Faculty of Science

Dept. of Information and Computing Sciences

Solving the Art Gallery Problem in Iterations

Author:
Simon Hengeveld *5500117*

First supervisor:
dr. T. Miltzow

Second examiner:
Prof. dr. M.J. van Kreveld

Acknowledgements

Firstly, I would like to thank my supervisor Tillmann Miltzow for introducing me to this interesting project. As the project was close to him, he was very interested in and positive about my progression. This alleviated the usual stress that comes with a master's thesis, making the process a lot less tedious and much more enjoyable. Additionally, I am grateful to my second examiner Marc van Kreveld for his valuable feedback and knowledge of the thesis process.

I would also like to thank my housemate and friend Christopher Bouma for letting me use his computer to run several experiments. Furthermore, I am thankful to my parents for always being available to offer good advice concerning difficult choices. Finally, special thanks go to my girlfriend Sofia Rosero Abad for bearing with me through the last stressful weeks and helping me make the graphs and figures look attractive.

Abstract

The Art Gallery problem is a famous problem in the field of Computational Geometry. The variant of the problem discussed in this thesis is defined as follows: given a polygon (possibly with holes) P, the goal is to find the smallest set of point guards $G \subset P$ such that every point $p \in P$ is at least seen by one of the guards $g \in G$. A guard g sees a point $p \in P$ if the segment pg is contained in P.

We discuss the practical implementation of the vision-stable iterative algorithm that is used to solve this famous visibility problem. This algorithm has theoretical guarantees, as shown by Hengeveld and Miltzow [17], in a paper written during this thesis. Aside from the vision-stable iterative algorithm, two of its subroutines were implemented as well. One of these algorithms is used to compute weak visibility polygons [1] and the other is used the answer weak visibility queries [16]. This is the first algorithm for the Art Gallery problem that both has theoretical guarantees and works arguably well in practice. The main question that we answer is whether the iterative algorithm is feasible in practice. We performed tests to measure the running time of the algorithm, and show that it does perform quite well practically. Additionally, several experiments using the practical implementation are discussed, answering questions about the running time of the algorithm such as the standard deviation, sensitivity to input parameters and workload distribution.

Contents

1 Introduction **4**

2 The Ingredients **8**
2.1 Preliminary concepts . . . 8
2.2 Computing the visibility polygon of a segment . . . 13
2.3 Computing the visibility polygon of a point . . . 16
2.4 Shortest path map . . . 17

3 Practical Algorithms for the Art Gallery problem **21**
3.1 Timeline . . . 22
3.1.1 Stony Brook 2007 . . . 23
3.1.2 Turin 2011 . . . 23
3.1.3 Braunschweig 2010/2012 . . . 24
3.1.4 Campinas 2013 (1) and (2) . . . 26
3.1.5 Braunschweig 2013 . . . 28
3.1.6 Campinas and Braunschweig 2013 . . . 28
3.2 Comparing the algorithms . . . 28

4 The iterative algorithm **30**
4.1 Speedup methods . . . 38

5 Experiments **41**
5.1 Practical running time . . . 41
5.2 The effect of the speedup methods . . . 45
5.3 Standard deviation . . . 46
5.4 Critical witness sizes . . . 47
5.5 Converge to the optimal solution . . . 48
5.6 Distribution of CPU usage . . . 51

6 Discussion and Future research **52**

A Intermediate Arrangements **56**

1 Introduction

The Art Gallery problem is a widely studied, complex problem in the field of Computational Geometry [3, 8, 20, 22]. In this problem, the goal is to guard an art gallery by using as few guards as possible. Geometrically, the art gallery is translated to a polygon P (possibly with holes) while the guards are translated to points within the polygon. We say a guard g can see a point $p \in P$ if the segment pg is contained within P. The guards are allowed to be anywhere inside the polygon. The goal is then to find a set $G \subseteq P$ such that every point $p \in P$ is seen by at least one guard $g \in G$. Section 3 describes the Art Gallery problem in more detail. Over the years, many practical algorithms have been introduced that attempt to efficiently find optimal solutions for this problem [4, 6, 11, 15, 18, 24, 25]. They work well in practice but offer no running time upper bounds. It is difficult to find such an upper bound because the solution space of the Art Gallery problem is continuous, making it hard to discretize. The practical algorithms presented so far often use heuristics to select candidate variables for an integer program which then finds candidate solutions. Afterwards, it is checked whether the candidate solution is optimal. If not, more candidate variables are added, and the IP is run again. Section 3.1 offers a more detailed review of these practical approaches. While this process works well in practice for many input polygons, these algorithms may run forever. Given these algorithms, it is not even clear if the Art Gallery problem is decidable. Only one algorithm that shows that the Art Gallery problem decidable has been published for the Art Gallery problem [12]. However, this algorithm uses algebraic methods. These methods are impractical for the Art Gallery problem because of the large number of variables that would have to be taken into account.

In this thesis, a practical implementation of a new algorithm is discussed. This algorithm is called the vision-stable iterative algorithm and relies on the computation of different types of visibility and several other subroutines. These subroutines and several more important concepts are discussed in Section 2. As the name of the algorithm suggests, it is based on the concept of *vision-stability*. To understand what vision-stability is, we imagine a version of the Art Gallery problem in which guards have either enhanced or diminished vision. We enhance the vision of the guards by letting them look "around the corner" by an angle of δ. We diminish vision of guards having their vision be more "blocked" by a corner by an angle δ. To explain this in more detail, consider the visibility polygons showcased in Figure 1. The visibility polygon of a point p inside a polygon P consists of the region of P visible from p. A more detailed definition of visibility and visibility polygons is given in Section 2. Using enhanced and diminished guards we can define the vision-stability. We say a polygon has vision-stability δ if the size of an optimal solution using δ-enhanced guards is the same as the size of an optimal solution using δ-diminished guards. Note that if a polygon has vision-stability δ, it also has vision-stability δ' for all $\delta' < \delta$.

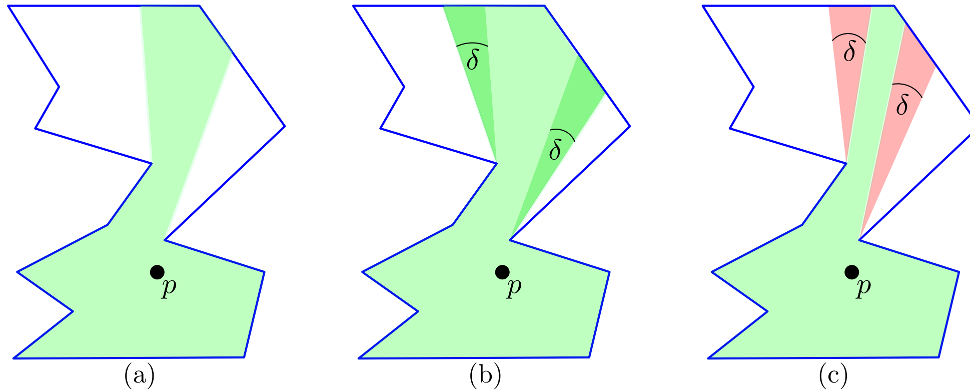


Figure 1: In (a), p is a normal guard and vis(p) is shown in green. Next to it, (b) shows a δ-enhanced guard p. The parts in darker green are added to its visibility polygon. The figure in (c) contains $\text{vis}_{-\delta}(p)$, the parts removed from its visibility polygon are shown in red.

The vision-stable iterative algorithm can be used to find an optimal solution for every vision-stable polygon [17].

From now on, we will refer to the vision-stable iterative algorithm simply as the iterative algorithm. The second part of the name of the algorithm comes from the fact that it solves the Art Gallery problem in iterations. At the end of every iteration, we use an IP to find intermediate solutions for an arrangement $\mathcal{A}$ inside P. These solutions act as lower bounds to the optimal solution. One of the reasons these are lower bounds is that aside from point-guards we also allow solutions that use face-guards. Once we have found such a "degenerate" solution using faces, we make sure that we update $\mathcal{A}$ and split the faces in question so that we cannot use them in the next iteration. Figure 2 shows three example iterations of the iterative algorithm and how the arrangement $\mathcal{A}$ inside P changes.

As Hengeveld and Miltzow [17] describe, the way we split faces is what guarantees the decidability of the iterative algorithm. Section 4 provides a more detailed description of the iterative algorithm, the IP we use and the way we use the subroutines from Section 2 to compute the necessary visibilities. Furthermore, Section 4.1 describes two ways in which practical speedup of the iterative algorithm can be achieved.

In this thesis, we answer the question of whether this algorithm is practically feasible. To that effect, the iterative algorithm was implemented in practice, including all of its subroutines and speedup methods. This was achieved using C++ and CGAL [23], a computational geometry library. The C++ source files of this implementation can be found at `https://github.com/simonheng/AGPracticalWithPerformance`. The code provided in the repository consists of C++ source and header files, in addition to a Visual Studio solution file. Bear in mind that the code is dependent on CGAL version 4.13.1 [23], IBM ILOG CPLEX version 12.10 [5], the boost library and Libxl version 3.9.0.0 (used to read and write excel result

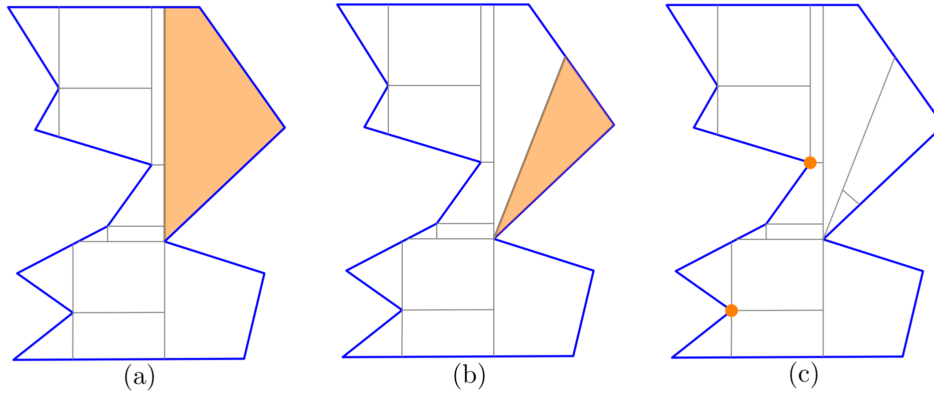


Figure 2: In (a) we create the initial arrangement by using the reflex vertices. We find a degenerate solution using 1 face-guard. In (b), we have split that face-guard from the previous iteration. We find another degenerate solution by using one of the new faces. In (c), we again split a face. Now, we cannot find a solution of size 1 anymore because all faces of the arrangement are too small. The IP finds an optimal solution using 2 point-guards.

files). In order to be able to compile and run the project, the above dependencies must be installed. Additionally, a video was created to further explain one of the subroutines. The video is publicly available on YouTube via `https://youtu.be/CFs8wPbmjlU`. To answer the question about practical feasibility and several other sub-questions about the algorithm, various experiments were conducted. These experiments are described in Section 5. Through these experiments, we found that we can use the algorithm to find optimal solutions for smaller to medium-sized input polygons with reasonable running times. We also compare the running times found to one of the existing practical algorithms for the Art Gallery problem [11]. We show that we are not far behind their running times, confirming the practical feasibility of the iterative algorithm. Moreover, upon closer examination of our results, we found that the median of our running times is often lower than the average. For many polygons, the iterative algorithm finds the solution in a very efficient manner. Some polygons, in particular polygons with low vision-stability, are more difficult for the algorithm. This re-affirms the importance of this theoretical concept. Furthermore, Hengeveld and Miltzow [17] show that the size of the IPs that we solve in the iterations can be bound by the vision-stability, showing that the running time of the iterative algorithm depends on δ. Moreover, because solving an IP, in theory, takes exponential running time, this means that this dependence is theoretically exponential. However, note that the running time of IP solvers is much faster than exponential in practice. Section 5.1 discusses the running times and the correlations between the vision-stability and the running times in more detail. In Section 5.5 we show that not only we find solutions for solvable polygons, but that the iterative algorithm provides an iteratively improving set of solutions to a polygon with irrational guards [2]. We demonstrate

this by computing the Hausdorff distance of the optimal solution of the polygon with irrational guards to the solutions found at the end of every iteration. Plotting these distances per iteration on a logarithmic scale (see Figure 27) shows that the iterative algorithm converges to the optimal solution in an exponential manner.

In a different experiment, discussed in Section 5.3, we show that several random factors of the iterative algorithm cause there to be a large standard deviation in its running times, even for the same polygon. Finally, we test the effects of the speedup approaches described in Section 4.1 and show that they have a large effect on the running time of the iterative algorithm.

The results found in all of these experiments confirm that the iterative algorithm is one of its kind, as it is the first algorithm for the Art Gallery problem has a theoretical running time upper bound (for vision-stable polygons) and works arguably well in practice. More discussion about the results of this thesis and some suggestions concerning future research can be found in Section 6.

2 The Ingredients

In order to find solutions for the Art Gallery problem, the iterative algorithm must answer different types of visibility queries. To achieve this, many different subroutines are necessary. This section will discuss and analyze these various subroutines. To start off, Section 2.1 will discuss some preliminary concepts. Sections 2.2 and 2.3 will discuss two algorithms that can be used to compute two types of visibility polygons. Finally, Section 2.4 will discuss a method of answering segment-point visibility queries.

2.1 Preliminary concepts

In order to understand the subroutines that we discuss later on, we must first define some concepts.

Visibility polygons. To define visibility polygons, we must first introduce the concept of visibility between two points in a polygon P. When we talk about a polygon P, we define P as all points in the interior and the boundary of the polygon. The visibility we will deal with is limited to two-dimensional visibility. A point q is visible from another point p if the segment pq does not intersect any obstacles. A point q is visible from a segment s if q is visible from any of the points in s. We can now define visibility polygons. Given a polygon P and a point $p \in P$, the visibility polygon $\text{vis}(p)$ is made up by all the points in P that are visible from p. Similarly, the visibility polygon $\text{vis}(s)$ for a polygon P and a segment $s \in P$ is defined as all the points in P that are visible from s. This type of visibility polygon is called a *weak* visibility polygon. These weak visibility polygons are useful because they let us compute the visibility polygons of larger shapes. Given an arrangement of line segments with a boundary polygon P, we can compute the visibility polygon of the faces of the arrangement by combining the weak visibility polygons of each of the boundary edges of the faces. Why this is relevant for the Art Gallery problem will be discussed in Section 4.

Figure 3 includes examples of $\text{vis}(p)$ and $\text{vis}(s)$. The blue outline shows the original simple polygon P, while the green outline is the resulting visibility polygon. In this case, s and p are both in the interior of P.

Beams, pockets and doors. We will discuss several notions that are important in the context of the first subroutine. The first concept we will define is the concept of *beams*. Given a segment $s = ab$ in P, a beam emanating from s is a segment pq, contained in P, drawn from any point p in s to a point q in the interior or on the boundary of P. The beams of a segment s are relevant when we want to compute $\text{vis}(s)$ because any point that they touch belongs to $\text{vis}(s)$, given our definition of visibility. These beams from s might create *pockets* in P, which are maximal regions from P that are not in $\text{vis}(s)$, which means that they are

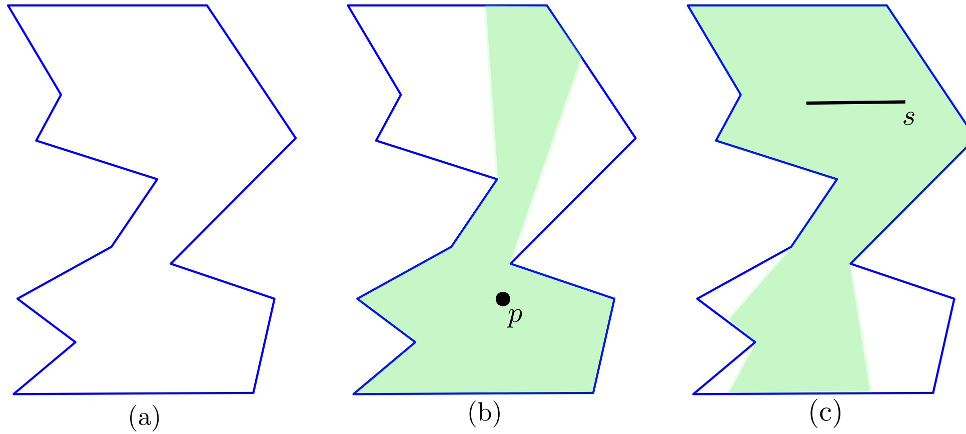


Figure 3: Includes (a) a polygon P, (b) an example of vis(p) in P and (c) an example of vis(s) in P.

cut off by one of the beams from s. The particular beam that separates a pocket from vis(s) is called a *unique maximal beam*, and the part of the beam that borders with the pocket is called a *door*. Since the door is not part of the pocket, pockets can be seen as objects that are part open and part closed. If a pocket is to the left of its beam it can be called a *left pocket*. Otherwise, it is a *right pocket*. At most two doors have an endpoint on any given edge in P [1]. This is because any edge contains at most one endpoint of a door from a left pocket and right pocket. Figure 4 shows a beam pq from the edge ab and a left pocket in gray, with as door the segment lq.

A segment ab has a left half-plane and a right half-plane noted LHP(ab) and RHP(ab). These half-planes consist of the areas to the left and the right of $\overleftrightarrow{ab}$ respectively, where $\overleftrightarrow{ab}$ is the line obtained by extending ab to infinity on both sides. Figure 4 shows the left and right half-planes of the segment cd. According to this definition LHP(ab) = RHP(ba). If pq is a beam with p on an edge ab of P and q on an opposing edge de of P, it is called a *proper* beam if if $pq \subset$ LHP(ab) and $pq \subset$ LHP(de). Moreover, pq is called the *rightmost* beam between ab and de if it is a proper beam and, among all other proper beams, p is as close to b as possible and q is as close to d as possible.

The *right support* of a beam pq is a reflex vertex r of P such that $r \in pq$ and the edges meeting at r are both contained in RHP(pq). The *left support* l of a beam pq is defined analogously. The beam pq in Figure 4 is an example of a rightmost beam between the segments ab and de, with left and right support l and r. Finding the rightmost beam is a key factor when computing *vis* for a segment.

A subroutine to find the rightmost beam is provided in [1]. This subroutine takes as input the vertices $v_0, v_1 ... v_{n-1}$ of a polygon P in counter-clockwise order and an index i, such that v_0v_1 and v_iv_{i+1} are opposing edges. The output of the algorithm is the rightmost beam of

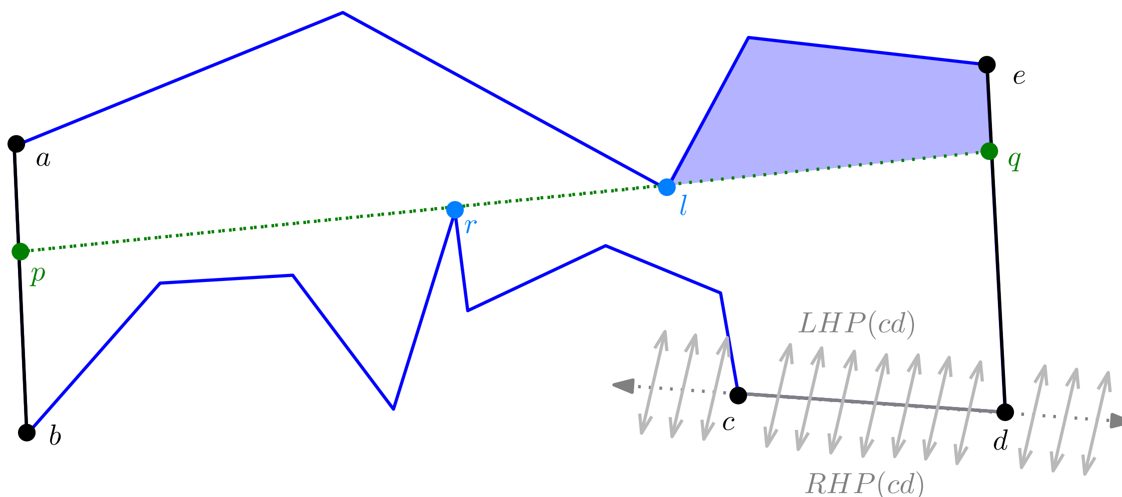


Figure 4: This figure shows an example of a (rightmost) beam pq between the edges ab and de contained in P with left and right support vertices l and r. The blue area is a pocket with a door lq. Additionally, the left and right half-planes of the edge cd are shown in gray.

between the segment $s = v_0v_1$ and the edge v_iv_{i+1}, expressed through the indices of its right and left support vertices. The subroutine finds the right and left supports by splitting P into two parts, one part being the vertices between v_1 and v_i, and the other part being the vertices between $v_i + 1$ and v_n. The vertices inside the right half of P are used as candidates for the right support while the vertices on the left half of P are used as candidates for the left support. The candidate vertices are tested by checking if there is a proper beam from v_0v_1 to v_iv_{i+1} that intersects both candidate vertices. If no proper beam can be made from v_0v_1 to v_{i+1}, the subroutine returns $NULL$. If the rightmost beam from v_0v_1 to v_{i+1} does not intersect any vertices (the entire segment v_{i+1} is visible from v_0v_1), the right and left supports returned are v_1 and v_{i+1} respectively. A limitation of this subroutine is that it requires the segment for which we are computing the rightmost beam to be an edge in P, i.e., it requires s to be a *boundary segment*. Section 2.2 will discuss how this subroutine is used when computing vis(s). Additionally, we will explore the way we can overcome this limitation and compute vis(s) when s is an *interior segment*, which is a segment that lies within P (see ab in Figure 4).

