Efficient Computation of Crossing Components and Shortcut Hulls

Nikolas Alexander Schwarz[1](✉) and Sabine Storandt[2]

[1] Christian-Albrecht University of Kiel, Kiel, Germany
nsch@informatik.uni-kiel.de

[2] University of Konstanz, Konstanz, Germany

Abstract. Polygon simplification is an important building block in many geovisualization algorithms. Recently, the concept of shortcut hulls was proposed to obtain a simplified polygon that fully contains the original polygon. Given a set of potential shortcuts between the polygon vertices, the computation of the optimal shortcut hull crucially relies on identifying edge crossings among the shortcuts and computing so called crossing components. In this paper, we present novel algorithms to significantly accelerate these steps. For a simple polygon P with n vertices and a set of shortcuts $\mathcal{C}$, we describe an algorithm for computing all edge crossings in $\mathcal{O}(n + m + k)$, where $m := |\mathcal{C}|$ and k is the number of crossings. This output-sensitive algorithm is clearly optimal and a significant improvement over general-purpose algorithms to identify edge crossings. Furthermore, we extend this algorithm to compute the crossing components in $\mathcal{O}(\min\{n + m + k, n^2\})$. As k could potentially be up to $\Theta(n^4)$, this is a significant speed-up if only the crossing components are needed rather than each individual crossing. Finally, we propose a novel crossing component hierarchy data structure. It encodes the crossing components and allows to efficiently partition the polygon based thereupon. We show that our novel algorithms and data structures allow to significantly reduce the theoretical running time of shortcut hull computation.

Keywords: Polygon Simplification · Edge Crossings · Intersection Computation · Hierarchical Data Structure

1 Introduction

Identifying crossings of line segments is a classic problem in the field of computational geometry with many applications in visual computing and computer graphics, including ray shooting [14], time-series analysis and visualization [25], as well as graph clustering and rendering [8]. Shamos and Hoey [23] were the first to describe an algorithm that decides whether there is at least one crossing in an arrangement of line segments. Given m line segments, they also show that the computational lower bound for this problem is $\Omega(m \log m)$. Bentley and Ottmann extended the algorithm to count and report all crossings [3]. Later, their algorithm was described in more detail and with regard for degenerate cases with a running time of $\mathcal{O}(m \log m + k \log m)$ for k crossings [4]. An optimal algorithm with a running time of $\mathcal{O}(m \log m + k)$ was given in [7]. However, for special cases algorithms with better performance are possible. For example, Eppstein et al. [12] showed that in a given geometric graph with n nodes and m edges with the number of edge crossings k being sublinear in n by an iterated logarithmic factor, the crossings can be computed in time $\mathcal{O}(n + k \log n)$. In [10], it was proven that if the line segments connect points that are all arranged on a circle and the cyclic sequence of the points is given, the segment crossings can be computed in linear time. They use this algorithm to efficiently compute optimal circular right angle crossing graph drawings, where all nodes are located on a circle and edge crossings have an angle of $\pi/2$.

In this paper, we focus on the following setup: Given a polygon P with n vertices, and a set of line segments that connect vertices of P, assess their crossing structure. Here, the line segment set is either contained exclusively in the interior of P or exclusively in its exterior. Thus, the line segments are a subset of the interior or exterior visibility graph of the polygon vertices. This poses a generalization of the setup discussed by Dehkordi et al. [10]. The intersection structure of these visibility graphs plays an important role, e.g., in illumination computation [24] or in motion planning in polygonal environments [2]. Furthermore, a novel method for polygon simplification was recently proposed that also heavily relies on crossing computation in such line segment arrangements [5]. The resulting simplification is called a *shortcut hull*. It can be interpreted as a generalization of convex hulls, which allows to smoothly adjust the level of detail via an input parameter λ. Figure 1 depicts shortcut hulls for an example polygon with varying λ values. The shortcut hull algorithm receives as input a polygon P and a set of potential simplification edges between the polygon vertices, so-called shortcuts. The shortcut set $\mathcal{C}$ is a subset of the exterior visibility graph of the polygon. The goal is to select a sequence of crossing-free shortcuts that form a closed walk around the polygon. We will show that our newly developed algorithms significantly reduce the running time to compute shortcut hulls.

