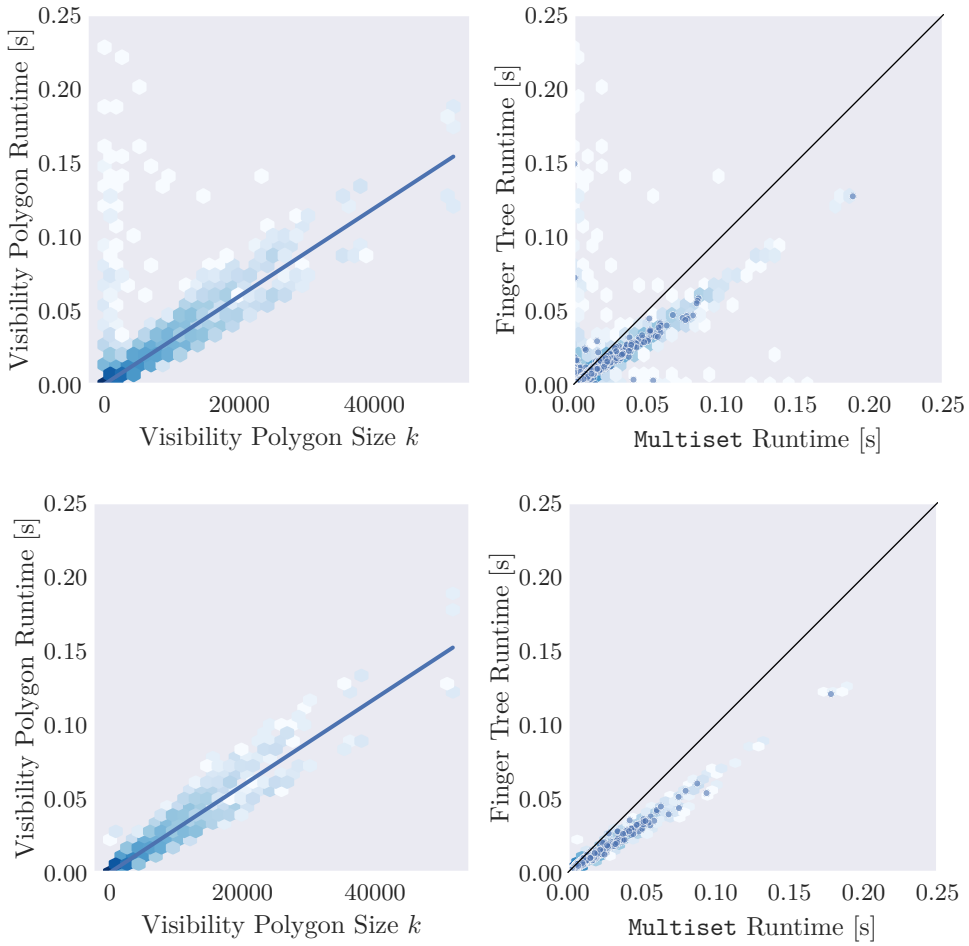


Figure 7 *Top Left:* Runtime dependence on the size of the output visibility polygon, excluding preprocessing such as triangulating the polygon. The hexagonal bin plot shows the underlying distribution similar to a scatter plot, for which we have too many data points; darker hues correspond to a higher concentration of data points, the gray area contains none. *Top Right:* Plot of the runtime using CGAL's `Multiset` and our finger tree implementation with a random sample of data points; the finger tree is better for points below the black diagonal. *Bottom:* The same data as the top row, but dropping the two worst runtime measurements (out of five) for each instance/data structure combination. As our algorithm is deterministic, it is likely that the noise in the top row is simply due to runtime artifacts such as interrupts and scheduling of background processes.

amortizing the cost of computing a window partition; similarly, for doing many queries for different sources and targets in the same polygon, the approximate algorithm can be used to amortize the cost of computing a constant number of window partitions. Depending on the relative importance of runtime versus solution quality, the approximation may be preferable.

5 Conclusions

We presented a scalable, robust and exact implementation for computing the weak visibility polygon of a line segment in a simple polygon. We also demonstrated two concrete applications: computing the window partitioning, and computing the link distance between two points inside a simple polygon. For further improvement on the algorithm engineering side, we could parallelize the shortest path tree and window partition implementations.