Shortest paths, funnels and merging. Section 2.4 introduces a method that uses paths within a polygon to answer visibility queries between segments and points. We will later show how, given the shortest paths between a segment and a point, we can check whether the segment can see the point. A *shortest path* between two points p and q in a polygon is the shortest sequence of points $a, a_0, \ldots, a_k, b$ such that all points in the sequence are inside or on the boundary of the polygon. We will denote the shortest path from p to q as $\pi(p, q)$. Figure 5 (a) gives an example of such a shortest path. Another concept that we will use

is is the concept of *funnels*. A funnel can be seen as the set of all shortest paths between a segment s and a point p. Figure 5 (b) shows such a funnel. One can represent a funnel between a segment $s = ab$ and a point p by using $\pi(a, p)$ and $\pi(b, p)$, which combined with s, is the boundary of the funnel. Given a funnel, we can test whether or not p is visible from s. If the paths $\pi(a, p)$ and $\pi(b, p)$ have no overlapping vertices ab can see p. This is shown in Figure 5 (b). In some cases, $\pi(a, p)$ and $\pi(b, p)$ can have one overlapping vertex such that ab can still see p (Figure 5 (c)). This happens when the ray from p to the first overlapping point intersects ab. If the funnel has more than one overlapping vertex, s can never see p (Figure 5 (d)).

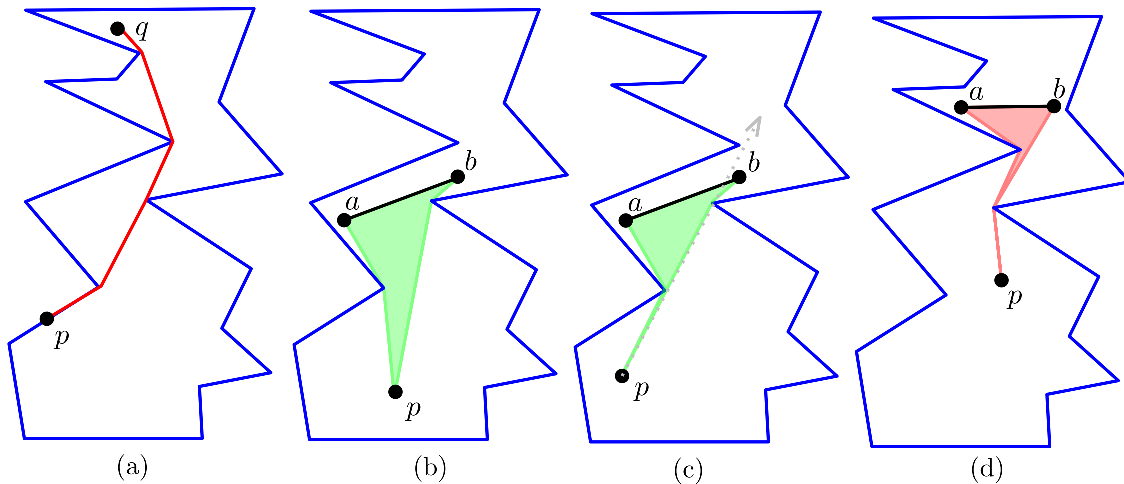


Figure 5: Includes (a) an example of a shortest path $\pi(p, q)$ in which point p does not see q, (b) an example of a funnel in which point p is visible from the segment $s = ab$, (c) an example of a funnel in which point p is visible from the segment $s = ab$ and where the funnel has one overlapping vertex and (d) an example of a funnel between a segment $s = ab$ and a point p where p is not visible from s.

Given a chord of the polygon $c = ab$ and two points p and q on opposing sides of c we can find $\pi(p, q)$ by *merging* the funnels from p to c and q to c. Theoretically, these funnels can be merged using an algorithm that runs in $O(\log m)$, where m is the number of vertices of the boundary of the funnels. However, the merging funnels subroutine that was implemented as part of this thesis uses a linear-time method. In practice, the funnels that were merged in the practical implantation were rarely very large. The linear-time method that was used is similar to existing methods that are used to find the convex hull of a polygon. We merge the two funnels by building two candidate shortest paths, one on the side of a and one on the side of b. We will run through the construction of the $a-$side path. Note that the second path is constructed similarly, in a mirrored fashion. We start in the middle, at point a. We then connect a with its two adjacent points in $\pi(a, p)$ and $\pi(a, q)$. We then check the position

of the middle points, now a, to the line between its two adjacent points. The position of the middle point compared to its adjacent point tells us whether or not this point should be included in the shortest path. Figure 6 shows how we do this. We keep adding points to the candidate shortest path, sometimes removing them until we run out of points on the a-side of the funnels. Meanwhile, we do the same for the b-side. We build both candidate paths in an alternating fashion so that we do not waste too much time on an incorrect shortest path.

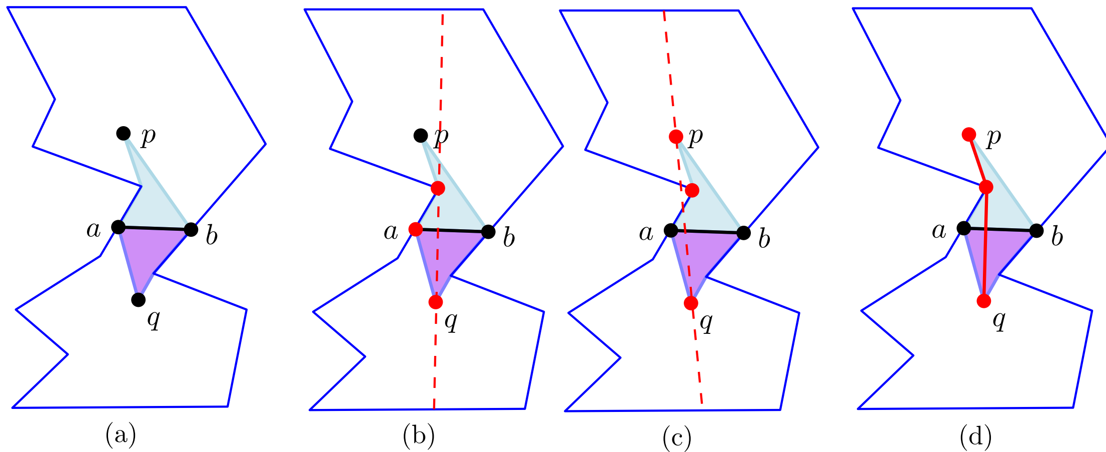


Figure 6: We see an example of how a funnel merge procedure is handled. In (a), we have the two funnels. In (b) we test the position of point a compared to the line of its adjacent points. Point a is to the left of the line, so we remove it from the chain. Then, in (c), we add p to the chain and again compare the middle point to the line of its adjacent points. The middle point is to the right of the line, thus we keep it in the chain. The final figure (d) shows the resulting shortest path. The path on the b-side is found to be invalid after only one positional check.

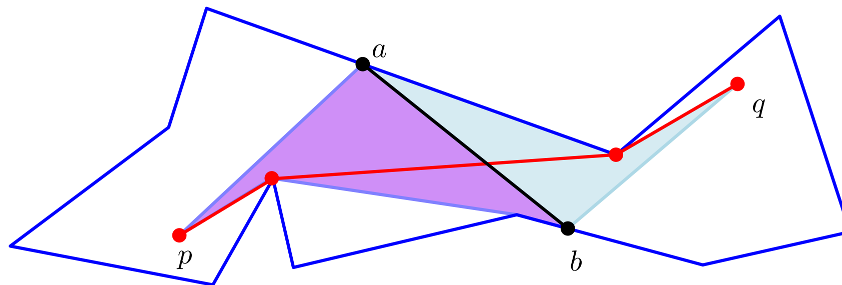


Figure 7: This figure shows an edge case. The funnel contains two reflex vertices of P, on opposing sides. The shortest path resulting from the funnel merge contains both reflex vertices.

Most of the time, the shortest paths will be either on the a-side or the b-side of the funnels. If this is the case, then the previously described algorithm functions correctly. However, there

are some cases in which this does not happen. Such a case is shown in Figure 7. In this case, the path is on both the a-side and b-side of the funnels because the funnels contain two opposing reflex vertices. We detect this case when after the procedure, we did not find a working shortest path on either the a-side or b-side of the funnels. In this case, we can find the path by merging two parts on both sides.

2.2 Computing the visibility polygon of a segment

Algorithm 1 is an algorithm taken from [1] that can be used to compute vis(v_0v_1). This algorithm assumes that v_0v_1 are vertices of P and are both convex. This means that this algorithm shares the same limitation as the rightmost beam subroutine, but is even more restrictive. We will first discuss how this algorithm works and then we will explore how to overcome these restrictions. This algorithm was implemented from scratch using C++ and CGAL as part of the first phase of this thesis.

Algorithm 1 Computing vis(v_0v_1)

1: **report** v_0, v_1
2: $i \leftarrow 1$
3: **while** $i < n - 1$ **do**
4: $(R, L) \leftarrow$ FindRightmostBeam(i)
5: **if** $L = i + 1$ **then**
6: **report** v_{i+1}
7: Let s be the point on v_0v_1 closest to v_0 that can see v_{i+1}
8: **if** $s \in$ LHP($v_{i+1}v_{i+2}$) **then**
9: $i \leftarrow i + 1$
10: **else**
11: Let x be the point where $\overrightarrow{sv_{i+1}}$ exits P
12: $i \leftarrow$ the first vertex of the edge containing x
13: **report** x
14: **end if**
15: **else**
16: Let q be the intersection point between v_iv_{i+1} and $\overrightarrow{v_Rv_L}$
17: **report** q, v_L
18: $i \leftarrow L$
19: **end if**
20: **end while**

The algorithm builds up vis(v_0v_1) by iterating through the vertices of P in a counter-clockwise manner. At the very beginning, v_0 and v_1 are added to vis(v_0v_1) because they are trivially visible from v_0v_1. For each edge v_iv_{i+1}, where i starts at 1, the rightmost beam between v_0v_1 and v_iv_{i+1} is computed. In the pseudo code, these supports are saved by their index in P as the integer variables R and L. If the right and left support found are not $NULL$,

two cases are identified, and points are added to vis(v_0v_1) based on these cases. These cases are illustrated in Figure 8.

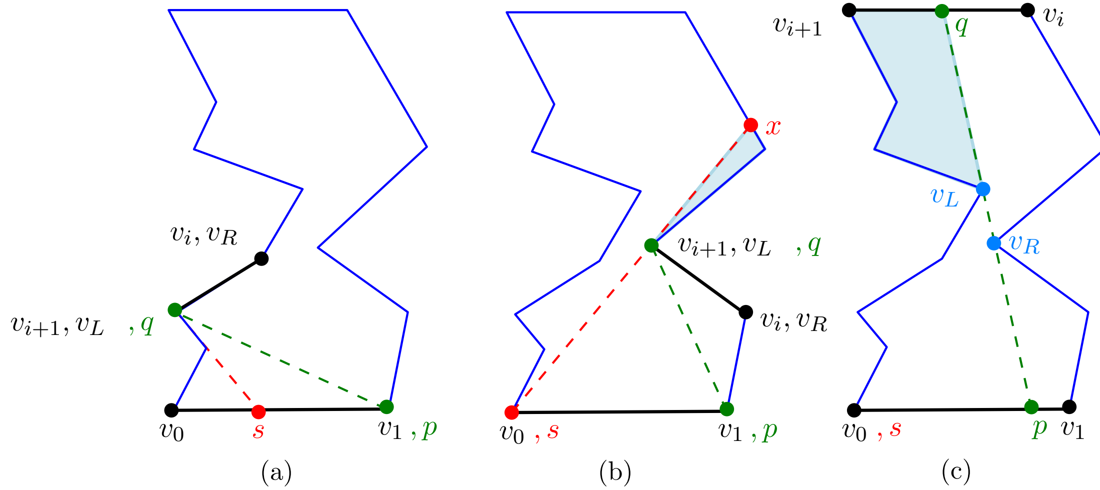


Figure 8: The different cases the algorithm will encounter. In (a) we see an example of Case 1, when $s \in \text{LHP}(v_{i+1}v_{i+2})$. Next to it, in (b) we see an instance of Case 1, but now s is not in $\text{LHP}(v_{i+1}v_{i+2})$. Finally, (c) shows Case 2.

The first case occurs when the left support is equal to v_{i+1} (lines 6 – 15). This case is shown in Figure 8 (a) and Figure 8 (b). When the vertex v_L is equal to v_{i+1} we know that v_{i+1} is visible from v_0v_1 and thus v_{i+1} is added to vis(v_0v_1). To find what else is visible we must use a point s that we set to be the point on v_0v_1, closest to v_0 that can see v_{i+1}. This can be achieved by using a subroutine that computes point to edge visibility, using v_{i+1} as the point and v_0v_1 as the edge. This subroutine is also discussed in [1] and works by checking whether v_0v_{i+1} or v_1v_{i+1} intersects with any vertices in P. If it does, the visible region of v_{i+1} on v_0v_1 is updated. To find s we simply let it be the left bound of this visibility region because that is the closest visible point to v_0. Based on the location of s we decide whether another point should be added to the output. If s is within $\text{LHP}(v_{i+1}v_{i+2})$ we can simply increment i and check the next rightmost beam. This is shown in Figure 8 (a) where s is on the the extended line $\overleftrightarrow{v_{i+1}v_{i+2}}$ and thus $s \in \text{LHP}(v_{i+1}v_{i+2})$. However, if s is not within $\text{LHP}(v_{i+1}v_{i+2})$ we can add another point to vis(v_0v_1). The point we can add is a point x where the ray sv_{i+1} exits P. This point is added because we know that the beam sx is a door to a right pocket. We cannot see this right pocket, but we might be able to see the rest of edge that x is on, so i is updated to be the source this edge. This is illustrated in Figure 8 (b), showing the right pocket in blue and the beam sx in red. That concludes the steps taken for the first case.

In the second case (lines 16 – 21), the left support L found by the rightmost beam subroutine is larger than $i + 1$. We then let q be the intersection point between the segment v_iv_{i+1} and the ray v_Rv_L, where R and L are the right and left supports of the rightmost beam.

Because the rightmost beam has these left and right supports, we know that the points in the pocket with the door v_Lq are not visible. This is shown in Figure 8 (c). This means that we can add q and v_L to vis(v_0v_1) and update i to L because we can skip the vertices between v_{i+1} and v_L. Once the loop is finished, the output will contain the correct visibility polygon of v_0v_1 in P.

The O(mn) complexity of this algorithm comes from the fact that every call to lines 5, 12 and 17 take O(n) time and the outer loop will iterate at most m times. This is because in every iteration when there exists a rightmost beam, at least one vertex is added to the output polygon and vertices are never removed.

As mentioned before, this algorithm only works when s is a boundary segment with convex vertices. We will now address when we want to compute vis(s) for s when s is either an interior segment or when s is a boundary segment with one or two reflex vertices. The idea is to split the original polygon P into multiple polygons by extending the segments and then use the above algorithm on each of the separate polygons. We must then combine these results in a smart way as to not have repeating vertices. This will only work if we split the polygons in a way such that the part of s in the new polygon is a boundary edge with convex vertices. Sometimes this part of s will simply be a point. We can then use the algorithm discussed in the next section to compute the visibility polygon for that point.

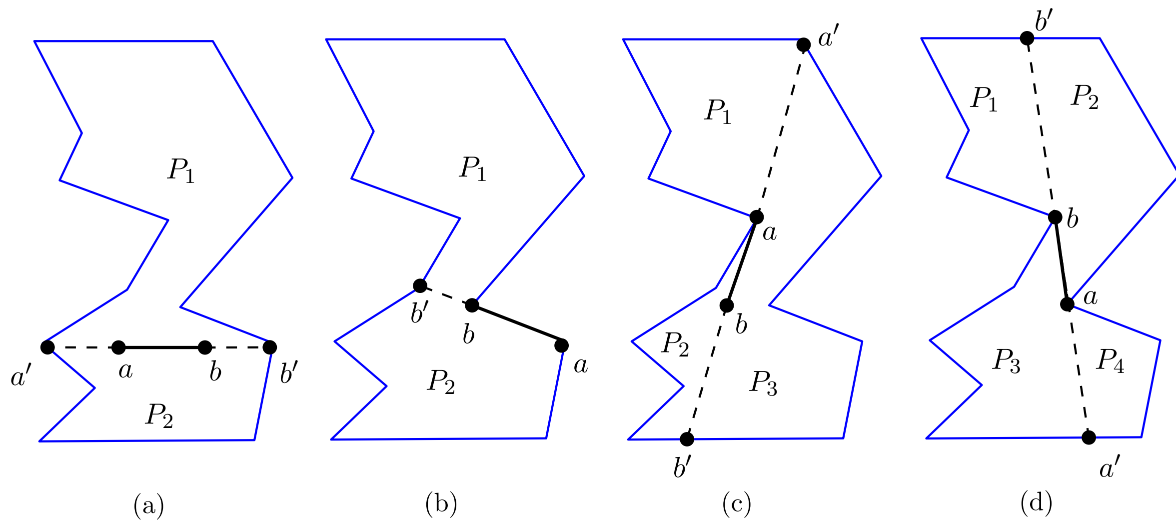


Figure 9: Three images with example splits for P when computing vis(s) for $s = ab$

Figure 9 shows several example splits for a polygon P when computing vis(s) for $s = ab$. In this figure, the splits are achieved by either extending a, b or both. The new polygons should have the correct vertices in the right order. For example in Figure 9 (a), P_2 should have, besides several original vertices, $v_0 = a$, $v_1 = b$, $v_2 = b'$, $v_{12} = a'$ while P_2 should have $v_0 = b$, $v_1 = a$, $v_2 = a'$, $v_7 = b'$. The visibility polygon of P_1 in Figure 9 (b) is computed

using the point visibility algorithm because the only point of s in P_1 is b. The same goes for P_1 in Figure 9 (c) but then for point a. Figure 9 (d) demonstrates that it is sometimes necessary to split the original polygon into four new polygons.

2.3 Computing the visibility polygon of a point

An apt algorithm to compute the visibility polygon for a point p in a simple polygon was introduced in 2013. This algorithm was developed in order to make a practical solution to the Art Gallery problem more viable. Practical testing of this solution showed that most computation time was spent computing the visibility polygon of points. To overcome this the authors of [7] decided to develop an algorithm that can compute the point visibility polygons in a more efficient manner. More about this practical solution to the Art Gallery problem can be found in Section 3.1.5.

