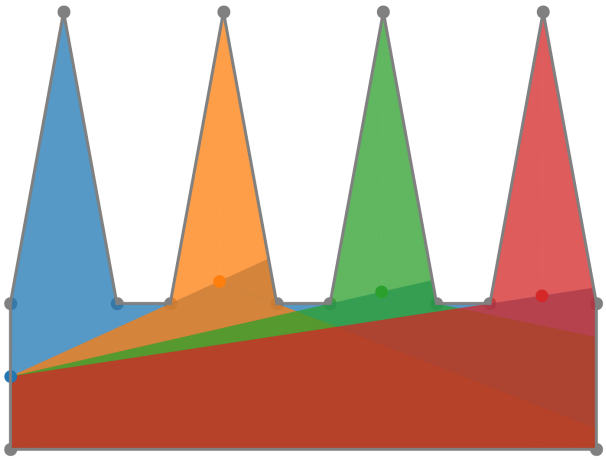
Solving The Art Gallery Problem Using Gradient Descent

Master's Thesis



Georgiana Juglan

Supervisor:
Tillmann Miltzow
Second Examiner:
Frank Staals

Computing Science
October 10, 2022
Student Nr. 2996307
Utrecht University
The Netherlands

Universiteit Utrecht

Abstract

The Art Gallery Problem is one of the central problems in computational geometry. It was recently shown that the problem is $\exists\mathbb{R}$-complete. Thus it is unlikely to be solvable by methods that are usually applied to NP-complete problems. These methods include heuristics with provable guarantees to come as close as possible to the optimal solution in polynomial time. Nonetheless, one of the most important methods to solve $\exists\mathbb{R}$-complete problems is called gradient descent. Curiously, there was no attempt yet to solve the Art Gallery Problem by gradient descent. By aiming to maximise the area seen by the guards in the Art Gallery Problem, we get a continuous cost function, which allows us to compute a gradient. Using the gradient and other heuristics inspired from neural networks, we will try to solve the Art Gallery Problem practically using gradient descent. Specifically, we aim to use the library CGAL. Implementations will show how feasible this approach is. We will visualize the results and discuss the performance of the algorithm on different input shapes and sizes.

Contents

1 Introduction

1.1 The Art Gallery Problem

The Art Gallery Problem [14] is a central problem in computational geometry. It can be introduced as follows: given a simple polygon P with n vertices, we are interested in finding the minimum number of guards (points) that are able to see the whole polygon. A simple polygon is a polygon that has no holes. Thus, we can define the visibility of a point g in the polygon $P \subset \mathbb{R}^2$. The point (guard) $g \in P$ sees another point $q \in P$ if the line segment $\overline{gq} \subseteq P$. The points that are visible from g form the visibility polygon (region) $Vis(g)$. In the Art Gallery Problem, we are looking for a minimum size guard set S that can see the whole polygon P.

Figure 1 displays an example of the Art Gallery Problem [14] with polygon P guarded by 2 points ($|S| = 2$). The visibility region Vis of each guard is marked with a different colour. For point $g \in S$, its visibility region $Vis(g)$ is emphasised with the pink contour. The vertex r blocks part of the view of g. For this reason, it is called "reflex", because the angle it forms on the inside of the polygon is larger than 180°. Reflex vertices are only found in concave polygons, so convex polygons can be guarded by only one guard.

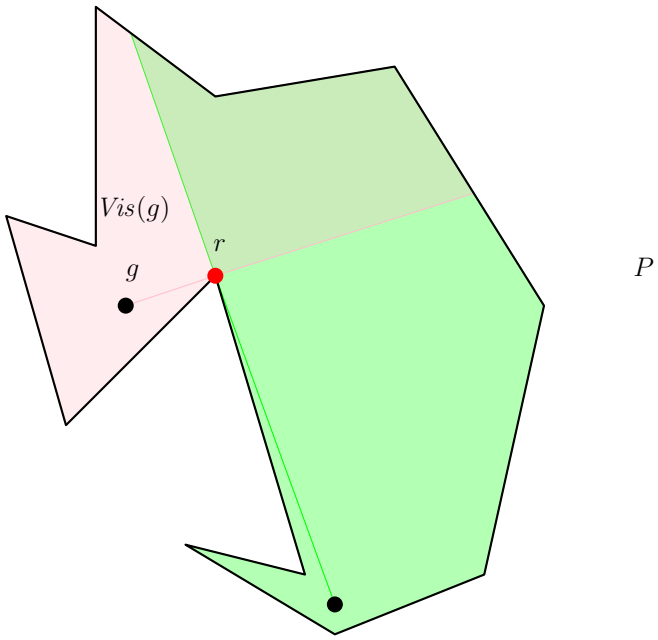


Figure 1: Example of an Art Gallery Problem instance with polygon P guarded by 2 points. The visibility area $Vis(g)$ is emphasised in pink.

The Art Gallery Problem [14] is $\exists\mathbb{R}$-complete [1], which means it is even harder to solve than NP-complete problems. For this reason, approximation algorithms have been extensively used to address it ([4], [8], [10]). Nonetheless, to the best of our knowledge there is no related work on approaching the Art Gallery Problem [14] using gradient descent. As such, we will approach the Art Gallery Problem [14] from a new perspective using gradient descent.

1.2 Gradient Descent

Gradient descent is an iterative optimisation algorithm for finding the minimum of a continuous differentiable function. The core idea of gradient descent is to repeatedly move in the opposite direction of the gradient of the function at the current point using a specific step size (learning rate). High learning rates result in approaching a local optimum faster, but risk overshooting it. Conversely, small learning rates are more precise, with the compromise of a longer computation and convergence time. When there is no more change in the gradient, then an optimum has been reached. Gradient descent does not guarantee that the found optimum is global. For this reason, it can remain stuck in local optima.

Figure 2 illustrates an intuitive example of applying gradient descent. The optimisation function takes the shape of a curve. Starting from an arbitrary point on the curve, the goal is to reach the minimum of the function (the bottom of the curve). This is done by computing the gradient (derivative) of the function at the current point and moving in its opposite direction. In this case, the gradient would indicate going up the curve. In order to reach the minimum, the point has to move down the curve with the given step size. Continuing from the new point on the curve, the process is repeated until the minimum is reached.

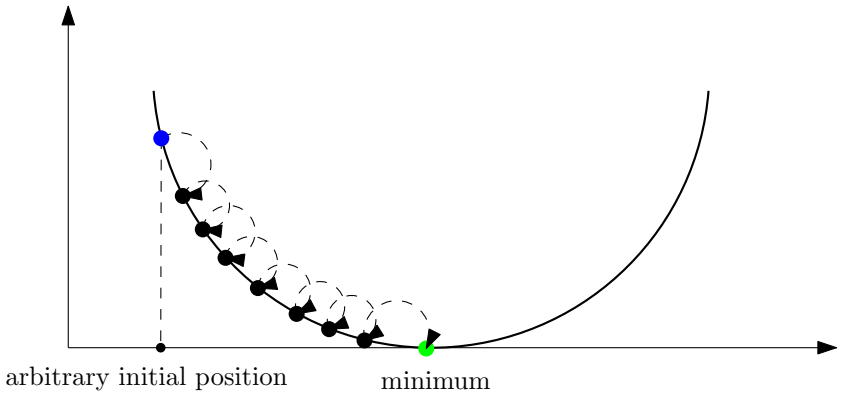


Figure 2: Illustrative example of applying gradient descent.

1.3 Thesis Goal

This thesis will thus be an optimisation algorithm engineering paper. The main goal is to create and implement an algorithm that uses gradient descent to approximate the solution to the Art Gallery Problem. The explored research question will be whether the Art Gallery Problem can be efficiently solved using gradient descent. Additional to gradient descent, the algorithm will deploy different heuristics to address various shortcomings and edge-cases. The algorithm will be implemented in C ++ using the CGAL library (https://www.cgal.org).

This paper is organised as follows: Section 2 offers an overview of existing work; Section 3 explains how in theory gradient descent is used to solve the Art Gallery Problem and what heuristics we can make use of to improve the performance of our algorithm; Section 4 offers an overview of how well our algorithm performs in practice. Lastly, Section 5 discusses issues this project has encountered.

1.4 Discussion

As previously mentioned, this thesis focusses on implementing and evaluating a gradient descent algorithm to find a solution to the Art Gallery Problem. Indeed, this goal has been achieved for a few cases. Nonetheless, it is worth discussing the development process and the performance of the algorithm. Gradient descent is quite a straight-forward approach. However, its implementation in the context of the Art Gallery Problem is not. There were numerous edge-cases to be taken into account. For many of them, we created various hyperparameterised heuristics. Other heuristics (momentum) were created as an improvement to the shortcomings of the vanilla version of gradient descent.

The resulting program is sensitive to hyperparameter choices, polygon shapes and starting guard positions. For this reason, we only provide qualitative evaluations for the heuristics used to extend gradient descent. Namely, Section 4 discusses the added benefits of each of the heuristics used. The way this is done is by running the program with all heuristics but the one whose practicality we are discussing. In this way, we will assess its significance in its absence. The reason behind this experimental setup is that combinations of all heuristics are both verbose and unnecessary. Given that the program is still limited to a few number of examples, we cannot use extensive statistical tools to test its performance. In this way, we provide a pragmatic overview to the usefulness of each of the heuristics and hyperparameters used. Namely, we analyse on what the type of polygons different heuristics and hyperparameter values provide concrete results.

Section 1.5 will further discuss how the program could be improved in the future.

1.5 Future Work

Currently, the program offers a state-of-the-art view about its practical possibilities. In the future, it would be suitable to extend it with more robust functionalities.

One of the most crucial aspects of improvement for the program would be to solve the existing errors and bugs. As of now, the program crashes for specific input polygons (see Section 5 for an extensive overview of its shortcomings). Another important element to consider would be to code it more efficiently. Some of the data structures used are naive and can be easily implemented in a smarter way. For this reason, the program does not scale well.

Additionally, some features and heuristics are not complete. For example, the greedy initialisation of the guards' position is deterministic. In order to quantitatively assess the performance of the algorithm, a truly randomised greedy initialisation would be required for statistical significance.

Lastly, it would be worth exploring how the algorithm would benefit from a different implementation. Namely, using another programming language like Python (or CGAL Python bindings) and other geometry libraries which are better documented and more reliable to use.

2 Literature Summaries

This section offers an overview of previous works addressing the Art Gallery Problem [14]. Working with the visibility region is a basic tool in computational geometry, specifically in the context of the Art Gallery Problem [14]. The Art Gallery Problem [14] is an $\exists\mathbb{R}$-complete problem [1]. The problem class $\exists\mathbb{R}$ consists of instances that can be reduced in polynomial time to a decision problem of whether a system of polynomial equations with integer coefficients and any number of real variables has a solution. Since NP $\subseteq \exists\mathbb{R}$, the $\exists\mathbb{R}$ class is an at least as hard to solve complexity class. That is due to the fact that $\exists\mathbb{R}$-complete problems are not know to have solutions of polynomial space.

Building on the aforementioned concepts, this section will further inspect how visibility in polygons can be efficiently computed [5], how it is not always possible to place guards at rational coordinates [1], in addition to two improved polygon guarding algorithms [13], [10].

2.1 Efficient Computation of Visibility Polygons

Bungiu, Hemmer, Hershberger, Huang and Kröller [5] introduce the implementations and their experimental evaluations for two existing algorithms ([11], [2]) and a newly developed one for computing visibility in polygons. These implementations are available in the CGAL library (https://www.cgal.org/), starting with version 4.5.

Therefore, Bungiu et al. present three algorithms and their practical performance.

2.1.1 Algorithm of Joe and Simpson

The algorithm of Joe and Simpson [11] runs in $O(n)$ time and space.

Let v_i, for $i = 1, 2, ..., n$, be the boundary vertices of the polygon P. Let g be a guard in P, and let s be a stack datastructure. The stack s will be used to keep track of the vertices determining $Vis(g)$.

The algorithm begins by scanning the boundaries of P. The scanning is done through shooting rays $p\vec{v}_i$, for $i = 1, 2, ..., n$ in this order. The endpoints v_i and v_{i+1} of each ray form a boundary edge $\overline{v_i v_{i+1}}$. In this way, the processing of v_i and v_{i+1} is done by checking whether the points are in $Vis(g)$. This means that the position of every v_{i+1} with respect to the ray $p\vec{v}_i$ is checked. If v_{i+1} is found in front of the ray $p\vec{v}_i$ (if v_{i+1} is seen from g), then both v_i and v_{i+1} are added to the processing stack s. For every newly pushed vertex on s, the algorithm checks whether the segment $\overline{v_i v_{i+1}}$ obscures any of the previously added vertices. If that is the case, then the endpoints of the obscured line segment are declared obsolete and deleted. The polygon comprised of the vertices from s forms at the end the visibility polygon $Vis(g)$.

Figure 3 displays an example run of the Algorithm of Joe and Simpson [11] for polygon P and guard g. First, the ray from g is shot through vertex v_1, and v_1 is added to s. Then, the ray from g is shot through v_2. Since ray $p\vec{v}_2$ takes a right turn from $p\vec{v}_1$, this means that v_2 is still in the visibility region of g. For this reason, v_2 is also added to s. The ray passing through v_2 also intersects the boundary of P in a point v_2'. To account for the fact that g can see "behind" v_2 and is still inside P, the boundary vertex v_2' is hence added to s. Next, the ray $p\vec{v}_3$ takes a left turn from v_2, which means that v_3 is not seen from g. Similarly, v_4 and v_5 are added to s. However, because ray $p\vec{v}_6$ takes a left turn from $p\vec{v}_5$, segment $\overline{v_5 v_6}$ obscures $\overline{v_4 v_5}$. So, v_4 and v_5 are removed from s. Ray $p\vec{v}_6$ then intersects the boundary of P in v_6'. At the end, $Vis(g) = \{v_1, v_2, v_2', v_6, v_6', v_7, v_8, v_9, v_{10}\}$, as

shown on the boundary of the green area.

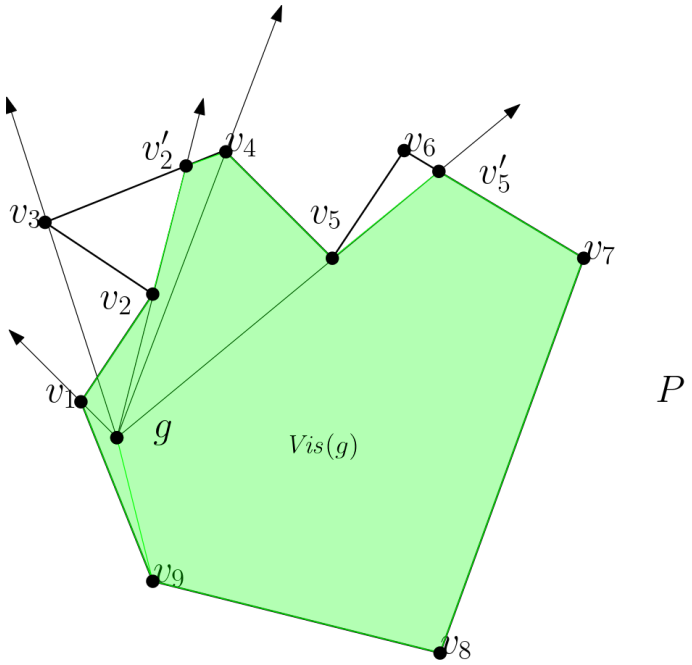


Figure 3: An example run of the algorithm of Joe and Simpson [11] for polygon P guarded by g and boundary vertices v_i, for $i = \{1, 2, ..., n\}$.

2.1.2 Algorithm of Asano

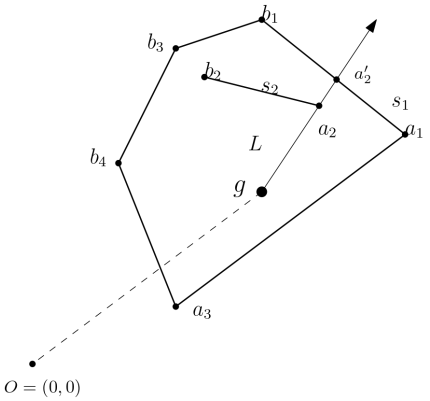
The algorithm of Asano [2] runs in $O(n \log n)$ time and $O(n)$ space. It uses a plane sweep approach with event line L.

Let P be a polygon determined by vertices $\{a_1, a_2, a_3, b_1, b_2, b_3, b_4\}$. P may have holes. Suppose we want to guard it by point g. The algorithm of Asano [2] begins by efficiently sorting all the vertices of P based on their polar angles with respect to the guard g. Figure 4 displays an example run of the algorithm. The points will be treated in the order of $a_2, a_1, a_3, b_4, b_3, b_2, b_1$ with respect to g and their angular comparisons

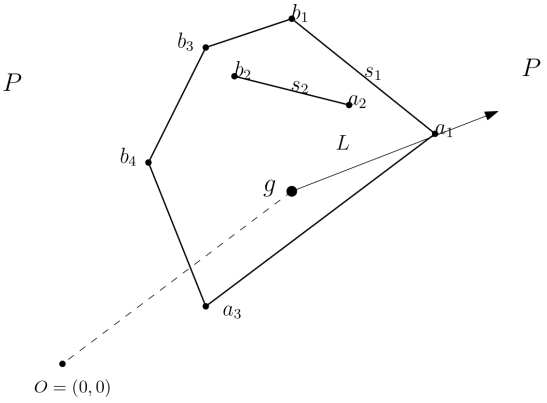
$$\measuredangle a_2Op < \measuredangle a_1Op < \measuredangle a_3Op < \measuredangle b_4Op < \measuredangle b_3Op < \measuredangle b_2Op < \measuredangle b_1Op,$$

where $O = (0, 0)$.