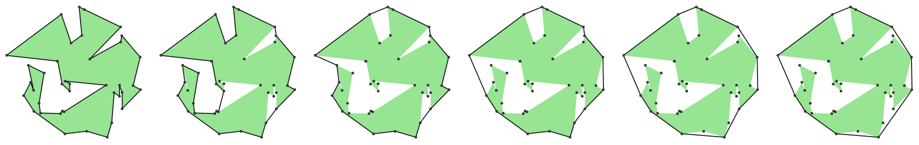


Fig. 1. Input polygon (green) and shortcut hulls (black outline) with varying level of detail. The right one coincides with the convex hull of the polygon. (Color figure online)

1.1 Related Work

The number of crossings in line segment arrangements and their intersection structure plays an important role in many application realms in computational geometry, visualization, and graph theory. A prominent example is graph drawing, where the typical objective is to find a drawing of the input graph that minimizes edge crossings [11,21,22]. Drawings with fewer intersections are easier to read and convey the structure of the input graph better to the viewer. It has been established that for large amount of edges m the lower bound for the amount of edge crossings is proportional to m^3/n^2. This is known as the crossing lemma. The respective coefficient has been refined over time [1,18,19]. There are also many graph layouts that restrict node position. Well-studied examples are layouts where all nodes must be on the outer face, which includes the special case of circular layouts [16]. This is closely related to our setup, where nodes need to coincide with the vertices of a polygon. The class of outerplanar graphs contains the graphs that admit a planar (that is, crossing-free) drawing with all nodes being placed on the outer face [6]. For non-outerplanar graphs, crossings are enforced. Here, the goal is often to maximize the crossing angle in addition to minimizing the number of crossings to still ensure readability [9].

In a visibility graph, the nodes represent geometric objects, and edges encode visibility between the objects. The visibility graph of a simple polygon contains all edges between polygon vertices that do not intersect the exterior of the polygon. Hershberger [15] presented an algorithm to construct such a visibility graph in $\mathcal{O}(m)$ where $m \in \mathcal{O}(n^2)$ denotes the number of edges. For polygons with holes, Overmars and Welzl proposed an algorithm to compute the visibility graph in $\mathcal{O}(m \log n)$ time [17]. Pocchiola and Vegter [20] improved the algorithm to run in $\mathcal{O}(m+n \log n)$ using the notions of pseudotriangulation and visibility complexes.

1.2 Contribution

We provide several new results regarding crossing computation in line segment arrangements. Given a polygon P with n vertices, and a set of m line segments that are contained in the interior (or exterior) of P and whose endpoints are vertices of P, we design an algorithm that reports all k crossings in $\mathcal{O}(n+m+k)$. This is clearly optimal and faster than the general algorithm by a factor logarithmic in m. Moreover, we propose a new method for crossing component computation. A naive way to obtain these components is to compute the intersection graph G_I and then extract its connected components. But as the number of segments I can be quadratic in n and any pair of segments might cross, G_I can have a size of up to $\Theta(n^4)$. Our new algorithm computes a smaller graph whose connected components are equal to those of G_I. We refer to this graph as the pseudo-intersection graph G_P. We show that the size of G_P is bounded by $\mathcal{O}(n^2)$ which is directly reflected in its construction time. Thus, our new algorithm is vastly faster, especially for large inputs with many crossings.

While these results are interesting on their own, we also show their importance for efficient shortcut hull computation. The construction algorithm proposed by Bonerath et al. [5] uses the crossing components to derive a suitable partition of the polygon's interior. However, in [5], the crossing components and the respective partition were assumed to be given, and thus their efficient computation was left as an open problem. We develop a tailored data structure, called *crossing component hierarchy* that allows to process the crossing components very efficiently and present an improved theoretical running time for the complete shortcut hull algorithm based on our novel results.

2 Preliminaries

Throughout the paper, we always assume to be given a simple polygon P as input. P is defined as a closed chain of line segments that connect an ordered set of points $p_1, \dots, p_n$. We will refer to these points also as polygon vertices and use $N := \{1, \dots, n\}$ as their index set. Everything enclosed by the polygon is called the interior of P and everything outside is called the exterior of P.

Two polygon vertices p_i, p_j are visible from each other inside of P if the line segment $\overline{p_i p_j}$ does not intersect the exterior of P. The visibility graph $\mathrm{Vis}(P) = (N, E)$ of P contains a node $i \in N$ for each polygon vertex p_i and edges $\{i, j\} \in E$ for all mutual visible point pairs p_i, p_j. Two edges cross each other if a curve from one endpoint to the other cannot be drawn without a intersection (except for their end points touching each other) when respecting the constraint of not intersecting the exterior of P. Note that for points in general position this is equivalent to their line segment representations having a point in common apart from their end points. We will use the terms crossing and intersection interchangeably in this paper (as it is the case in most existing literature on the topic). Note, that we can also define an exterior visibility graph by demanding that line segments connecting polygon vertices do not intersect the interior of P. Given a set of line segments, the respective intersection graph contains a node for each segment, and edges between segment nodes if the respective pair of segments crosses. In this paper, we are interested in intersection graphs induced by a subset of the visibility graph edges.

