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

Sándor P. Fekete[1,2], Prahlad Narasimhan Kasthurirangan[3], Phillip Keldenich[1], Fabian Kollhoff[1], Chek-Manh Loi[1], and Michael Perk[1]

1 Department of Computer Science, TU Braunschweig
s.fekete@tu-bs.de, {keldenich,kollhoff,loi,perk}@ibr.cs.tu-bs.de

2 L3S, Germany

3 Department of Applied Mathematics and Statistics, Stony Brook University
prahladnarasim.kasthurirangan@stonybrook.edu

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. 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 confirm that the end-to-end runtime 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.

Related Version *Full version and code* [11, 12]

1 Introduction

In this paper, we 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 [22, 21]. 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 [15]. However, to this date, we are unaware of *any* open source implementation. Our implementation has the following 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 addition, we showcase two major use cases 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.

* Supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) as part of project *Computational Geometry: Solving Hard Optimization Problems* (CG:SHOP) - 444569951. Work by PNK was carried out during a research stay at TU Braunschweig.

42nd European Workshop on Computational Geometry, Hagen, Germany, March 25–27, 2026.
This is an extended abstract of a presentation given at EuroCG'26. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear eventually in more final form at a conference with formal proceedings and/or in a journal.

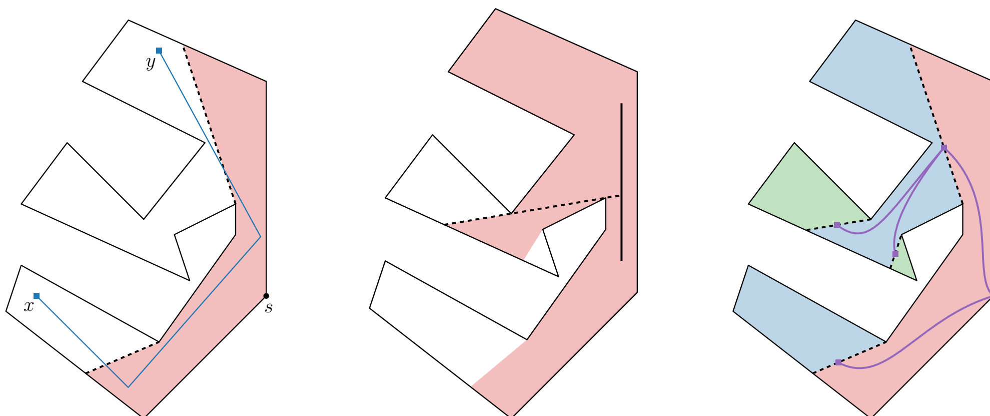

Figure 1 *Left:* $V_P(s)$ is given in red and the windows are marked using dashed lines. A minimum link path between x and y is marked in blue; $d_L(x, y) = 3$. *Middle:* A segment $\ell \in P$ is marked with a solid black line and $V_P(\ell)$ is marked in red. *Right:* The window partition $\mathcal{W}(s)$ 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.

Related Work Due to restricted space, we focus on weak visibility polygons of segments inside simple polygons; the interested reader can find a more general overview of visibility problems in [27, 23, 24, 13, 9]. Given a triangulation of a polygon, Guibas et al. [15] 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 [1, 3, 5, 8, 15]. Implementing these more recent results with improved output sensitive query time with exact predicates seems extremely challenging. It is also unclear if these will lead to improvements in runtime in practice (see Section 3 for experiments).

Computing segment visibility is a crucial subroutine in several geometric data structures; perhaps the most well-known one being window partitions [25]. These can be used to efficiently compute minimum link paths between two points. For a comprehensive overview of minimum link paths, we refer the reader to [18, Chapter 12] and [19].

2 Overview

We implemented the algorithm by Guibas et al. [15] as well as the window partition construction by Suri [25] which allows exact and approximate minimum link path computation. All our implementations use exact number types and handle real-world degenerate cases.