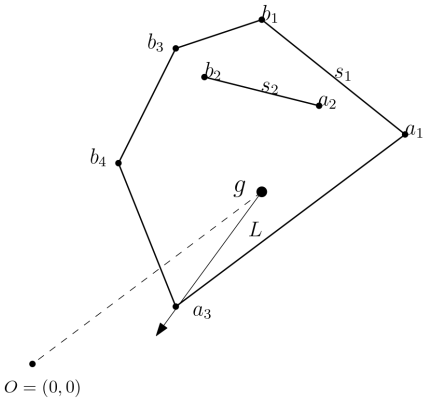
Then, the event line L starts sweeping around g as shown in Subfigures 4a - 4g. Every line segment that L intersects is stored in a balanced binary tree T in the order of their intersection angle. As T is updated, a new vertex of $Vis(g)$ is stored each time the segment closest to g in T changes. It is important to mention that the intersection between L and the line segments is not explicit, but is instead determined by comparisons between the endpoints' coordinates. For instance, in Figure 4, the endpoint b_2 of line segment $\overline{b_2a_2}$ is the first one L intersects. Point b_2 is thus added to T. Then, L continues sweeping and adds b_1, a_2 and a_1 to T. Although $\overline{b_2a_2}$ and $\overline{b_1a_1}$ represent line segments s_2 and s_1, respectively, the intersection of L with them is not explicitly computed, but is determined based solely on the positions of their endpoints: s_1 is farther away from g because q, a_2 and b_2 are on the same side of s_2.



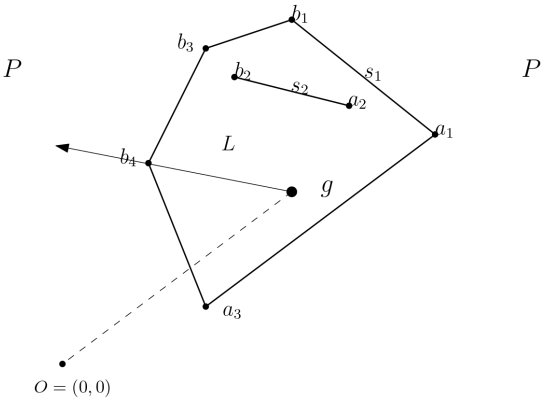
(a) a_2 is added to the empty $Vis(g)$ such that $Vis(g) = \{a_2\}$.



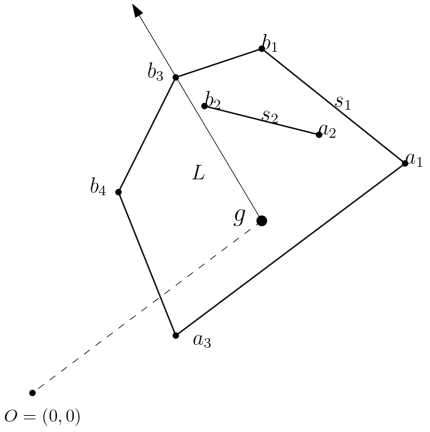
(b) a_1 is added to $Vis(g)$ such that $Vis(g) = \{a_1, a_2\}$, and line segment $\overline{b_1a_1}$ is added to the empty binary tree $T = \{\overline{b_1a_1}\}$.



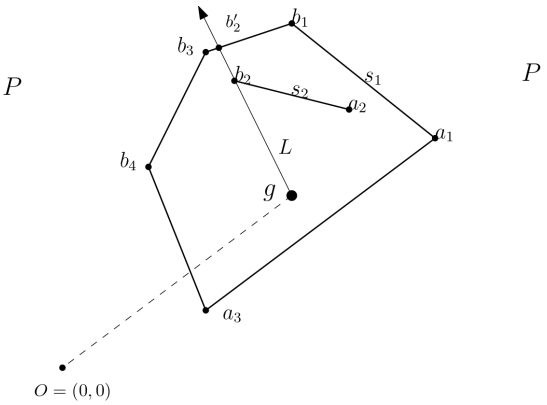
(c) a_3 is added to $Vis(g)$ such that $Vis(g) = \{a_1, a_2, a_3\}$, and line segment $\overline{a_1a_3}$ is added to T such that $T = \{\overline{b_1a_1}, \overline{a_1a_3}\}$.



(d) b_4 is added to $Vis(g)$ such that $Vis(g) = \{a_1, a_2, a_3, b_4\}$, and line segment $\overline{a_3b_4}$ is added to T such that $T = \{\overline{b_1a_1}, \overline{a_1a_3}, \overline{a_3b_4}\}$.

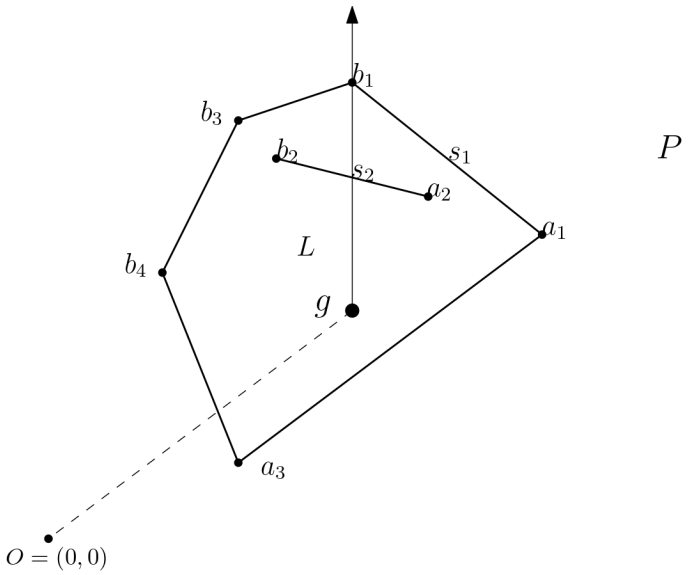


(e) b_3 is added to $Vis(g)$ such that $Vis(g) = \{a_1, a_2, a_3, b_4, b_3\}$, and line segment $\overline{b_4b_3}$ is added to T such that $T = \{\overline{b_1a_1}, \overline{a_1a_3}, \overline{a_3b_4}, \overline{b_4b_3}\}$.



(f) b_2 is added to $Vis(g)$ such that $Vis(g) = \{a_1, a_2, a_3, b_3, b_4, b_2\}$, and line segment $\overline{b_2a_2}$ is added to T such that $T = \{\overline{b_1a_1}, \overline{a_1a_3}, \overline{a_3b_4}, \overline{b_4b_3}, \overline{b_2a_2}\}$.

Figure 4: Example run of the Algorithm of Asano [2] on polygon P and guard g. The vertices of P are added to the binary tree T in the order of their angle between g and the origin $O = (0, 0)$. The result of the algorithm is visibility region $Vis(g) = \{a_1, a_2, a_3, b_3, b_4, b_2\}$.



(g) b_1 is not added to $Vis(g)$ because it is obstructed by the line segment $\overline{b_2a_2}$ which is already in T. For the same reason, line segments $\overline{b_3b_1}$ and $\overline{b_1a_1}$ are also not added in T.

Figure 4: Example run of the Algorithm of Asano [2] on polygon P and guard g. The vertices of P are added to the binary tree T in the order of their angle between g and the origin $O = (0, 0)$. The result of the algorithm is visibility region $Vis(g) = \{a_1, a_2, a_3, b_3, b_4, b_2\}$.

2.1.3 New Algorithm: Triangular Expansion

The algorithm introduced by Bungiu et al. [5] is named Triangular Expansion and runs in $\Omega(n^2)$ time and $O(n)$ space. It begins by triangulating P in $O(n \log n)$ time if P has holes, and $O(n)$ otherwise. The implementation runtime is constrained by CGAL, which makes use of the Delaunay triangulation algorithm [7] with $O(n^2)$ time for the worst case, but with better performance in practice.

Taken from [5] and annotated to suit the explanations in these summaries, Figure 5 depicts an example run of the algorithm on a polygon with holes P. Starting from the viewpoint g, the triangle containing g is located by performing a simple walk. Trivially, g sees the entire triangle it is contained in. The algorithm continues by recursively expanding the view of g from one triangle into the next, until there are no more triangles to expand into. The view of g becomes restricted by the reflex vertices l and r of the third triangle entered by the recursive step. Since l and r are reflex vertices, the view past them is further restricted until the boundaries l' and r' of P, respectively, are reached. Line segments $\overline{ll'}$ and $\overline{rr'}$ are added to $Vis(g)$ in their angular order around g. At the end, $Vis(g)$ will contain the segments delimiting the visibility polygon of g.

2.1.4 Experiments

Bungiu et al. do not report on benchmarks with query points on edges in the interior polygon. This is because they claim that their implementations perform similarly to other already implemented algorithms. Instead, they use two real-world scenarios (a simple polygon of Norway with 20981 vertices, and a cathedral polygon with 1209 vertices) and a worst-case polygon for the Triangular Expansion algorithm.

In terms of results on the real-world polygons, the Triangular Expansion algorithm has a 2-factor improved performance when compared to Asano's algorithm [2], and performs "one order

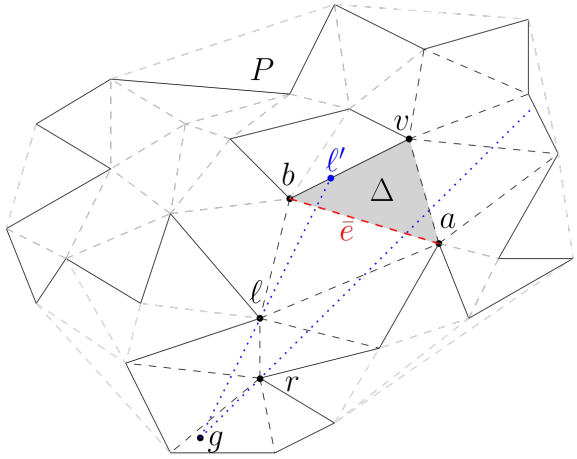


Figure 5: The Triangular Expansion Algorithm Example - recursion entering triangle Δ through edge e [5].

of magnitude" faster than Joe and Simpson's algorithm [11]. For the worst-case scenario, Asano's algorithm [2] outperforms the Triangular Expansion algorithm with increasing input complexity.

Thus, despite the Triangular Expansion algorithm being outperformed in the worst-case scenario, Bungiu et al. add efficient implementations for 3 different polygon visibility algorithms in the CGAL library. The choice of algorithms when using the library can be adapted based on the input polygons.

2.2 Irrational Guards Are Sometimes Needed

Abrahamsen, Adamaszek, and Miltzow [1] study the placement of the guards in the context of the Art Gallery Problem [14]. Namely, it focusses on and confirms that there are polygons with integer coordinates that require guards placed at points with irrational coordinates. Generalising, Abrahamsen et al. show that $\forall n \in \mathbb{N}$, there is a family of simple monotone polygons that can either be guarded by $3n$ guards with irrational coordinates, or requires at least $4n$ guards with rational coordinates. The result is also extended to a rectilinear polygon that can be guarded by 9 guards with irrational coordinates, or requires at least 10 guards with rational coordinates. The family of simple monotone polygons that can be guarded by $3n$ guards with irrational coordinates, as well as the rectilinear polygon that can be guarded by 9 guards with irrational coordinates are also discussed.

First, we will address the question regarding whether polygons given by integer coordinates require guard positions with irrational coordinates in any optimal solution by introducing a crafted monotone polygon P. A polygon is monotone if there exists a line l such that every line orthogonal to l intersects its boundary at most twice. Polygon P is depicted in Figure 6 and is constructed using a rectangle, six triangular pockets (in green), three rectangular pockets (in blue) and four quadrilaterial pockets (in red) are added. The intuition behind the polygon will be explained in the upcoming subsections. The figure is accredited to Abraham et al. [12].

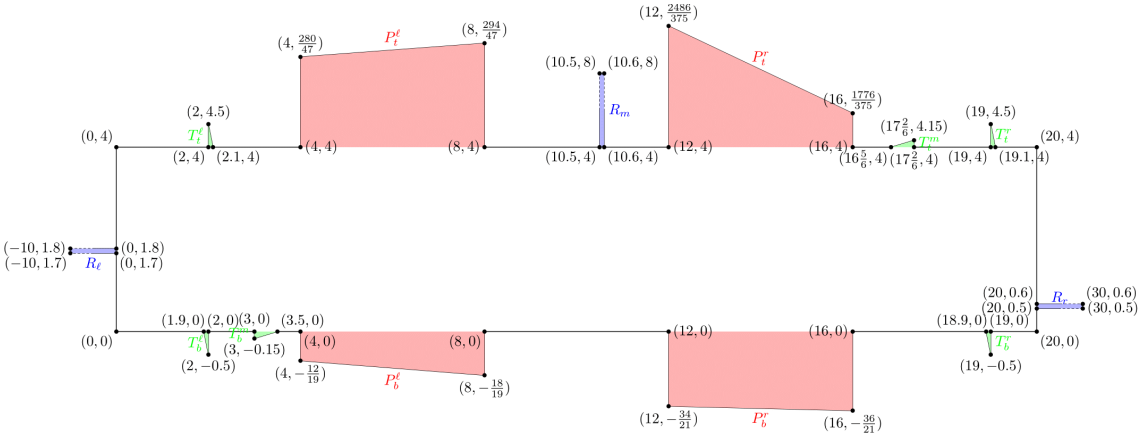
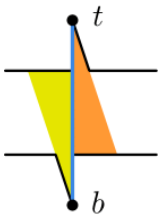


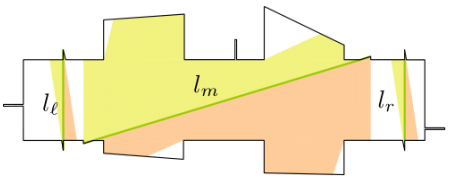
Figure 6: The polygon P developed by Abraham et al. [12] to emphasise the need for irrational guards sometimes.

2.2.1 Intuition for Triangular Pockets

The six triangular pockets are created in order to force a guard on a line segment, as depicted in Figure 7. The figure is taken from [1]. As such, they are grouped into three pairs: top pockets are paired with bottom pockets corresponding to their position in P. For every pair (leftmost, middle, rightmost) of triangular pockets, there is a corresponding line l_ℓ, l_m, l_r, respectively, that joins the peaks of each triangular pocket in a pair. A guard can see both the peaks t, b of the pockets only if it is placed on the line segment $\overline{tb}$ (Figure 7a). Hence, one guard is needed per pair, resulting in 3 guards in total.



(a) The only way that one guard can see both t and b is when the guard is on the blue line.



(b) The only way to guard the polygon with three guards requires one guard on each of the green lines l_ℓ, l_m, l_r.

Figure 7: Forcing guards to lie on specific line segments [12].

2.2.2 Intuition for Quadrilaterial Pockets

The four quadrilateral pockets are created in order to force a guard in a region bounded by a curve. Figure 8 was taken and adapted from [12], and depicts the visualisation for this case. Consider some position of g_ℓ on l_ℓ. As g_ℓ moves up on l_ℓ, the point p_t^ℓ slides towards the right on segment e_t^ℓ, and point p_b^ℓ slides to the left on segment e_b^ℓ. In this way, the lines $p_t^\ell b$ and $p_b^\ell d$, where b, d are reflex vertices, intersect and form the curve c_ℓ. Considering thus the fixed position of a guard g_m either on the left or on the right of the curve c_ℓ, there exists a position of guard g_ℓ on l_ℓ such that edge e_t^ℓ and line $p_b^\ell g_m$ are seen by g_ℓ. Then, only if g_m is on the left or on c_ℓ can it see the remaining parts of the pockets that are not seen by g_ℓ (edge e_b^ℓ, line $p_t^\ell g_\ell$). Analogously, in order for the right side of P and the right pair of quadrilateral pockets to be seen by both guards g_m

and g_r, g_m has to be on the curve c_r or on its right. Since g_m has to also satisfy its position on l_m, its only feasible position is as the intersection point between c_ℓ, c_r and l_m.

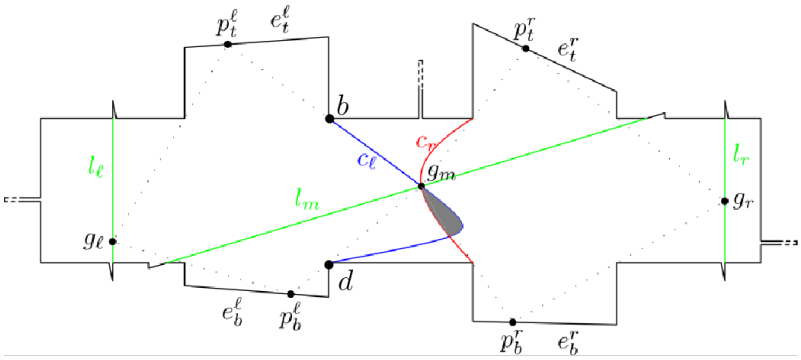


Figure 8: Restricting a guard to a region bounded by a curve [12].

2.2.3 Intuition for Rectangular Pockets

The three rectangular pockets are created in order to force a guard to a single irrational point. Further building onto the previously discussed instance from Figure 8, the three rectangular pockets on the laterals and top of P are added. They allow for additional constraints for the positions of the guards g_ℓ, g_m, g_r. Based on the curve equations of c_ℓ and c_r, we can thus calculate the irrational position of $g_m = (3.5 + 5\sqrt{2}, 1.5\sqrt{2})$. Subsequently, we can fix the positions of g_ℓ and g_r on lines l_ℓ and l_r.

2.2.4 Monotone Polygon Construction

The polygon P in question was found through experimentation in GeoGebra (https://www.geogebra.org/). The triangular and rectangular pockets were fixed. Finding then the position of the rectangular pockets required the existence of a rational line that contained the irrational coordinates of guard g_m from Figure 8. The irrational coordinates of the two other guards g_ℓ, g_r were chosen such that they would be able to see the rest of the polygon that is unseen by g_m. The rectangular pockets were added based on their coordinates.

In this way, by placing each guard placed on the lines l_ℓ, l_m, l_r at unique irrational positions, respectively, dependencies between guard positions were created. The resulting guard set becomes thus $S = \{g_\ell, g_m, g_r\}$. Given the unique and irrational positions of the guards, the only method for seeing the whole polygon P using guards with rational positions would be to use at least 4. This statement can also be generalised such that a family of polygons $(\mathcal{P}_n)_{n\in\mathbb{Z}_+}, \forall n$ can be guarded by $3n$ guards with irrational coordinates, or requires $4n$ guards with rational coordinates. The coordinates determining the polygons are hence polynomial in n.