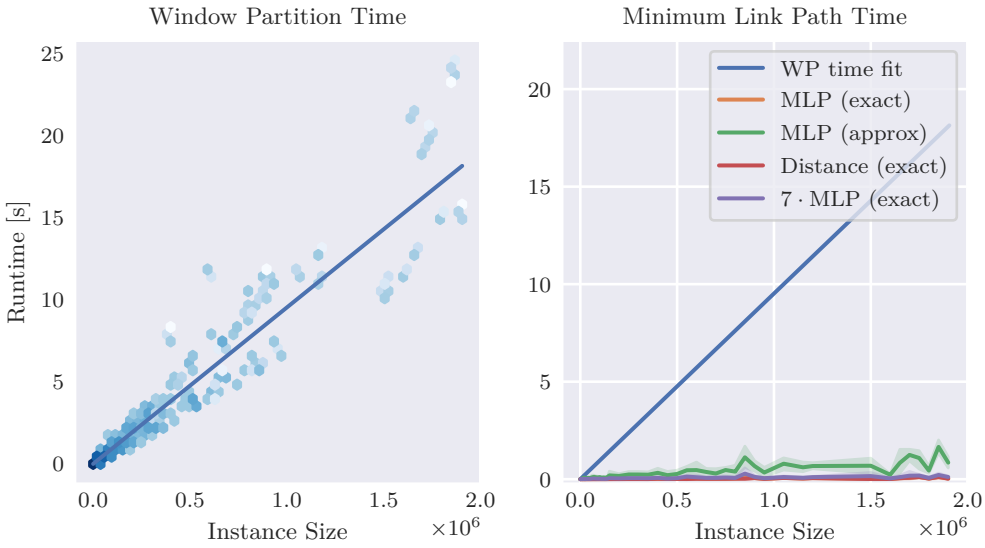


Figure 8 *Left:* The runtime distribution of our window partition implementation, which overall appears to scale linearly, with some instances slightly below and some slightly above the average. *Right:* Comparison of the individual components, showing only the regression line for the window partition time to improve readability. It can be seen that, once a window partition is computed, computing the link distance is very cheap. A single path reconstruction has almost negligible cost, whereas $k = 7$ reconstructions for the approximate approach are more noticable, but still clearly dominated by the window partition time.

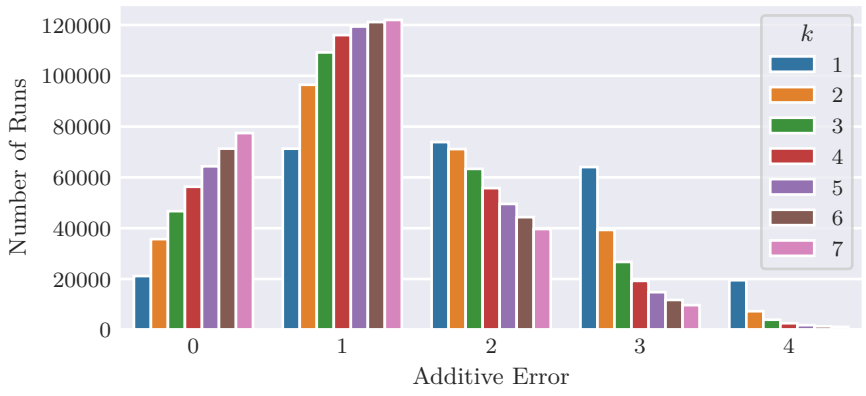


Figure 9 The additive approximation error of our approximate minimum link paths, depending on k, the number of source points used to compute window partitions. In particular, for $k \geq 4$, the majority of queries result in an error of 0 or 1.

An immediate generalization is to non-simple polygons. Suri and O'Rourke [51] studied algorithms for computing visibility polygons with holes. Following the approach described in [10], we can extend our implementation to handle polygons with holes by first splitting the polygon into simple subpolygons and then computing the union of these visibility polygons. The same can be considered for link distance in polygons with holes [42, 40].