Weak Visibility of a Segment Let P be a simple polygon with n 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 (see Figure 1, *Left*). 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 any point on ℓ, i.e., $V_P(\ell) := \{p \in P \mid \exists s \in \ell : p \in V_P(s)\}$, see Figure 1 *Middle*. Constructing a triangulation $\mathcal{T}$ of a polygon P takes linear time using Chazelle's algorithm [7]. Thus, computing $V_P(\ell)$ for a segment ℓ in P takes linear time – first triangulate P [7], then use the algorithm from Guibas et al. [15], which we describe in the full version [11]. As we are unaware of any implementation of [7], we compute a

constrained Delaunay triangulation (CDT) of P using CGAL [2]. Due to the lack of more direct polygon triangulations or other types of constrained triangulations implemented in CGAL, this is, to the best of our knowledge, the recommended way of obtaining a polygon triangulation using CGAL. Although CDT can incur a worst-case cost of $\mathcal{O}(n^2)$ [4], they perform very well in practice. Our implementation of [15] nevertheless runs in linear time on the computed triangulation. For this runtime, we also implemented a specialized data structure called a *finger tree* [17, 16] (for details see the full version [11]). A finger tree is a balanced search tree augmented with pointers (*fingers*) to its extremal elements, enabling access and splits in $\mathcal{O}(\log \delta)$ time, where δ is the distance to the nearest finger. This yields an amortized constant time per operation in the algorithm and thus overall linear runtime. A simpler alternative is to use a standard balanced binary search tree such as CGAL's `Multiset` (a red-black tree), which supports the same operations in $\mathcal{O}(\log n)$ time, resulting in an $\mathcal{O}(n \log n)$ algorithm. To the best of our knowledge, all previous implementations of finger trees are in functional programming languages like Haskell or Scala [16, 14]; we used C++ for their iterative versions.

Minimum Link Paths and Window Partitions For two points $x, y \in P$, the *link distance* $d_L(x, y)$ 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 the part of P that does not contain s. The *window partition* $\mathcal{W}_P(s)$ is the partition of P obtained by recursively computing the visibility polygons of the windows of $V_P(s)$ within their respective subpolygons $P(w, s)$. The *window tree* T_s of $\mathcal{W}(s)$ is rooted at s and has the windows as its vertices, with an edge between two vertices whenever their regions share a common boundary; see Figure 1, *Right*. Following Suri [25], we use $\mathcal{W}(\cdot)$ to compute exact and approximate minimum link paths; see the full version [11] for details. The approximation algorithm precomputes window partitions from k random seed points and answers a query by finding the best path among all seed trees, guaranteeing an additive error of at most four links. We improved the visibility polygon routine so it runs in time linear in the number of triangles of $\mathcal{T}$ that intersect $V_P(\ell)$, allowing the window-partition construction to run in overall linear time.

3 Experimental Evaluation

In this section, we present the experimental evaluation of our implementations, see Figure 2 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, running Ubuntu Linux 24.04.3 LTS. 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 [12].

3.1 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?