3 Shortcut Hull

As a showcase application of our improved algorithms for crossing detection, we consider the computation of shortcut hulls [5]: Given a polygon P, a shortcut hull Q is a non-crossing polygon that encloses P. The boundary segments of Q, also called shortcuts, have to be part of the exterior visibility graph of P. One can use the full exterior visibility graph or a subset $\mathcal{C}$ of shortcut candidates. The cost $c(Q)$ of Q depends on its perimeter $\beta(Q)$, its area $\alpha(Q)$, and a user-defined parameter $\lambda \in [0, 1]$: $c(Q) = \lambda \cdot \beta(Q) + (1 - \lambda) \cdot \alpha(Q)$. The goal is to find the shortcut hull of minimum cost for given λ. Bonerath et al. present an exact algorithm for shortcut hull computation [5] with the following main steps:

1. Construct a box that fully contains P (with some wiggle room) and connect its top left corner to the uppermost point of P with an edge. The weakly-simple polygon that consists of the box, the inserted edge, and P, is called the sliced donut $\mathcal{D}$.

2. Compute an enrichment $\mathcal{C}^+ \supset \mathcal{C}$ with $\mathcal{C}^+$ being a subset of the interior visibility graph of $\mathcal{D}$. For every set $\mathcal{C}'$ of pair-wise non-crossing edges in $\mathcal{C}$, the enrichment needs to contain a triangulation of $\mathcal{D}$ that is a superset of $\mathcal{C}'$.
3. Use a dynamic program that iterates through triangles in $\mathcal{C}^+$ to find the cost-optimal shortcut hull $Q \subset \mathcal{C}$ with respect to λ.

By Bonerath et al.'s analysis [5], the above algorithm has a running time of $\mathcal{O}(n^3)$. More precisely, a running time of $\mathcal{O}(h\chi^3 + \chi n)$ is obtained where $\chi \in \mathcal{O}(n)$ denotes the spatial complexity of the regions and $h \in \mathcal{O}(n)$ the amount of regions induced by the crossing components of $\mathcal{C}$. A region is defined as the minimal polygon that encloses a crossing component and respects the boundaries of P and the spatial complexity is the maximum over the amount of vertices of each region. It is proven that for any $\mathcal{C}$, an enrichment $\mathcal{C}^+$ of size $\mathcal{O}(h\chi^2 + n)$ exists and that it can be computed in $\mathcal{O}(h\chi^3 + \chi n)$. This step dominates the overall running time. However, the algorithm relies on the availability of the crossing components and the regions induced by them. Computing the crossing components naively takes $\mathcal{O}(n^4)$ and thus constitutes the most expensive part of the algorithm. But as the crossing components are simply assumed to be given, the respective running time is not factored in. We present a novel method for crossing component computation in Sect. 5, and an efficient method to compute the enclosing polygons, i.e. regions in Sect. 6.

Furthermore, we propose the following modification to make the overall algorithm simpler to implement and more efficient in practice: The sliced donut, as computed in step 1, is not necessary for the algorithm to work. Instead, one can simply compute the convex hull of P in $\mathcal{O}(n)$ using e. g. the algorithm by Graham [13]. Then the algorithm can be run separately for each pocket, that is, every polygon enclosed by convex hull segments and the boundary of P as illustrated in Fig. 2. This also allows for solving the subproblems in parallel.

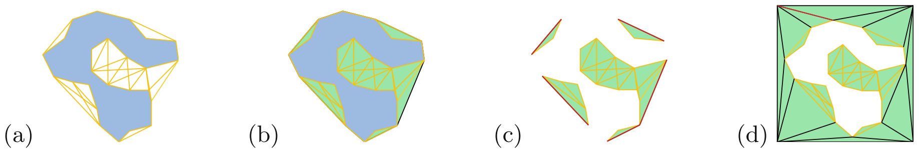


Fig. 2. Segmentation of a polygon P into smaller problems. (a) P with shortcut candidates $\mathcal{C}$. (b) Convex hull of P. (c) Our approach: pockets of the convex hull. (d) Sliced donut approach from [5] for comparison.

4 Computation of Edge Crossings

In this section, we present an optimal algorithm for computing crossings in the interior visibility graph of a polygon. Given a simple polygon with an index set $N = \{1, 2, ..., n-1, n\}$ and a subset of the visibility graph's edges $\mathcal{C}$ with $|\mathcal{C}| = m$, we want to compute and report all k intersections among the segments in $\mathcal{C}$. See Fig. 2a for an example of a possible input.

From here on we will consider all edges as *directed* in topological order, i.e. for any $(u, v) \in \mathcal{C}$ it holds that $u < v$. Furthermore, we will assume that the neighborhoods of a vertex v, i.e. the targets of outgoing edges $N^+(v)$ and the sources of incoming edges $N^-(v)$ are represented as adjacency lists with vertices sorted in descending order. We will also use them like stacks, such that `top()` gets the last element in the list and `pop()` deletes the last element in the list. If the adjacency lists are not sorted, we can easily sort them with an algorithm similar to counting sort in $\mathcal{O}(n + m)$.