2.2.5 Rectilinear Polygon Construction

Similarly, a rectilinear polygon P_R can be created given integer coordinates. Polygon P_R would require guards with irrational coordinates in any optimal solution. It can be guarded by 9 guards if guards can be placed at points with irrational coordinates. Otherwise, the smallest optimal guard set S with rational coordinates would be required to have size 10.

Polygon P_R can be constructed by extending P with the 6 non-rectiliniar gray parts $(Q_1, Q_2, Q_3, Q_4, T_1, T_2)$ in Figure 9, such that each of them requires at least 1 guard in the interior. The figure was taken from [12]. When placing 6 guards in the gray parts, the white areas of P_R remain unseen. As such, 3 guards must still be similarly placed at the 3 irrational points on lines l_ℓ, l_m, l_r as in the case of P, in order to guard the rest of P_R.

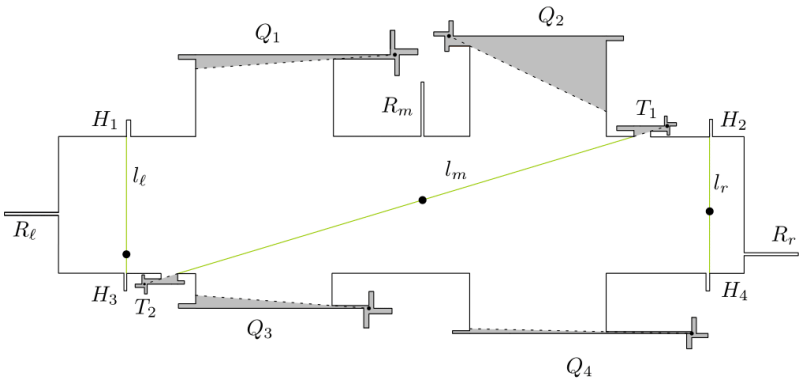


Figure 9: Rectilinear Polygon P_R [12].

2.3 Implementation of Polygon Guarding Algorithms for Art Gallery Problems

Maleki and Mohades [13] implement efficiently two existing approximation algorithms ([8], [3]) for computing visibility in simple polygons. Namely, they introduce practical implementations for visibility algorithms that already offer theoretical guarantees. Additionally, they develop their own visibility algorithm that aims to offer performance guarantees for polygon visibility computation. They lastly evaluate experimentally their implementations for the three algorithms in the context of the Art Gallery Problem [14].

To begin with, we will introduce some terminology used throughout this summary. The algorithms distinguish in their implementation between vertex guards and point guards in a polygon P. Vertex guards can be placed only on the vertices of P. Point guards can be placed without restriction inside P. Lastly, the algorithms are tested on weak visibility polygons. A polygon P has weak visibility if all boundary vertices of P can see all the points in P.

2.3.1 Algorithm of Ghosh

The algorithm of Ghosh [8] runs in $O(n^4)$ time and yields an $O(\log n)$-approximation algorithm for computing the minimum vertex guard for simple polygons.

One of the most important concepts the algorithm works with is that of a convex component. Given a polygon P, we can form subsets of the vertices in P such that all the subsets form convex subpolygons of P. We call these subsets of vertices the convex components of P.

The algorithm begins by computing the set of all the convex components C of a given polygon P. Then, each vertex $j \in P$ creates a set F_j with the convex components from C that it is able to fully see. Let $F := \{F_j \mid \forall j \in P, F_j \subseteq C \text{ seen by } j\}$. Then, the algorithm checks for overlaps between the sets of F. For every vertex i and its corresponding convex components set F_i, the algorithm checks whether any of the other sets $F \setminus F_i$ sets are included in F_i. That is, if $F_j \subseteq F_i, F_j \in F, i \neq j$. If that is the case, then vertex i sees at least as little as j. The vertex j is thus not needed as a guard, so F_j is removed from F. Vertex i is added in the final guard set S. The algorithm repeats until the set F becomes empty. When that happens, it means that the algorithm found a set S of guards who see all the convex components of P without overlap.

An example run of the algorithm is depicted in Figure 10. Let P be the polygon in question, and its boundary vertices $\mathcal{V} = \{1, 2, 3, 4, 5, 6\}$. Polygon P is divided into two convex components C_1 and C_2, such that $C = \{C_1, C_2\}$. The sets $F_j, \forall j \in P$ are also displayed. Since $F_1 \subseteq F_2 \subseteq F_3 \subseteq F_4 \subseteq F_6$ and $F_5 \subseteq F_3 \subseteq F_4 \subseteq F_6$, F_1, F_2, F_3, F_4, and F_5 are removed. The remaining set F_6 yields the final guard set $S = \{6\}$, which can see both convex regions of P, and hence the whole polygon.

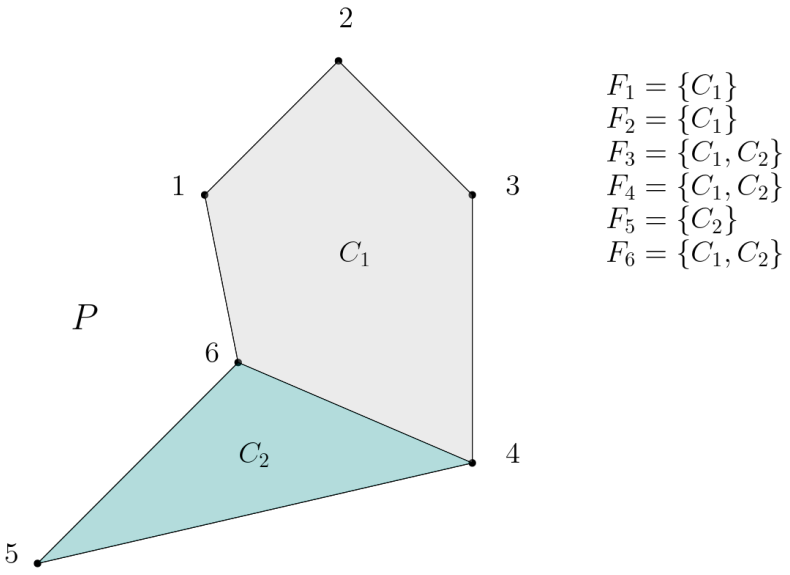


Figure 10: Example run of the Algorithm of Ghosh [8] with polygon P divided into two convex components C_1 and C_2. The resulting guard set is $S = \{6\}$.

2.3.2 Algorithm of Bhattacharya, Ghosh and Roy

The algorithm of Bhattacharya et al. [3] runs in $O(n^2)$ time and yields a 6-approximation for computing the minimum vertex guard for weak visibility polygons without holes.

It begins by choosing two neighbours u and v as parents for every vertex in P. It then computes the Shortest Path from each pair of parents (u, v) to every other vertex in P. The Shortest Path is a path between two vertices such that the distance between them is minimal. If all distances between two adjacent vertices are the same, then the length of the path corresponds to the number of edges between the vertices. The Shortest Path from a pair of vertices (u, v) to any other vertex w is the length of the minimum path between u and w, and v and w.

Then, all vertices that can be reached from u and v are unmarked. In increasing angular order from $\overline{uv}$, every vertex $w \in P$ is checked for visibility from u, and v. If all vertices w are visible, then u, and v are added to S. Otherwise, w is added to S and the procedure continues with w as the starting node. All vertices that become seen from S are marked. At the end, the algorithm checks whether the vertices in S have overlapping visibility regions, and duplicates are removed.

An example run of the algorithm is depicted in Figure 11. Let P be the polygon in question, and its boundary vertices $\mathcal{V} = \{a, b, c, d, e, f\}$. The algorithm starts with vertices a and c as parents of b. Clockwise, vertex d is visible from $\overline{ac}$, but e is not. For this reason, e is added to the guard set S, and vertices d and f are chosen as the new parents. In increasing angular order, all vertices a, b, c, e are visible from $\overline{ef}$, so d and f are added to S. Since the visibility regions of d and f coincide ($Vis(d) = Vis(f)$), and the visibility region of e is included in that of d and f ($Vis(e) \subseteq Vis(d) \subseteq Vis(f)$), e and d can be removed from S. As such, the final guard set becomes $S = \{f\}$.

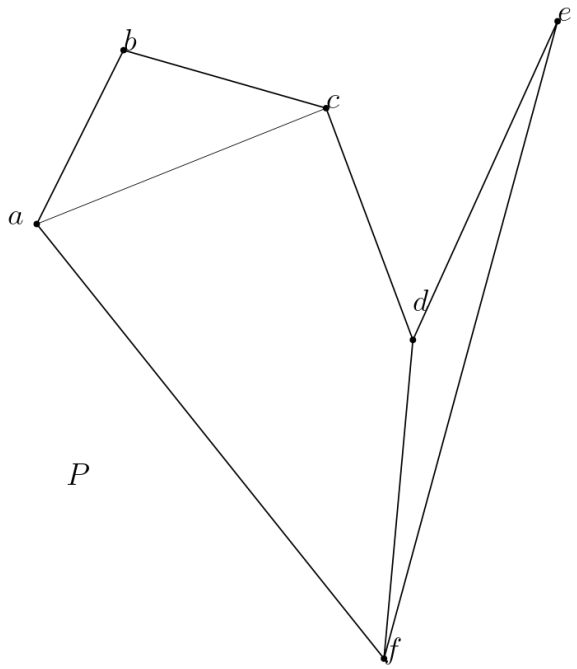


Figure 11: Example run of the Algorithm of Bhattacharya et al. [3] for polygon P. The final guard set is $S = \{f\}$.

2.3.3 New Algorithm

The algorithm introduced by Maleki and Mohades is focussed on polygons with large number of vertices n and different amounts of reflex vertices r. If the number of reflex vertices is significantly lower than the total number of vertices ($r \leq \log\log n$), guards are placed at all reflex vertices. Otherwise, they are placed according to the algorithm of Ghosh [8].

2.3.4 Experiments

Algorithms of Ghosh [8] and Bhattacharya et al. [3] are tested on weak visibility polygons, and the newly introduced algorithm is tested on simple polygons.

Let Procedure 1 be a procedure for generating weak visibility polygons that is illustrated in Figure 12: given two points $p = (k, 0), q = (-k, 0)$, n random points $\{x_1, ..., x_n\}$ sorted on the distance from p on $\overline{pq}$, n sorted random angles $\{\alpha_1, ..., \alpha_n\} \in (0, \pi)$, and n vertices $\{y_1, ..., y_n\}$ are created by shooting n rays at the corresponding angle $\alpha_i, \forall x_i$. Then, n reflex vertices $\{z_1, ..., z_n\}$ are added in the quadrilateral formed by vertices $x_i y_i y_{i+1} x_{i+1}$. The figure is accredited to [13].

The algorithms of Ghosh [8] and Bhattacharya et al. [3] were tested using polygons generated by Procedure 1 with $n \in \{10, 11, ..., 15\}$ vertices and $r \in \{2, 3\}$ reflex vertices. The results suggest that for low values of n and r, the algorithm of Ghosh [8] performs better when using the number of guards as evaluation criteria. Similarly, the algorithm of Ghosh [8] performed better both when tested on a weak visibility polygon with low $n = \overline{10, 15}$ and $\frac{n}{2} \leq r$, and when tested with large $n \in \{100, 400\}$.

Secondly, let Procedure 2 be a procedure for generating simple polygons with custom number of reflex vertices r is devised in Figure 13. The figure was taken from [13] and annotated to suit the explanations in these summaries. Starting from a simple convex polygon P with n vertices $(u, v, x1, x2, ..., x7)$, P is triangulated such that every triangle has a joint edge with its boundaries;

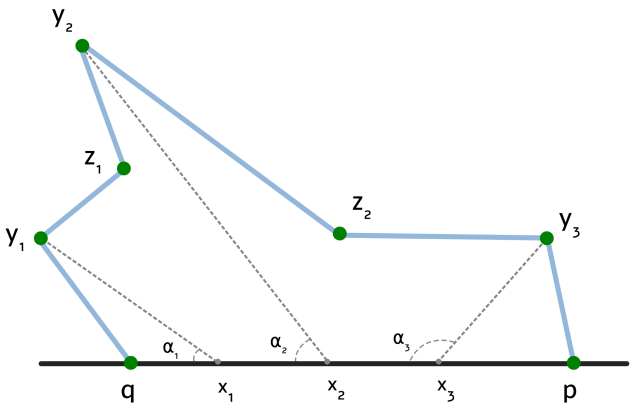


Figure 12: Generated weak visibility polygon for $n = 3$ [13] through Procedure 1.

r reflex vertices $(r1, r2, r3)$ are randomly added inside P, and the boundaries outside of the reflex vertices are moved such that all r points are now forming boundaries.

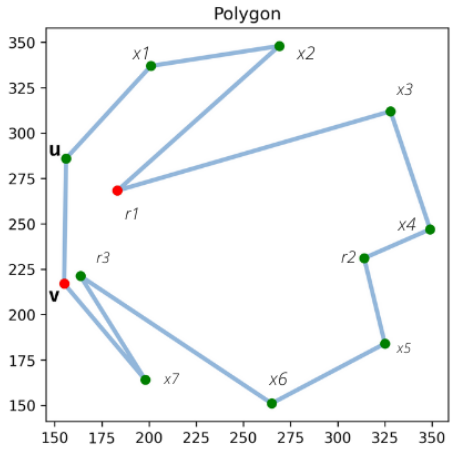


Figure 13: Generated simple polygon P for $n = 12, r = 3$ [13] through Procedure 2.

The new algorithm is tested on simple polygons constructed as mentioned by starting from low r and gradually increasing it. The results are reported positively in the sense that the $|S|$ always remains close to the optimal, as a 2-approximation solution.

Therefore, through the newly implemented algorithm, Maleki and Mohades [13] testify that the algorithm of Ghosh [8] performs like a constant approximation in practice, and often better than its theoretical bound when tested on complex simple polygons.

2.4 A Practical Algorithm with Performance Guarantees for the Art Gallery Problem

Hengeveld and Miltzow [10] introduce the idea of vision-stability, and describe an iterative algorithm that guarantees to find the optimal guard set for every vision-stable polygon in order to address the Art Gallery Problem [14]. Additionally, it proves that the set of vision-stable polygons admits an FPT algorithm when parametrised by the number of vertices visible from a chord of the

polygon, and gives a one-shot algorithm for that matter.

Hengeveld and Miltzow [10] start by introducing the notion of vision-stability (see Figure 14). The figure was taken from [10] and annotated to facilitate the understanding of this summary. The authors define guards as "enhanced" if they can "see around the corner" by an angle of δ ($vis_\delta(q)$), or "diminished" if their vision is "blocked" by reflex vertices by an angle of δ ($vis_{-\delta}(q)$). As such, the size of the minimum δ-guarding set is defined as $opt(P, \delta)$. A polygon P has vision-stability δ if the optimal number of enhanced guards is the same as the optimal number of diminished guards for that matter ($opt(P, -\delta) = opt(P, \delta)$).

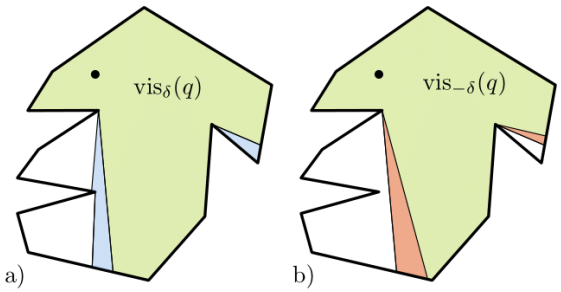


Figure 14: Enhanced ($vis_\delta(q)$) and diminished ($vis_{-\delta}(q)$) visibility regions of the one point q [10].

The core idea of Hengeveld and Miltzow's approach is to discretise the problem in order to improve the computation time. In this way, only a subset of potential guards and points to be guarded from P would be considered. In order to do so, they introduce the notions of a candidate guard set C and a witness set W. The role of the candidate guard set C is to discretise the guard searching space. As such, we can find an upper bound $opt[C]$ on the final guard set. Thus C is "pretty close" on the actual optimum guard set opt. The candidate guard set C is then initialised with a subset of all the potential guards. The guards that are included should have their visibility regions overlap as little as possible. Similarly, the witness set W discretises the space of the guards' visibility regions. At initialisation time, W contains a subset of all vertices and faces of P. It is important to also include faces in W. In this way, we account for the fact that if a point q is included in a face f, then f sees at least as much as q. So, we can determine an approximation of how much more f sees when compared to g. Thus, by adding the face as a witness, we are creating a discretisation of the space containing a superset of possible guard positions.

Firstly, rays are shot from every reflex vertex such that the angle between any two rays is at most δ, as observed in Figure 15, as taken from [10]. All the intersection points of the rays within P define the candidate guard set C, together with the faces they are part of. A witness guard set W is then defined by picking an arbitrary interior point of every face of P, together with all faces of P. In this way, the problem can be discretised in a way in which the solution to the discretised problem are also solutions to the continuous problem.

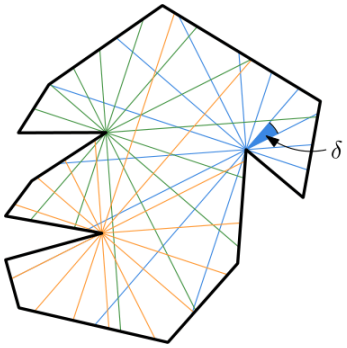


Figure 15: Shooting rays from all reflex Vertices with angles between them at most δ [10].

2.4.1 One-Shot Vision-Stable Algorithm

At first, an Integer Program (IP) is built and named the One-Shot Vision-Stable Algorithm. It computes the minimum number of point guards in polynomial time in a vision-stable polygon P with vision-stability δ and r reflex vertices. The guards are encoded by binary integers, and linear equations and inequalities are used to encode visibility. Assuming that δ is part of the input, the algorithm returns the optimal solution only if the vision-stability of P is larger or equal to δ. The one-shot vision-stable algorithm is reliable, as it also verifies that its result is an optimal solution.