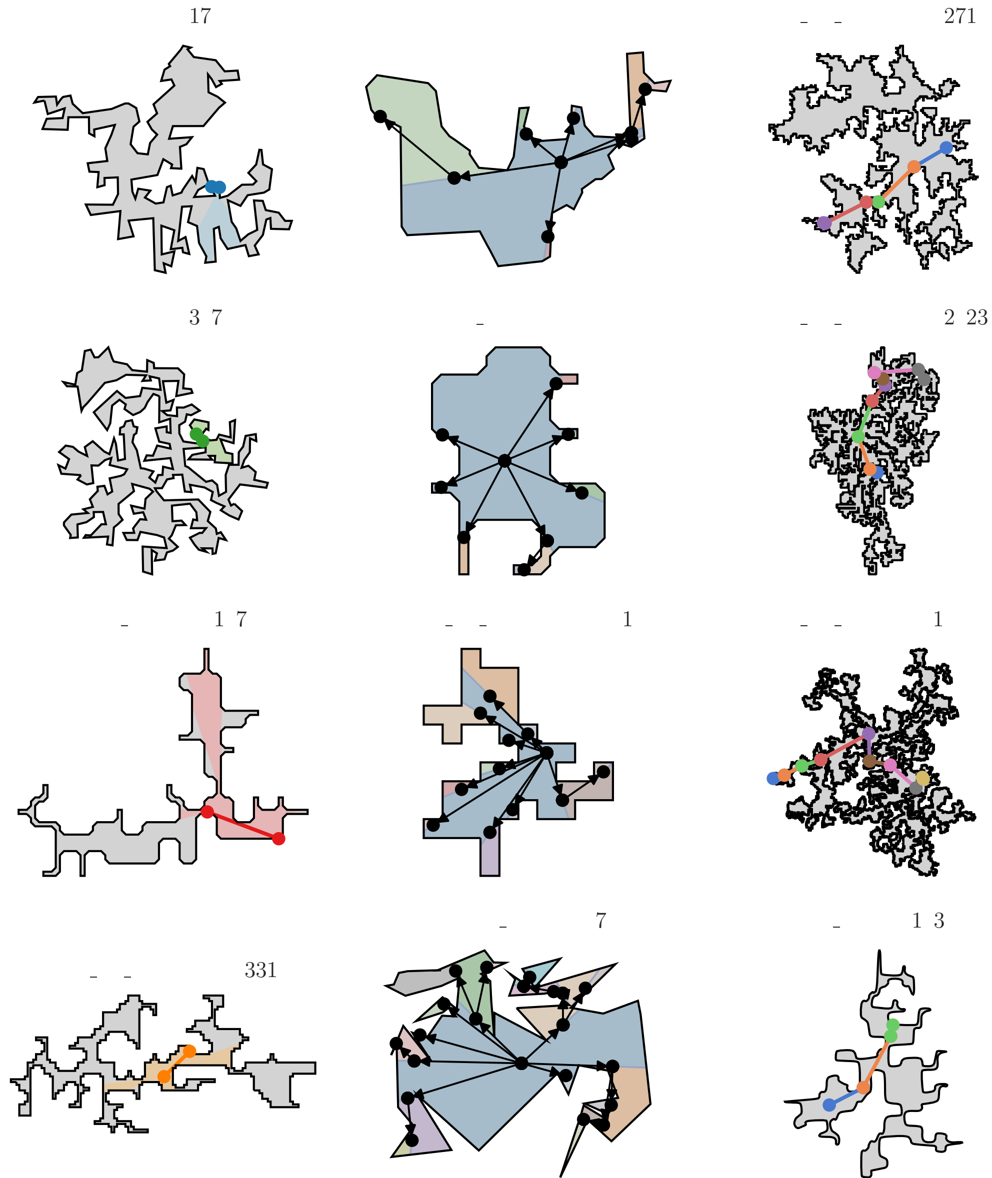

Figure 2 Example outputs. *Left Column:* Weak visibility polygon of a segment. *Middle:* Window partition from a source point s. *Right:* Minimum link path between two points s and t.

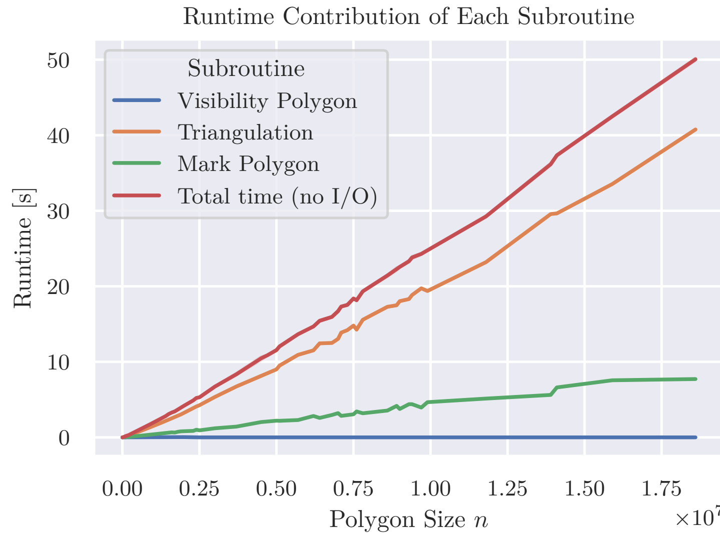

Figure 3 The runtime of different subroutines (triangulating the polygon, marking faces in the polygon, and computing the visibility polygon itself).

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[1] [10], the largest of which has 18 556 888 vertices. 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. 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 3 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 runtime for the visibility computation seems to be almost unaffected by the size of the input polygon because it mostly depends on the size of the output polygon; see Figure 4, *Left*. On our inputs, there is a small but detectable runtime advantage resulting from using a finger tree compared to CGAL's `Multiset`; see Figure 4, *Right*. This is despite the fact that funnels are rarely large; when building a funnel by left insertion as part of a preliminary micro-benchmark, we found that our finger tree implementation already becomes significantly faster at around 15 funnel vertices. This threshold might be reached even earlier if the comparison operation is particularly expensive. In light of this observation, in the remaining experiments, we used our finger tree-based funnel.