Given two edges that cross each other, one index of the second edge must lie between the indices of the first edge and the other one must lie outside. Based on that, we make the following observation:

Observation 1. *Given two edges $e = (a, b) \in C$ and $e' = (u, v) \in C$ with $u < a$, the following characterization holds: $a < v < b \iff e$ and e' cross*

Using this observation, we design algorithm that finds all crossings. For this purpose, we take a look at the adjacency matrix, see Fig. 3. Observe that if we fix an edge $e' = (u, v)$, then all edges in the rectangle to the lower right will fulfill Observation 1. In coordinates, this is the rectangle $[u + 1, v - 1] \times [v + 1, n]$. In the example in Fig. 3 this rectangle and the corresponding edge $e' = (1, 5)$ are marked in green and blue in the matrix.

So we need an algorithm that iterates over all edges and for each edge traverses the edges in the corresponding rectangle. We do this column-wise from left to right and traverse each column from bottom to top. After we have visited an edge, we delete it. This simplifies traversing each rectangle because we will no longer have to worry about its left-hand border as all edges to its left will already have been deleted by the time we traverse it. Since constructing and traversing an adjacency matrix is computationally expensive, we simulate the method we just described with adjacency lists. Each column can be traversed using the adjacency lists for incoming edges and each rectangle can be traversed using the adjacency lists in the range $[u + 1, v - 1]$ for outgoing edges. The links between list elements are shown in red in Fig. 3, right. Furthermore, we maintain a doubly-linked list that contains all non-empty adjacency lists for outgoing edges. This allows us to skip empty adjacency lists and avoid overhead. But since we also need constant access to each element, we do not store this doubly-linked list in a conventional way. Instead, we simulate it using two arrays `next` and `prev` of size n. Before we run our algorithm, we fill those two arrays such that `next[v]` and `prev[v]` contain the next and previous index u with $N^+(u) \neq \emptyset$ if $N^+(v) \neq \emptyset$, and -1 if $N^+(v) = \emptyset$.

Based on Observation 1 and the described approach to compute the intersections by a sweep over the edges, we arrive at our first main result.

Theorem 1. *The algorithm computes all k crossings in time $\mathcal{O}(n + m + k)$.*

5 Computation of Crossing Components

We now present a novel algorithm for computing crossing components. Given a candidate segment set $\mathcal{C}$ of size m, which as before needs to be a subset of the

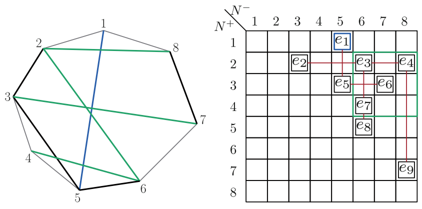


Fig. 3. Subset of a polygon's visibility graph with corresponding adjacency matrix

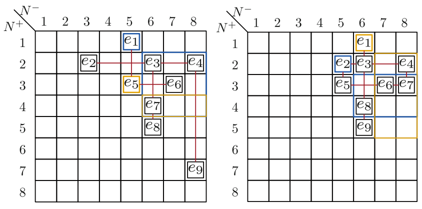


Fig. 4. Adjacency matrix with rectangles that allow for optimization of the algorithm

polygon's visibility graph, the naive algorithm computes the intersection graph G_I of $\mathcal{C}$ and then extracts its connected components. However, this algorithm has a running time of $\mathcal{O}(m^2)$. As m might be quadratic in n, the running time amounts to $\mathcal{O}(n^4)$, which is impractical.

We will show that we can compute the crossing components way more efficiently by constructing a so called pseudo intersection graph G_P instead of G_I, which is much smaller than G_I but has the same connected components. To obtain G_P, we need the following lemma:

Lemma 1. *Given three edges $\mathfrak{e} = (a, b) \in C$, $e_1 = (u_1, v_1) \in C$ and $e_2 = (u_2, v_2) \in C$ with $u_1 < a$, $u_2 < a$ and $a < v_1 \leq v_2$, the following implication holds: e_2 and $\mathfrak{e}$ cross $\Rightarrow$ e_1 and $\mathfrak{e}$ cross*

Proof. Let e_2 and $\mathfrak{e}$ intersect. $a < v_2 < b$ follows from Observation 1. According to prerequisites, $v_1 \leq v_2$ holds, therefore $v_1 < b$ follows. $a < v_1$ already holds according to the prerequisites. Thus, it follows from Observation 1 that e_1 and $\mathfrak{e}$ intersect.