The algorithm is called triangular expansion and works for simple polygons with or without holes. The worst-case time complexity of the algorithm is $O(n^2)$, but in practice, much better results are achieved. The algorithm is composed of a pre-processing stage and a computation phase. The pre-processing stage consists of triangulating the polygon. Theoretically, this can be done in $O(n)$ for polygons without holes and $O(n \log n)$ for polygons with holes, but in the practical implementation of CGAL [23] an algorithm with a worst-case running time of $O(n^2)$ is used. Similarly to the second part of the algorithm, this pre-processing phase also performs much better in practice.

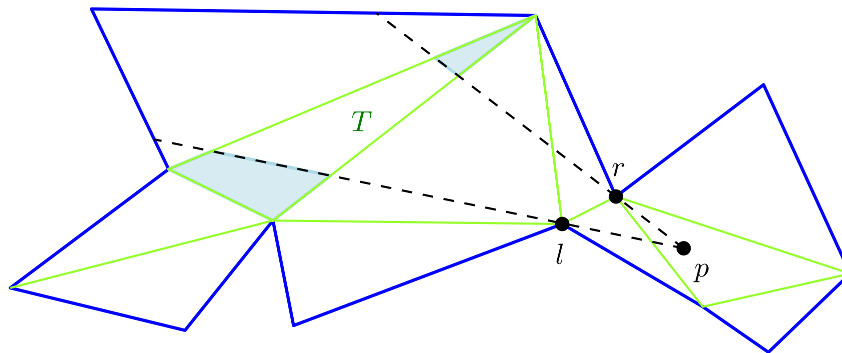


Figure 10: This figure shows an example of how the view of a point p might be restricted in a triangle from the triangulation of P. In this case, two reflex vertices l and r restrict the view. The parts of triangle T that p cannot see are highlighted in blue.

The computation phase starts by finding the triangle of the triangulation in which p is located. This is done by performing a simple walk. The edges of P in this triangle are added to vis(p) because they are visible. For every other edge in the triangulation, a recursive procedure is done that expands the view of p through that edge into the next triangle. This initially just restricts the view using the two endpoints of the edge but as the recursion

proceeds, the view may be further restricted. This happens if, between p and the new triangle, one or two reflex vertices blocks the view from p. An example of this is shown in Figure 10 below with l and r being the left and right reflex vertices blocking the view.

In this case, vis(p) is updated to reflect this restriction. If p is between l and r like in the figure, the recursion might split into two instances. As there are n vertices, the recursion might be split into n calls which then iterate $O(n)$ triangles. This suggests a worst-case complexity of $O(n^2)$. However, a true split into two visibility cones that may reach the same triangle independently can happen only at a hole of P. This means that the worst case is $O(nh)$ where h is the number of holes. For simple polygons, the running time is linear. Furthermore, [7] performed experiments showing that, in practice, the triangular expansion method performed much better than the state of the art methods for visibility polygon computation.

2.4 Shortest path map

Technically, we could use the above two methods to answer all types of visibility queries. Given the visibility polygon of a candidate point or segment, all we have to do to check visibility is to compute whether or not the target point lies within this visibility polygon. Using the triangular expansion method for point-point visibility is very fast in practice. However, as shown in the previous section, the algorithm to compute this weak visibility polygon has a running time of $O(n^2)$. As we will be doing large amounts of visibility queries, relying on the above method would be inefficient. This creates the need for a faster way of computing segment-point visibilities. Such a method was introduced in 1987 [16]. This method includes pre-processing the polygon P by computing all shortest paths between all pairs of vertices. Given a chord $c = ab$ and a point p, we let m be the size of the funnel, i.e., the size of $\pi(a, p)$ and $\pi(b, p)$. Then, given this shortest path map, we can use it in combination with a triangulation of the polygon to answer segment-point visibility queries in $O(\log m)$ time. However, note that in our implementation, we actually solve them in $O(m)$ time because of the way in which the merge routine was implemented. We know that $m \ll n$, and that m is often rather low. This is because, in practice, the shortest paths will not be very complicated, as a result of the weak visibility polygon tree that we used (see Section 4.1). The fact that m is often low means that the linear time funnel merge method is still rather efficient in practice. This then means that using the shortest path map is much more efficient than using the $O(n^2)$-running time technique described in Section 2.2.

Building the shortest path map. To build the shortest path map inside a polygon P, we need a triangulation of P. As mentioned in Section 2.3, CGAL[23] has a method available for triangulating polygons. This is what is used in the practical implementation of this pre-processing step. Once we have access to the triangulation, we can begin building the shortest paths. First, we build all shortest path from one vertex to all others. Then, we repeat

this for every other vertex to build the complete map. To find all shortest paths from a vertex v to all other vertices, we find the triangle T_v that contains v. We then traverse the adjacency tree of the triangulation, starting at T_v, building the paths. The shortest path between two vertices of the same triangle is trivial: it is the triangle edge. All other shortest paths we can find by iteratively merging funnels. To find the shortest path of two vertices in adjacent triangles, we must merge the funnels. The funnels that we are merging are actually the two triangles themselves, with the diagonal that they share as the edge that both funnels have in common. Figure 11 (a) and (b) show an example of this.

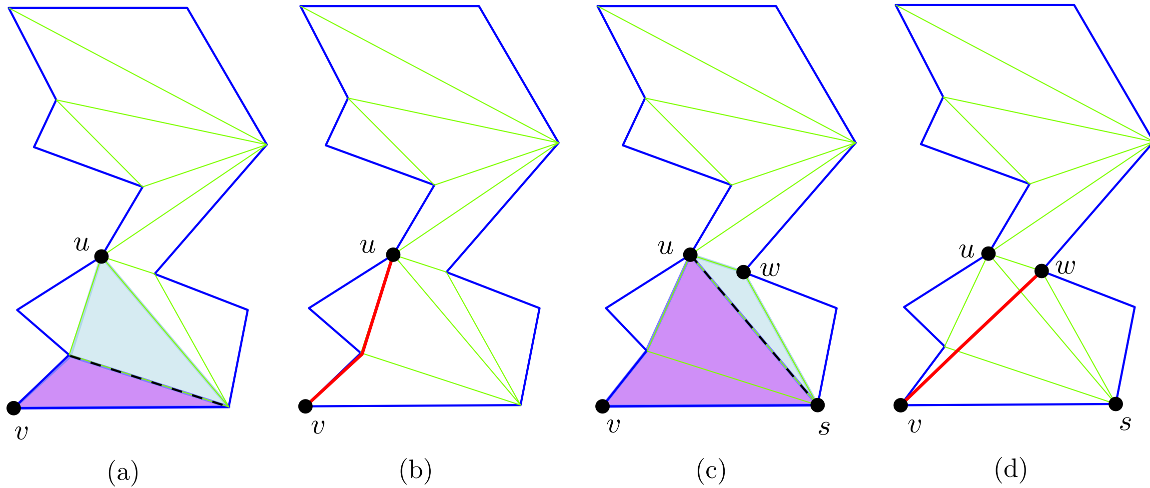


Figure 11: While building the map, we perform funnel merges to find shortest paths. In (a) and (b) we show that to find $\pi(v, u)$ between two vertices v and u in adjacent triangles, we merge two simple triangle funnels. In (c) and (d) we show that to find $\pi(v, w)$ between two vertices v and w in non-adjacent triangles, we can use the paths computed so far $(\pi(v, u), \pi(v, s))$ to find the necessary funnels for merging.

As we traverse the triangulation tree, we will reach triangles that are not adjacent to T_v. However, since we have already build up the shortest path up and to this point, we can use the already pre-computed shortest paths to continue. Figure 11 (c) and (d) show the funnels that we then need to merge. Note, that while building the map, at least one of the two funnels will always be a simple triangle. After we finish building the shortest path map from v, we must do this for every other vertex of P. For every vertex, we must traverse the complete triangulation of P. At every stage, we are performing a merge, which theoretically can be done in logarithmic time. This means that the running time complexity of building the shortest path map is $O(n^2 \log n)$. In our implementation, it means the running time is $O(n^3)$. Furthermore, since we are storing paths between all vertices of the polygon, this indicates a memory usage of $O(n^3)$. However, this is not a very accurate upper bound. In practice, the running time and memory usage are much lower because paths are not generally very complex.

Moreover, the use of the weak visibility polygon tree, described in Section 4.1, eliminates the need for computation and storing of many paths inside the polygon. Using the concept discussed in that section, [17] proves that the running time and memory use of the shortest path map depend on another parameter of the weak visibility polygon tree, thus giving a tighter upper bound on both of these aspects of the data-structure.

Computing visibilities using shortest paths. As mentioned before, the goal of using the shortest path data-structure was to be able to check whether a segment $s = ab$ can see a point p in logarithmic time. To achieve this, we must construct the funnel between s and p. This involves finding $\pi(a, p)$ and $\pi(b, p)$. Let us explore how we find the shortest path from $\pi(a, p)$ using the shortest path map. Finding $\pi(b, p)$ can be done the same way. First we locate the triangles T_a and T_p in the triangulation that contain a and p respectively. If a and p are both vertices, we can use the shortest path map to find $\pi(a, p)$ in $O(1)$. If one of the two points is a vertex and the other is not, we must perform one funnel merge. Let's say a is a vertex and p an point inside T_p. Both funnels share as target edge the diagonal of T_p that faces T_a. We can easily identify this diagonal using the triangulation tree. The funnel from p to this diagonal is a simple triangle. The boundary of the second funnel consists of the shortest paths from a to the vertices of this diagonal. We can find these paths by looking inside the pre-computed map. Finally, we merge these funnels, finding $\pi(a, p)$. The hardest case is when both a and p are interior points of their triangles. In that case, we must merge three funnels. We again identify the diagonal from T_p that faces T_a. We then find the shortest path from a to this diagonal. This time, we cannot simply look up these paths. Instead, for each path, we must perform an additional funnel merge. Both merges share the triangle funnel T_a. To find the second funnel for both merges, we must look up the shortest paths between each of vertices of the two identified diagonals of T_a and T_p. Doing this, we will obtain four paths. These four paths are the boundaries of the second funnels necessary for the funnels. Once we have obtained the two paths from the diagonal of T_a to p, we can perform the final merge, similarly as in the previous case.

Once we compute both the $\pi(a, p)$ and $\pi(b, p)$, we can answer the visibility query. Only if the paths of the funnel have less than two overlapping points, s can see p, as shown in Figure 5. Locating a point in a triangulation, using existing methods in CGAL, has an average running time of $O(\sqrt{n})$. However, in our algorithm, we will do this only once for every point. We then save the triangle together with the point in a data-structure. This means that after we have located the points once, subsequent lookups can be done in $O(1)$. This is why we do not consider this when discussing the running time of the visibility queries. To solve the queries all we have to do is a constant number of funnel merges and $O(1)$ lookups. This means that the total running time of each visibility query amounts to $O(\log m)$, or $O(m)$, in the case of our implementation. As before, m is the number of vertices of the funnels that we need to

merge.

3 Practical Algorithms for the Art Gallery problem

This section will give a definition of the Art Gallery problem and will then explore and compare various algorithms that aim to solve it. We will explore five different algorithms and their results.

The Art Gallery problem bears this name because when the problem was originally sketched it was described as the problem of determining the minimum number of guards necessary to keep an eye on all of the paintings in an art gallery. A challenging part of the problem was that this art gallery might be awkwardly constructed with all kinds of corners and winding passages. The translation of this problem to a geometrical problem treats the art gallery as a polygon P and the guards as points and their visibility polygon as the area that they are guarding. Several variants of this problem exist and are studied. The most general definition of the problem states that the both interior and boundary of the polygon should be guarded and that guards are allowed to be placed anywhere inside or on the boundary of the polygon. Variants of the problem can be constructed by either changing what needs to be guarded or changing where the guards may be located. For instance, we can relax the problem by only requiring the boundary of the polygon to be guarded. Another example may be only allowing guards to be placed on the vertices of the polygon, simplifying the problem. We will focus on the complex, general variant of the Art Gallery problem. This variant is more complicated because the candidate set of guards now becomes exponentially large. Compare this to the easier vertex guard variant which has a candidate set size of n, the set of vertices of P. Below we properly define the variant that we will be using.

We are given a polygon P, with or without holes, in the plane with vertices V and $|V| = n$. We assume that P does not self intersect. If for a set $G \subset P$, $\text{vis}(G) = P$ holds, G is called a *guard set* of P and a $g \in G$ is called a *guard*.

The Art Gallery problem then asks for such a guard set G of minimum size. We call a point a *guard candidate* if we consider adding it to a guard set G, and we call a point a *witness* when we use it as a certificate for coverage. The different variants of the Art Gallery problem can also be defined as an Integer Linear Program (IP), where C is the set of candidate guards that we can choose from and W the set of witnesses that we need to guard. The IP is shown below.

$$AGP(C,W) = \min \quad \sum_{c \in C} [\![c]\!] \tag{1}$$

$$\text{s.t.} \quad \sum_{c \in \text{vis}(w) \cap C} [\![c]\!] \geq 1, \quad \forall w \in W \tag{2}$$

$$[\![c]\!] \in \{0, 1\}, \qquad \forall c \in C \tag{3}$$

We use a binary variable $[\![c]\!] = 1$ when we use the candidate guard $c \in C$ to guard one of the witnesses and $[\![c]\!] = 0$ otherwise. This is essentially a set cover problem in which we attempt to minimize the sum of the variable $[\![c]\!]$ thus minimizing the number of guards we pick from the guard candidate set C. The second line of the IP is the constraint which makes sure that all witnesses are guarded. It states that for every witness $w \in W$ there should be at least one guard candidate c from C in $\text{vis}(w)$ for which $[\![c]\!] = 1$. If the guard candidate $c \in \text{vis}(w)$ it implies that $w \in \text{vis}(c)$ because visibility is a symmetric function.

The variant of this problem that we are interested in is when both C and W are equal to (all the points in) P. For this problem, we have an infinite number of variables and constraints because with infinite precision there are infinitely many points in P. We will denote this variant as $AGP(P, P)$, the variant that only considers vertex guards as $AGP(V, P)$ and the variant in which only the boundary needs to be guarded as $AGP(P, \partial P)$.

Chvátal [8] proved the "Art Gallery Theorem" in 1975, stating that $\lfloor n/3 \rfloor$ guards are sometimes necessary but always sufficient. The proof is based on the fact that if you triangulate and then 3-colour P you can put a guard *near* every vertex of the least popular colour, providing a solution that is always valid. Various variants of the Art Gallery problem, including $AGP(P, P)$, were proven to be NP-hard. Recently, $AGP(P, P)$ was even shown to be $\exists\mathbb{R}$-complete [3]. $AGP(V, P)$ has been shown to be in NP while $AGP(P, P)$ has not yet been shown to be in NP because it is unknown if there is a polynomial-size representation of the point guard locations. Furthermore, $AGP(P, P)$ being $\exists\mathbb{R}$-complete strongly implies it is not in NP unless $\exists\mathbb{R} = \text{NP}$.

In [13], Eidenbenz et al. established a lower bound of the achievable approximation ratio for the $AGP(P, P)$ for polygons with holes. They found a lower bound of $\Omega(\log n)$, as well as establishing that the Art Gallery problem is an APX-hard problem. In [11] several algorithms solving various variants of the Art Gallery problem were compared. In the sections below we will summarize the results of this survey pertaining to the $AGP(P, P)$ variant. Both algorithms for simple polygons and polygons with holes will be considered.

3.1 Timeline

In this section, we will discuss seven different algorithms and explore the way they work. We will go through them chronologically. The seven different algorithms we will discuss are:

1. The oldest algorithm which uses a heuristic approach and is among the first experimented with for this variant of the problem. The algorithm was developed in Stony Brook in 2007. [4]. (Section 3.1.1)

2. An algorithm developed in Turin in 2011 [6] with had good results for smaller sized polygons. (Section 3.1.2)

3. The Braunschweig 2012 algorithm [15] which is an improvement over an earlier version [18] from 2010 that could not handle integer solutions. (Section 3.1.3)
4. The first version of the Campinas 2013 algorithm [24] which can efficiently find optimal solutions for simple polygons *without* holes. (Section 3.1.4)
5. The second (journal) version of the Campinas 2013 algorithm [25] which can efficiently find optimal solutions for simple polygons *with* holes. (Section 3.1.4)
6. An improvement of the Braunschweig 2012 algorithm published in 2013 with much better results than the previous versions. (Section 3.1.5)
7. The resulting algorithm of a collaboration between the universities of Braunschweig and Campinas in 2013, combining the results of the previous years. (Section 3.1.6)

3.1.1 Stony Brook 2007

Among the first to attempt to create a solver for $AGP(P, P)$ were Amit et al. [4] in 2007. The algorithms that the authors presented were more of a heuristic approach rather than an exact one. Several strategies were explored, all following a similar setup. A large set G of guards is picked following a heuristic such that P is guarded by G. Afterwards, guards are removed one by one from G following a priority function μ. The authors considered 16 different combinations of choices to construct G and to use μ. The most successful approach was when G was chosen to consist of all the vertices of P and several additional points. To find these additional points the edges of the polygon were extended to create a new arrangement. From each face in this arrangement, one point is picked and added to G. The function μ is then chosen such that it gives priority to guards that can see the most positions in G that are not yet guarded. Experiments were done with this algorithm on 40 input sets of polygons with up to 100 vertices. The solutions were found to be within factor 2 of the optimum while the most successful strategy found the optimal solution 12 out of 37 times.

3.1.2 Turin 2011

The lack of a practical algorithm that could be used to compute near-optimal solutions for the complex $AGP(P, P)$ led to the development of this algorithm in Turin in 2011 [6]. It is based on an earlier algorithm from 2008 by the same author which could find solutions for the $AGP(P, \partial P)$ variant when only the boundary needs to be guarded. The first step of the 2011 algorithm consists of finding multiple optimal solutions for the $AGP(P, \partial P)$ by using the 2008 algorithm as a subroutine. These solutions are then tested to check whether they also offer a solution for the $AGP(P, P)$ variant. The authors of [6] claim that if such a solution exists it is automatically also a near-optimal solution for $AGP(P, P)$. There are cases that a

correct solution is not among the ones found for the $AGP(P, \partial P)$ variant. When this occurs, a greedy search is done starting from one of the incorrect solutions, iteratively adding guards until a valid solution is found. The algorithm was tested with 400 polygons with sizes ranging from 30-60 vertices and an optimal solution was found in 68% of the cases.

3.1.3 Braunschweig 2010/2012

The second algorithm proposed to solve the difficult point guard Art Gallery problem was developed in 2012 in Braunschweig [15]. This algorithm builds on an earlier version of the algorithm [18] in 2010 that could provide *fractional* solutions for the point guards variant of the problem. What this means is that the third constraint of the IP in the previous section is relaxed. We will now also allow fractional values, i.e., $0 \leq [\![c]\!] \leq 1$. The Art Gallery problem demands integer solutions, which is why the newer variant was proposed. Both versions use a primal-dual approach. The main observation that leads to this approach is that when the sets of candidate guards and witnesses are finite and not too large, the problem can be solved easily using an LP-solver. The idea is then to construct such finite, relatively small sets C and W. We start by carefully choosing points and adding them to the initial sets C and W. Afterwards, these sets are iteratively extended using a primal and a dual separation phase. During the primal phase, an upper bound for $AGP(P, P)$ is computed while during the dual phase a lower bound is computed. In the 2010 version, the primal separation phase consists of *cutting planes* while the dual separation part uses *column generation*.