[1] We used all valid simple polygons not labeled as "contrived".

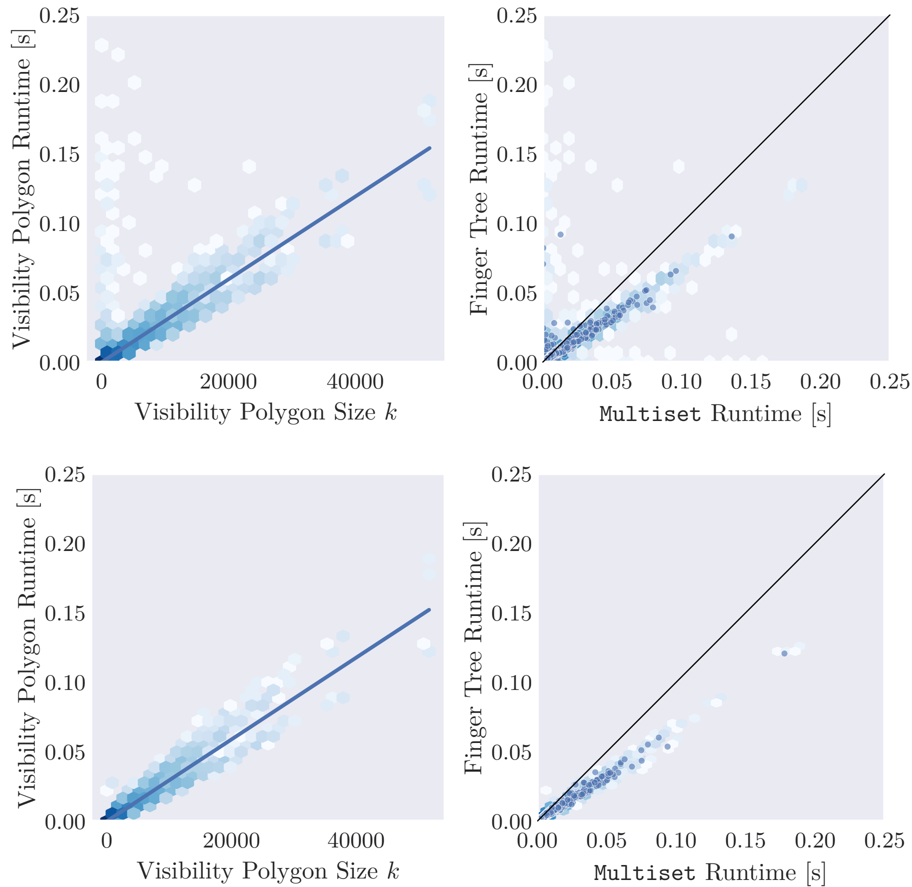

Figure 4 *Top Left:* Runtime dependence on the size of the output visibility polygon, excluding preprocessing. The hexagonal bin plot shows the underlying distribution similar to a scatter plot, for which we have too many data points; darker hues correspond to a higher concentration of data points, the 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 noise in the plot can be attributed to random disturbances such as interrupts, page faults on code pages hitting the network file system, and similar issues; it completely disappears if we drop the worst two out of five runs for each (instance, method) pair.