Another set of applications arises from using our implementation as a critical tool for dealing with more complex optimization problems that require frequent visibility queries as the critical subroutine. An important example is the Watchman Route Problem (WRP), in which the goal is to find a cheapest tour in a (simple or non-simple) polygon P for a watchman along which every point in P can be seen; here "cheapest" may refer to a minimization of total length or correspond to the number of involved edges. (There are numerous other variants, such as using multiple watchmen.) While a minimum-length watchman tour inside a simple polygon can be computed in polynomial time inside a simple polygon, with the best known time bounds [18] being $\mathcal{O}(n^3 \log n)$ for the "anchored" WRP, in which the tour is required to pass through a specified anchor point, and $\mathcal{O}(n^4 \log n)$ for the "floating" WRP, in which no anchor point is specified; this is only efficient in theory, and we are not aware of any practical implementations. Moreover, the problem is NP-hard for non-simple polygons (with an immediate reduction from the TSP), so practical computation requires more involved exact algorithms that combine the computation of shortest tours with set cover, i.e., many segment visibility computations along the way. This resembles practical approaches for computing optimal solutions for instances of the excruciatingly difficult Art Gallery Problem; as previous work has shown, this may actually hinge on the theoretically "easy" visibility computations instead of the "hard" set cover. For example, of the computation time reported by Kröller, Baumgartner, Fekete and Schmidt [36], 80% was spent on (point) visibility computations: "It is obvious that data structure updates and geometric support procedures are – by far – computationally most expensive, whereas solving the LPs has virtually no runtime impact." See [16] for a deeper dive into how provably optimal solutions for polygons with up to thousands of vertices can be computed. We are optimistic that our newly developed implementation for segment visibility will be a milestone towards similar progress for Watchman Problems, as well as many of the ensuing variants.

References

1 Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. The art gallery problem is $\exists\mathbb{R}$-complete. *Journal of the ACM*, 69(1):1–70, 2021. doi:10.1145/3486220.

2 Esther M. Arkin, Joseph S. B. Mitchell, and Subhash Suri. Optimal link path queries in a simple polygon. In *Symposium on Discrete Algorithms (SODA)*, pages 269–279, 1992. URL: http://dl.acm.org/citation.cfm?id=139404.139462.

3 Boris Aronov, Leonidas J. Guibas, Marek Teichmann, and Li Zhang. Visibility queries and maintenance in simple polygons. *Discrete and Computational Geometry*, 27(4):461–483, 2002. doi:10.1007/S00454-001-0089-9.

4 Moritz Baum, Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner. Scalable exact visualization of isocontours in road networks via minimum-link paths. *Journal of Computational Geometry*, 9(1):27–73, 2018. doi:10.20382/JOCG.V9I1A2.

5 Jean-Daniel Boissonnat, Olivier Devillers, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. In *Symposium on Computational Geometry (SoCG)*, pages 11–18, 2000. doi:10.1145/336154.336165.

6 Prosenjit Bose, Anna Lubiw, and J. Ian Munro. Efficient visibility queries in simple polygons. *Computational Geometry*, 23(3):313–335, 2002. doi:10.1016/S0925-7721(01)00070-0.

7 Reilly Browne, Prahlad Narasimham Kasthurirangan, Joseph S. B. Mitchell, and Valentin Polishchuk. Constant-factor approximation algorithms for convex cover and hidden set in a simple polygon. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1357–1365, 2023. doi:10.1109/FOCS57990.2023.00083.

8 Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller. Efficient computation of visibility polygons. *CoRR*, abs/1403.3905, 2014. arXiv:1403.3905.

9 Mojtaba Nouri Bygi and Mohammad Ghodsi. Weak visibility queries in simple polygons. In *Canadian Conference on Computational Geometry (CCCG)*, 2011. URL: http://www.cccg.ca/proceedings/2011/papers/paper95.pdf.

10 Mojtaba Nouri Bygi and Mohammad Ghodsi. Weak visibility queries of line segments in simple polygons and polygonal domains. *CoRR*, abs/1310.7197, 2013. arXiv:1310.7197.

11 Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991. doi:10.1007/BF02574703.

12 Bernard Chazelle. The computational geometry impact task force report: An executive summary. In *Workshop on Applied Computational Geometry (WACG)*, pages 59–65, 1996. doi:10.1007/BFB0014485.

13 Danny Z. Chen and Haitao Wang. Weak visibility queries of line segments in simple polygons. *Computational Geometry*, 48(6):443–452, 2015. doi:10.1016/j.comgeo.2015.02.001.