The primal separation phase aims to prevent constraint violations of the primal relaxed linear program while the dual separation phase attempts to do the same for the dual linear program. Both the primal and dual linear programs are shown below.

$$\min \sum_{c \in C} [\![c]\!] \qquad (4)$$

$$\text{s.t.} \sum_{c \in \text{vis}(w) \cap C} [\![c]\!] \geq 1, \quad \forall w \in W \qquad (5)$$

$$0 \leq [\![c]\!] \leq 1, \qquad \forall c \in C \qquad (6)$$

$$\max \sum_{w \in W} y_w \qquad (7)$$

$$\text{s.t.} \sum_{w \in \text{vis}(c) \cap W} y_w \leq 1, \quad \forall c \in C \qquad (8)$$

$$0 \leq y_w \leq 1, \qquad \forall w \in W \qquad (9)$$

The left side shows the relaxed primal version of $AGP(C, W)$. The only change compared to the previously discussed IP is that it allows $[\![c]\!]$ to have decimal values as shown in (6). The right side shows the dual variant of this problem which now contains a maximisation instead of a minimisation.

The cutting planes technique works as follows: we check if there is a witness $w \in P \setminus W$ that is not guarded and if so add it to W. If there is no such w, the current solution is feasible for the relaxed version of $AGP(C, P)$ and thus an upper bound for the $AGP(P, P)$.

If we then solve for the dual version we will effectively find a lower bound for the primal linear program. This notion is used in the dual separation phase using the column generation technique. If a constraint is violated in the dual linear program there must be a point $c \in P$ that is not yet contained in C. This candidate c should then be added to C. If there is no such guard c the current guard set G optimally guards W and thus provides a lower bound for the relaxed $AGP(P, P)$. This is known as column generation because using this dual linear program we identify which additional column should be added to the primal problem.

Both of these phases are applied iteratively and thus the sets of C and W grow each iteration. It is not necessary to perform both the primal and dual phase in every iteration, one can "steer" the computation towards an upper or lower bound by choosing to perform either of the two phases more often than the other. This steering is heuristic. This approach also has other heuristic aspects, such as the choice for the starting sets of C and W and how new guards and witnesses are chosen. The authors experimented with several ideas and found that two settings gave them the best results. The first was to set C to initially contain all reflex vertices of P and the second was setting W to have a witness on every polygon edge that is incident to a reflex vertex. An example of how the initial sets would be chosen is shown in Figure 12. This idea was based on prior work done in [9] by Chwa et al.

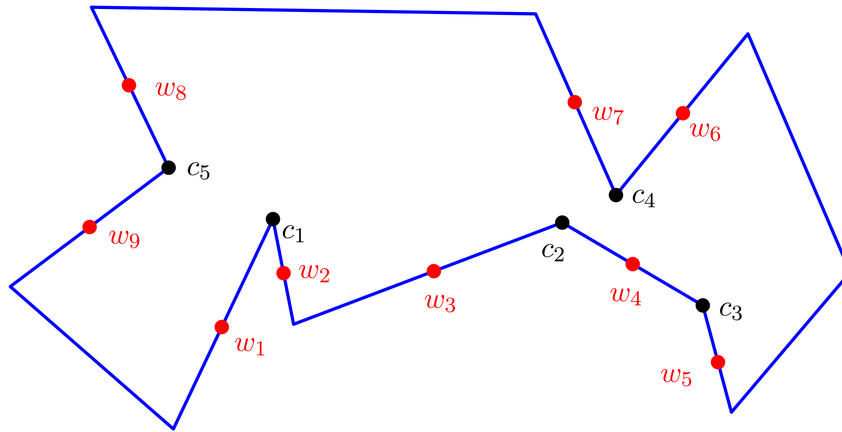


Figure 12: This figure shows an example of how the sets C and W are initially constructed for a polygon. The set of guard candidates are shown in black while the set of witnesses is shown in red.

This approach was then extended in 2012 so that it only finds integer solutions and is thus a valid method for $AGP(P, P)$. To reflect this, the primal and dual phases were changed. In the primal phase, $AGP(C, W)$ is solved using IP for the current sets C and W. Then, the visibility polygon of the sets of points C is scanned for insufficiently covered spots and additional witnesses are generated accordingly. If there are no more witnesses to be added we know that the current solution is optimal and thus an upper bound for $AGP(P, P)$. The

dual-phase computes a lower bound for the fractional variant in a similar fashion as in 2010. This leaves the issue of closing the gap between the upper and lower bounds. This has to be done to make sure that the algorithm ends with integer solutions. The lower bounds are raised using the cutting planes technique but the gap may still lead to sub-optimal solutions. However, in this case, we at least have a lower bound. The algorithm was tested on polygons with 60-1000 vertices including polygons with or without holes and orthogonal polygons. Different setups were tested and even the version that did not apply the extra cutting planes technique could identify fairly good integer solutions. However, the polygons with 1000 vertices were too large for this algorithm.

3.1.4 Campinas 2013 (1) and (2)

The next two algorithms to be discussed were both developed at the University of Campinas in 2013. The first algorithm [24] was designed to only handle simple polygons while the second one [25] could also handle polygons with holes. Additionally, the second algorithm includes various improvements over the first one in terms of computational efficiency. The approach the algorithms take is globally similar to the Braunschweig approach in the way that it iteratively computes upper and lower bounds until it finds an optimal solution. It does this by solving two semi-infinite discretized variants of $AGP(P, P)$. For the lower bound, $AGP(P, W)$ is solved where P is infinite and W is finite. This might seem impossible to solve but through a clever observation, [24] showed that is it possible.

Given a finite set of points $S \subset P$ we can construct an arrangement that divides the polygon into several regions. For each $s \in S$ we overlay their visibility polygons. The more visible a region, i.e. the more witnesses w can see it, the lighter the region is coloured. Figure 13 shows this for a small set S. In every region of the polygon, the figure displays the subset of points that can see this region.

The aforementioned clever observation is that if we construct the above arrangement for the finite set of witnesses W and then add the vertices of the lightest coloured regions to the candidate set C we can draw some conclusions about W and C. If we assume that $|W|$ is bounded by a polynomial in n then the size of this candidate guard set C is also bounded in n. This means that if we solve $AGP(C, W)$ using an IP we will find a correct lower bound for $AGP(P, W)$.

To find an upper bound, we solve $AGP(C, P)$. The procedure starts with the same sets C and W used for the lower bound computation and $AGP(C, W)$ is solved as before. We then check if this solution covers P. If it does, then we have found an upper bound. If not, new witnesses are added to W and the procedure continues.

A caveat of this method is that for some polygons it might run indefinitely and not find an optimal solution. To prevent this, the program must either terminate once it finds the optimal solution (there is no gap between the bounds) or when a certain time limit is reached.

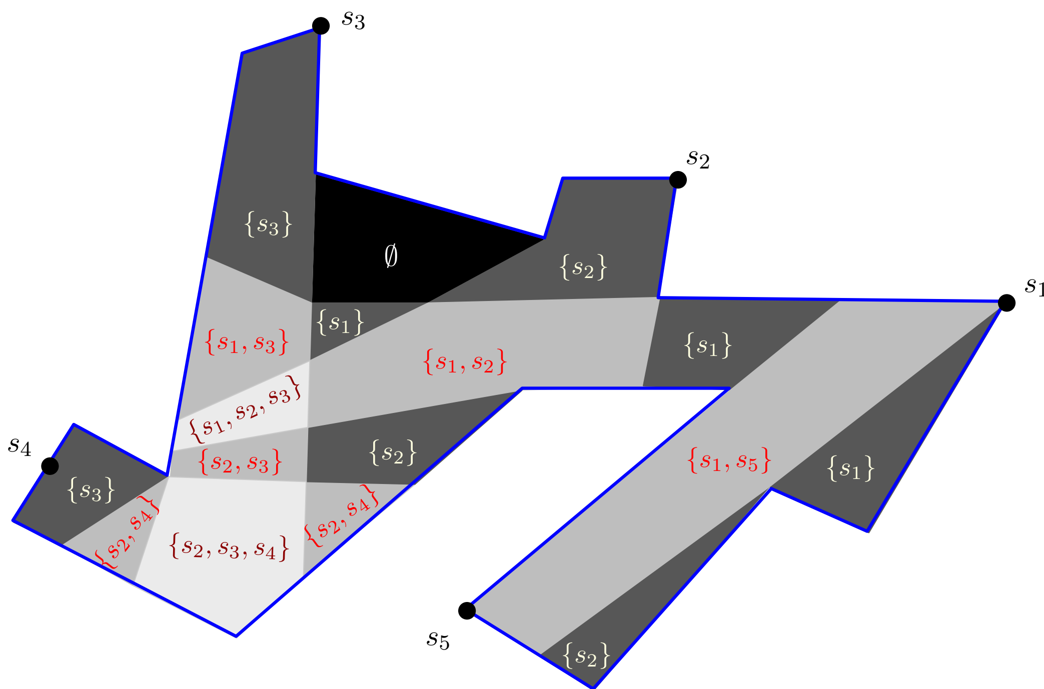


Figure 13: This figure shows an example of how we can create an arrangement of a set of points $S = \{s_1, s_2, s_3, s_4, s_5\}$ and their visibility polygons. An overlay is created. The darker the region the fewer points that can see it. Inside the regions the points that can see this region are shown in different colours such that the text is legible.

The initial witnesses and guards are chosen in a similar way as in Section 3.1.3 based on the results of [9].

The first variant of this algorithm was tested on 1440 simple polygons with a maximum of 1000 vertices. The results were quite good because, for all polygons, optimal results were found in a matter of minutes. The second variant was tested on 2440 polygons with or without holes and consisting of up to 2500 vertices. Again, the results were very promising as the algorithm found an optimal solution in 98% of the cases. The speedup of this version compared to the first one comes from the improvements made. Two changes were done to speedup the IP process: a Lagrangian Heuristic method was added and a procedure for removing redundant variables and constraints from the set cover formulation was used. The authors noticed that during the computations of the algorithm there are often more points in C than in W. They exploited this observation by, instead of computing the joint visibility polygons of guards and then testing if the witnesses are contained within these arrangements, to do the reverse. Because there are fewer witnesses, fewer visibility polygons need to be computed, saving a good amount of time.

3.1.5 Braunschweig 2013

A study of the previous algorithm found that about 90% of the run time of the algorithm was spent on geometric computation rather than on the linear program. The 2013 version improved the geometric subroutines dramatically leading to a much more efficient algorithm. Three major bottlenecks were: (1) the way visibility polygons were computed, (2) the way the visibility polygons were combined and finally (3) the way the location of a point was checked. For each of these bottlenecks, the code was changed to be more efficient. Visibility polygons were now computed with a new triangular expansion method [7]. The authors of this method published it to CGAL as a library. It is also used in the practical implementation of this thesis and is described in Section 2.3 The union of the visibility polygons was changed such that it used a lazy-exact kernel [21]. This kernel is "lazy" because it delays the computation of intersection points as long as possible. By doing this, it can sometimes even avoid the computation at all and thus save a lot of time. Finally, the point location algorithm was changed so that it used a better strategy that uses known locations of points to compute new ones in a better way [14]. The Braunschweig algorithms were implemented in CGAL [23] and the triangular expansion approach to compute visibility polygons for a point was published as a new CGAL library [7]. These improvements made the algorithm much faster and also made it possible for it to handle polygons with more vertices. How this algorithm compares to the other ones will be discussed in Section 3.2 later on.

3.1.6 Campinas and Braunschweig 2013

After the previous events, the research groups in both universities decided to work together to create an algorithm based on both of their findings in the previous year. At the core of the new algorithm was the second Campinas version discussed in Section 3.1.4. The improvements that the research group in Braunschweig had found in 2013 were made to this algorithm as well. The lazy-exact kernel approach was very effective for the Campinas algorithm because the approach computes many combined visibility polygons and computes many intersections. Additionally, some changes were done to the core algorithm. These alterations include postponing the upper bound computation until a good lower bound has been found, and the inclusion of a new strategy for guard selection. Moreover, for the linear program, the CPLEX solver was used instead of XPRESS which is a more optimized tool. All of these improvements led to better computation times and optimality of the algorithm, as we will see in the next section.

3.2 Comparing the algorithms

The survey done in [11] compares the running time and accuracy of the five algorithms developed in Braunschweig and Campinas. Again, we will only look at the results pertaining

to $AGP(P, P)$, the complex point guarding problem. Each of the five algorithms was tested with *random* polygons of six classes and multiple sizes. The six different classes are: (a) simple, non-orthogonal polygons, (b) non-orthogonal polygons with holes, (c) floorplan-like simple polygons with orthogonal edges, (d) floorplan-like orthogonal polygons with holes, (e) polygons inspired by randomly pruned Koch curves, (f) spike polygons with holes as presented in [18] . Small examples of the different test polygons are shown below in Figure 14.

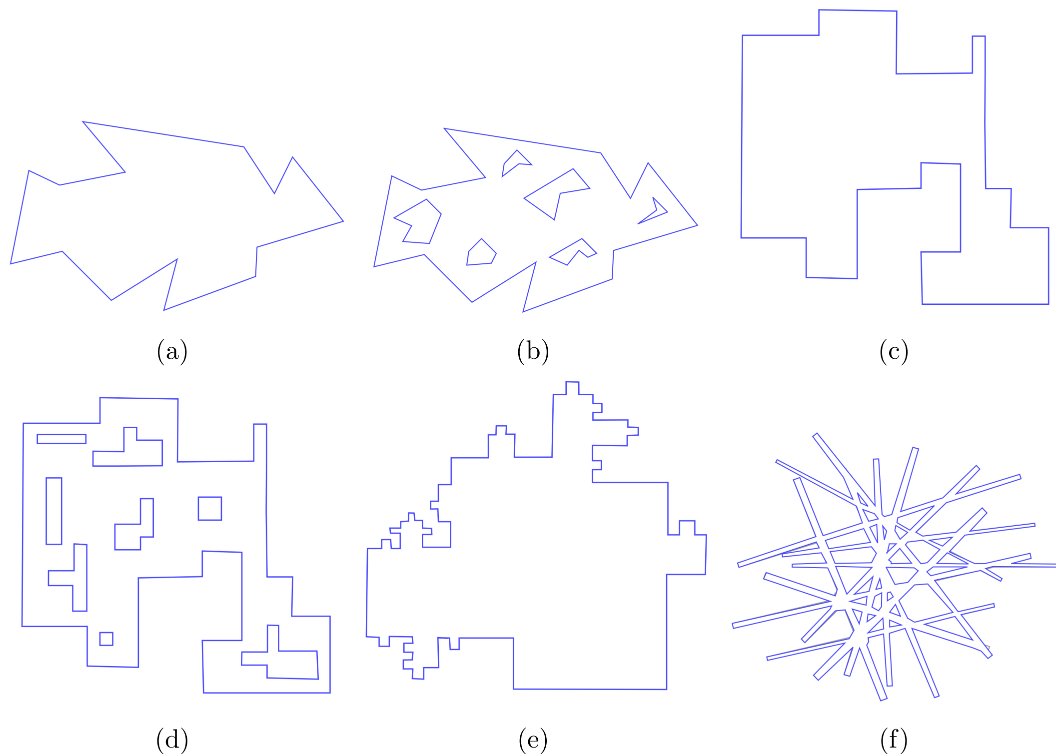


Figure 14: Examples of the six different types of test polygons.

The results from [11] show, unsurprisingly, that the latest version that resulted from the collaboration of the two universities was the most consistently accurate of the five algorithms. The improved 2013 Braunschweig algorithm is just behind the collaboration algorithm, while the Campinas algorithms are slightly worse. The three classes with holes, (b), (c) and (e) were much harder for most of the algorithms. For these three classes, even the collaboration algorithm could not find the optimal solution in any case for polygons with around 5000 vertices. The hardest class to solve were the orthogonal polygons with holes. For the other three classes, the collaboration algorithm could almost always find optimal results, even for these large 5000-vertex polygons. These results were quite promising especially comparing them to the best implementation proposed by 2011 from [6], mentioned in Section 3.1.2, which found the optimal solution in 68% of test cases for relatively small 60-vertex polygons.

4 The iterative algorithm

In this section, we will discuss the new algorithm that was implemented as part of this thesis. The selling point of this algorithm is that besides performing well in practice, it is the first algorithm for the Art Gallery problem that is both practical and has theoretical guarantees. The paper written during this thesis, [17], describes the theoretical guarantees in detail. In this section, we will describe how the iterative algorithm works in general and how it uses the subroutines described in Section 2. Additionally, we will discuss vision-stability and its importance. The algorithm uses an integer linear program solver to find solutions, similar to the newer algorithms discussed in Section 3.2. The algorithm solves the Art Gallery problem in iterations, which is why we call it the iterative algorithm. As discussed in Section 3, the reason that the Art Gallery problem is so complex is that the solution space of the problem is difficult to discretize. Overcoming this challenge is why the iterative algorithm has theoretical guarantees, it manages to discretize the solution space, without losing correctness. This is achieved by one important aspect of the algorithm. Namely, throughout the iterations, we will not allow only points to be guards, but we will also allow face-guards. Similarly, we also use both points and faces as witnesses. This use of face-guards is the reason that we need the shortest path map discussed in Section 2.4.

Solving the visibility queries. Now that we are using point-guards, face-guards, point-witnesses and face-witnesses we should discuss how we compute the visibilities between these types of guards in more detail. To check whether a point-guard sees a point-witness, we can use the triangular expansion method discussed in Section 2.3. For each point-guard candidate c that we have we use this method to compute $\text{vis}(c)$, and save it. Then, when we want to check if a point-guard sees a point-witness w, all we have to do is check whether w is inside $\text{vis}(c)$ or not.

To check whether a face-guard candidate c can see a point-witness w, we use the shortest path map from Section 2.4. For each edge s of c, we compute the funnel between s and w using the shortest path map. We can then use this funnel to test the visibility, as shown in Figure 5. Visibility queries involving face-witnesses seem more complicated. However, we can make some assumptions about the faces that make this easier. As we will show later in this section, all faces we will use as guards or witnesses are convex. Additionally, we only consider polygons without holes. With these assumptions, we can show that a point-guard or face-guard c sees a face-witness w if and only if all vertices of w are seen by c. This is shown in the proof for Lemma 3. This means that to answer these types of visibility queries, we can use the same techniques as for guards and point-witnesses. Note that in our practical implementation, we did queries involving face-guards in parallel. This is because these queries turned out to be the bottleneck, as shown in Section 5.6.

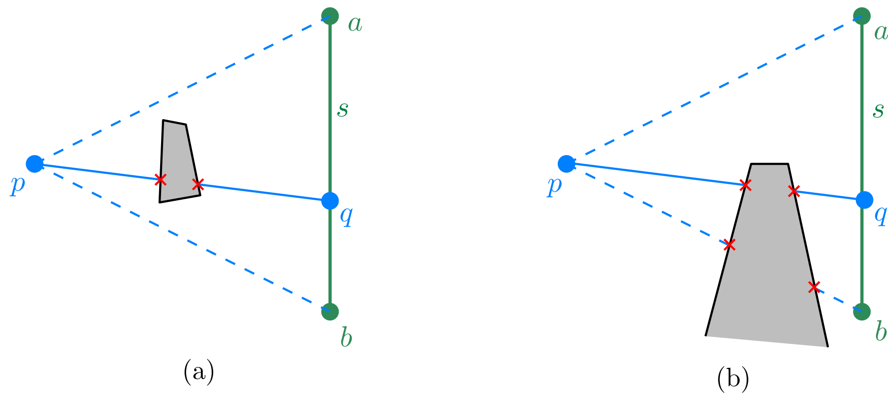


Figure 15: (a): A hole might block p from seeing s entirely if p sees a and b. (b): Part of P can block p from seeing s. This means that p cannot see both a and b.