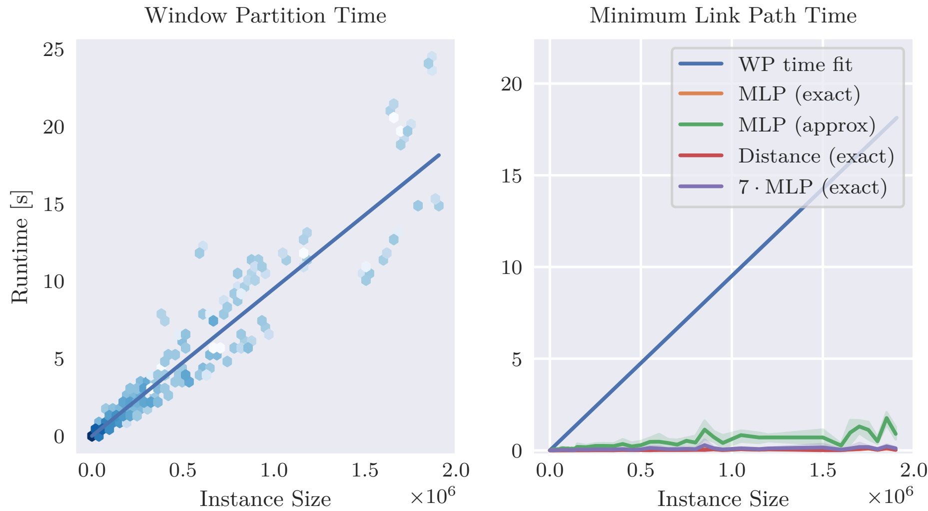

Figure 5 *Left:* The runtime distribution of our window partition implementation, which overall appears to scale linearly, but does have a significant amount of outliers; these may be related to the cost of exact computations arising more commonly in polygons with many collinear vertices. *Right:* Comparison 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 noticeable, but still clearly dominated by the window partition time.

3.2 Minimum Link Paths

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 minimum link path implementation?

RQ6 How does the approximate 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 [10]. 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 compute 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 5. For each path computed using the approximation algorithm, Figure 6 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.

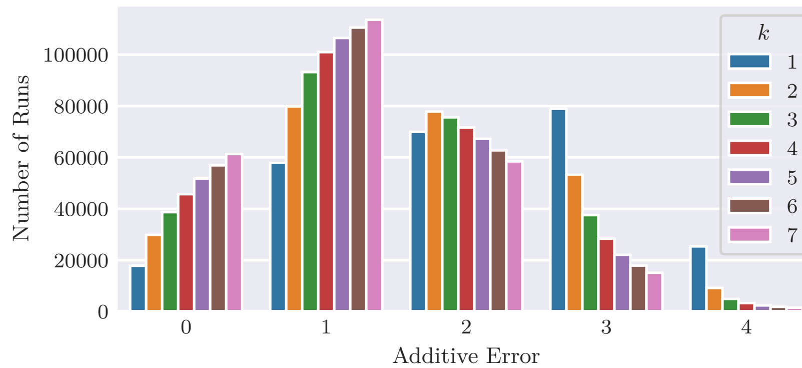

Figure 6 The additive approximation error of our approximate minimum link paths, depending on k, the number of source points used to compute window partitions. In particular, for $k \geq 4$, the majority of queries result in an error of 0 or 1.

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 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.

4 Conclusion

We presented a scalable, robust and exact implementation for computing the weak visibility polygon of a line segment in a simple polygon, which can handle large inputs with more than 18 000 000 vertices on commodity hardware. The bottleneck of the implementation is the computation of a triangulation of the input polygon. We also demonstrated two concrete applications: computing the window partition, and computing the link distance between two points inside a simple polygon. Our experiments confirm that the visibility and link-path computations themselves are efficient and output-sensitive, while the approximate minimum link path algorithm already achieves an additive error of at most 1 for the majority of queries with $k \geq 4$ random source points. For further improvement on the algorithm engineering side, we could parallelize the shortest path tree and window partition implementations.

An immediate generalization is to non-simple polygons. Suri and O'Rourke [26] studied algorithms for computing visibility polygons with holes. Following the approach described in [6], 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 [20, 19].

References

1 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.

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

3 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.

4 Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller. Efficient computation of visibility polygons. 2014. arXiv:1403.3905.

5 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.

6 Mojtaba Nouri Bygi and Mohammad Ghodsi. Weak visibility queries of line segments in simple polygons and polygonal domains. 2013. arXiv:1310.7197.

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

8 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.

9 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.

10 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.

11 Sándor P. Fekete, Prahlad N. Kasthurirangan, Phillip Keldenich, Fabian Kollhoff, Chek-Manh Loi, and Michael Perk. Line segment visibility in simple polygons: Exact, robust, scalable computation and applications. In *Symposium on Compuational Geometry (SoCG)*, 2026. to appear.

12 Sándor P. Fekete, Prahlad N. Kasthurirangan, Phillip Keldenich, Fabian Kolhoff, Chek-Manh Loi, and Michael Perk. Line segment visibility in simple polygons, 2025. doi:10.5281/zenodo.17800685.

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

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

15 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.

16 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.

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

18 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.

19 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.

20 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`.

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

22 Joseph SB Mitchell et al. Geometric shortest paths and network optimization. In *Handbook of Computational Geometry*, volume 334, pages 633–702. North Holland, 2000.

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

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

25 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`.

26 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`.

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