14 Mark de Berg. On rectilinear link distance. *Computational Geometry*, 1:13–34, 1991. doi:10.1016/0925-7721(91)90010-C.

15 Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008. doi:10.1007/978-3-540-77974-2.

16 Pedro J de Rezende, Cid C de Souza, Stephan Friedrichs, Michael Hemmer, Alexander Kröller, and Davi C Tozoni. Engineering art galleries. In *Algorithm Engineering: Selected Results and Surveys*, pages 379–417. Springer, 2016. doi:10.1007/978-3-319-49487-6_12.

17 Nakju Lett Doh, Chanki Kim, and Wan Kyun Chung. A practical path planner for the robotic vacuum cleaner in rectilinear environments. *IEEE Transactions on Consumer Electronics*, 53(2):519–527, 2007. doi:10.1109/TCE.2007.381724.

18 Moshe Dror, Alon Efrat, Anna Lubiw, and Joseph SB Mitchell. Touring a sequence of polygons. In *Symposium on the Theory of Computing (STOC)*, pages 473–482, 2003.

19 Günther Eder, Martin Held, Steinþór Jasonarson, Philipp Mayer, and Peter Palfrader. Salzburg database of polygonal data: Polygons and their generators. *Data in Brief*, 31:105984, 2020. doi:10.1016/j.dib.2020.105984.

20 Robert Fitch, Zack Butler, and Daniela Rus. 3d rectilinear motion planning with minimum bend paths. In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180)*, volume 3, pages 1491–1498. IEEE, 2001. doi:10.1109/IROS.2001.977191.

21 Submir Kumar Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, 2007.

22 Andrew Gibiansky. Finger trees, 2014. URL: https://andrew.gibiansky.com/blog/haskell/finger-trees/.

23 Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972. doi:10.1016/0020-0190(72)90045-2.

24 Leonidas J. Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert Endre Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987. doi:10.1007/BF01840360.

25 Leonidas J. Guibas, John Hershberger, Joseph S. B. Mitchell, and Jack Snoeyink. Approximating polygons and subdivisions with minimum link paths. *International Journal of Computational Geometry and Applications*, 3(4):383–415, 1993. doi:10.1142/S0218195993000257.

26 Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Symposium on the Theory of Computing (STOC)*, pages 49–60, 1977. doi:10.1145/800105.803395.

27 Mart Hagedoorn and Valentin Polishchuk. Link diameter, radius and 2-point link distance queries in polygonal domains. In *Workshop on Algorithms and Data Structures (WADS)*, pages 34:1–34:15, 2025. ISSN: 1868-8969. doi:10.4230/LIPIcs.WADS.2025.34.

28 Michael Hemmer, Kan Huang, Francisc Bungiu, and Ning Xu. 2D visibility computation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 6.1 edition, 2025. URL: https://doc.cgal.org/6.1/Manual/packages.html#PkgVisibility2.

29 Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006. doi:10.1017/S0956796805005769.

30 Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta informatica*, 17(2):157–184, 1982. doi:10.1007/BF00288968.

31 Hiroshi Imai and Masao Iri. Polygonal approximations of a curve—formulations and algorithms. In *Machine Intelligence and Pattern Recognition*, volume 6, pages 71–86. Elsevier, 1988.

32 Phillip-Raphaël Keldenich, Fabian Kollhoff, Michael Perk, Chek-Manh Loi, and Prahlad Narasimhan Kasthurirangan. Line segment visibility in simple polygons, March 2026. doi:10.5281/zenodo.19194126.

33 Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry*, 40(1):61–78, 2008. doi:10.1016/J.COMGEO.2007.06.003.

34 K. Kolarov and B. Roth. On the number of links and placement of telescoping manipulators in an environment with obstacles. In *International Conference on Advanced Robotics (ICAR)*, pages 988–993 vol.2, 1991. doi:10.1109/ICAR.1991.240543.

35 Irina Kostitsyna, Maarten Löffler, Valentin Polishchuk, and Frank Staals. On the complexity of minimum-link path problems. *Journal of Computational Geometry*, 8(2):80–108, March 2017. doi:10.20382/jocg.v8i2a5.

36 Alexander Kröller, Tobias Baumgartner, Sándor P Fekete, and Christiane Schmidt. Exact solutions and bounds for general art gallery problems. *Journal of Experimental Algorithmics*, 17:2–1, 2012. doi:10.1145/2133803.2184449.