Lemma 1. *Let $s = ab$ be a segment and p be a point, both contained inside a simple polygon P. If p sees a and b then p sees the entire segment s.*

Proof. See Figure 15 for an illustration of this proof. Let q be any point on s. If q is a vertex, p sees q by assumption of the lemma. If q is not a vertex, then we look at the segment pq. The point p can see q if pq is fully contained in P. By assumption of the lemma, pq cannot be interrupted by a hole. The only way pq is not fully contained in P is if it properly intersects the boundary of P. Because s is contained in P, pq could only properly intersect the boundary of P if an edge e of P intersects it between s and p. By assumption of the lemma, this is not possible because e would also intersect pa or pb. Thus, pq must be contained in P and p sees q. Because p can see any point q on s, p sees s completely. □

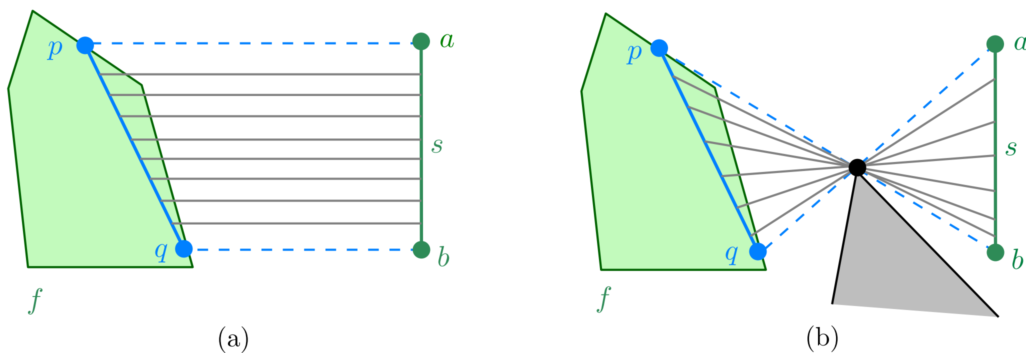


Figure 16: Every point in s has a corresponding point in pq, thus pq sees s. In (b) there is a reflex vertex of P is in between f_1 and f_2, but the same still holds.

Lemma 2. *Let $f \subseteq P$ be convex polygonal region and $s = ab$ be a segment, both contained inside a simple polygon P. If f sees a and b then f sees s.*

Proof. Let q be any point on s. If q is a vertex, f sees q by assumption of the lemma. If q is not a vertex, we will show that f contains a point p such that pq is contained in P. Because f sees ab, we know that f must have the points p and q corresponding to a and b, as shown in Figure 16 (a) or (b). Because f is convex, pq must be contained in f and in P. In both cases, we can draw line segments from every point in s such that they hit pq. All of these line segments are contained in P This means that pq sees s and thus f sees s. □

Lemma 3. *Let $f \subseteq P$ be a convex polygonal region inside the simple polygon P. Let g be a point or convex polygonal region in the same polygon. If g sees all the vertices of f, then g sees f entirely.*

Proof. We will prove this by showing that g can see any given point on the boundary or in the interior of f. Let a be a point on the boundary of f. If $a$is a vertex of f, then a is seen by assumption of the lemma. If a is not a vertex, then it is contained in a segment uv, where u and v are vertices of f. Because u and v are seen by g by assumption of the lemma, g can also see a by Lemma 1 and 2. Now let b be a point in the interior of f. We can construct a segment pq, with p and q on the boundary of f such that b is contained in pq. We know that g sees p and q because we showed that g can see any point on the boundary of f. Furthermore, we know that pq is contained in f because f is convex. Then, g can see b by Lemma 1 and 2. We showed that g can see any given point on the boundary or interior of f, thus g sees f completely. □

Arrangements and the normal IP. Throughout the iterations of the algorithm, we keep track of an arrangement inside the polygon. We will refer to this arrangement as $\mathcal{A}_i$, where i denotes the iteration index. We update $\mathcal{A}_i$ at the end of every iteration i.

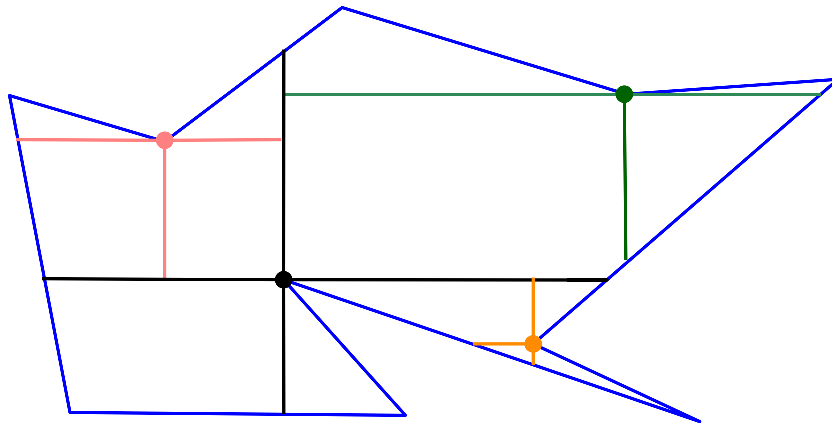


Figure 17: This figure shows an example of how a polygon would be pre-processed into an arrangement for the iterative algorithm.

To create $\mathcal{A}_0$, we take the original polygon P and from all of its reflex vertices, we shoot a maximum of four rays in axis-parallel directions from each vertex. As soon as a ray hits another ray or the boundary of the polygon we stop, resulting in a less complex arrangement. The figure below shows an example of such an initial arrangement.

At the end of every iteration i, we use an IP to find an intermediate solution set G_i. As candidates and witnesses, the IP uses the vertices and faces of $\mathcal{A}_i$. We call the set of candidate faces face(C), the set of candidate vertices vertex(C), the set of witness faces face(W) and the set of witnesses vertices vertex(W). The IP that we use at the end of every iteration is shown below. Note that for every witness w the set of candidates that see w completely. This IP is called the normal IP. Hengeveld and Miltzow [17] describe different IPs that can be used to prove theoretical correctness. Here, we will only consider the normal IP, as it was used for the variant of the algorithm with the best practical performance.

$$\min \quad \sum_{c \in \text{vertex}(C)} [\![c]\!] + \sum_{c \in \text{face}(C)} (1+\varepsilon) \; [\![c]\!] + \varepsilon \sum_{w \in \text{face}(W)} [\![w]\!] \tag{10}$$

$$\text{s.t.} \quad \sum_{c \in \text{VIS}(w)} [\![c]\!] \geq 1, \qquad \forall w \in \text{vertex}(W) \tag{11}$$

$$\sum_{c \in \text{VIS}(w)} [\![c]\!] + [\![w]\!] \geq 1, \qquad \forall w \in \text{face}(W) \tag{12}$$

$$[\![c]\!], [\![w]\!] \in \{0,1\}, \qquad \forall c \in C, \forall w \in \text{face}(W) \tag{13}$$

The ε is used so that we will always prefer vertex guards over face guards, i.e., if there is a possible solution with m vertex guards we will always prefer it over a mixed solution with m vertex and face guards. If we choose $0 < \varepsilon \leq \frac{1}{|C|+|W|+1}$ we will guarantee that ε is small enough to prefer vertex guards over face guards, but not too large to make sure it always chooses the solution with the lowest number of guards in total. Note that we also include a special rule concerning witness faces. The variables $[\![w]\!] \in \text{face}(W)$ denote whether or not a face is wholly seen by one of the guards in the solution set. We relax the IP a little bit by allowing some witness faces not to be wholly seen (12). Because of the other constraints, we know that its vertices are seen. This IP finds a lower bound to the optimal solution. This is because we have included face guards in the candidate guard set and we have relaxed the witness constraints. If there would be an optimal solution S_1 only containing point guards we can always convert it into an optimal solution of maximally the same size S_2 that contains faces of our arrangement as guards. This is because every point in S_1 must

be contained in a face in our arrangement and a face always sees at least as much as any of the points inside it. This means that if we find a solution when allowing face guards, we will always find solutions at least as good as the solutions found when only allowing point guards. Solutions are considered degenerate if they have a 1 for any of the variables $[\![c]\!]$ or $[\![w]\!]$, $c \in \text{face}(C), w \in \text{face}(W)$. This means that if, at the end of an iteration i, we find such a degenerate solution, we should update $\mathcal{A}_i$. Furthermore, we want the update $\mathcal{A}_i$ in such a way that in the next iteration we will find a different solution. We can achieve this by splitting the faces for which $[\![c]\!]$ or $[\![w]\!]$ was set for 1. The reasoning behind this is that if we find a solution using a face-guard, we would like to make that face-guard smaller so that in the next iteration, it will be able to see fewer witnesses. If we have a solution with a witness face that is not wholly seen by one guard, we want to make it smaller, such that in the next iteration, it may be wholly seen by one guard. The different methods used for splitting are discussed in the next paragraph. The methods in which we split, guarantee that after a certain number of iterations we will always find a solution that only uses point-guards. Since the normal IP guarantees a lower bound if there are no partly seen face witnesses, this solution is optimal, which means that we are done. Note that this guarantee only works under some assumptions. Some polygons, such as one described in [2], require guards that have irrational coordinates. For this polygon, the algorithm will never finish. However, after running some tests, we can show that the intermediate guard sets found by the iterative algorithm converge to the optimal solution. More about this experiment can be found in Section 5.5.

Splitting methods and the Power of a Face. For the iterative algorithm, we used different splitting methods to split a face f. The most simple type of split is called a *square split*, shown in Figure 18 (a). To perform this split we first identify the orthogonal bounding box of f. Then, we draw a horizontal and a vertical segment inside this box that intersect in the center. We use these two segments to split f. When f is incident to two reflex vertices after a square split each new face will only be incident to at most one reflex vertex (see (Figure 19 (a)). This split performs well in practice and is easy to implement. However, from a theoretical standpoint, the square split is not very useful. The reason for this is that it is hard to show an upper bound on the number of square splits needed before finding an optimal solution. For the theoretical guarantees to exist, different types of splits are necessary. How these splits lead to the theoretical guarantees is discussed in more detail by Hengeveld and Miltzow [17]. The first type of these splits, the *angular split*, is justified using a concept that we call the *power of a face*. The power of a face quantifies how much a face can maximally see "around a corner". This is determined by the reflex vertices that are visible from this face. For each reflex vertex r that a face f can see, we define the power-angle $\alpha(r)$, as the angle of the minimum cone with apex r that fully contains f. We then define the power of f as the maximal power-angle amongst the reflex vertices that are seen by f. The other three

types of splits make use of reflex vertices seen by f. The goal of the angular split is to reduce the power of f after each split. An example of an angular split is shown in Figure 18 (b). To do this split, we take all reflex vertices that see f and shoot rays in 2^k equally diverging directions, for a given parameter k. To make sure we reduce the power of f using these rays, we would want to use the sine and cosine function here. However, this is not possible without losing some precision. Instead, we use a workaround. We imagine a bounding box around the reflex vertex and use this box to shoot the rays as shown in Figure 19 (c).

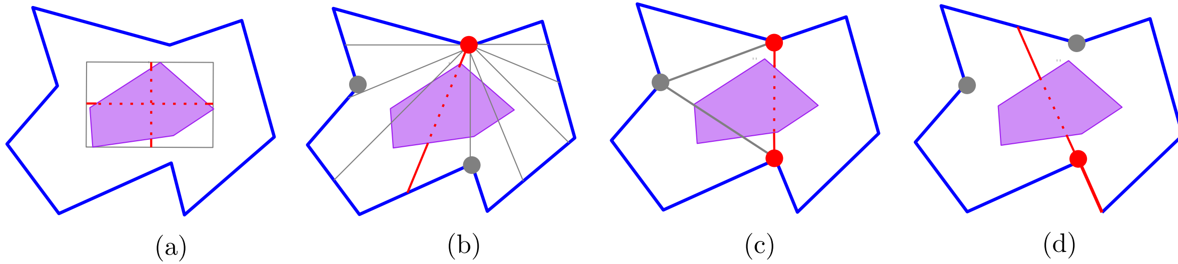


Figure 18: This figure shows four types of splits. In (a) we see an example of a square split. Next to it, (b) shows an angular split. The figure in (c) contains a reflex chord split. Finally, (d) shows an extension split. The reflex vertices in red are the ones we used for the splits.

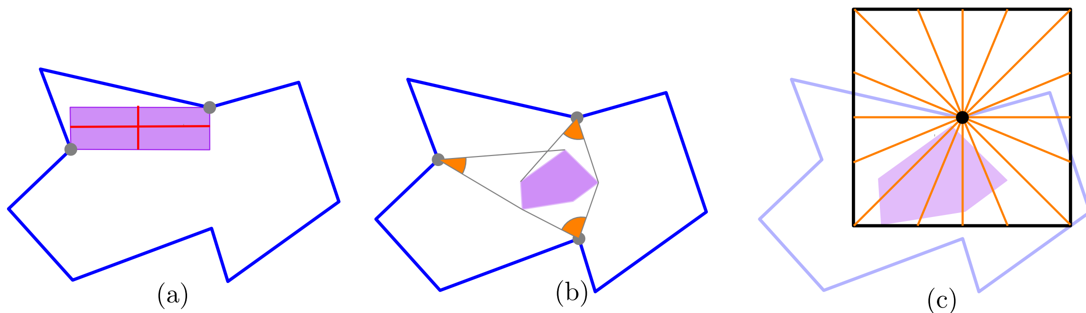


Figure 19: (a): After a square split, none of the new faces are incident to two reflex vertices. (b): The power of f is the maximal angle from a reflex vertex visible from f. (c): We use a bounding box around the reflex vertex to shoot the rays for the angular split.

We then identify the reflex vertex which has the most intersecting rays and use its middle ray to split f. We start with $k = 4$, and double it if none of the reflex vertices has any rays that intersect f. In practice, we do not shoot all rays from all reflex vertices. A more efficient way of doing this in practice is by using a technique similar to a binary search to find the rays intersecting f. Another useful split that we will discuss is the *reflex chord split*. This split is illustrated in Figure 18 (c). In this split, we identify all chords between reflex vertices that are seen by f. If any such chord exists, and it properly intersects f, we use it to split f in two parts. In practice, we save all the reflex chords at the start of the algorithm. This way

we do not have to compute them more than once. The final type of split is the *extension split*, shown in Figure 18 (d). Again, we identify the reflex vertices that f sees. Then, we extend the two edges that are incident to the reflex vertex into rays. If either of the rays of a reflex vertex r intersect f, we use it for the split. The usefulness of this split is that it produces points in $\mathcal{A}$ that are useful guard candidates because they can perfectly look around the corner created by r.

For the practical version of the iterative algorithm, we mostly use these last three types of splits because they have a theoretical basis. Square splits are only used when f is incident to more than one reflex vertex. Otherwise, we use a random number to choose the other types of splits. With 80% probability, we will perform an angular split. The other two splits both have a 10% probability of occurring. If it turns out that a certain type of split is not possible, we try the other options. We used this approach in all experiments, except for the one described in Section 5.5. In that experiment we only used square splits because that led to better results.

Vision-stability and granularity. An important factor that is part of one of the assumptions which lead to the theoretical guarantees of the iterative algorithm is *vision-stability*. Imagine a new version of the Art Gallery problem in which guards have *enhanced* or *diminished* vision. This version of the problem is parameterized by an angle of δ. When the guards have enhanced vision, they can look “around a corner” by δ. When the guards have diminished vision, their vision is “blocked” by a corner with an angle of δ. We define the visibility polygon of a guard p with δ-enhanced vision as $\text{vis}_\delta(p)$ while the visibility polygon of a δ-diminished guard p is defined as $\text{vis}_{-\delta}(p)$. See Figure 20 to see an example of both of these types of vision.

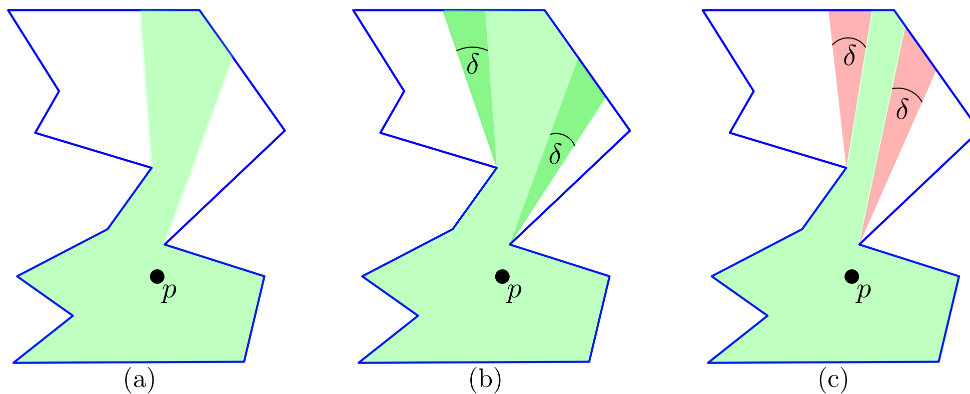


Figure 20: In (a) p is a normal guard and $\text{vis}(p)$ is shown in green. Next to it, (b) shows a δ-enhanced guard p. The parts in darker green are added to its visibility polygon. The figure in (c) contains $\text{vis}_{-\delta}(p)$, the parts removed from its visibility polygon are shown in red.

This leads to the definition of vision-stability. We say a polygon P has vision-stability δ if the size of an optimal solution using δ-enhanced guards is the same as the size of an optimal solution using δ-diminished guards. For some polygons, particularly those that have optimal solutions that rely on collinearity, no δ exists for which the above is true. These polygons are not vision-stable. The iterative algorithm only has the theoretical performance guarantees for polygons that are vision-stable. However, in practice, the iterative algorithm will often still find the optimal solutions for these polygons. This is because the extension split described above makes sure that the right guard candidates are added to the arrangement in collinear positions. An example of such a non-vision-stable polygon is shown in Figure 21. This polygon has one unique solution which relies on collinearities.

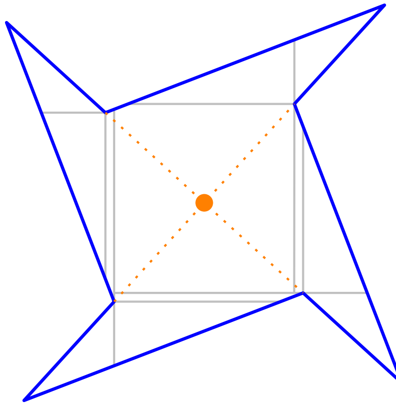


Figure 21: The polygon shown here is not vision-stable because the solution relies on colinearity. The orange line segments show possible extension splits that we could make. If we make at least two of these, we will find the solution.

If we split faces at least twice using extension splits, we will add the candidate point in the middle, enabling us to find the right solution. There is one polygon for which the iterative algorithm cannot find the optimal solution. This is a polygon that is not vision-stable and requires guards at irrational coordinates, as described in [2]. However, Section 5.5 describes an experiment in which we show that for this polygon, the iterative algorithm does provide a series of “degenerate” solutions that use face-guards or allow some faces to not be seen completely. Moreover, this series of solutions converges to the optimal solution. Also, only one such concrete polygon is known at the time of the writing of this thesis. Another interesting aspect of the vision-stability of a polygon is that has explanatory value for the running time of the iterative algorithm. The lower δ, the more splits we have to do to find an optimal solution. We also found this correlation during our practical experiments. Recall that to perform an angular split we shoot 2^k rays, for a value k that we double throughout the iterations when necessary. We then call $\lambda = 2^{-k}$ the *granularity* of such a split. The lowest granularity we need before finding an optimal solution can be used as a crude estimation of

the vision-stability of an input polygon. Section 5.1 shows that we found fairly strong positive correlations between this minimal granularity and the running time of the iterative algorithm.

4.1 Speedup methods

In this section, we will explore two ways in which we improved the running time of the algorithm. The first way a data-structure that can be used to both reduce the size of the shortest path map and answer certain visibilities queries efficiently. The second way involves making the set of witnesses smaller, while still obtaining qualitative solutions at each iteration. This way the IP has to deal with fewer variables and we have to compute fewer visibilities. The effect that these methods have on the running times of the iterative algorithm is shown in Section 5.2.

Weak visibility polygon tree. In large polygons, many vertices cannot see each other. This is because large polygons will most likely contain many turns and reflex vertices, which will obstruct many visibilities. The idea behind the weak visibility polygon tree is that if we save the location of the points and faces of the arrangement in an useful manner, we can prevent the unnecessary computation of many visibilities. We achieve this by dividing the arrangement into weak visibility polygons. Figure 22 shows an example of a large polygon divided into weak visibility polygons. This figure was generated using the practical implementation.