Let f_1 be the optimisation fuction of the IP. The function f_1 uses variables $[[c]]$ for every candidate $c \in C$, and $[[w]]$ for every witness $w \in W$. Its role is to count the number of guards that are used with the purpose of minimising this number. Additionally, it makes use of parameter factor $1 + \varepsilon$ to ensure that vertex candidates are preferred over face candidates. By choosing $\frac{1}{\varepsilon} = |C| + |W| + 1$, we ensure that ε is sufficiently small. As such, the optimisation function f_1 of the IP is

$$f_1 = \sum_{c \in \text{vertex}(C)} [[c]] + (1+\varepsilon) \sum_{c \in \text{face}(C)} [[c]] + \varepsilon \sum_{w \in \text{face}(W)} [[w]].$$

For every witness w, its visible set of candidates are denoted as $vis(w)$, where $face$ and $vertex$ refer to whether the candidate is a face or a vertex, respectively.

Additionally, let vis(w) be the set of candidates that see witness $w \in W$ completely. For every vertex-witness $w \in \text{vertex}(W)$, we formulate a constraint such that all $w \in \text{vertex}(W)$ is seen:

$$\sum_{[[c]] \in vis(w)} c \geq 1, \forall w \in \text{vertex}(W).$$

We similarly formulate a constraint for every face-witness $w \in \text{face}(W)$. However, we add $[[w]]$ as a variable to relax it, so that the focus is placed on first seeing all vertex-witnesses

$$[[w]] + \sum_{[[c]] \in vis(w)} c \geq 1, \forall w \in \text{face}(W).$$

Nonetheless, the one-shot vision-stable algorithm proves to be too slow for practical reasons. The bottleneck is however not solving the IP in itself, but computing visibilities. Hence, an iterative vision-stable algorithm is devised, that is faster and retains similar performance guarantees.

2.4.2 Iterative Vision-Stable Algorithm

Starting with the smaller sets of candidates C and witnesses W, another IP is deployed in order to find a minimum guard set $S \subseteq C$ that sees all vertex-witnesses. This algorithm is then called the Iterative Vision Stable Algorithm, as it changes C and W in each iteration.

As the Iterative Vision Stable Algorithm is also named the Big IP, it also has an objective function f_2. The function f_2 minimises the number of face-guards used in S and the number of unseen face-witnesses. If S contains only point guards and sees all face-witnesses, the algorithm reports the optimal solution by using the One-Shot IP. As such, f_2 assures that all witnesses w are seen by their set of candidates $vis(w)$, and enforces that only the number of guards s resulted from the One-Shot IP is used. The optimisation function with its constraints of the Big IP becomes

$$f_2 = \sum_{x \in splittable(W \cup C)} [[x]] \tag{1}$$

$$\sum_{[[c]] \in C} c = s, s \in \mathbb{Z}, s \text{ the number of used guards in the One-Shot IP} \tag{2}$$

$$\sum_{[[c]] \in vis(w)} c \geq 1 \tag{3}$$

$$1 - (\varepsilon \sum_{[[c]] \in vis(w)} c) \geq [[w]]. \tag{4}$$

If not all point guards are used, or not all face-witnesses are seen, the faces of P are split (discretised). Then the algorithm continues to the next iteration. The One-Shot IP is then used as soon as all faces are deemed unsplittable (they have reached a certain granularity λ).

Inspired by the Iterative Algorithm, an FPT algorithm for the Art Gallery Problem can be created. In this case, the FPT algorithm would have as parameter of chord-visibility width. Given a chord c of P, we count the number $n(c)$ of vertices visible from c. The chord-visibility width of P ($cw(P)$) is the maximum $n(c)$ over all possible chords c. By using this measure, it is possible to describe the local complexity of a polygon: many synthetic arbitrary polygons have much smaller chord-visibility width than they have reflex vertices, and polygons constructed with hardness reductions have the chord-visibility width proportional to the total number of vertices.

2.4.3 Experimental Results

The previously introduced algorithms are tested experimentally in terms of their running times and solution quality with respect to the input size, chord-visibility width and vision-stability of the input polygons P. The tests were run on 30 instances of arbitrary simple polygons of sizes 60, 100, 200 and 500 vertices. The algorithms have been implemented both with the theoretical performance guarantees $\mathcal{T}$ (if no solution is found in $\mathcal{T}$, abort) and with the practical optimisation of critical witnesses.

When running the Iterative Algorithm with optimisations, reasonable practical results were found: all tested instances were solved to optimality, although the overall running time was not improved when compared to state-of-the-art algorithms. The median running time was also lower than the average running time. This could be accounted for by the sensitivity of the algorithm to the vision-stability of a polygon. Thus, a few polygons that are hard to solve (have low vision-stability) outweigh the rest of the running times.

The Iterative Algorithm without optimisations was on the 60 vertices instance and with the

theoretical limit $\mathcal{T}$ as a safe guard. A solution was found in an efficient manner (within an hour) only for 25 out of 30 instances. For the rest of the 5 instances, the Big IP performed unnecessary splits, which negatively influenced the running times.

Additionally, the correlation between the granularity λ of the input polygon was tested for the iterative algorithm with performance guarantees $\mathcal{T}$. There appeared to be a strong correlation between the running times and λ. Thus, lower granularity implied a shorter running time.

In terms of the convergence to the optimal solution, the iterative algorithm with optimisations has been used to show the fast convergence in the first few iterations. The speed of the convergence slows down as the algorithm nears its end due to the increasing number of candidates and witnesses that are added to C and W^*, respectively, in the later iterations. Nonetheless, it appears that computing the visibility area of the polygon is still the bottleneck in the algorithm's performance.

3 Theory

This section will investigate the theory aspects of solving the Art Gallery Problem using gradient descent. Namely, we will explain what the optimisation function in the context of the Art Gallery problem is, and how it can be solved with gradient descent. We will initially apply gradient descent to only one guard, then extend it to multiple guards.

3.1 Guarding the Polygon with One Point

First, we will explore how gradient descent can be applied for the case that we want to guard the polygon using only one guard.

3.1.1 Gradient Descent

Let P be a polygon and $g = (x, y) \in P$ a guard. We are interested in computing the best direction for moving g inside P such that the visibility area $Vis(g)$ increases. That is, exploring what would be a better position g' to move g to such that g "sees more" of the polygon P.

We define $f(g) = \text{Area}(g)$ as the area seen by a guard g. Let $\bigtriangledown\text{Area}_r(g)$ be the local change in the area guarded by point g around a reflex vertex r seen by guard g, thus $r \in Vis(g)$. Given all reflex vertices i, the total (global) change in the area seen by g can be thus summed up to $\text{Area}(g) = \sum_i \text{Area}_i(g)$. Figure 16 offers an example for this case for a polygon P and its reflex vertices r_1 and r_2. The polygon P is guarded by g, and its position is modified to g' by a small change ∂y in its y-coordinate. The visibility areas of g are $Area_{r_1}$ and $Area_{r_2}$ around reflex vertices r_1 and r_2, respectively. In this way, the total change in the visibility area of g is computed as $\bigtriangledown\text{Area}(g) = \bigtriangledown\text{Area}_{r_1}(g) + \bigtriangledown\text{Area}_{r_2}(g)$.

Thus, we consider $f(g)$ as the continuous objective function of the Art Gallery Problem. We can then use gradient descent as a method to optimise the objective function f. We will define below what the methodology of gradient descent consists of.

Let $\bigtriangledown f$ be the gradient of f. The gradient then indicates the direction of the steepest descent for the objective function $f(g)$. The learning rate (step size) α is the size of the steps taken to reach the optimum. It is typically a small value, and it is evaluated and chosen based on the behaviour of the optimisation function.

After the gradient $\bigtriangledown f$ is computed, we can use it to calculate the new optimised position g' of guard g:

$$g' = g + \alpha \bigtriangledown f(g).$$

In later sections we will experiment with various learning rates. As such, we will explore how they influence the performance of our algorithm in relation to different test polygons.

3.1.2 Computing the Gradient

Given that f is a function that describes the visibility area of a point g, we first need to define how its gradient is computed. We will simplify the gradient computation without losing generality. As such, we will rotate the plane with rotation matrix R, so that g and any reflex vertex r have the same x-coordinate. In this way, we only need to compute the gradient when we vary the y-coordinate. The computation of the gradient remains the same regardless of the rotation applied to the plane.

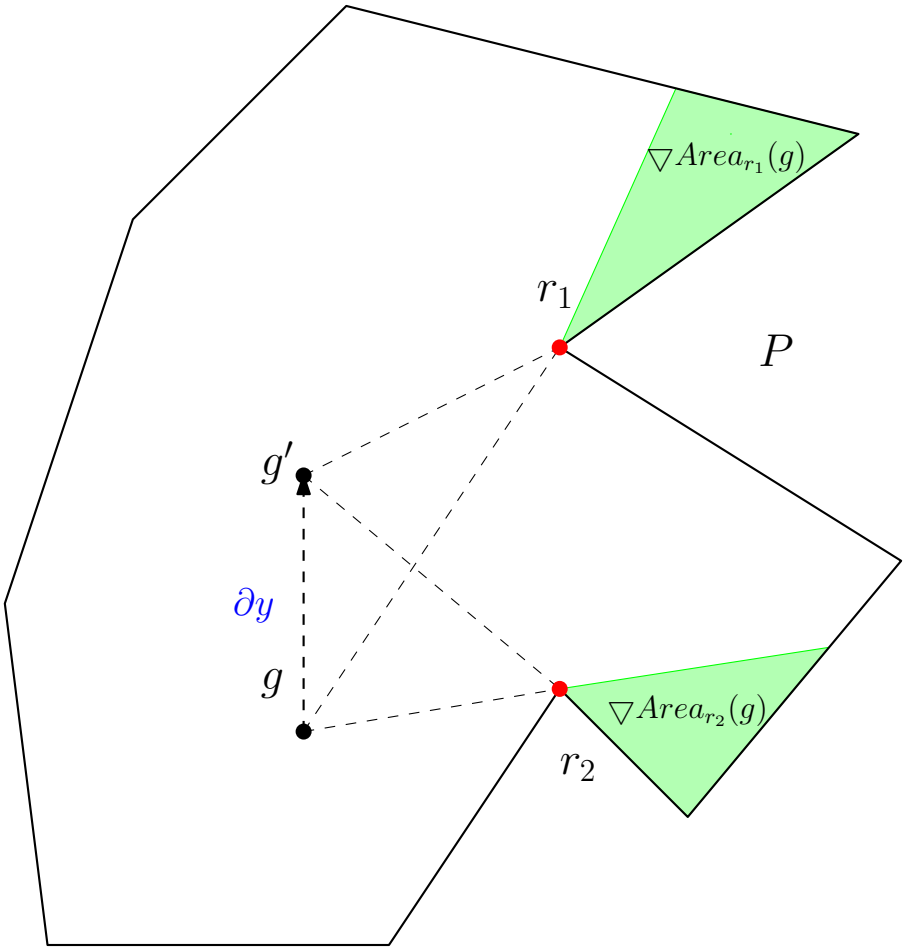


Figure 16: Global change in the area seen by g when moved by ∂y to a new position g'.

We will use the notation $\frac{\partial f}{\partial y}$ to denote the change in the visibility area $f(g)$ when the plane is rotated and then the y-coordinate is modified by a small amount $\partial y \to 0$. In this way, we define

$$\bigtriangledown f(g) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)^{\intercal} \tag{5}$$

to be the gradient of f given that P is guarded by a point g.

We will now create a canonical geometrical construction that allows us to further define and compute $\bigtriangledown f$. In this case, we consider the normalised length of the gradient as $|| \bigtriangledown f(g)|| = 1$. This canonical construction is displayed in Figure 17.

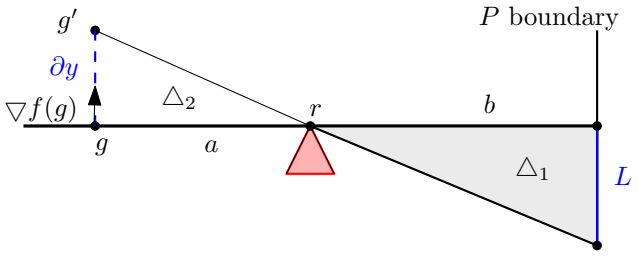


Figure 17: Canonical gradient construction for when the position of g is varied by a small amount ∂y around reflex vertex r to the new position g'.

Take a boundary line segment of P, r a reflex vertex inside P and g a guard whose optimal position we are interested in. The reflex vertex r is seen by g. Let $\overline{gr} = a$ be known, and let $\triangle rgg' = \triangle_2$. Similarly, let b be the known distance between r and the polygon boundary in question.

Let ∂y be a tiny change in the y-coordinate of g. Let g' be the new position of g given the change ∂y. The point g' can see more around r on the polygon boundary. Let L be the new segment that g' sees on the polygon boundary. As such, let $\triangle_1$ denote the increase in the visibility area of g when it moves to position g':

$$\bigtriangledown \text{Area}_r(g) = \triangle_1. \tag{6}$$

We are now interested in computing how the area seen by guard g increases given the change ∂y in the position of g. The distances a and b are known. As such, we aim at expressing the gradient $\bigtriangledown\text{Area}_r$ for point g and reflex vertex $r \in Vis(g)$ using a and b. Since $\bigtriangledown\text{Area}_r$ depends on the change in the coordinates of g, computing it is tightly connected to the change in the area of triangle $\triangle_1$. We will proceed to calculate the area of $\triangle_1$ below.

Given that triangles $\triangle_1$ and $\triangle_2$ are square triangles, their areas can be calculated as:

$$\begin{aligned}\text{Area}_{\triangle_1}(g) &= \frac{bL}{2},\\ \text{Area}_{\triangle_2}(g) &= \frac{a\partial y}{2}.\end{aligned}$$

Given that $\overline{gg'}$ is parallel to polygon's boundary, we can use Thales's Theorem in triangles $\triangle_1$ and $\triangle_2$ to compute the length L:

$$\begin{aligned}\frac{\partial y}{L} &= \frac{a}{b}\\ L &= \frac{b\partial y}{a}.\end{aligned} \tag{7}$$

So, the area of $\triangle_1$ can be computed:

$$\begin{aligned}\text{Area}_{\triangle_1}(g) &= \frac{Lb}{2}\\ &\overset{(7)}{=} \frac{\frac{b\partial y}{a}b}{2}\\ \text{Area}_{\triangle_1}(g) &= \frac{b^2\partial y}{2a}.\end{aligned} \tag{8}$$

We can now compute the gradient $\bigtriangledown\text{Area}_r$ in a plane rotated by R for a point g and a reflex guard r seen by g as

$$
\begin{aligned}
R \bigtriangledown \text{Area}_r(g) &\overset{(5)}{=} \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)^\intercal \\
&\overset{(6)}{=} \left(0, \frac{\text{Area}_{\triangle_1}}{\partial y}\right)^\intercal \\
R \bigtriangledown \text{Area}_r(g) &\overset{(8)}{=} \left(0, \frac{b^2}{2a}\right)^\intercal . \qquad (9)
\end{aligned}
$$

Therefore, for all the reflex vertices r guard g can see, the total gradient $\bigtriangledown f$ becomes the sum of all the partial gradients $\bigtriangledown f_r$ as

$$
\bigtriangledown f(g) = \sum_{i \in R(g)} \bigtriangledown \text{Area}_i(g), \qquad (10)
$$

$$
R(g) = \{\text{reflex vertices of } P \text{ seen by } g\}.
$$

3.1.3 Computing the New Guard's Position

We can now use the coordinates of the gradient $\bigtriangledown f$ to compute the movement direction of the guard g given all the reflex vertices from P seen by g. In order to do so, we will use the construction depicted in Figure 18.

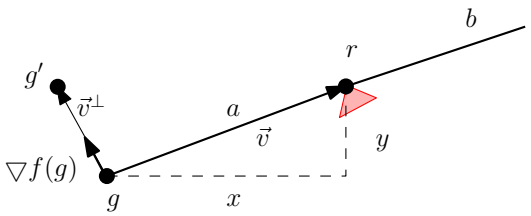


Figure 18: Computing the new position g' of guard g around reflex vertex r based on the gradient $\bigtriangledown f$.

Let $\vec{v}$ be the vector corresponding to the direction of movement from guard g to a reflex vertex r, such that $\vec{v} = (r - g) = (x, y)^\intercal$, with norm $||\vec{v}|| = a$. So, $||\frac{\vec{v}}{a}|| = 1$.

Let $\vec{v}^\perp = (g' - g)$ be the vector corresponding to the direction of movement from guard g to its new position g'. Vector $\vec{v}^\perp$ is orthogonal to $\vec{v}$, in the same direction as $\bigtriangledown f$, such that $||\vec{v}|| = ||\vec{v}^\perp|| = a$. We will use the coordinates of $\vec{v}^\perp$ to compute the coordinates of $\bigtriangledown f$ and thus the direction in which g needs to move.

The coordinates of $\vec{v}^\perp$ can then be computed using the construction from Figure 19. Since $\vec{v}^\perp \perp \vec{v}$, and g and r are on the right-hand side of $\vec{v}^\perp$, the coordinates of $\vec{v}^\perp$ will be rotated by $-90°$ so that $\vec{v}^\perp = (-y, x)^\intercal$. Analogously for the case when g and r are rotated by 180° to the left-hand side of $\vec{v}^\perp$, the coordinates of $\vec{v}^\perp$ will be rotated by 90° to $\vec{v}^\perp = (-x, y)^\intercal$.

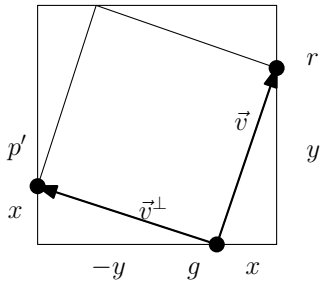


Figure 19: Computing the coordinates of $\vec{v}^{\perp}$ given the guard g and the reflex vertex $r = (x, y)$.