Now, we will again take a look at the matrix representation. Observe in Fig. 4 that the yellow rectangle belonging to e_5 is a subset of the blue rectangle belonging to e_1. This is a direct consequence of Lemma 1. It also implies that all these edges are part of the same crossing component. We can exploit this as follows: When traversing a column from bottom to top, we find the first edge whose corresponding rectangle is not empty. From here on, we connect all edges in the column in the pseudo intersection graph G_P. When reaching the last (i.e. topmost) edge in the column, we report all edges in its corresponding rectangle as intersections. This works because all of the edges in this rectangle must belong to the same crossing component and it is a superset of all the rectangles that came before in this column.

The improvement we just described will already save us many intersection reports, but there are still plenty of edges that are visited multiple times. So we still need to improve this to construct a truly sparse graph and improve the running time asymptotically. Observe that we report many redundant edges when we traverse a row of a rectangle if that row has already been traversed as part of a previous rectangle, because these edges are already part of the same connected component in G_P, see Fig. 4. So instead of reporting all edges in such a row, it suffices to only report the first edge.

Theorem 1. *The construction yields a graph G_P whose connected components are identical to the connected components of the intersection graph G_I.*

Once we have computed the pseudo intersection graph G_P, we use a graph search algorithm, such as BFS or DFS, to identify all connected components. Since G_P has m vertices and at most $\min\{k, n^2\}$ edges, this can be done in $\mathcal{O}(m + \min\{k, n^2\})$.

Theorem 2. *Crossing components can be computed in $\mathcal{O}(\min\{n + m + k, n^2\})$.*

This result is noteworthy, because it means that even for very large amounts of edge crossings $k \in \omega(n^2)$ the algorithm is still bounded by $O(n^2)$. Such large amounts of crossings are not unlikely, as the crossing lemma tells us that $k \in \Omega(m^3/n^2)$ is possible for sufficiently large m [1]. So k can be in the order of $\Theta(n^4)$ if $m \in \Theta(n^2)$. Nevertheless, our algorithm only has a quadratic running time in this case, and an even better one in case k is small.

6 The Crossing Component Hierarchy

Our improved methods for crossing detection and for crossing component computation are crucial for the efficient computation of shortcut hulls. Unlike in the previous sections, we look at the exterior of the polygon. However, this is equivalent because all the relevant edges are in the interior of the pockets formed by the convex hull of the polygon.

However, the crossing components $C_1, \ldots, C_h$ themselves are not sufficient. We also need to compute for each C_i the smallest subset of vertices of P that avoids conflicts with the polygon's boundary and forms an enclosing polygon for C_i. These are referred to as *regions*. In this section, we describe how to accomplish this. We first present a basic algorithm and then introduce a data structure for acceleration, which we call the crossing component hierarchy.

6.1 Basic Algorithm

Our goal is now to compute the boundary of each crossing component C_i. As a first step, we create a list X_i for each crossing component C_i that contains the component vertices in ascending order. This can be achieved with a counting sort like procedure in linear time. However, the polygon described by X_i does not coincide with the component boundary because the areas defined by the crossing component vertices can intersect with the input polygon's boundary, see Fig. 5a. Those areas might also intersect each other as shown in Fig. 6. As defined by Bonerath et al. [5], the boundary of a region is the shortest path (measured by vertex count) within the visibility graph around the crossing component. So if the full visibility graph is available, we can easily trace the boundary on it. But since computing the visibility graph might take time quadratic in n, we now describe a different way to compute the boundary.

We consider each consecutive pair a, b in X_i separately. (Since X_i represents a polygon, the last and first entry also form a consecutive pair.) A path from a to b that respects the aforementioned properties must be a convex chain formed by the polygon vertices. We first define a search space by giving the two most extreme forms this path can take: The lower bound is the line segment from a to b. If it does not intersect the polygon boundary, this is the solution. An upper bound can be constructed as follows: Consider the edges (a, a') and (b, b') in the crossing component with the smallest angle to the line segment $\overline{ab}$. Starting at a, we can follow the edge until we reach an intersection, then follow that edge and repeat this process until we reach b. This must be possible without encountering a polygon point because otherwise the crossing component would fall apart at such a point. The path we just traced is a conflict-free path through the plane because it follows visibility edges and as such does not intersect any polygon edges. It is also a convex chain and as such it forms an upper bound for the convex chain through polygon vertices we want to obtain.

We need to find an optimal conflict-free path within this search space. To get an intuitive idea of what this path has to look like, we first give a physical experimental method to obtain it: Span a rubber band from a to b tracing the upper bound described above, fixing it at the edge intersections using pins. Now remove the pins, only keeping the rubber band fixed at a and b. The rubber band will snap into the optimal conflict-free configuration.