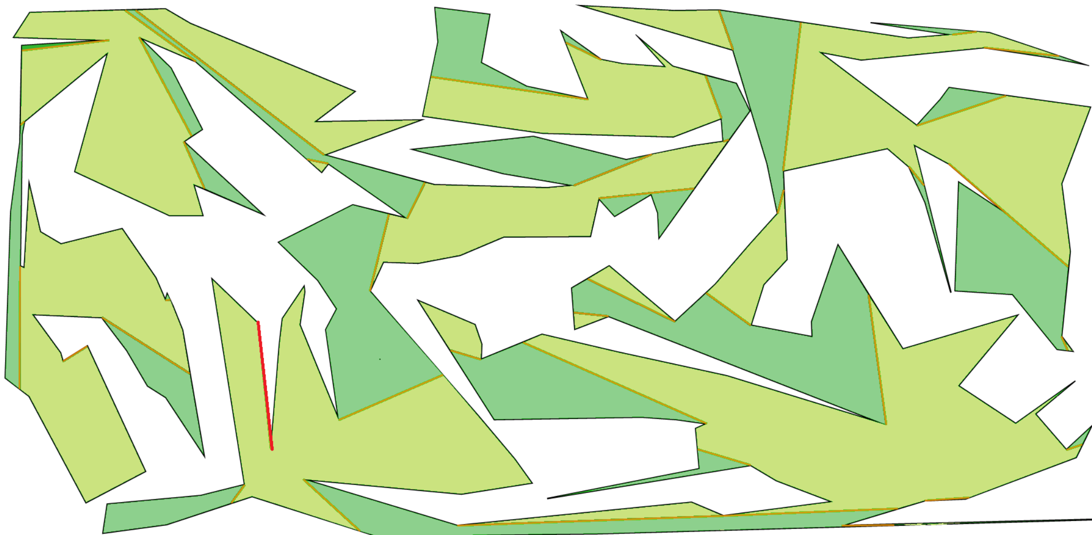


Figure 22: The polygon together with a weak visibility polygon tree. The polygon has 200 vertices, but each node in the weak visibility polygon tree has only about 20 vertices. The red segment indicates the starting edge of the weak visibility polygon tree.

To build the weak visibility polygon tree, we begin by picking a random starting edge. This edge is shown in red in Figure 22. We compute the weak visibility polygon of this edge, using the subroutine described in Section 2.2. This edge and its visibility polygon become the root of the tree. Afterwards, we identify the doors introduced by the weak visibility polygon shown in orange in Figure 22. For each of these doors, we compute its weak visibility polygon inside the area of the polygon that is left. These visibility polygons are considered children of the root node. We repeat this process recursively for each of these new polygons, adding nodes to the tree until all the weak visibility polygons cover the entire polygon. Given this tree, we know that any point within a weak visibility polygon can only be visible from points in either in the parent, direct children or sibling visibility polygons. The proof for this can be found in [17]. Using this knowledge, we can optimize the shortest path map described in Section 2.4. For all the vertices that we know cannot see each other as a result of the weak visibility polygon tree, we do not have to compute or store the paths between them. This means a dramatic reduction in both the memory of the data-structure and the running time of our visibility queries.

Critical witnesses. During the iterations, the arrangements $\mathcal{A}_i$ have many vertices and faces that are not particularly important to be guarded. The idea of the critical witness set is to identify a subset $W^* \subseteq W$ of critical witnesses which are relevant. At the beginning, we initialize the *critical witness* set W^* by randomly picking 10 percent of vertices and faces, for each weak visibility polygon separately. In this way, the critical witness set is roughly equally distributed over the whole polygon. Later, throughout the iterations, we add to the critical witnesses set as necessary, by using the guards G given by the integer program. See Section 4 for a detailed description of the integer program. We compute all the vertices and faces that G sees. This also gives us the sets of unseen face-witnesses and vertex-witnesses U which were not marked as critical before. We then randomly choose a small constant size subset U^+ of vertices and faces from U that we add to W^*. The size of U^+ is an interesting parameter to the program. In the practical implementation tested in [17] and in Section 5.1, 5.3, 5.5 and 5.6 this size depends on the number of cores of the processor of the system that it is run on. Using the number of cores is plausible because our practical implementation runs the visibility queries in parallel. In Section 5.4 we discuss an experiment in which different sizes for U^+ were tested. Note that if we were to mark all unseen witnesses as critical this would lead to very large numbers of visibility queries, thus defeating the purpose of using the critical witness set. It would also increase the size of all subsequent integer programs that we need to solve. It is important to find a good balance between adding too few critical witnesses and adding too many.

Every time we update the critical witness set, we re-run the IP to check if we can find a better solution given the critical witnesses. We keep adding to the critical witness set as long

as there are unseen witnesses left that are not marked as critical. We then check if we need to update the new critical witness set again using this new-found guard set. We keep doing these *critical cycles*, until we find a guard set that can see the entire polygon. Only then we split the faces and continue with the next iteration.

A face can only be removed from the critical witness set if it is split. For every critical witness face, we also add a critical vertex to the interior of that face. Critical witness vertices are only removed from the critical witness set, if they are interior to a face that is removed.

To summarize, we do not need to compute all the visibilities between all the candidates and all the witnesses. Instead, we compute all visibilities between the critical witnesses W^* and candidates C and then all visibilities between the guards G and the witnesses W. As there are much fewer guards than candidates and also much fewer critical witnesses W^* than witnesses overall, this saves a lot of running time in practice.

5 Experiments

Various experiments were conducted with the algorithm. Section 5.1 described an experiment in which the running time of the algorithm was tested on several input polygons. The goal of this experiment was to find out more about the average running time of the practical implementation, and how it correlates with certain factors such as size and vision-stability. Secondly, in Section 5.2, we also discuss how the two speedup methods, the weak visibility polygon tree and the critical witnesses, improve the running time.

In Section 5.3, we discuss tests in which we solved the same polygons several times, in order to find out the standard deviation introduced by the randomness of different parts of the iterative algorithm. Furthermore, different ways of adding critical witnesses are tested and analyzed in Section 5.4. Section 5.5 discusses tests on the irrational-guard polygon from [2], showing that the iterative algorithm finds a series of solutions that converge to the optimal solution. Finally, Section 5.6 analyzes the CPU usage of the program and identifies where future improvements could be made. All experiments conducted use the version of the algorithm that uses all speedup methods. Except for the experiment on the irrational-guard polygon, all tests were also done using the normal split protocol.

Apart from the polygon with irrational guards [2], all input polygons were obtained from AGPLIB library [10], a library used for other papers on the Art Gallery problem as well [11]. This library offers different types of randomly generated polygons, that are described in Section 3.2 and illustrated in Figure 14. These polygons were generated by the research group at the University of Campinas while developing practical solutions for the Art Gallery problem. For all our experiments, except the one discussed in Section 5.5, we used simple, non-orthogonal polygons, the class illustrated in 14 (a). Figure 23 also shows an example of such a polygon. The polygons are all similar to the polygon showed in 23. They generally have small to medium-sized rooms and a larger number of corridors. The experiments were run on a computer with a 64-bit Windows 10 operating system, an 8-core Intel(R) Core i7-7700HQ CPU at 2800 Mhz and 16 GB of main memory.

The practical implementation heavily makes uses of version 4.13.1 of CGAL [23]. The IP solver used was IBM ILOG CPLEX version 12.10 [5]. This IP solver was also used by de Rezende et al. [11].

5.1 Practical running time

In this experiment, the practical running time of the fastest variant of the iterative algorithm was tested. Note that in [17], we also tested a variant that is based more on theory. However, in practice, this variant could not always find the optimal solution. The results that we found using the more practical variant of the algorithm always led to the optimal solution and did so in a reasonably efficient manner.

As mentioned previously, the input polygons were taken from the AGPLIB library [10]. We had access to random simple polygons of different size classes. We tested on 30 instances of polygons of sizes 60, 100, 200 and 500 vertices. An example of one of the 200-vertex polygons and its solution is shown in Figure 23.

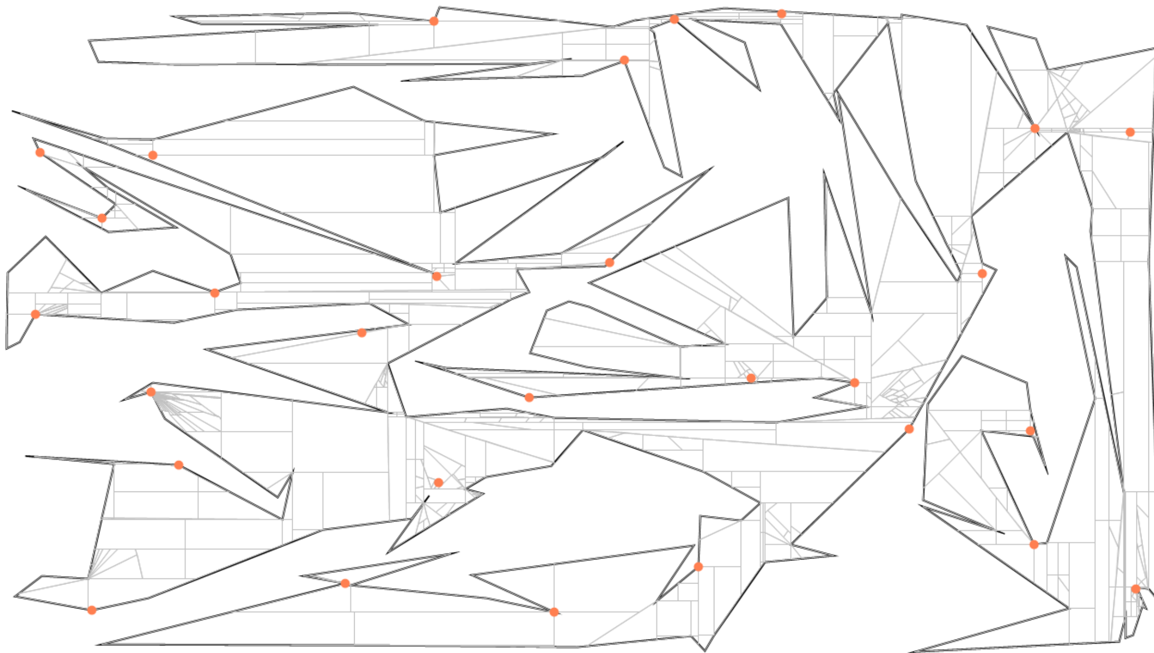


Figure 23: An example of a 200-vertex input polygon and its solution, as found by the iterative algorithm. We have also included the final arrangement at the end of the algorithm.

The averages and medians for the size classes tested are shown in Table 3. Note that the times we report exclude the pre-processing time of computing the weak visibility decomposition before the first iteration. This table also shows the results by Tozoni et al. [25] pertaining to polygons of these sizes. We acknowledge the fact that the results from Tozoni et al. [25] were improved on by de Rezende et al. [11]. However, we were not able to find the exact running times from this improved version corresponding to these sizes. This table shows that the running time of our algorithm is not faster than the algorithm from Tozoni et al. [25], but it is not too far behind. Also, the median running times of this variant of the iterative algorithm is lower than the average running times. We believe this is, amongst other reasons, because the algorithm is sensitive to the vision-stability of a polygon. This means that some polygons are very hard to solve and overly influence the average. The results in Table 2 confirm that this is indeed true. In Table 2 we see all the individual running times of every polygon. Furthermore, see Appendix A for an illustration of how the algorithm finds a solution for a 60-vertex polygon in 9 iterations.

Size	60	100	200	500
Correlation	0.73	0.38	0.77	0.45

Table 1: The correlation coefficients between the measured minimum granularity λ and the running time, computed per size.

Sizes	**Average time (s)**		**Median time**
	Tozoni et al. (2016)	**Ours**	
60	0.26	0.67	0.47
100	0.94	2.45	1.6
200	3.77	19.67	6.29
500	35.04	64.52	56.49

Table 3: A comparison of the iterative algorithm with the results from Tozoni et al. [25]. Note that the reported times for our algorithm do not include the pre-processing time and that the results from Tozoni et al. [25] were found on a system with slightly less powerful hardware: Intel Core i7-2600 at 3.40GHz and 8GB of RAM

Correlation of granularity and running time. As mentioned before, the iterative algorithm is sensitive to the vision-stability of the input polygon. To test this, we saved the smallest granularity λ necessary to find the solution. We then computed the correlation coefficients between the running times and the granularity λ. These coefficients are shown in Table 1. These are fairly strong correlations. Note that the minimum granularity λ might not be the best indication of the vision-stability of a polygon. This is because a larger polygon might need a very fine subdivision in one part of the polygon, but can be relatively coarse in other parts. We have no efficient way to compute the vision-stability efficiently.

Besides this, there are several other random factors which influence the running time. For example, the IP chooses an arbitrary optimal solution out of several possible options and the splits we do at each iteration are also chosen randomly. Additionally, the complexity of the weak visibility polygon tree has some effect on the running time. In particular, the number of weak visibility polygons inside the tree and their size have fairly strong effects on the running time. We believe that these factors account for the fluctuation in the correlations. Section 5.3 has a more in-depth analysis of the standard deviations in the running time of instances of the same input polygons.

Running time (s)			
n = 60	**n = 100**	**n = 200**	**n = 500**
0.07	0.24	0.60	6.09
0.10	0.42	1.30	8.37
0.15	0.42	1.31	9.69
0.18	0.43	1.56	10.31
0.18	0.46	2.45	11.74
0.20	0.51	2.59	21.24
0.21	0.53	3.33	21.50
0.22	0.72	3.82	22.87
0.26	0.74	4.16	23.70
0.26	0.82	4.45	25.36
0.26	0.82	4.79	26.09
0.27	0.87	4.90	51.69
0.30	0.88	5.00	53.08
0.31	1.21	5.39	55.87
0.34	1.53	6.00	57.11
0.47	1.67	6.26	59.71
0.47	1.74	6.32	60.37
0.48	1.74	6.78	61.86
0.53	1.95	7.07	68.00
0.53	2.32	7.62	73.07
0.56	2.92	9.28	79.93
0.73	4.10	11.00	85.69
0.76	4.48	13.53	85.95
0.78	4.70	16.75	94.95
1.06	4.78	18.12	115.46
1.28	4.94	21.39	120.40
1.62	5.67	37.82	128.37
1.68	6.36	40.48	151.03
2.10	7.57	150.98	160.68
3.78	7.96	184.81	185.56

Table 2: The full results of the experiment. For every one of the 4 sizes, we tested 30 polygons. We see, in seconds, how long it took for the iterative algorithm to find the optimal solution for each instance. The running times are ordered in an ascending manner.

5.2 The effect of the speedup methods

To find out the added value of the two speedup methods, we will look at different statistics that were measured in the experiments. We first look at the effect of the weak visibility polygon tree and then at the effect of the use of the critical witnesses. We show that both methods offer large speedups.

The value of the weak visibility polygon tree. In the experiment from the previous section, we also measured several things about the weak visibility polygon trees computed for the different input polygons. In particular, we tracked three different characteristics of the weak visibility polygon tree: the number of weak visibility polygons in the tree, the largest number of normal vertices and reflex vertices of the largest weak visibility polygon in the tree. Table 4 summarizes these three characteristics for each size class.

We see that the tree size seems to almost grow linearly with the size of the polygon. However, the other two statistics grow much more slowly. This attests to the effectiveness of the weak visibility polygon tree because it means that even the larger polygons are divided into relatively small weak visibility polygons. This prevents the computation of many unnecessary visibilities throughout the course of the iterative algorithm.

Size	60	100	200	500
Tree size	14.2	24.2	50.6	116.2
Largest polygon	45.6	50.7	58.6	66.7
Largest number of reflex vertices	10.8	11.0	13.7	15.0

Table 4: We tested 30 input polygons from the AGPLIB library [10] of four sizes. For each size class we see the averages of three characteristics of the weak visibility polygon trees.

Critical witnesses speedup. To find if our second speedup method was also as effective, we conducted an additional experiment. We tested the same polygons of sizes 60, 100, 200 and 500 without critical witnesses. Table 5 compares the running times of the versions with and without critical witnesses. Additionally, we see the number of witness points and faces used on average. For the method without critical witnesses, we list the number of total witnesses while the method with critical witnesses shows the number of critical witnesses only. The table shows that the critical witnesses cause large speedups of up to a factor 9. When we compare the size of the critical witness sets to the total witness sets, we see that the ratios are about 3 : 1 and 5 : 1 for points and faces respectively. This means we have to compute much fewer visibilities when using critical witnesses, and the IP has to deal with smaller sets of variables. This confirms that the overhead of maintaining and adding to the critical witness set is small compared to the time that we save by using it.

Sizes	Average time (s)			Witness points		Witness faces	
	With	**Without**	**Speedup**	**With**	**Without**	**With**	**Without**
60	0.79	7.36	9.4	172.48	518.31	39.55	211.24
100	2.53	19.67	7.8	314.27	914.59	73.13	376.24
200	22.15	59.53	2.7	801.93	2012.38	215.23	842.48
500	63.99	320.64	5.0	1768.86	5256.85	426.90	2192.10

Table 5: A comparison of the iterative algorithm with and without the use of critical witnesses, tested 30 polygons of sizes 60, 100, 200 and 500. We compare the running times and the number of witnesses used in the IP. Depending on the version of the algorithm, the witnesses shown are either the number of critical witnesses or the total number of witnesses.

5.3 Standard deviation

In the previous section, we explained that the differences between the running times for polygons within the same size class can be very large. However, it would also be interesting to know the standard deviation of the running times if we test the same polygon multiple times. For this experiment, we ran the algorithm 15 times on each of the 30 input polygons of size 100 and measured the running times and computed the averages standard deviations of each input polygon. These statistics are visualized in a graph in Figure 25. We see that for some polygons the standard deviations can be quite large. Several random factors in the iterative algorithm could account for these standard deviations. Firstly, the weak visibility polygon tree has a small degree of randomness in its construction. This is because we choose a random first edge. However, the results showed that for the 15 different instances of the same polygon, the complexity of the weak visibility polygon tree did not change much. In fact, the number of weak visibility polygons and the largest weak visibility polygon were often very similar. Next, we randomly choose the type of split that we perform. For some polygons, this may be largely responsible for the deviation from the average. That is because if a solution relies on collinearity, an extension or chord split can often create a candidate vertex at the necessary position. Interestingly we see that for the same polygon the minimal granularity varies. When we look at the instances with lower granularity, it seems that extension splits and chord splits were used more often than the instances with higher granularity for the same polygon. Figure 24 (a) and (b) show a simple example of how this could happen. This suggests that we can lower the standard deviation and minimal granularity by doing these types of splits more often. Finally, when several solutions are available, the IP solver will choose a random one. This means that the solver sometimes chooses a face that is useful to split and other times it may choose a face that is not so interesting.

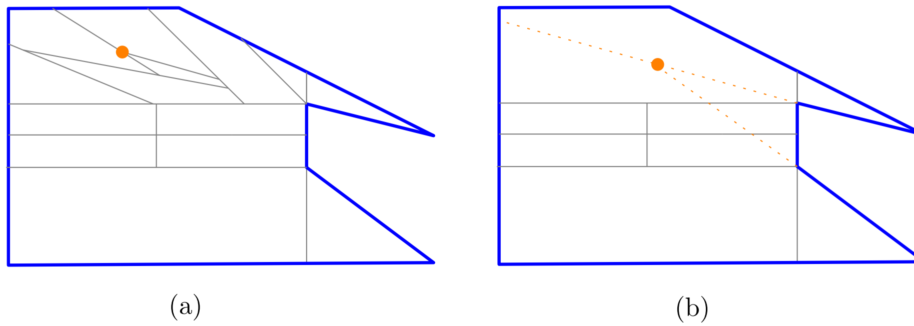


Figure 24: In (a), we perform several angular splits in a row. We are unlucky, and never choose to do an extension split. The minimum granularity gets quite low before we find a solution. In (b) we are lucky and do two extension splits in a row. The minimum granularity is never updated.