We know that the norm of the total gradient $\bigtriangledown f$ is

$$\begin{aligned} || \bigtriangledown f(g)|| &\overset{(10)}{=} || \sum \left(0, \frac{b^2}{2a}\right)^{\intercal} || \\ &\overset{(9)}{=} \sqrt{\left(\frac{b^2}{2a}\right)^2} \\ || \bigtriangledown f(g)|| &= \frac{b^2}{2a}. \end{aligned} \tag{11}$$

Since $\bigtriangledown f$ has the same direction as $\vec{v}^{\perp}$, we wish to normalise it with $\frac{1}{a}$ (the norm of $\vec{v}^{\perp}$). Therefore, the gradient for guard g and one reflex vertex $r \in Vis(g)$ can be computed as

$$\bigtriangledown f(g) = \vec{v}^{\perp} \frac{b^2}{2a} \frac{1}{a}. \tag{12}$$

As mentioned before, the total gradient for guard g and all the reflex vertices r the guard can see is

$$\begin{aligned} \bigtriangledown f(g) &= \sum_{r \in R(g)} \bigtriangledown \text{Area}_r, \\ R(g) &= \{\text{reflex vertices of } P \text{ seen by } g\}. \end{aligned}$$

The new position g' of guard g based on all the reflex vertices it can see is:

$$g' = g + \alpha \bigtriangledown f(g). \tag{13}$$

3.2 Guarding the Polygon with Multiple Guards

In this subsection we will investigate how to generalise the computation of gradient descent to multiple guards.

3.2.1 Computing the Gradient for Two Guards

Let point g_1 be the guard we have previously computed the gradient for. Let point g_2 be another guard in the polygon. The position of g_2 is optimised around reflex vertex r_2, as seen in Figure 20. Guard g_2 sees a part of the visibility region already seen by g_1. Let the shared seen region be $\triangle_2$, and the region that is only visible by g_1 be $\triangle_1$.

The visibility regions of guards g_1 and g_2 are computed. Let p be the intersection point between them. Let d be the point seen by g_1 on the polygon boundary. Point p divides the segment seen by

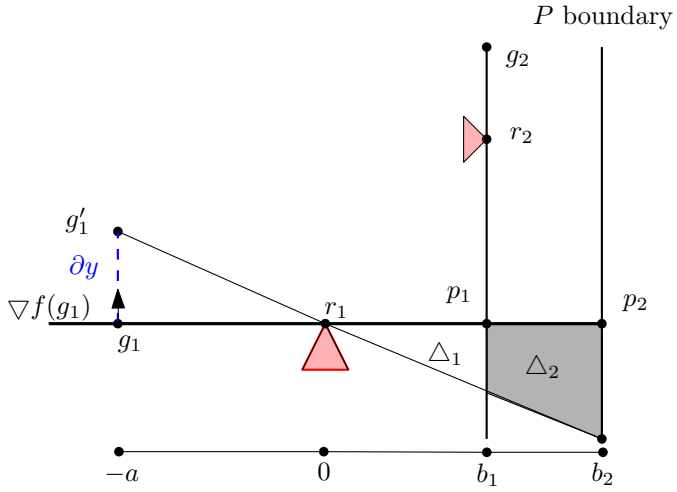


Figure 20: Canonical gradient construction for when the visible area of g_1' is also seen by guard g_2.

g_1 behind the reflex vertex r_1 into two subsegments $\overline{r_1 p_1}$ and $\overline{p_1 p_2}$. The lengths of the subsegments are b_1 and b_2, respectively, with $b_1 + b_2 = b$. Recall that b is the distance between the reflex vertex r_1 and the polygon boundary. Thus, b_2 corresponds to the length of the shared seen segment.

As before in equation 8, the area seen by g_1 is

$$\text{Area}_{\triangle_1+\triangle_2}(g) = (b_1 + b_2)^2 \frac{\partial y}{2a}.$$

However, $\triangle_2$ is already seen by guard g_2. Since this area is already covered, we are only interested in how g_1 can cover the remaining $\triangle_1$. So, we do not need to take $\triangle_2$ into account when computing the gradient of g_1. We thus need to subtract the area of $\triangle_2$ from the total area seen by g_1.

First, we need to compute the shared area seen $\triangle_2$ as:

$$\begin{aligned}
\text{Area}_{\triangle_2}(g) &= \text{Area}_{\triangle_1+\triangle_2}(g) - \text{Area}_{\triangle_2}(g) \\
&= (b_1 + b_2)^2 \frac{\partial y}{2a} - b_1^2 \frac{\partial y}{2a} \\
\text{Area}_{\triangle_2}(g) &= \left[(b_1 + b_2)^2 - b_1^2\right] \frac{\partial y}{2a}. \qquad (14)
\end{aligned}$$

Then, we can compute the area $\triangle_1$ seen exclusively by g_1 by subtracting the shared area of $\triangle_2$ from the total area seen by g_1:

$$\begin{aligned}
\text{Area}_{\triangle_1}(g) &= \text{Area}_{\triangle_1+\triangle_2}(g) - \text{Area}_{\triangle_1}(g) \\
&= (b_1 + b_2)^2 \frac{\partial y}{2a} - \left[(b_1 + b_2)^2 - b_1^2\right] \frac{\partial y}{2a} \\
\text{Area}_{\triangle_1}(g) &= b_1^2 \frac{\partial y}{2a}.
\end{aligned}$$

3.2.2 Computing the Gradient for Multiple Guards

We can now generalise the gradient computation to m guards and m intersection points. Let g be the guard whose position we wish to optimise around reflex vertex r as before. Let the areas highlighted in blue in Figure 21 be the parts of the visibility region of g that are also seen by the other $m - 1$ guards. Let the visibility region of g intersect other guards' visibility regions in m

intersection points $p_1, p_2, ..., p_m$. Let the distance from the reflex vertex r to the intersection points be $b_1, b_2, ..., b_m, b$.

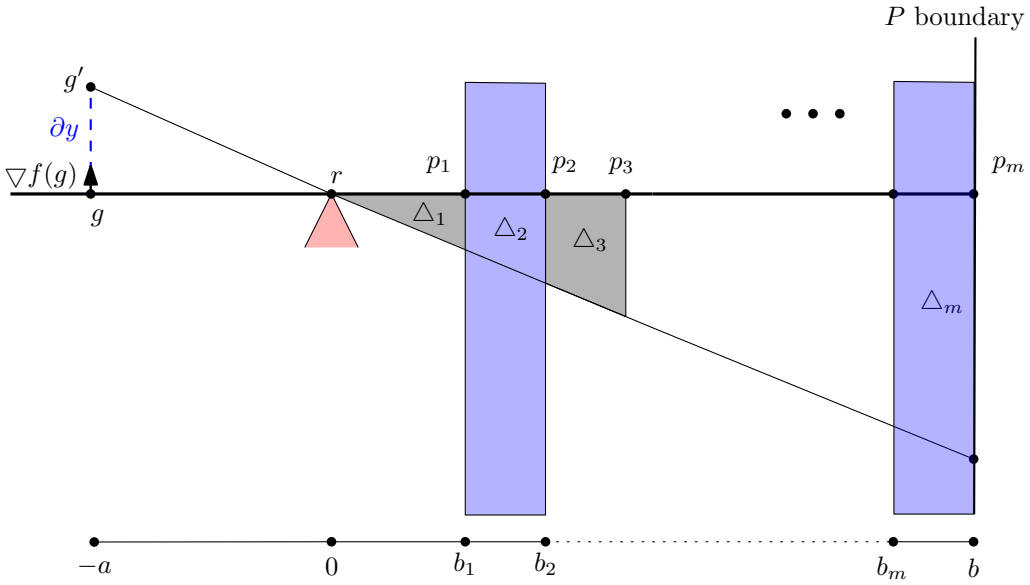


Figure 21: Canonical gradient construction for where the pink parts of the visible area of g are also seen by other guards. The grey polygons $\triangle_1, \triangle_3, ..., \triangle_{m-1}$ are the areas that are exclusively seen by g.

We can now generalise equation 14. Namely, we need to subtract the areas that are seen by other guards from the total area seen by guard g.

We start to move in the direction from the polygon boundary to reflex vertex r. We know the intersection points $p_{m-1}..., p_3, p_1$ of the shared seen regions with the visibility region of g. Thus, we can compute the distances $\overline{rp_{m-1}}, ..., \overline{rp_3}, \overline{rp_1}$ between the reflex vertex r and the beginning of the shared visibility regions. Analogously, we can compute the distances $\overline{rp_m}, ..., \overline{rp_4}, \overline{rp_2}$ between the reflex vertex r and the end of the shared visibility regions.

The areas of polygons $\triangle_1, \triangle_2, ..., \triangle_m$ do not grow linearly. For this reason, we cannot simply subtract the sum of the shared areas from the total area seen by g. An explanation about why this is the case is given in Figure 22. Take overlapping polygons s_1, s_2, s_3, s_4 with areas $\text{Area}_{s_1} = 2$, $\text{Area}_{s_2} = 3$, $\text{Area}_{s_3} = 4$, $\text{Area}_{s_4} = 5$. We want to compute the area of polygons s_1 and s_2. However, it is incorrect to simply add their areas together as $\text{Area}_{s_1+s_3} = 2 + 4 = 6$, as s_2 is overlapping in between them. Instead, from the total area $\text{Area}_{s_1+s_2+s_3+s_4} = \text{Area}_{s_4} = 5$ we can sequentially subtract the areas we are not interested in $(\text{Area}_{s_2}, \text{Area}_{s_4})$ and add the ones we are interested in $(\text{Area}_{s_1}, \text{Area}_{s_3})$. Namely, $\text{Area}_{s_1+s_3} = \text{Area}_{s_1+s_2+s_3+s_4} - \text{Area}_{s_4} + \text{Area}_{s_3} - \text{Area}_{s_2} + \text{Area}_{s_1} = 5 - 5 + 4 - 3 + 2 = 3$.

Similarly to Figure 22, we can compute the area seen exclusively by g. From the total area $\text{Area}_{\triangle_1+\triangle_2,...,\triangle_m}$ seen by g we need to alternatively subtract the shared areas $\text{Area}_{\triangle_m}, ..., \text{Area}_{\triangle_4}$, $\text{Area}_{\triangle_2}$, then add the exclusively seen areas $\text{Area}_{\triangle_{m-1}}, ..., \text{Area}_{\triangle_3}, \text{Area}_{\triangle_1}$.

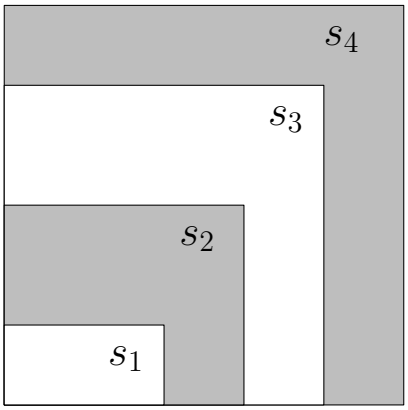


Figure 22: Example for computing the area of overlapping polygons with areas $\text{Area}_{s_1} = 2$, $\text{Area}_{s_2} = 3$, $\text{Area}_{s_3} = 4$, $\text{Area}_{s_4} = 5$. It is incorrect to compute $\text{Area}_{s_1+s_3} = 2 + 4 = 6$. Instead, $\text{Area}_{s_1+s_3} = \text{Area}_{s_1+s_2+s_3+s_4} - \text{Area}_{s_4} + \text{Area}_{s_3} - \text{Area}_{s_2} + \text{Area}_{s_1} = 5 - 5 + 4 - 3 + 2 = 3$.

This results in the fact that the area seen by g exclusively can be computed as

$$\begin{aligned}\text{Area}_{\triangle_1+\triangle_3+\ldots+\triangle_{m-1}}(g) &= \text{Area}_{\triangle_1+\triangle_2,\ldots,\triangle_m}(g) \\ &\quad - \text{Area}_{\triangle_{m-1}}(g) + \text{Area}_{\triangle_{m-2}}(g) \\ &\quad - \ldots \\ &\quad - \text{Area}_{\triangle_2}(g) + \text{Area}_{\triangle_1}(g) \\ &= \left(b^2 - b_{m1}^2 + b_{(m-1)2}^2 - \ldots - b_{12}^2 + b_{11}^2\right) \frac{\partial y}{2a}.\end{aligned}$$

Alternatively, if $\triangle_1$ is first seen by other guards (so $b_{11} = 0$), then all the signs are flipped. This claim can also be supported by the intuition of Figure 22. If we are interested in the areas of s_2 and s_4, then it is incorrect to compute them as $\text{Area}_{s_2+s_4} = 3 + 5 = 8$. This is because s_1 and s_3 are overlapping in between them. Instead, $\text{Area}_{s_2+s_4} = \text{Area}_{s_1+s_2+s_3+s_4} + \text{Area}_{s_4} - \text{Area}_{s_3} + \text{Area}_{s_2} - \text{Area}_{s_1} = 5 + 5 - 4 + 3 - 2 = 7$.

3.3 Momentum

In this section we introduce an extension to the regular gradient descent algorithm: momentum. Its aim is to smoothen out the large noisy jumps in the gradient descent computation [9]. Momentum builds upon the idea of considering the past states of the gradient descent computation. In this way, the past states create "inertia" to the newly computed gradient state. This results in the overall optimisation trajectory to be smoother.

So, the position g_i at iteration i of a guard g will not be calculated anymore based only on the current computation of gradient descent. Instead, it will also take into account past values of gradient descent. In order to decide how much past states influence the value of the current position, we will take each gradient value with a weight.

Let $M_i(g_i)$ be the momentum for a guard at iteration i. Let $\Delta f_i(g_i)$ be the computation of the gradient descent in iteration i. Let the weight of a gradient descent value be a hyperparameter γ. As such, we can take into account the value of the previous gradient $\Delta f_{i-1}(g_{i-1})$ with weight γ. However, we still want the current gradient $\Delta f_i(g_i)$ to influence the guard's movement. This can

happen with a weight of $1-\gamma$. So, the computation for the momentum at iteration i becomes

$$M_i(g_i) = \gamma M_{i-1}(g_{i-1}) + (1-\gamma)\Delta f_i(g_i).$$

We start with

$$M_0(g_0) = (1-\gamma)\Delta f_0(g_0).$$

We can then check for correctness how the next iterations of the momentum are carried out:

$$\begin{aligned} M_i(g) &= \gamma M_{i-1}(g_{i-1}) + (1-\gamma)\Delta f_i(g_i) \\ &= \gamma(\gamma M_{i-2}(g_{i-2}) + (1-\gamma)\Delta f_{i-1}(g_{i-1})) + (1-\gamma\Delta f_i(g_i)) \\ &= \gamma^2 M_{i-2}(g_{i-2}) + (1-\gamma)(\gamma\Delta f_{i-1}(g_{i-1}) + \Delta f_i(g_i)) \\ &= \gamma^3 M_{i-3}(g_{i-3}) + (1-\gamma)(\gamma^2\Delta f_{i-2}(g_{i-2}) + \gamma\Delta f_{i-1}(g_{i-1}) + \Delta f_i(g_i)) \\ &... \end{aligned}$$

In this way, our momentum computation exponentially decreases the influence of a past state over the guard's current movement. So, the previous value of the momentum will exert its inertia with γ influence over the new value of the momentum.

Similarly, the new position g_i of guard g with learning rate α will now be calculated as

$$g_i = g_{i-1} + \alpha M_i(g_{i-1}).$$

3.4 Line Search

In this section we introduce another extension to the regular gradient descent algorithm: Line Search [15]. Its aim is to use gradient descent's descending direction as a guide for finding the optimal solution to the optimisation function. Then, given a step size factor, multiple solutions along the descending line are computed. The optimal one for the specific iteration is chosen.

Figure 23 illustrates an example for this extension. Take guard g and reflex vertex r. Let $\frac{1}{x}$ be the starting search factor, and s the step size factor. Recall that $M_i(g)$ is the optimal movement direction for a guard at iteration i. As such, line search will compute the optimal guard position $g_{i+1} = g_i + \frac{s^t}{x}M_i(g_i)$, with t the best step power for the guard from $\{\frac{1}{x}M_i, \frac{s}{x}M_i(g_i), \frac{s^2}{x}M_i(g_i), ...\}$. The dashed line represents the direction of gradient descent, as computed based on the gradient and reflex vertex pull computations. Given a step size s, 3 different possible new guard positions can be identified: $g'_{i_1} = g_i + \frac{1}{x}M_i(g_i)$, $g'_{i_2} = g_i + \frac{s}{x}M_i(g_i)$, $g'_{i_3} = g_i + \frac{s^2}{x}M_i(g_i)$. The optimal guard will be the one with the largest global area seen increase. If the global area is not increased, a best solution is also chosen if a guard is gaining visibility to a previously unseen part of the polygon, regardless of the area increase.

It is worth mentioning that Line Search deems the use of a learning rate obsolete. Because the search factor finds the optimal position on the direction line, it acts thus as a learning rate.

3.5 Reflex Vertex Pull

In this section we introduce an additional idea to our gradient descent algorithm: pulling a guard towards reflex vertices. The pull strategy makes use of the second derivative of our optimisation function $f(g) = \text{Area}(g)$.

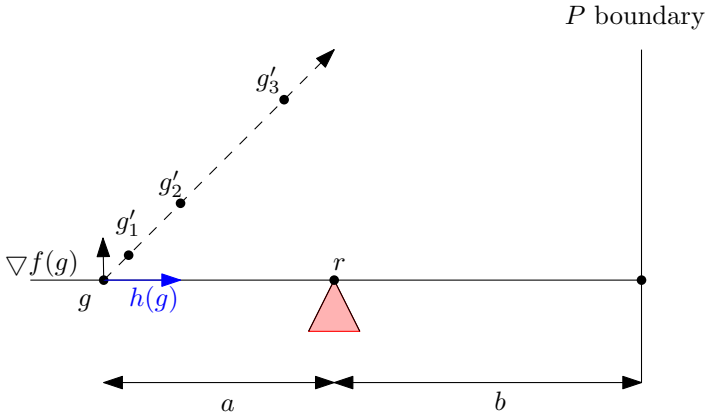


Figure 23: The gradient descent line gives the direction of searching for the optimal guard position. Given a step size, there are 3 possible guard positioning on the dashed gradient descent line: $g'_{i_1}, g'_{i_2}, g'_{i_3}$. The best of them is chosen based on the size of the newly visible area seen.