Now, we need to solve this problem computationally: First, we compute a polyline of points between each consecutive pair a, b of all points within the area defined by the vertices of the crossing component, see Fig. 5b. To do this, we traverse all indices between each pair and check for each two indices $i \in \{a, a+1, ..., b-2, b-1\}$ and $j = i + 1$ if the line segment formed by them enters or exits the area defined by the crossing component vertices. Here, we maintain which entrance occurs closest to a and which exit occurs closest to b. The polyline is then formed by these two intersections and all vertices between them. (Note here that we only include the intersections to avoid self-intersections of the polyline. This makes the following step easier to compute. The final result can only contain polygon vertices.) For the smallest and largest index, which also form a pair a, b with $a > b$, this simply wraps around, i.e., once we reach n, we continue with 1. Subsequently, we compute a convex chain along each polyline, see Fig. 5c. The following lemma shows the correctness of this approach.

Lemma 2. *Given a pair of two consecutive points of the crossing component boundary a, b (i.e. b follows directly after a in clockwise order around the component), only points between*[1] *them are relevant for computing the convex chain.*

[1] As a reminder: As mentioned in the paragraph above Lemma 2, this refers to the indices $\{a, a+1, ..., n, 1, ...b-1, b\}$ if $a > b$.

Proof. We have previously defined an upper bound of the search space. All vertices not between a and b lie behind this upper bound, so they are not relevant for the optimal convex chain.

Doing this for all crossing components yields a running time of $\mathcal{O}(nh)$ in total. This is because in the first step we visit exactly n line segments for each crossing component and perform one intersection check for each of them. For the second step, the polylines around one crossing component can at most be of length n in total. Since we can compute the convex chain of each of them in linear time, this also yields $\mathcal{O}(nh)$ for all crossing components.

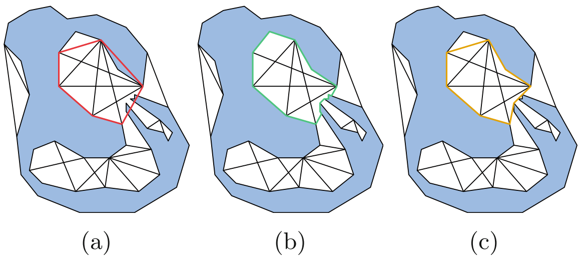


Fig. 5. Steps of isolating a region. (a) shows the crossing component's vertices in order with conflicts. In (b) the conflicts are resolved and in (c) each path is optimized.

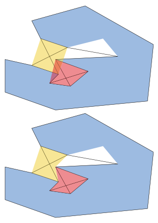


Fig. 6. Crossing components with overlap (top) and how it is resolved (bottom)

6.2 Improved Algorithm

The basic algorithm described above already yields an improvement over the naive approach especially for polygons with many visibility edges but few crossing components. However, the number of crossing components h could potentially be linear in n. In this case, the basic algorithm exhibits a quadratic running time. To get an improved running time in practice, we propose the *crossing component hierarchy* data structure.

Observe that crossing components cannot extend beyond each other. So given a pair of neighboring vertices from a crossing component, we can simply skip over other crossing components to speed up the computation. To do this efficiently, we will first need to bring them into a useful order. Note that if we represent each crossing component as an open interval $]a, b[$ of the lowest and highest vertex index it contains, then the subset relation induces a hierarchy. More precisely, this is the Hasse diagram of the partially ordered set of crossing components $(S, \subseteq)$ where S contains all crossing components represented as intervals. We will now take a closer look at the properties of this Hasse diagram.

Lemma 3. *For two crossing components $C, D \in S$, the following property holds:* $C \nsubseteq D \wedge D \nsubseteq C \Rightarrow C \cap D = \emptyset$

Proof. Let $C, D \in S$ with $C \nsubseteq D \wedge D \nsubseteq C$. Assume that $C \cap D = O \neq \emptyset$. Then at least one edge of each crossing component would have to extend into O creating a crossing. But then the two would not be separate crossing components in the first place. ↯

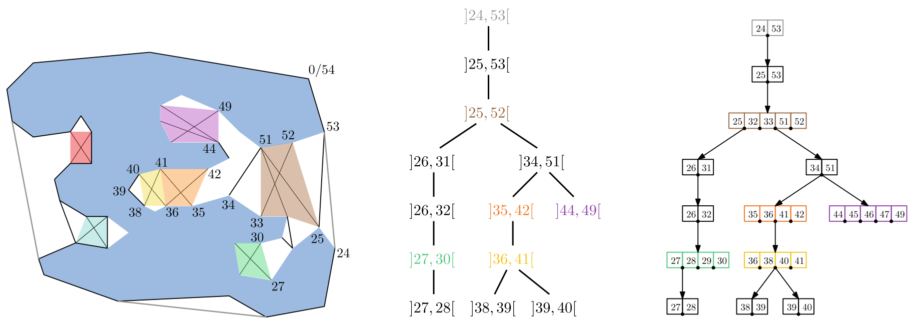
This means that the interval representations of two crossing components cannot overlap each other. (Note that this is only true for the interval representations. The areas defined by them can indeed overlap.) This also means that two unrelated interval representations cannot have a common subset. For the structure of the Hasse diagram, this implies that it is a forest. If we also include the interval for the convex hull pocket, we will obtain a tree. For an example, see Figs. 7a and 7b. Now we need to construct the Hasse diagram efficiently. We proceed as follows:

- Sort S in descending order by the upper bounds of the intervals using counting sort.
- Sort S again in ascending order by the lower bounds of the intervals.