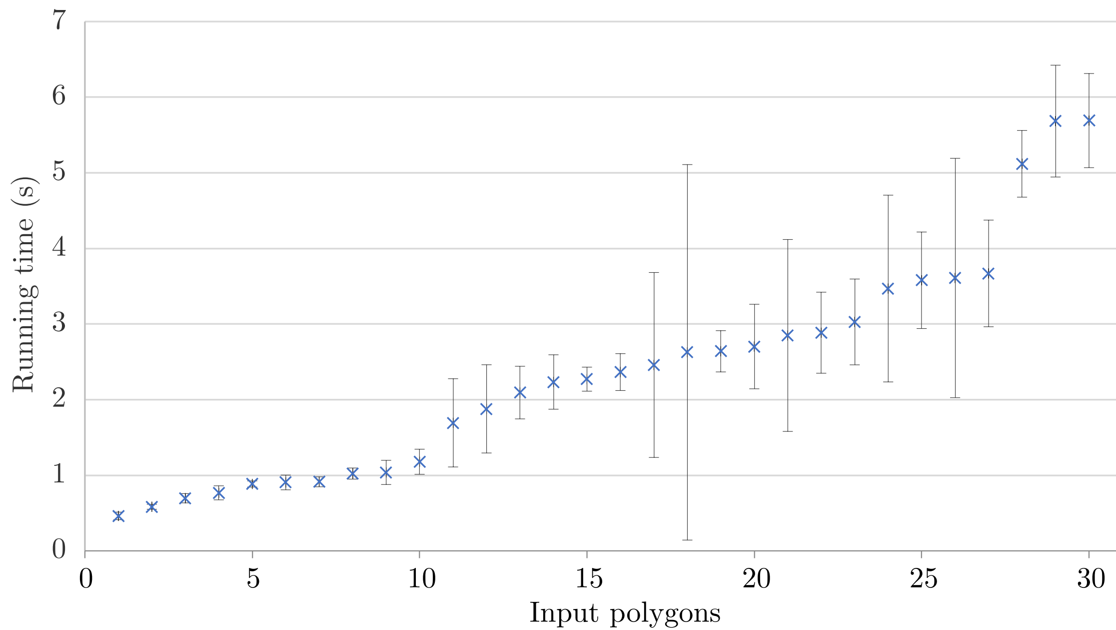


Figure 25: We tested 30 polygons of size 100 from the AGLIP [10]. Each polygon was tested 15 times. This graph shows the standard deviation of the running times for each of the 30 polygons.

5.4 Critical witness sizes

In Section 4.1 we discussed the concept of critical witnesses, which gave the practical implementation a huge speedup. A parameter to this approach is the number of witnesses U^+ from the unseen witnesses U we add to W^*. Two considerations come into this. On the

one hand, we do not want W^* to become too large because this would mean that we would have to solve a lot of unnecessary visibility queries. On the other hand, if we add very few critical witnesses before solving the IP again, we might be wasting a lot of time on the high number of instances of the IP. This is because we will keep finding candidate solutions which see all critical witnesses, but perhaps not all witnesses. If we add more critical witnesses to W^*, the quality of these solutions will be higher, requiring fewer iterations of the IP. For the algorithms used in most of the experiments we discussed here, we based the size of U^+ on the number of physical cores of the system the program is run on. The reasoning behind this is that we do the face-visibility queries in parallel. As Section 5.6 will show, these queries are the bottle-neck of the running time. It makes sense to do at least as many queries as CPU cores that we have available because the queries will be computed in parallel. The experiment that we will describe here was conducted to verify whether this is indeed correct in practice. Moreover, perhaps it is more advantageous to add even more witnesses to W^* every cycle. In this experiment, we tested 30 input polygons of size 100 and 30 of size 500, with different sizes of U^+, ranging from 1 to 16. Considering that we found a large standard deviation in the running times of the input polygons in the previous section, we tested every size of U^+ 5 times and averaged the resulting running times. Table 6 shows the results of this experiment. We see that for size 100, 4 seems to be the best option, while for size 500, 8 is the best. This can be explained because, for larger sizes, we will have a larger set of witnesses. This means that U will be larger at every critical witness cycle. Then it makes sense to use larger values for U^+ as well. Also, note that the system we tested on only has 8 cores. Then it makes more sense for U^+ to also be slightly larger. Perhaps the best option would be to make the size of U^+ depend on a combination of the number of cores in a system and the input size of the polygon.

$\lvert U^+ \rvert$	1	2	4	8	12	16
Running time (s), $n = 100$	2.85	2.62	2.14	2.53	2.66	2.81
Running time (s), $n = 500$	99.82	78.61	72.30	68.89	71.68	94.07

Table 6: For every different size of U^+ we tested 30 polygons of size 100 and 30 polygons of size 500. We tested each input polygon 5 times and averaged all the results. Note that these tests were done using an 8-core Intel(R) i7-7700HQ CPU.

5.5 Converge to the optimal solution

In this experiment, we ran the iterative algorithm for 30 minutes for the irrational-guard-example from [2]. For these experiments, we only used square splits. The results with angular splits and reflex chord splits are similar in spirit, but the convergence is slower. Appendix A

shows illustrations of the first 22 iterations, demonstrating the way we split faces and how the intermediate solutions approach the optimal solution. Additionally, see Figure 26 for an illustration of this special polygon and its optimal solution. What makes this polygon so interesting is that the 3 point guards in the solution are at irrational coordinates.

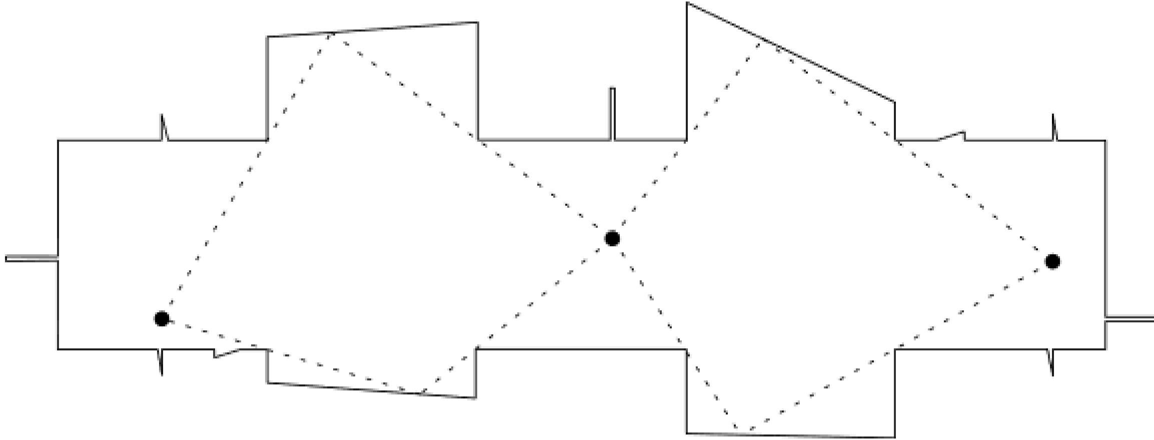
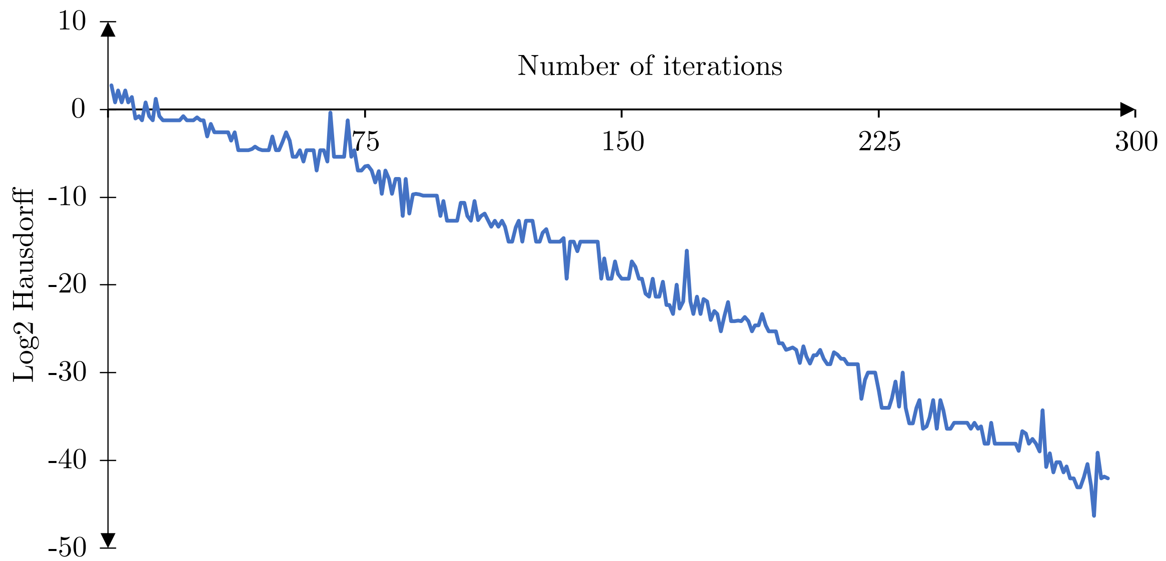
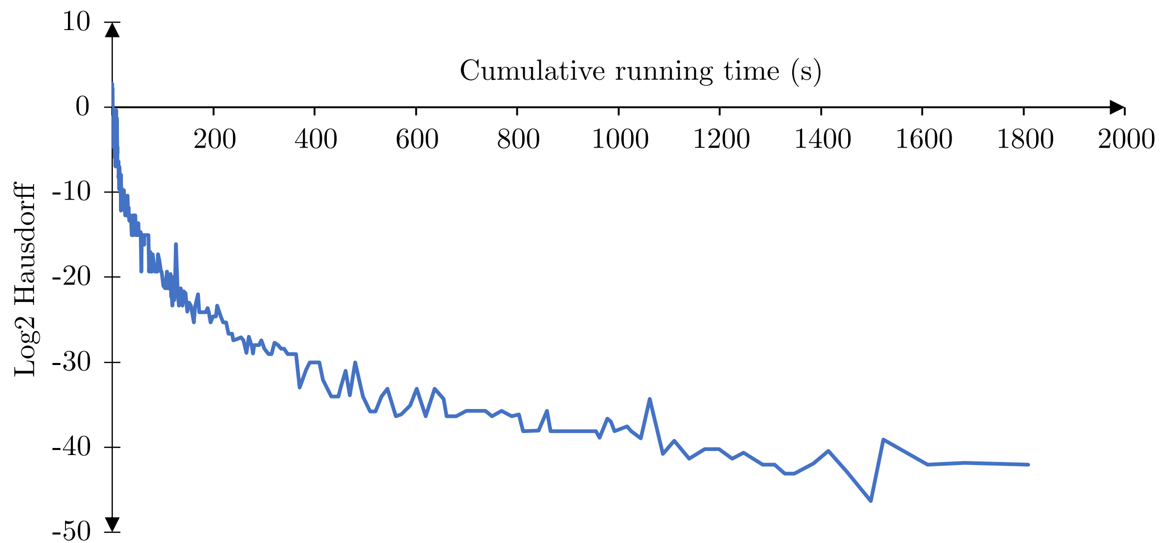


Figure 26: This polygon has a unique solution with three guards that have irrational coordinates guards [2].

Even though the algorithm will not be able to find the optimal solution, we get a set of guards G_i at the end of iteration i. Note that G_i is often a combination of point-guards and face-guards. As we know the optimal solution F, we can compute the Hausdorff distance $d_i = \text{dist}_{\text{H}}(G_i, F)$ at the end of each iteration. See Figure 27 (a) for an illustration of the convergence speed per iteration. Interestingly, the distance approaches the optimum very quickly. Additionally, Figure 27 (b) shows the Hausdorff distance plotted against the cumulative running time. Here we see that when plotted against the running time, the Hausdorff distance does not decrease as dramatically. This comes from the fact as the algorithm progresses, the later iterations take much more time per iterations. This is plausible because at the end of every iteration we introduce new candidates and witnesses. Thus, in the later iterations, the IP has to deal with more candidate variables than at the earlier iterations. Having a larger number of candidates and witnesses not only slows down the IP but also means we have to compute a larger number of visibilities. Furthermore, as we see in the next section, the face visibility queries are the bottle-neck of the CPU-time.



(a)



(b)

Figure 27: The iterative algorithm reports a sequence of solutions. Graph (a) shows on the x-axis the iterations from 1 to about 300 and on the y-axis, the $\log_2$ of the Hausdorff distance to the optimal solution. Graph (b) shows on the x-axis the cumulative running time and on the y-axis, the $\log_2$ of the Hausdorff distance to the optimal solution. All times are in seconds.

5.6 Distribution of CPU usage

This section describes how the workload of the algorithm is distributed on the CPU. To achieve this, we picked three polygons of different sizes and analyzed their CPU usage.

These polygons were chosen how their running times compare to the average of their size class. The first, smaller polygon performed slower than the average. The second, 500-vertex polygon performed around average while the last 800-vertex polygon was solved faster than average. The 200-vertex polygon used for this analysis is visualized in Figure 23. We performed this analysis using the Visual Studio profiler [19]. Figure 28 shows the results. The CPU usage distribution for other polygons that we tested seemed very similar. We divided the algorithm into several parts. These different parts are shown in a pie-chart in Figure 28.

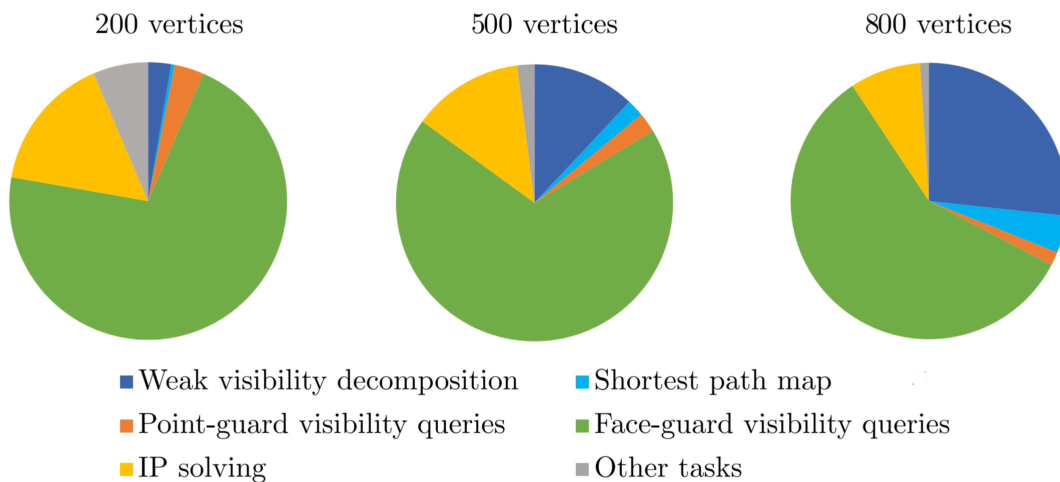


Figure 28: The left chart shows this distribution for a 200-vertex polygon that was slower than average. The middle chart shows the distribution of the workload for a 500-vertex polygon, with average completion time. Finally, the right chart shows the distribution for a faster-than-average 800 vertex polygon.

The first parts, in dark and light blue, are both parts of the pre-processing step. As described in Section 2.4 and Section 4.1, this is the time needed to set-up the shortest path map we compute weak visibility polygons and shortest paths. We show in dark blue the time it took to compute the weak visibility polygons of each window while we show in light blue the time spent finding the necessary shortest paths within each of these weak visibility polygons. The part shown in orange is point visibility queries, computing whether a point sees a witness face or witness point. Practically, this is achieved by using the method described in Section 2.3. The chart shows in green the percentage of the CPU time spent on face visibility queries, the question of whether a candidate face guard can see a face or point witness. We also compute these visibilities when checking whether we should add new critical witnesses, as we must

make sure that a candidate solution sees the complete polygon. These queries are solved using the shortest path map computed in the pre-processing stage and described in Section 2.4. These face guard visibility queries are done in parallel in our practical implementation. The yellow part shows the CPU time spent on solving the IP using the CPLEX solver. Finally, in gray, other, smaller tasks are combined into one section. These tasks typically consist of updating the intermediate arrangement, splitting faces using our different split techniques and keeping track of the results.

In the resulting figure we see that, for an average-time polygon such as the 500-vertex example here, most of the time is spent on face-guard visibility and solving the IP. This shows that much running-time improvement could be gained by improving the face-guard visibility routine. When we move on to the slower-than-average polygon, we can see that the workload of most tasks increase at the expense of the pre-processing part. This is logical because the pre-processing time does not depend on the difficulty of the polygon like the latter part of the routine does. Finally, looking at the 800-vertex polygon, the inverse happens. The pre-processing time takes a larger chunk of the distribution compared to the other sections of the workload.

The results shown here are for three specific polygons. For the most accurate results, it would have been desirable to repeat this experiment for a large number of polygons in the testbed. However, running the profiler slows down the algorithm, to the point that running such a large number of tests would be infeasible. Furthermore, after performing several more of these experiments, we found that these results generalize. From this, we can conclude that the running time of the algorithm is dominated by the face-visibility queries combined with the weak visibility decomposition. Both these subroutines involve computing weak visibilities. We thus believe that most speedup to the algorithm could be achieved by implementing more efficient weak visibility routines.

6 Discussion and Future research

In the previous sections we showed that the iterative algorithm performs quite well in practice. This is an important finding because it means that this algorithm is the first algorithm for the Art Gallery problem that both practical and has a theoretical upper bound on the running time [17]. We also provided empirical validation for the use of the two speedup methods introduced in Section 4.1. Both the use of critical witnesses and the use of the weak visibility polygon tree gave us large amounts of speedups. Moreover, we showed that the iterative algorithm provides a series of solutions that converge to the optimal solution for a polygon with guards at irrational coordinates [2]. As this is the only known concrete example of such a difficult polygon, so we cannot clearly say if this would be the case for all polygons with unique solutions with irrational coordinates. Future research could be done in this direction,

answering the question of whether we can find out if the iterative algorithm converges for all polygons that are not vision-stable.

Some results from our experiments suggest that further improvements could be made to the practical implementation of the algorithm. For instance, the way we handle critical witnesses could be optimized. The results from Section 5.4 imply that the size of U^+, the amount witnesses we add to the critical witness set, should by dependent on the amount of cores of the system and the size of the input polygons. Also, further experimentation could be done with the different types of splits. In Section 5.1 and Section 5.3 we demonstrated that the running times the iterative algorithm have large standard deviations, both when comparing polygons of the same size and instances of the same polygon. Making the protocol used for choosing split types more consistent and less dependent on probability might decrease these standard deviations. Finally, in Section 5.6 we saw that the most CPU time was spent on face visibility queries, despite the fact that we perform them in parallel. This indicates that future improvements could be made to the segment-point visibility queries. While we have done quite exhaustive tests on the randomly generated simple polygons that we used in the thesis, further experiments could be done with different types of polygons. For instance, the types displayed in Figure 14. Because of the fact that we use extension splits, it might be expected that the algorithms performs well on orthogonal polygons. However, how it performs for the other types of polygons remains a question. Finally, a very important limitation of the implementation is that it only works for polygons without holes. The weak visibility subroutines, both used for the weak visibility polygon tree and for face visibility queries rely on this assumption. It is also unclear if the algorithm would be much slower for polygons with holes. A possible way of computing the weak visibility polygon of a segment could be found by adapting the triangular expansion algorithm [7] so that it works with segments as well as points. This would largely depend on the implementation of the weak visibility subroutines for polygons with holes.

References

[1] Mikkel Abrahamsen. Constant-Workspace Algorithms for Visibility Problems in the Plane. Master's thesis, University of Copenhagen, 2013.

[2] Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. Irrational Guards are Sometimes Needed. In Boris Aronov and Matthew J. Katz, editors, *33rd International Symposium on Computational Geometry (SoCG 2017)*, volume 77 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. The Art Gallery Problem is $\exists\mathbb{R}$-complete. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 65–73, New York, NY, USA, 2018. ACM.