As mentioned previously, we are using the first derivative to compute the gradient of a guard's movement. That is, we use the first derivative of $f(g)$ to find the direction of a guard's movement that increases its seen area. Additionally, we can also make use of the second derivative of $f(g)$ to explore what the rate of change in the area increase is.

We can achieve a more rapid increase in the seen area if we move a guard closer to a reflex vertex. The closer a guard is moved to a reflex vertex, the larger the increase in the area seen past the vertex. If a guard is moved directly on a reflex vertex, then the area seen past the vertex is maximised.

We will now explore how the pull towards a reflex vertex r can be computed, with an example in Figure 24. Let the guard $g = (0, 0)$, the reflex vertex $r = (2, 0)$ and the polygon boundary intersection point be $(5, 0)$. So, the distances $a = 2$ and $b = 3$ are known. Let h_r be the pull in question. Let g'_y be the new coordinates of guard g when only the gradient is taken into account. In that case, we are only interested in the small movement ∂y. Analogously, let g'_x be the new coordinate of guard g when g moves by a small amount ∂x towards the reflex vertex. As previously, and without losing generality, assume that g and r have the same x-coordinate by rotating the plane R. Combining the two movements together results in g' being the final position of g.

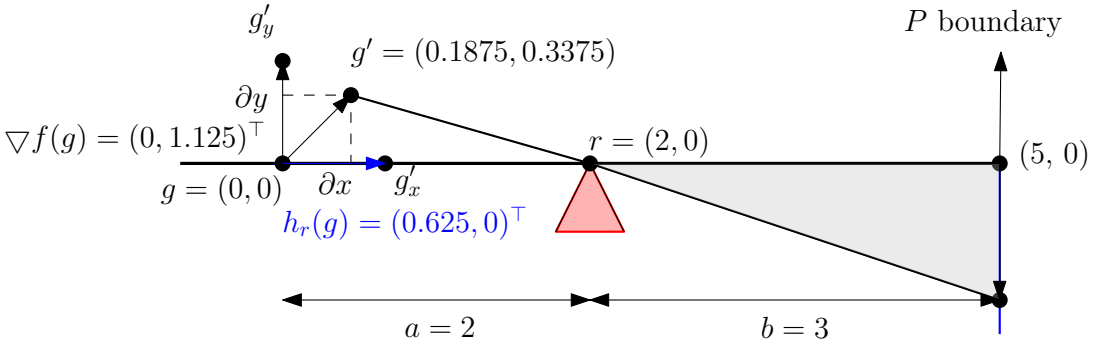


Figure 24: Computing the movements of the guard g based on both the gradient and the pull towards reflex vertex r. The new position of the guard with learning rate $\alpha = 0.3$ becomes $g' = (0.1875, 0.3375)$.

The gradient is computed as

$$\bigtriangledown f(g) \overset{(12)}{=} (0, \frac{b^2}{2a^2})^\intercal = (0, \frac{9}{8})^\intercal = (0, 1.125)^\intercal.$$

The pull is computed as

$$h_r(g) \overset{(15)}{=} (\frac{b^2}{2a^3}, 0)^\intercal = (\frac{9}{16}, 0)^\intercal = (0.625, 0)^\intercal.$$

So, the new position g' of the guard g given the learning rate $\alpha = 0.3$ becomes

$$g' = g + \alpha(\bigtriangledown f(g) + h(g)) = 0.3((0, 1.125)^\intercal + (0.625, 0)^\intercal) = (0.1875, 0.3375).$$

We will now explain how we arrived at the pull computation. The pull is the second derivative of the norm of the gradient, so

$$h_r(g) = \bigtriangledown || \bigtriangledown f_r(g)|| = \left(\frac{\partial \bigtriangledown f(g)}{\partial x}, \frac{\partial \bigtriangledown f(g)}{\partial y}\right)^\intercal = \left(\frac{\partial \bigtriangledown f(g)}{\partial x}, 0\right)^\intercal.$$

We can now calculate h_r as follows: the Euclidean norm of the gradient is $|| \bigtriangledown f(g)|| = \frac{b^2}{2a}(11)$, so the norm of the second gradient is

$$\begin{aligned} ||h_r(g)|| &= || \bigtriangledown || \bigtriangledown f(g)|||| \\ &= || \bigtriangledown (\frac{b^2}{2a})|| \\ &= \frac{b^2}{2a}\frac{d}{da} \\ &= b^2 \frac{1}{2a}\frac{d}{da} \\ ||h_r(g)|| &= -b^2 \frac{1}{2a^2}. \end{aligned}$$

Note that we are using the norm approximation heuristic

$$\bigtriangledown || \bigtriangledown f(g)|| \approx \bigtriangledown \sum_{i \in R(g)} ||f_i(g)||,$$
$$R(g) = \{\text{reflex vertices of } P \text{ seen by } g\}.$$

The reason for this choice is that computing the norm of the partial gradients, summing them and then computing their gradient is much easier than computing the gradient of their sum

$$\bigtriangledown || \bigtriangledown f|| = \bigtriangledown || \sum_{i \in R(g)} f_i||,$$
$$R(g) = \{\text{reflex vertices of } P \text{ seen by } g\}.$$

We are aware of the fallacies of this approximation. Take the opposing unit vectors $a_1 = (1, 0)^\intercal$, $a_2 = (-1, 0)^\intercal$. The sum of their norms $\sum_i ||a_i|| = 2$, whereas the norm of their sum is $|| \sum_i a_i|| = 0$. Clearly, they are not the same. Nonetheless, we still consider this approximation due to computation speed improvements and easiness of use.

The pull $h_r(g)$ is directed towards the reflex vertex r. So, it has the same orientation as vector

$\vec{v} = (r - g)$. We will normalise $h_r(g)$ with the norm of vector $\vec{v}$. Its norm is $||\vec{v}|| = \frac{1}{a}$. So,

$$h_r(g) = \vec{v}\frac{-b^2}{2a^2}\frac{1}{a} = \vec{v}\frac{-b^2}{2a^3}. \tag{15}$$

Let $h(g)$ be the total pull for guard g. As for the gradient, the total pull for guard g and all reflex vertices r the guard can see is

$$h(g) = \sum_{r \in R(g)} h_r(g),$$
$$R(g) = \{\text{reflex vertices of } P \text{ seen by } g\}.$$

Therefore, the movement of a guard g to the new position g' will take both the gradient and the pull into account:

$$g' = g + \alpha(\bigtriangledown f(g) + h(g)).$$

Additionally, we can choose how much influence the pull itself can have in the movement of the guard by adding a hyperparameter β:

$$g' = g + \alpha(\bigtriangledown f(g) + \beta h(g)).$$

3.5.1 Pull onto the Reflex Vertex

We have now created a heuristic for pulling a guard closer to a reflex vertex based on the increase in the seen area behind the reflex vertex. It could happen however that the pull towards the reflex vertex is very strong. In this case, the guard could be moved past the reflex vertex, in between the reflex vertex and the polygon boundary. Although the area seen behind the reflex vertex would be maximised, the guard would "unsee" (parts of) the area seen from its initial position g. In order to address this particular edge-case, the guard will be placed on the reflex vertex when the pull is strong enough.

This section will thus expand on a procedure that decides when a guard is best placed on a reflex vertex based on its pull towards it.

Firstly, we must define the condition of what a "strong enough" pull means. Such a pull would move the guard g "close enough" to the reflex vertex. Hence, we need to define what a minimum the distance between the new guard position g' and reflex vertex r would be. The most straightforward way to do so would be to assign a hyperparameter to the distance between g and r. Recall that $||\overline{gr}|| = a$. For now, let $\frac{2}{3}$ be the factor of closeness of g' to r. This means, that when the guard is more than two thirds of the distance close to the reflex vertex, then the guard is moved onto the reflex vertex:

$$||h_r(g)|| > \frac{2}{3}a. \tag{16}$$

Next, we need to account for the case where the guard is close to multiple reflex vertices. Namely, we consider the specific edge-case when a guard is in between two reflex vertices, illustrated

in Figure 25. Let D be the minimum distance in between any two reflex vertices of polygon P:

$$D = \min_{q \neq r} \text{ distance}(q, r),$$
$$\forall q, r \in P \text{ reflex vertices.}$$

Let g be a guard in between two reflex vertices r_1 and r_2 at distance D from each other. Let the pulls $h_{r_1}(g)$ and $h_{r_2}(g)$ be strong towards r_1 and r_2, respectively. However, because they are opposites in directions, they cancel each other out, resulting in g possibly not changing its position at all.

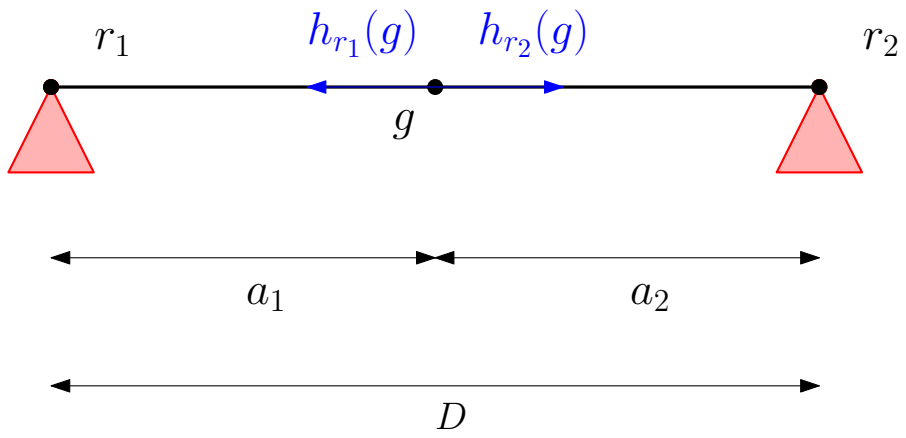


Figure 25: Guard g is in between two reflex vertices r_1 and r_2. Because they are collinear, g is equidistant from them and the pulls are equally strong $||h_{r_1}(g) = h_{r_2}(g)$, they cancel each other out.

We will now introduce another condition for pulling the guard towards the closer reflex vertex when pulls are cancelling each other out. If a guard g is not equidistantly placed in between two reflex vertices, we are interested in computing what the closer reflex vertex is. So, if g is closer to a reflex vertex than half of the minimum distance in between any two reflex vertices $\frac{D}{2}$, then we consider it close enough if:

$$a < \frac{D}{2}.$$

In this case we can choose to still move towards one of the reflex vertices and make progress.

An example of moving on top of a reflex vertex can be found in Figure 26. Let the guard $g = (1, 0)$, the reflex vertex $r = (2, 0)$ and the polygon boundary intersection point be $(5, 0)$. So, the distances $a = 1$ and $b = 3$ are known. Let $h_r(g)$ be the pull in question. The gradient is computed as

$$\bigtriangledown f(g) \overset{(12)}{=} (0, \frac{b^2}{2a^2})^\intercal = (0, \frac{9}{2})^\intercal = (0, 4.5)^\intercal.$$

The pull is computed as

$$h_r(g) \overset{(15)}{=} (\frac{b^2}{2a^3}, 0)^\intercal = (\frac{9}{2}, 0)^\intercal = (4.5, 0)^\intercal.$$

The guard is close enough to r to be moved onto it (16):

$$||h_r(g)|| > \frac{2}{3}a \iff 4.5 > \frac{2}{3}.$$

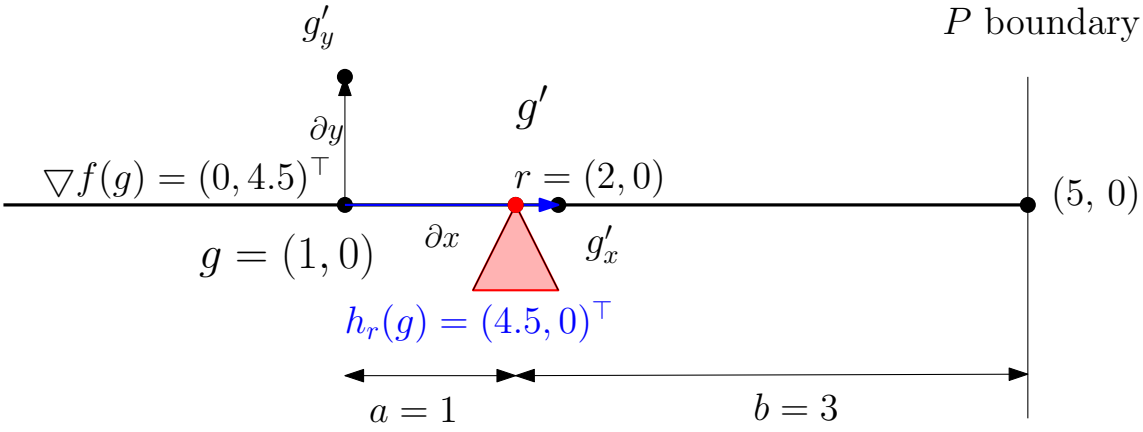


Figure 26: Computing the movements of the guard g based on both the gradient and the pull towards reflex vertex r. The guard g is close enough to r (16), so g is placed on top of the reflex vertex r.

So the new coordinate of the guard is $g' = r = (2, 0)$.

Note that Figure 26 displays how the closer the guard to a reflex vertex is, the stronger its pull. This comes in contrast with Figure 24. Given the same reflex vertex r and polygon boundary coordinates, the guard $g = (0, 0)$ was farther away, and its pull $h_r(g) = (0.625, 0)^\intercal$ was thus not as strong. Hence, we only want to place guards on reflex vertices when their pull is strong enough.

3.5.2 Pull Capping

Nonetheless, it can happen that the pull towards a reflex vertex is significantly larger in comparison to the gradient and the momentum. In that case, if the guard is not pulled onto the reflex vertex, it would at least have a very large jump towards the reflex vertex.

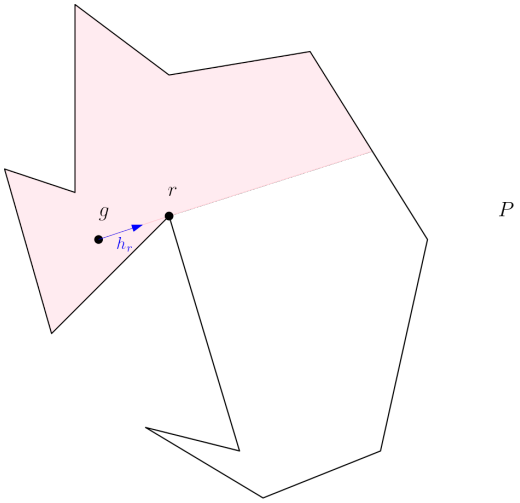
We want to smoothen out possibly erratic movement jumps. Just like in the case of momentum, we want our pull to be smoothened out when it is "suddenly too large". We define what "suddenly too large" is based on the momentum. If the pull is larger than a factor of μ than the momentum and a constant c, then we cap it at that value. So, at step i, the momentum $h_i(g_i)$ can be capped as:

$$\text{if } ||h_i(g_i)|| > \mu||M_i(g_i)|| + c, \text{ then}$$
$$h_i(g_i) = h_i(g_i)\frac{\mu||M_i(g_i)||}{||h_i(g_i)||}.$$

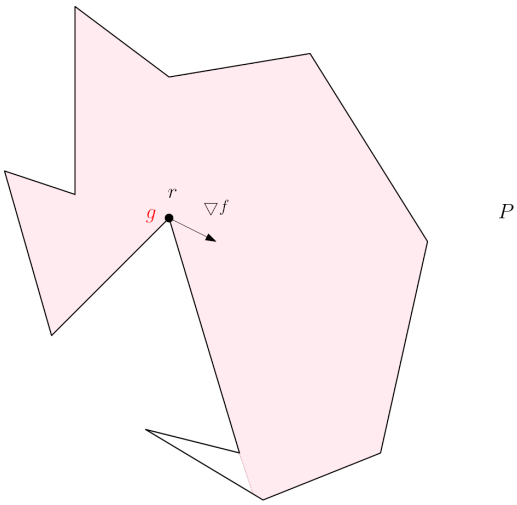
The factor μ becomes thus a hyperparameter to experiment with.

3.6 Reflex Area

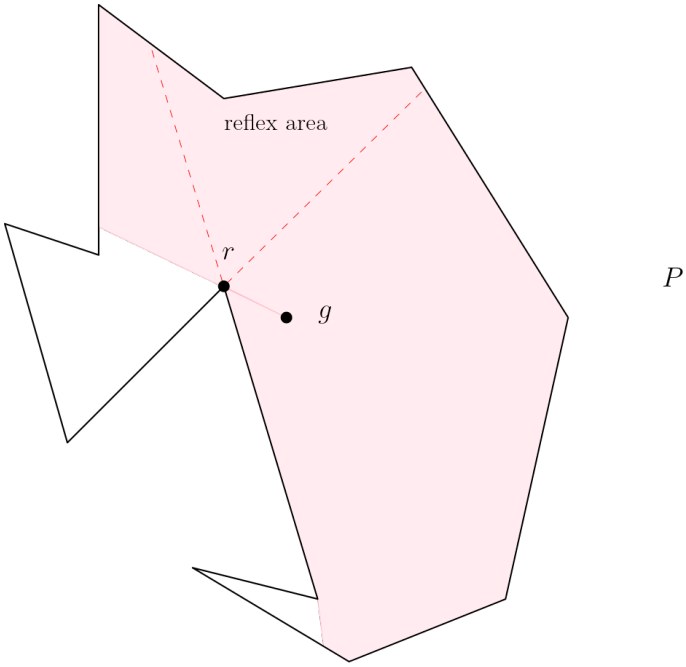
In this section we introduce an additional heuristic to our algorithm: the concept of *reflex area*. The reflex area design choice was made to counter-act the edge-case of a guard moving away from a reflex vertex and "unseeing" the area that it was already seeing. This case is illustrated in Figure 27. In Subfigure 27a, guard g starts moving with pull $h_r(g)$ towards reflex vertex r. The pull is strong enough to place g on r in Subfigure 27b. In this case, g can see everywhere around r. However, the new gradient $\bigtriangledown f(g)$ of g moves it past r in Subfigure 27c. The initially seen area before r is now not completely seen anymore by g.