37 D. T. Lee and Franco P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984. doi:10.1002/NET.3230140304.

38 Jyh-Ming Lien and Nancy M. Amato. Approximate convex decomposition of polygons. *Computational Geometry*, 35(1):100–123, 2006. doi:10.1016/j.comgeo.2005.10.005.

39 Anil Maheshwari, Jörg-Rüdiger Sack, and Hristo N. Djidjev. Link distance problems. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 519–558. North Holland / Elsevier, 2000. doi:10.1016/B978-044482537-7/50013-9.

40 Joseph S. B. Mitchell, Valentin Polishchuk, and Mikko Sysikaski. Minimum-link paths revisited. *Computational Geometry*, 47(6):651–667, 2014. Special issue on EuroCG'11. doi:10.1016/J.COMGEO.2013.12.005.

41 Joseph S. B. Mitchell, Valentin Polishchuk, and Mikko Sysikaski. Minimum-link paths revisited. *Computational Geometry*, 47(6):651–667, August 2014. doi:10.1016/j.comgeo.2013.12.005.

42 Joseph S. B. Mitchell, Günter Rote, and Gerhard J. Woeginger. Minimum-link paths among obstacles in the plane. In *Symposium on Computational Geometry (SoCG)*, pages 63–72. ACM, 1990. doi:10.1145/98524.98537.

43 Joseph S. B. Mitchell and Subhash Suri. Separation and approximation of polyhedral objects. *Computational Geometry*, 5:95–114, 1995. doi:10.1016/0925-7721(95)00006-U.

44 Joseph SB Mitchell. Shortest paths and networks. In *Handbook of Discrete and Computational Geometry*, pages 445–466. CRC Publishing, 1997.

45 Joseph SB Mitchell et al. Geometric shortest paths and network optimization. In *Handbook of Computational Geometry*, volume 334, pages 633–702. North Holland, 2000. doi:10.1016/B978-044482537-7/50016-4.

46 Joseph O'Rourke. *Art gallery theorems and algorithms*. Oxford University Press, Inc., USA, September 1987.

47 John H. Reif and James A. Storer. Minimizing turns for discrete movement in the interior of a polygon. *IEEE Journal on Robotics and Automation*, 3(3):182–193, 1987. doi:10.1109/JRA.1987.1087092.

48 Jörg-Rüdiger Sack and Jorge Urrutia. *Handbook of computational geometry*. Elsevier, 1999.

49 Subhash Suri. *Minimum link paths in polygons and related problems*. PhD thesis, The Johns Hopkins University, 1987.

50 Subhash Suri. On some link distance problems in a simple polygon. *IEEE Transactions on Robotics and Automation*, 6(1):108–113, 1990. doi:10.1109/70.88124.

51 Subhash Suri and Joseph O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In Alok Aggarwal, editor, *Proceedings of the Second Annual ACM SIGACT/SIGGRAPH Symposium on Computational Geometry, Yorktown Heights, NY, USA, June 2-4, 1986*, pages 14–23. ACM, 1986. doi:10.1145/10515.10517.

52 Ichiro Suzuki and Masafumi Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM Journal on Computing*, 21(5):863–888, 1992. doi:10.1137/0221051.

53 Mikko Sysikaski. Rectilinear minimum link paths in two and higher dimensions. Master's thesis, University of Helsinki, 2019. Available at https://helda.helsinki.fi/bitstreams/9c8e7b5f-f4ac-42ca-bbc6-bb39a4b24782/download.

54 Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987. doi:10.1137/0216030.

55 The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 6.1 edition, 2025. URL: https://doc.cgal.org/6.1/Manual/packages.html.

56 Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. *Handbook of discrete and computational geometry*. CRC press, 2017.

57 David Phillip Wagner. *Path planning algorithms under the link-distance metric*. Phd thesis, Dartmouth College, 2006. Available at https://digitalcommons.dartmouth.edu/cgi/viewcontent.cgi?article=1015&context=dissertations.

58 Hu Xu, Lei Shu, and May Huang. Planning paths with fewer turns on grid maps. In *Proceedings of the International Symposium on Combinatorial Search*, volume 4, pages 193–200, 2013. doi:10.1609/SOCS.V4I1.18298.