After applying the sorting method described above, S is the pre-order traversal of its Hasse diagram. This is because if we consider an interval representation of a crossing component $]a, b[$, all of its subsets come next in S because for any subset $]c, d[$ it holds that $a \leq c \leq d \leq b$. These correspond exactly to its sub-tree. For any other intervals $]c, d[$, it is $c \leq a$ such that it comes before $]a, b[$ in S or $c \geq b$ such that it comes after all subsets of $]a, b[$ in S.

From the pre-order traversal, we can easily reconstruct the Hasse diagram. We also maintain the order of siblings in this reconstruction. We can slightly extend this tree as follows: For each node, we list all indices that are part of the crossing component in order. Then, each child can be associated with a consecutive pair of indices between which its interval representation lies. This can easily be done in a merge-like manner. The resulting structure is very similar to a B-tree. We call this structure the *crossing component hierarchy*. An example is given in Fig. 7c. This entire procedure can be done in $\mathcal{O}(n)$. Once we have obtained the crossing component hierarchy, we can compute the polylines between each consecutive pair of indices as described above. But instead of iterating over all indices in between we can skip the indices that are in the intervals of the child node. For the polyline that must be computed for the pair consisting of the smallest and largest index, we can instead use its parent. Here, we need to be a bit careful, though, as parts might peak into these intervals and these conflicts also need to be resolved.

Running Time. A precise running time cannot be given here as it depends on the precise arrangement of candidate edges. The worst case scenario is still $\mathcal{O}(nh)$, but in cases where all multi-edge crossing components are separated from each other by at least one single-edge crossing component, this results in a running time of $O(n)$. We can also parametrize the running time as $O(nd)$, where d is the maximum amount of multi-edge children of a node in the crossing



(a) Polygon with candidate set and crossing components (b) Crossing component intervals in a Hasse diagram (c) Crossing component hierarchy data structure

Fig. 7. Crossing components and their representations for the convex hull pocket on the right of the polygon

component hierarchy. This is because we have to traverse the space between them at most d times. In practice, we expect d to be much smaller than h.

6.3 Improved Shortcut Hull Computation

Bonerath et al. [5] describe that their algorithm traces the decomposition trees of the regions' triangulations. If we rebuild the crossing component hierarchy from the regions that we just computed, it has the same structure as these decomposition trees. This means that we can adapt Bonerath et al.'s algorithm to run directly on the crossing component hierarchy. A huge benefit of this is that it makes it unnecessary to compute a constrained triangulation before running the algorithm.

Running Time for Shortcut Hull Computation. For the algorithm itself, our adapted approach of running it on the crossing component hierarchy can bring a small speed-up, but the asymptotic analysis doesn't change from what was given by Bonerath et al., i.e. $\mathcal{O}(n^3)$ in the worst case, more precisely $\mathcal{O}(h\chi^3 + \chi n)$ [5]. However, contemporary approaches to do the required pre-computations have a running time in $\mathcal{O}(n^4)$ in a worst case scenario. With our novel approach, we brought this down to $\mathcal{O}(n^2)$, such that the entire computation can adhere to the running time of $\mathcal{O}(n^3)$.

7 Conclusions and Future Work

We have developed algorithms and methods that can take care of the pre-computations required for Bonerath et al.'s [5] algorithms for shortcut hulls. As part of this, we came up with a new output-sensitive algorithm that can find all edge crossings in linear time given a polygon and a subset of its visibility graph that also has the advantage of not being prone to numerical errors. The algorithm might also be useful for applications beyond shortcut hulls. Furthermore, we extended the algorithm to compute crossing components without finding all crossings k. For very large k, this is very useful because it limits the running time to $\mathcal{O}(n^2)$. While this is a significant improvement, it is not clear whether this is optimal. A trivial lower bound is $\Omega(m)$. In future work, this gap could be narrowed. Moreover, it would be interesting to implement the shortcut hull algorithm with our novel building blocks and to study its performance on real-world inputs.