(a) Guard g starts moving with pull $h_r(g)$ towards reflex vertex r. Guard g cannot see past r.



(b) Guard g is placed on reflex vertex r. Guard g can see both before r and past it.



(c) Guard g has moved past reflex vertex r. Guard g cannot see anymore before r.

Figure 27: Guard g moves towards and on the reflex vertex r. Eventually, its newly computed position is away from r and the reflex area. This results in g not seeing the initial area anymore.

Let rr' and rr'' be the extensions to the polygon boundary segments whose intersection is the reflex vertex r. We call *reflex area* the area between line segments rr' and rr'' that is contained inside the polygon. Figure 28 draws this concept. If a guard g has to move outside the reflex area, we project its new position g' onto the closest reflex line (in this case, rr'). Naturally, if g has to move inside the reflex area, it can do so unaffectedly. In this way, we maintain the gained property of a guard seeing everything around a reflex vertex while still allowing it to move away from the reflex vertex.

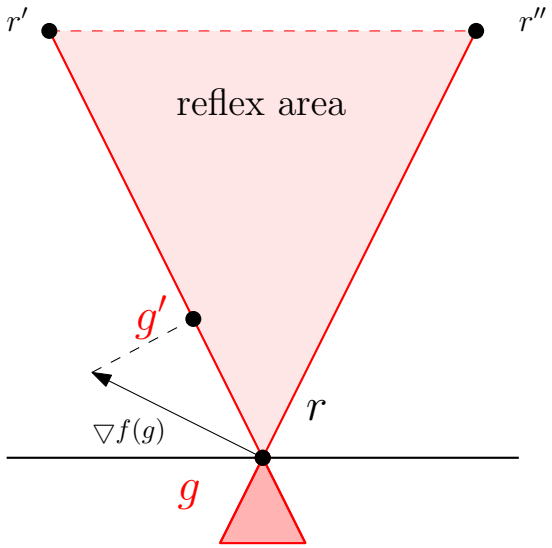


Figure 28: Guard g has been placed on the reflex vertex r. Hence, its movement is restricted to the reflex area created by the two line segments rr' and rr''. So, if g needs to move outside the reflex area, its new position will be projected on the closest reflex line.

3.7 Angle Behind Reflex Vertex

In this section we will introduce a heuristic that will further fine-tune the factor with which a guard's movement is influenced by a reflex vertex. Currently, we only take into account the distance b between the reflex vertex and the polygon boundary. Intuitively, the unseen area behind the reflex vertex should also play a role in the computation of the gradient: guards should be drawn faster to larger areas. In order to do so, we will take into account the normalised value of angle θ behind the reflex vertex.

A visualisation for this heuristic can be found in Figure 29. The pull of the guard g towards the reflex vertex r will be influenced by the normalised value of θ as follows:

$$g' = g + (\frac{\theta}{2\pi} + c)(\triangledown f(g) + \beta h(g)).$$

We will additionally add a constant c to account for very small angles. For example, if the normalised value of θ is close to 10^{-5}, then $c = 10^{-2}$. Then the guard still has a significant move towards the unseen area, no matter how small it is. In this way, smaller areas are not overlooked.

3.8 Hidden Movement

In this section we will introduce a computation speed-up for the case in which guards have a movement vector of 0 (they don't move). This can happen when the area seen by a guard is already completely seen by other guards. We consider that having a movement vector of 0 is detrimental to the progress of the algorithm. Note that we call *movement vector* the sum of all heuristics that apply to computing the new position of a guard. The reason behind this is that it is unlikely that a guard's optimal position has been found when its movement vector is 0. Hence, we would like every guard to move, no matter how little.

In order to allow guards to still move when their movement vector is 0, we deployed the *hidden gradient* heuristic. This heuristic is based on the fact that we allow guards whose movement vector

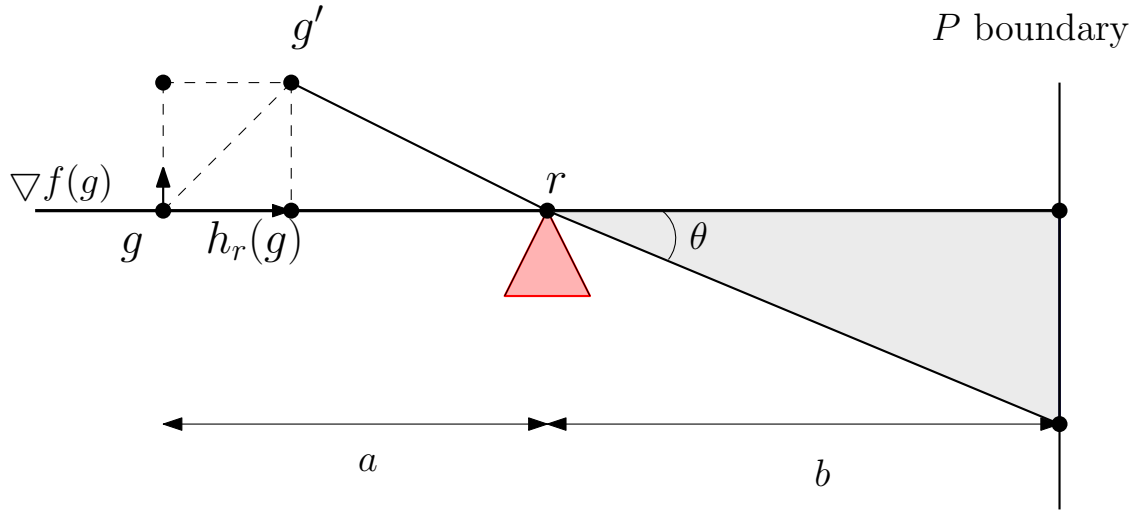


Figure 29: The normalised angle μ behind the reflex vertex r takes into account the factor with which the guard g is drawn to r.

is 0 to still move with a newly computed "hidden" movement vector. So, if there are guards whose movement vector is 0, we will recompute their movement vector by not taking into account the area seen by the guards who have a non-zero movement vector.

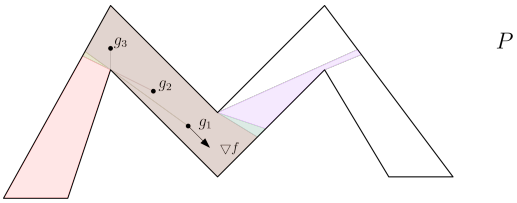
An example of this approach can be found in Figure 30. Let $G = \{g_1, g_2, g_3\}$ be the complete set of guards. Initially in Subfigure 30a, only g_1 has a non-zero movement vector. The visibility regions of g_2 and g_3 are overlapping with that of g_1, so their movement vectors are 0. Let $G_0 = \{g_1\}$ be the set of guards who at step 0 have a non-zero movement vector. In this case, only g_1 Then, Subfigure 30b will display the remaining set $G \setminus G_0$ of guards with a zeroed movement vector. The visibility area of guard g_1 overlaps with g_2, so only guard g_2 will have a non-zero movement vector. So, g_2 will be part of the set $G_1 = \{g_2\}$ of guards who at step 1 have a non-zero movement vector. Lastly, we can compute the non-zero movement vector for guard g_3 in Subfigure 30c. The guard g_3 can now be part of the set $G_2 = \{g_3\}$ which contains the guards who at step 2 have a non-zero movement vector. In Subfigure 30d all the guards have been moved to their new positions g_1', g_2', g_3', respectively. So the movement vectors have been computed as $G = G_0 \cup G_1 \cup G_2$, where G_1 and G_2 contain the guards with hidden movements.

The hidden movement heuristic can now be generalised. Let G be the complete set of guards. For each step i, let $G_i = G \setminus G_{i-1}$ be the set of guards with a non-zero movement vector after removing the guards with a movement vector from step $i-1$. The set G_0 will be the set of guards with a non-zero movement vector before removal of any guards. At the end, the union of all subsets G_i will comprise the set of all guards $G = G_0 \cup G_1 \cup \ldots$. In this way, all guards will have a non-zero movement vector and make progress.

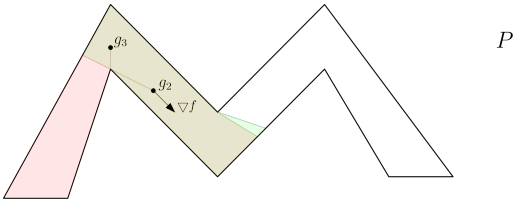
3.9 Greedy Initialisation

In this section we will introduce another heuristic for our algorithm: *greedy initialisation*. This heuristic sequentially places guards at starting positions in areas that are unseen by other guards. In this way, the algorithm will have a head start with a larger covered area than when guards are placed arbitrarily.

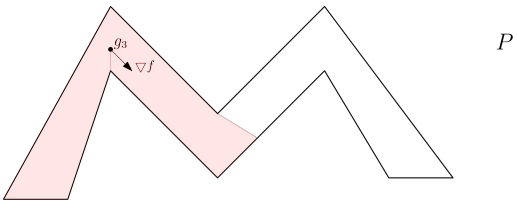
Figure 31 offers an example of a greedy initialisation. The first guard g_1 is arbitrarily placed



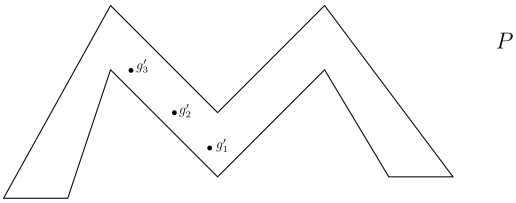
(a) Guard g_2 and g_3 have a movement vector of 0, because guard g_1 sees all the areas they see. So, the set of guards with a non-zero movement vector is $G_0 = \{g_1\}$.



(b) Guard g_1 had a non-zero movement vector, so we compute the movement vectors of guards g_2 and g_3 without g_1. Guard g_2 sees everything that g_3 sees, so g_3 will have movement vector 0. So, the set of guards with a non-zero movement vector is $G_1 = \{g_2\}$.



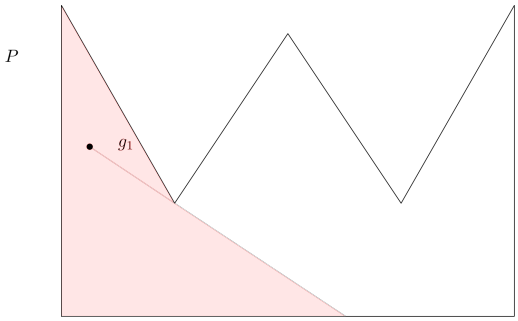
(c) Guard g_2 had a non-zero movement vector, so we compute the movement vector of guard g_3 without g_2. Guard g_3 will now have a non-zero movement vector. So, the set of guards with a non-zero movement vector is $G_2 = \{g_3\}$.



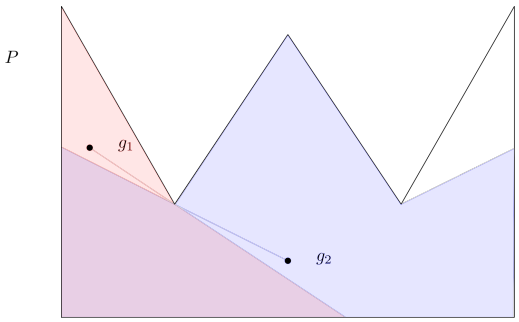
(d) The new positions g'_1, g'_2, g'_3 of the guards such that $G = G_0 \cup G_1 \cup G_2$.

Figure 30: Example of hidden movement computation for guard set $G = \{g_1, g_2, g_3\}$ in a corridor-like polygon. The visibility areas of each of the guards g_1, g_2, and g_3 are shown in purple, green and orange, respectively.

inside polygon P as shown in Subfigure 31a. Then, guard g_2 is arbitrarily placed in an unseen part of P, as displayed in Subfigure 31b. In this way, the algorithm gains a head start to continue with the optimisation of the guards' positions.



(a) Guard g_1 has been arbitrarily placed inside the polygon P.



(b) Guard g_2 has been arbitrarily placed inside the polygon P, outside the visibility region of guard g_1.

Figure 31: Greedy Initialisation for guards g_1 and g_2 inside polygon P. The visibility regions of the guards are displayed in orange and purple, respectively. In this way, the algorithm gains a head start for optimising the positions of the guards.

4 Practice

This section will provide the implementation details of the theory part (Section 3). Then, we will introduce our experimental setup. The experiments include basic test polygons to showcase the execution of the program. We will also observe the importance of each of the heuristics used. We analyse then how the program scales in the context of the comb polygon. Lastly, we will mention the importance of the hyperparameter values used.

The algorithm is implemented in C ++ and makes use of the CGAL library (https://www.cgal.org). The project can be publicly found at https://github.com/geo-j/master-thesis. It contains 22 code files and 1873 lines of code (excluding comments and empty lines). From the CGAL library we are using the Triangular Expansion Visibility algorithm [5] to compute the visibility polygons.

The program takes the segments of a boundary of a simple polygon, and the initial coordinates of the guards as input. For each iteration, the new position of the guards, the area seen by each guard and their movement direction details (momentum, pull, events i.e. pulling on top of reflex vertices) are logged in a file. Using the logfile, the program can create a visualisation for the position of each guard, their visibility polygon area and their movement direction vector for each iteration. The program can also plot the area seen both by each guard and in total. The visualisations are created using the Python 3 libraries matplotlib (https://matplotlib.org/) and the scikit-geometry (https://github.com/scikit-geometry/scikit-geometry).

4.1 Heuristics

In this section we will observe the role played by each of the heuristics used. We will additionally notice how different heuristics are relevant for different types of polygons. In order to do so, we will run the program with all the heuristics but one, for each of the heuristics. By analysing the difference in movement for each of the guards, we will be able to assess the influence every heuristic has on different types of polygons.

We will use fixed hyperparameters for all the runs. This will allow us to focus on the differences between the heuristics. As such, we will use the momentum hyperparameter $\gamma = 0.8$ and pull attraction $\beta = 1$. The hyperparameter values were chosen through experimentation. The algorithm is sensitive to the hyperparameter choices, so for other values it often becomes stuck in local optima.

4.1.1 Without Momentum

In this section we will discuss the impact momentum has on the overall behaviour of the algorithm. As introduced in Section 3.3, momentum takes into account the position history of the guards. In this way, the overall trajectory of a guard is smoothened out.

A suggestive way to observe this is with the arbitrary polygon from Figure 32. The polygon requires a minimum of 3 guards to be fully seen. We will compare how the guards move when we are using all the heuristics to when we are not using momentum. They will start at the same fixed position in both cases.

Figure 33 displays the area seen per iteration for the arbitrary polygon. Both the total and the individual areas seen by each guard are shown. Starting with almost the whole polygon seen, the guards are eventually optimally placed. Nonetheless, using momentum clearly makes a difference in Subfigure 33a, than when not using it in Subfigure 33b. Momentum allows the overall seen area

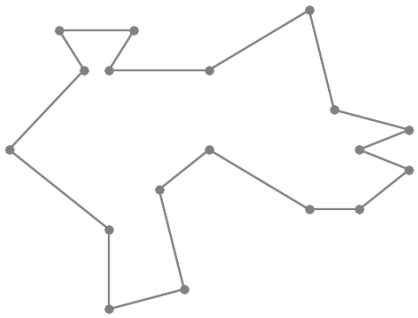
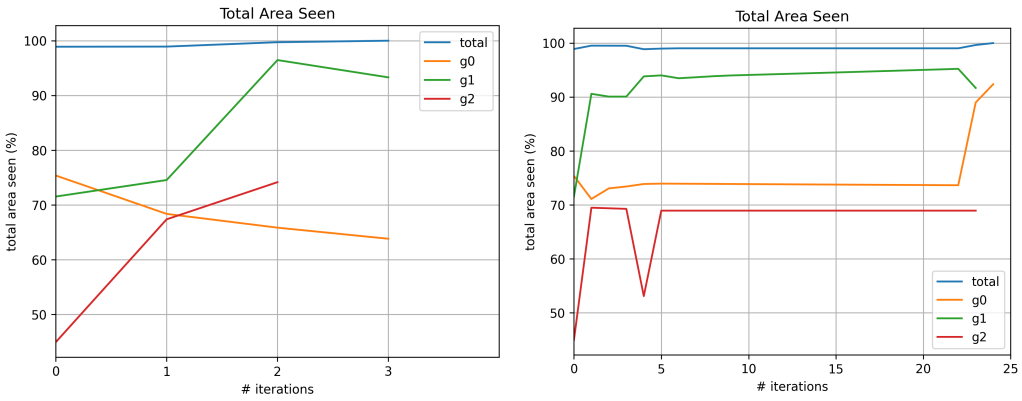


Figure 32: Arbitrarily shaped polygon.

to keep a steady trajectory towards its maximum. Additionally, guards quickly find their optimum in only 4 iterations, without many oscillations. In Subfigure 33b however we can observe how the total area fluctuates. The guards display large jumps close to iterations 5 and 20. These jumps cause the overall progress towards the optimum to be less stable. For example, when guard 2 ($g2$) has a sudden drop in the area it sees around iteration 5, the total area seen naturally drops as well. The algorithm only recovers after iteration 20, when guard 0 ($g0$) makes another large jump. This behaviour also emphasises how the movement of one guard heavily influences the other guards' trajectories. As a result, the guards' trajectories to optimality become noisier and slower (more iterations are needed).



(a) All heuristics.

(b) No momentum.

Figure 33: Total area seen per iteration for an arbitrary polygon guarded by 3 guards.