[4] Yoav Amit, Joseph S.B. Mitchell, and Eli Packer. Locating guards for visibility coverage of polygons. *International Journal of Computational Geometry & Applications*, 20(05):601–630, 2010.

[5] Robert Bixby. IBM ILOG CPLEX. `https://www.ibm.com/analytics/cplex-optimizer`.

[6] Andrea Bottino and Aldo Laurentini. A nearly optimal algorithm for covering the interior of an art gallery. *Pattern recognition*, 44(5):1048–1056, 2011.

[7] Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller. Efficient Computation of Visibility Polygons. *CoRR*, abs/1403.3905, 2014.

[8] Václav Chvátal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B*, 18(1):39–41, feb 1975.

[9] Kyung-Yong Chwa, Byung-Cheol Jo, Christian Knauer, Esther Moet, René van Oostrum, and Chan-Su Shin. Guarding art galleries by guarding witnesses. *International Journal of Computational Geometry & Applications*, 16(02n03):205–226, 2006.

[10] Marcelo C. Couto, Pedro J. de Rezende, and Cid C. de Souza. Instances for the Art Gallery Problem, 2009. `www.ic.unicamp.br/~cid/Problem-instances/Art-Gallery`.

[11] Pedro J. de Rezende, Cid C. de Souza, Stephan Friedrichs, Michael Hemmer, Alexander Kröller, and Davi C. Tozoni. *Engineering Art Galleries*, pages 379–417. Springer International Publishing, Cham, 2016.

[12] Alon Efrat and Sariel Har-Peled. Guarding galleries and terrains. *Inf. Process. Lett.*, 100(6):238–245, 2006.

[13] Stephan Eidenbenz, Christoph Stamm, and Peter Widmayer. Inapproximability results for guarding polygons and terrains. *Algorithmica*, 31(1):79–113, 2001.

[14] Efi Fogel, Dan Halperin, and Ron Wein. *CGAL arrangements and their applications: A step-by-step guide*, volume 7. Springer Science & Business Media, 2012.

[15] Stephan Friedrichs. Integer solutions for the art gallery problem using linear programming. Master's thesis, TU Braunschweig, 2012.

[16] Leonidas Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1-4):209–233, 1987.

[17] Simon Hengeveld and Tillmann Miltzow. A Practical Algorithm with Performance Guarantees for the Art Gallery Problem. *arXiv preprint arXiv:2007.06920*, 2020.

[18] Alexander Kröller, Tobias Baumgartner, Sándor P. Fekete, and Christiane Schmidt. Exact Solutions and Bounds for General Art Gallery Problems. *J. Exp. Algorithmics*, 17, May 2012.

[19] Microsoft. Visual Studio Profiler. `https://docs.microsoft.com/en-us/visualstudio/profiling/?view=vs-2019`.

[20] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, Inc., USA, 1987.

[21] Sylvain Pion and Andreas Fabri. A generic lazy evaluation scheme for exact geometric computations. *Science of Computer Programming*, 76(4):307 – 323, 2011. Special issue on library-centric software design (LCSD 2006).

[22] T. C. Shermer. Recent results in art galleries (geometry). *Proceedings of the IEEE*, 80(9):1384–1399, 1992.

[23] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14.1 edition, 2019.

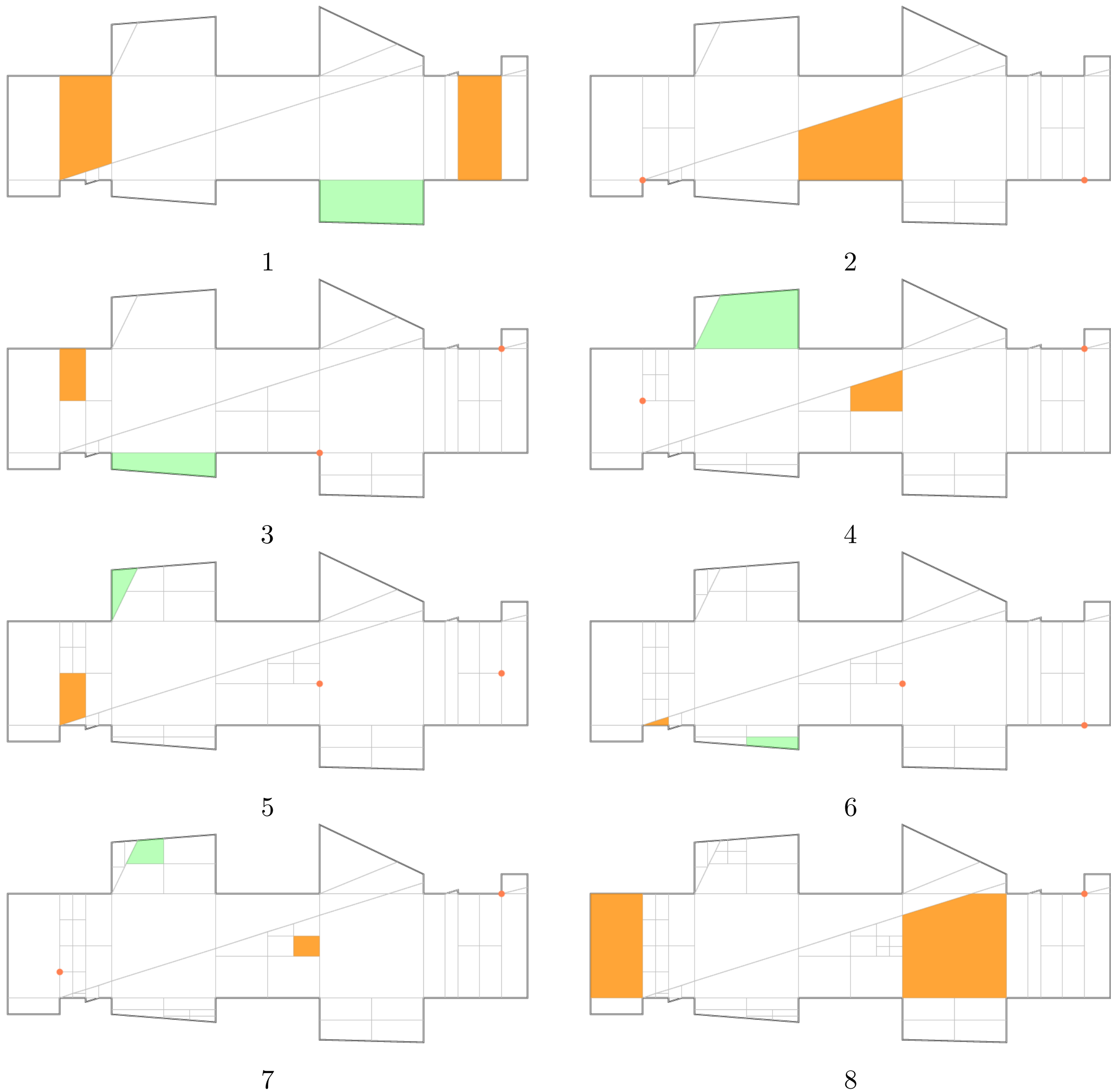
[24] Davi C. Tozoni, Pedro J. de Rezende, and Cid C. de Souza. The Quest for Optimal Solutions for the Art Gallery Problem: A Practical Iterative Algorithm. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms*, pages 320–336, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

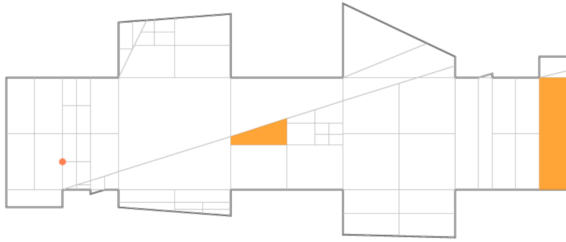
[25] Davi C. Tozoni, Pedro J. de Rezende, and Cid C. de Souza. Algorithm 966: a practical iterative algorithm for the art gallery problem using integer linear programming. *ACM Transactions on Mathematical Software (TOMS)*, 43(2):16, 2016.

A Intermediate Arrangements

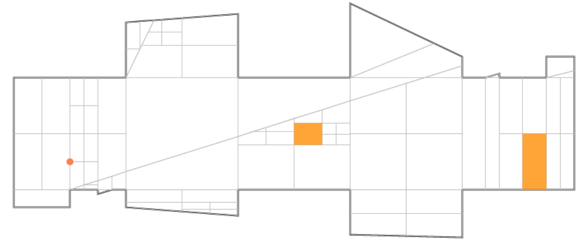
Irrational-Guard Polygon.

Below we show the first 22 iterations of the Irrational-Guard polygon. Section 5.5 shows that this iteratively updating set of guards converges to the optimal solution. The orange points and faces represent point- and face-guards in the intermediate solution. The green faces represent faces not fully seen by the current candidate solution. Both orange and green faces are split in the next iteration.

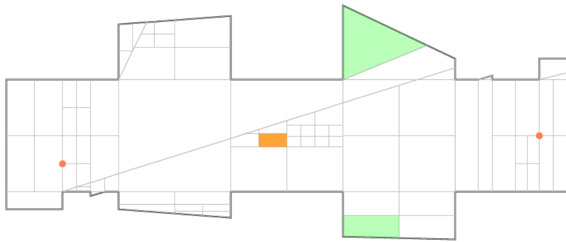




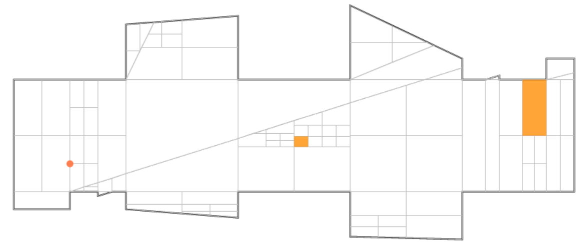
9



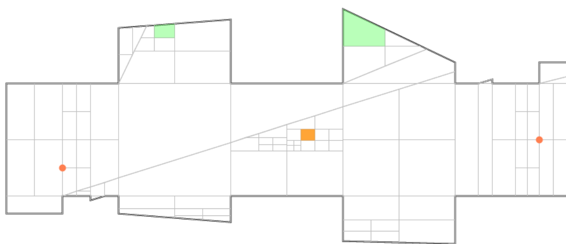
10



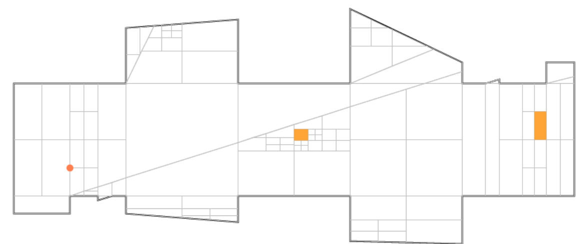
11



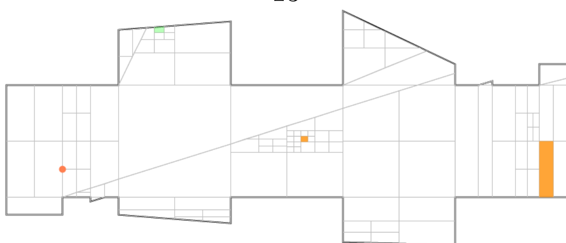
12



13

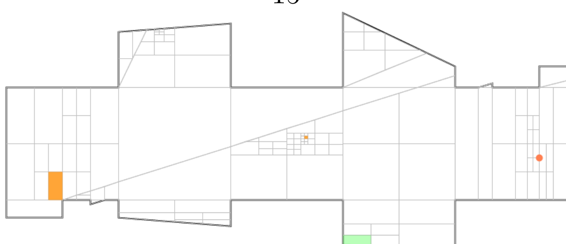


14



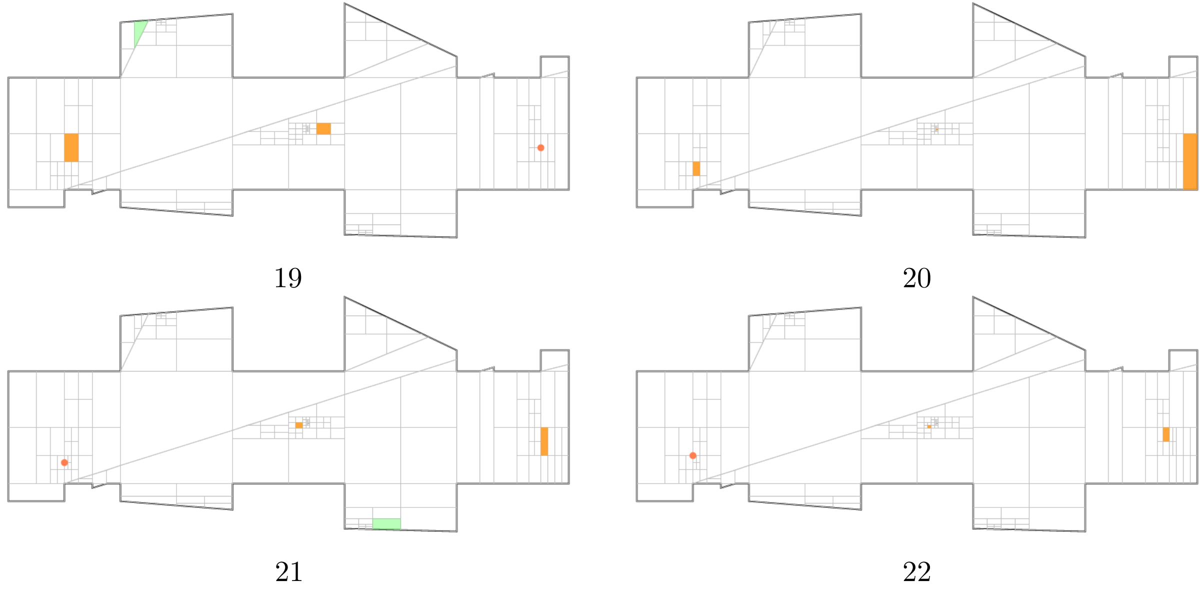
15

16



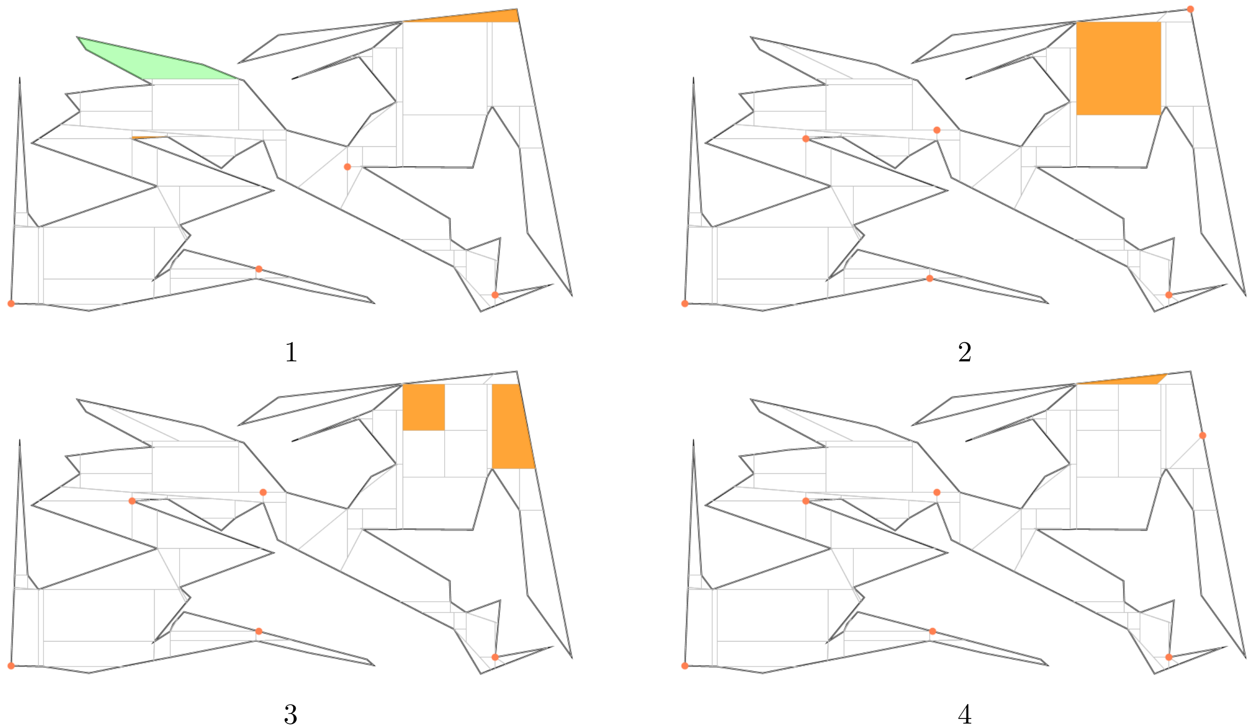
17

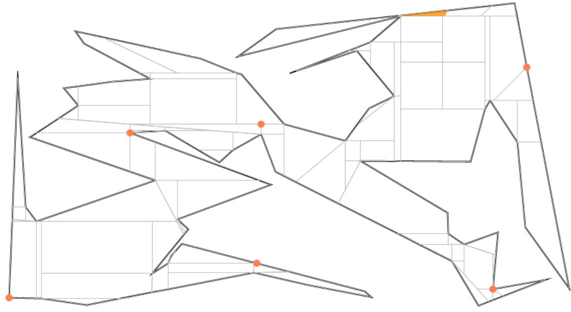
18



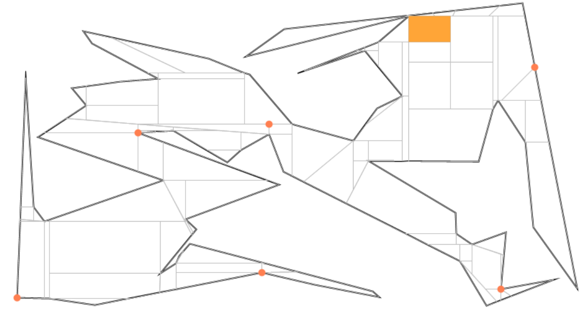
Polygon with 60 vertices.

Here, we show how the iterative algorithm finds a solution in 9 iterations for a polygon with 60 vertices. This is one of the input polygons from Couto et al. [10].

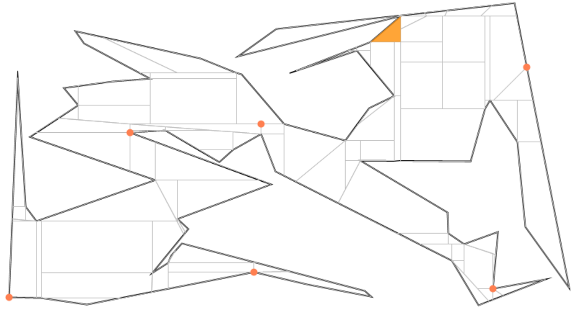




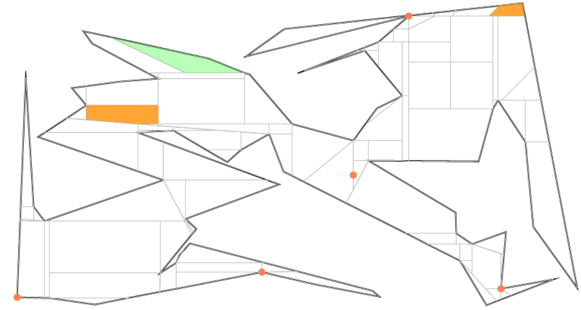
5



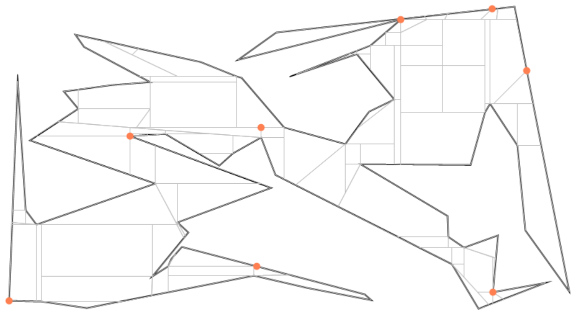
6



7



8



9