References

1. Ackerman, E.: On topological graphs with at most four crossings per edge. Comput. Geom. **85**, 101574 (2019). https://doi.org/10.1016/j.comgeo.2019.101574. https://www.sciencedirect.com/science/article/pii/S0925772119301154
2. Belta, C., Isler, V., Pappas, G.J.: Discrete abstractions for robot motion planning and control in polygonal environments. IEEE Trans. Rob. **21**(5), 864–874 (2005)
3. Bentley, Ottmann: Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput. **C-28**(9), 643–647 (1979). https://doi.org/10.1109/TC.1979.1675432
4. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer, Germany (2008). https://doi.org/10.1007/978-3-540-77974-2
5. Shortcut hulls: vertex-restricted outer simplifications of polygons. Comput. Geom. **112**, 101983 (2023). https://doi.org/10.1016/j.comgeo.2023.101983. https://www.sciencedirect.com/science/article/pii/S0925772123000032
6. Chartrand, G., Harary, F.: Planar permutation graphs. In: Annales de l'institut Henri Poincaré. Section B. Calcul des probabilités et statistiques, vol. 3, pp. 433–438 (1967)
7. Chazelle, B., Edelsbrunner, H.: An optimal algorithm for intersecting line segments in the plane. J. ACM **39**(1), 1–54 (1992). https://doi.org/10.1145/147508.147511
8. Cui, W., Zhou, H., Qu, H., Wong, P.C., Li, X.: Geometry-based edge clustering for graph visualization. IEEE Trans. Vis. Comput. Graph. **14**(6), 1277–1284 (2008)
9. Dehkordi, H.R., Eades, P.: Every outer-1-plane graph has a right angle crossing drawing. Int. J. Comput. Geom. Appl. **22**(06), 543–557 (2012)
10. Dehkordi, H.R., Eades, P., Hong, S.H., Nguyen, Q.: Circular right-angle crossing drawings in linear time. Theor. Comput. Sci. **639**, 26–41 (2016)
11. Eades, P., Wormald, N.C.: Edge crossings in drawings of bipartite graphs. Algorithmica **11**, 379–403 (1994)
12. Eppstein, D., Goodrich, M.T., Strash, D.: Linear-time algorithms for geometric graphs with sublinearly many edge crossings. SIAM J. Comput. **39**(8), 3814–3829 (2010)
13. Graham, R.L., Frances Yao, F.: Finding the convex hull of a simple polygon. J. Algorithms **4**(4), 324–331 (1983). https://doi.org/10.1016/0196-6774(83)90013-5. https://www.sciencedirect.com/science/article/pii/0196677483900135
14. Guibas, L., Overmars, M., Sharir, M.: Intersecting line segments, ray shooting, and other applications of geometric partitioning techniques. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 64–73. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-19487-8_7

15. Hershberger, J.: An optimal visibility graph algorithm for triangulated simple polygons. Algorithmica **4**(1–4), 141–155 (1989). https://doi.org/10.1007/BF01553883
16. Klute, F., Nöllenburg, M.: Minimizing crossings in constrained two-sided circular graph layouts. J. Comput. Geom. **10**(2), 45–69 (2019)
17. Overmars, M.H., Welzl, E.: New methods for computing visibility graphs. In: Proceedings of the Fourth Annual Symposium on Computational Geometry, pp. 164–171 (1988)
18. Pach, J., Radoicic, R., Tardos, G., Toth, G.: Improving the crossing lemma by finding more crossings in sparse graphs. Discret. Comput. Geom. **36**(4), 527–552 (2006). https://doi.org/10.1007/s00454-006-1264-9
19. Pach, J., Tóth, G.: Graphs drawn with few crossings per edge. Combinatorica **17**(3), 427–439 (1997). https://doi.org/10.1007/BF01215922
20. Pocchiola, M., Vegter, G.: Topologically sweeping visibility complexes via pseudo-triangulations. Discret. Comput. Geom. **16**, 419–453 (1996)
21. Purchase, H.C.: Metrics for graph drawing aesthetics. J. Vis. Lang. Comput. **13**(5), 501–516 (2002)
22. Radermacher, M., Reichard, K., Rutter, I., Wagner, D.: Geometric heuristics for rectilinear crossing minimization. J. Exp. Algorithmics (JEA) **24**, 1–21 (2019)
23. Shamos, M.I., Hoey, D.: Geometric intersection problems. In: 17th Annual Symposium on Foundations of Computer Science (SFCS 1976), pp. 208–215 (1976). https://doi.org/10.1109/SFCS.1976.16
24. Teller, S., Hanrahan, P.: Global visibility algorithms for illumination computations. In: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, pp. 239–246 (1993)
25. Zhao, Y., Wang, Y., Zhang, J., Fu, C.W., Xu, M., Moritz, D.: KD-box: line-segment-based KD-tree for interactive exploration of large-scale time-series data. IEEE Trans. Vis. Comput. Graph. **28**(1), 890–900 (2021)