Thus, it becomes clearer how momentum allows the smoothening of noisy guard movements. We reckon that because guards are holding a steadier trajectory, they are more likely to achieve the optimum in less iterations. When not using momentum, the number of iterations increases substantially. Momentum thus is a crucial improving heuristic to our whole algorithm, both in terms of speed-up and in the smoothness of the process.

4.1.2 Without Line Search

In this section we will discuss the impact line search has on the overall behaviour of the algorithm. As introduced in Section 3.4, line search determines how far a guard should move towards the direction movement vector. In this way, it computes the optimal position of a guard on the movement direction given a step size.

For our experiments, we will start with a factor of $\frac{1}{x}$ for the movement vector. We will increase it by a step size of s up to factor x. We will choose step size $s = 2$ and maximum movement factor $x = 32$. In this way we are able to find the best new guard position between 10 options at every iteration. This will allow us to search a larger space of position possibilities knowing the direction of the movement vector. We will pick the best solution along all the step size based on the largest area increase.

A suggestive way to observe how well line search works is with the comb polygon with four teeth from Figure 34. Comb polygons with t teeth require t guards to be guarded. In our case, $t = 4$. We will compare how the guards move when we are using all the heuristics to when we are not using line search. They will start at the same fixed position in both cases, so we can focus on observing their movements.

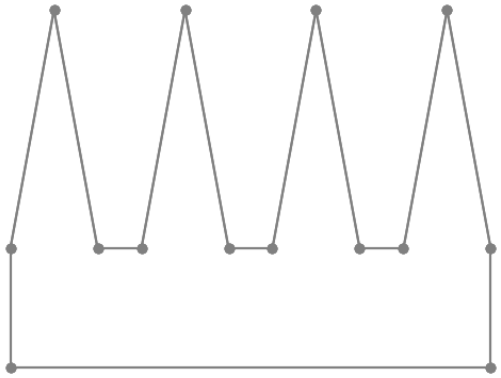
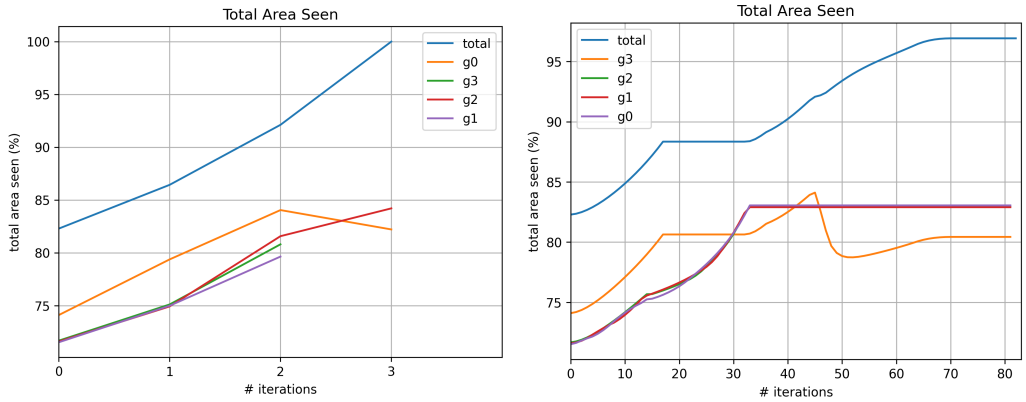


Figure 34: Polygon in the shape of a comb with four teeth.

Figure 35 displays the area seen per iteration for the comb polygon with four teeth. Both the total area seen and the individual area seen by each guard are shown. Starting with around 82.5% total area seen, the global optimum is eventually found. Nonetheless, using line search clearly makes a difference between Subfigures 35a and 35b. The first noticeable difference is the number of iterations. Using line search allows the guards to find their optimal positions in 3 iterations, with a steady increase in the total area seen. On the other hand, not using line search results in the optimal position to be found in more than 80 iterations. What is more, 3 of the guards found their optimal position after the 30^{th} iteration, whereas the last 50 iterations are spent on only one guard finding its own.

Therefore, we reckon that line search significantly and more efficiently speeds up the process of finding the optimal position for each guard. In this way, each guard moves faster to its optimal position. The situation where multiple guards that have found their optimal position have to wait for only one guard to find its own is also avoided.



(a) All heuristics.

(b) No line search.

Figure 35: Total area seen per iteration for the comb polygon with four teeth, guarded by four guards.

4.1.3 Without Pulling Onto Reflex Vertex

In this section we will discuss the impact the pull onto the reflex vertex heuristic has on the behaviour of the algorithm. Section 3.5.1 introduced the pull onto reflex vertices heuristic. This heuristic tackles the idea that if a guard is "close enough" to a reflex vertex, then the locally maximum seen area would be achieved by placing the guard on top of the reflex vertex. We have defined "close enough" as a guard being closer than two thirds of the minimum distance between any two reflex vertices in the polygon.

We will compare how the guards move when we are using all the heuristics to when we are not pulling them onto reflex vertices. The guards will start at the same fixed position in both cases.

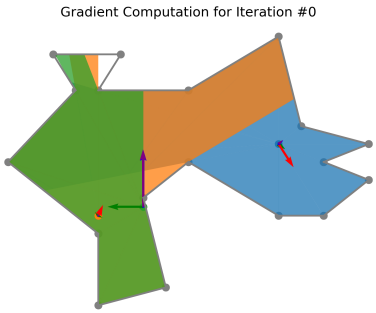
Figure 36 displays the two reflex vertex pull cases: Subfigures 36a and 36b show how the green guard's movement changes when it is placed onto the reflex vertex; Subfigures 36c and 36d show how the guard's movement changes when it is pulled towards the reflex vertex, but not onto it. We can observe how in Subfigure 36b the green guard is placed onto the reflex vertex. It then strives to reach the unseen polygon part in the upper left corner. On the other hand, in Subfigure 36d the guard moves away from the reflex vertex.

These events can also be observed in the two seen area plots in Figure 37. The case when the green guard is not placed on top of the reflex vertex results in more iterations (Subfigure 37b). This is due to the fact that the green guard cannot maximise its locally seen area by being placed on the reflex vertex. So, it needs to move around more before finding its optimal path. This is also displayed in how the total area seen lowers and then rises in between iterations 1 and 4. Conversely, Subfigure 37a displays how the green guard quickly moves towards the last part of the polygon that was not yet seen. This results in a steady increase in the total seen area.

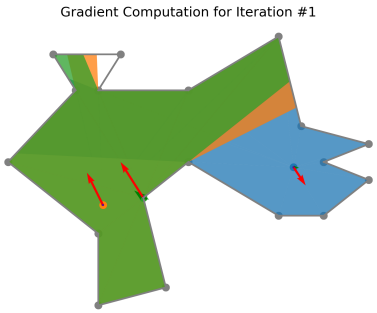
Therefore, we believe that pulling guards on top of reflex vertices when they are "close enough" results in maximising the local area seen by those guards. The number of iterations needed to reach the global maximum is thus reduced. So, the efficiency of the algorithm is also improved.

4.1.4 Without Pull Capping

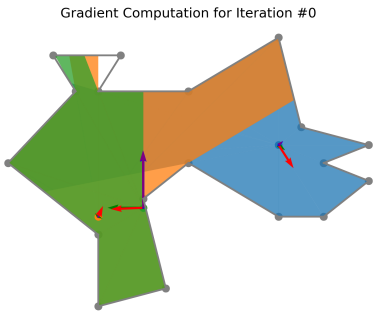
In this section we will discuss how not capping the pull towards a reflex vertex influences the progress of the algorithm. Section 3.5.2 introduced the notion of capping the pull towards a



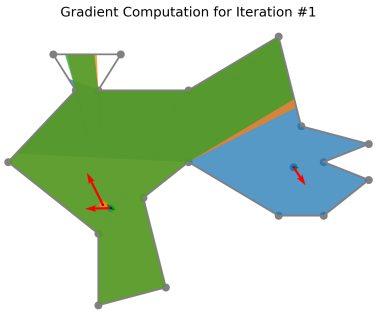
(a) All heuristics. The green guard is pulled towards the reflex vertex.



(b) All heuristics. The green guard is onto the reflex vertex.

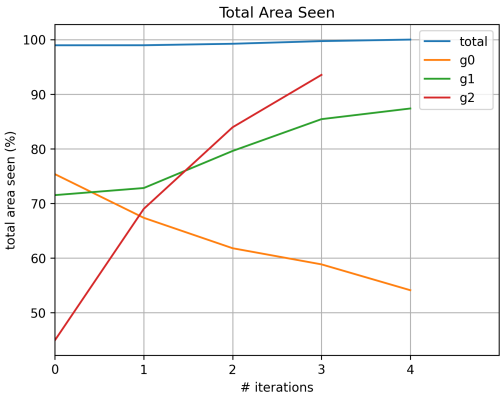


(c) No pull onto the reflex vertex. The green guard is pulled towards the reflex vertex.

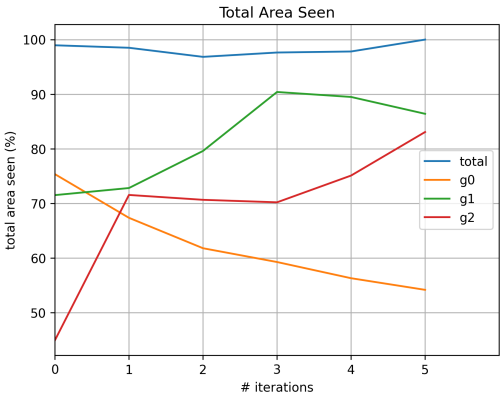


(d) No pull onto the reflex vertex. The green guard is placed according to the momentum computation.

Figure 36: Example of different movements of guards with and without pulling them onto reflex vertices in an arbitrarily shaped polygon guarded by three guards.



(a) Seen area for all heuristics.



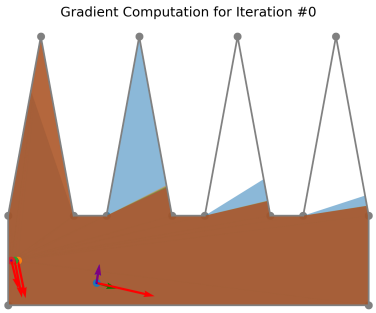
(b) Seen area without pull onto reflex vertices.

Figure 37: Total and individual areas seen per iteration for an arbitrarily shaped polygon guarded by three guards.

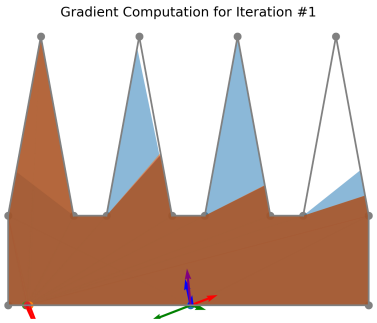
reflex vertex. The reason for this choice was to not allow the pull to overpower the value of the momentum. So, if the pull is larger than a factor μ than the momentum, it is capped at the momentum value of a factor of μ.

We will compare how the guards move when we are using all the heuristics to when we are not capping the pull towards reflex vertices. We will use the comb polygon with four teeth as example. The guards have a pull cap hyperparameter $\mu = 1$. Thus, if the pull is larger than the momentum, we cap it at the size of the momentum. This value was chosen experimentally in order to clearly show the benefits of using this heuristic. So, if the pull is as large as the momentum, we prioritise pulling guards onto reflex vertices if they are "close enough".

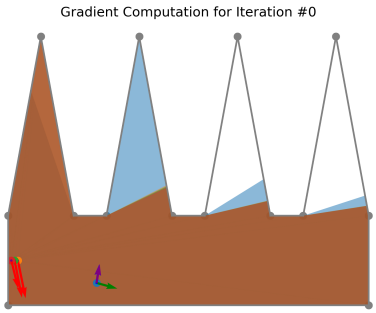
Figure 38 displays the two reflex vertex pull cases: Subfigures 38a and 38b show how the blue guard movement changes when its pull is capped; Subfigures 38c and 38d show the opposite. We can observe how in Subfigure 38a the blue guard has its pull towards the reflex vertex capped. In that case, the movement vector is larger than the pull, so the guard doesn't move towards the reflex vertex. So, the blue guard moves as its movement vector dictates to the position in Subfigure 38b. This encourages the guard to explore the polygon more rather than directly moving towards a reflex vertex. Conversely, Subfigure 38d displays the blue guard moving closer to the reflex vertex it is pulled to. In subsequent iterations, it is placed on the reflex vertex.



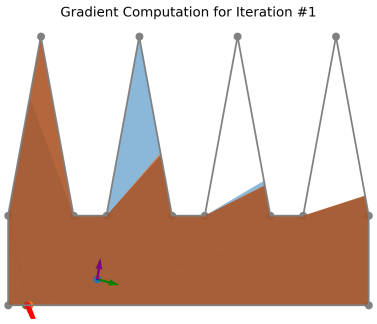
(a) All heuristics. The guard's pull towards the reflex vertex is capped.



(b) All heuristics. The guard does not act upon the pull towards the reflex vertex.



(c) No pull onto the reflex vertex. The guard's pull towards the reflex vertex is not capped.

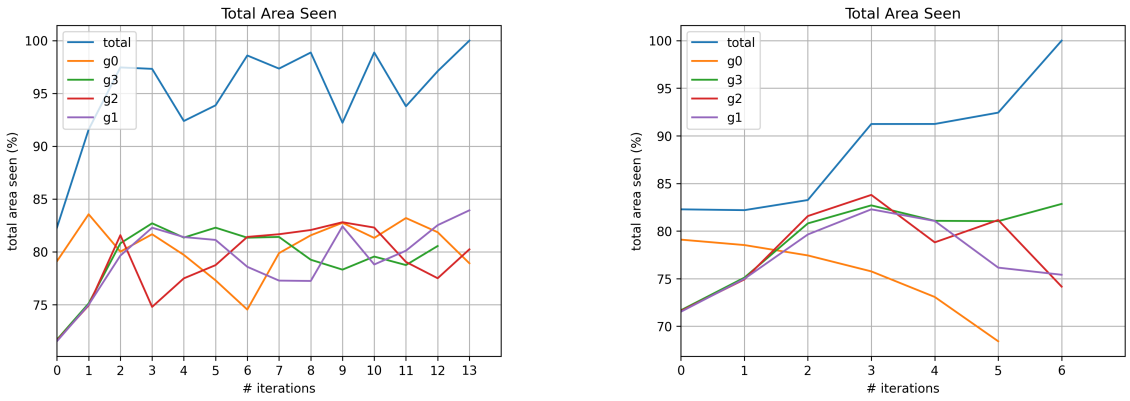


(d) No pull onto the reflex vertex. The guard is drawn closer to the reflex vertex.

Figure 38: Example of different movements of guards with and without pull capping in a comb polygon with four teeth.

Figure 39 shows how the global behaviour of the algorithm is influenced by the capping. Interestingly enough, when capping the pull, the guards' behaviour is more erratic. This results in more iterations before the whole polygon is seen (Subfigure 39a). On the other hand, using no capping allows a steadier increase in the total area seen (Subfigure 39b). Eventually, the blue guard ($g0$)

is drawn and placed onto the reflex vertex in iteration 2. The reflex vertex placement does not improve the locally seen area. However, it reinforces the previously discussed idea that placing guards on reflex vertices is globally beneficial for the algorithm. As such, tuning the hyperparameter μ is a crucial point of exploration for deciding how and when pull capping would always prove most favourable.



(a) Seen area for all heuristics.

(b) Seen area without pull capping.

Figure 39: Total and individual areas seen per iteration for a comb polygon with four teeth.

We believe that capping the guards' pull towards a reflex vertex highly depends on the hyperparameter μ. When not capped, it emphasises the importance of placing guards on reflex vertices. However, when the pull of guards is capped, we are using a more exploratory approach to discovering the polygon. Thus, the choice of using this heuristic can be made through experimentation. The shape of the polygon can influence this decision as well.

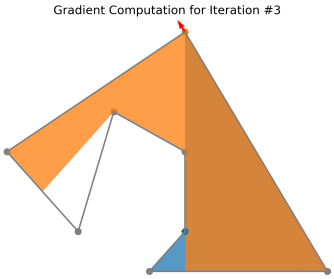
4.1.5 Without Reflex Area

In this section we will discuss the importance of keeping guards in the reflex area of a reflex vertex they had been placed on. Section 3.6 introduced this heuristic. The idea counteracts the edge-case of guards moving away from reflex vertices they had been placed on. When they move away from the reflex vertex, they might unsee a large previously seen area. It is possible that afterwards their movement vector directs them to move back onto the reflex vertex. We want to prevent them from unnecessarily moving away and then back on the vertex in a loop. For this reason, we will keep them in the reflex area. The reflex area is then the area determined by the two segment lines meeting in the reflex vertex. By staying in this area, the guard would be able to move away from the reflex vertex in a direction that does not undo its progress.

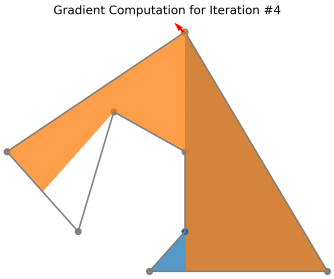
We will compare how the guards move when we are using all the heuristics to when we are not keeping guards into a reflex area. We will use another arbitrary polygon as an example, that needs two guards to be fully seen.

Figures 40 and 41 compare the algorithm progress when using and not using the reflex area heuristic, respectively. Subfigures 40a - 40d show the guards' movement when one of them is bound to the reflex area. In this case, the blue guard wants to move away from the reflex vertex. The reflex area does not allow it to move away from the reflex vertex, so it stays there. Subfigures 40e - 40h display the guards' movement when the reflex area heuristic is not applied. In this case, the blue guard moves away from the reflex vertex (Subfigure 40f), only to be drawn back to it in the

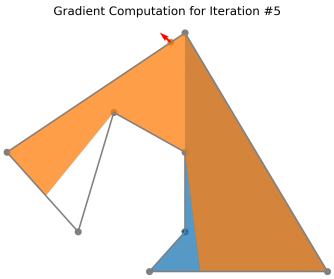
next iteration (Subfigure 40g). This repetitive behaviour continues without eventually reaching the optimum state.



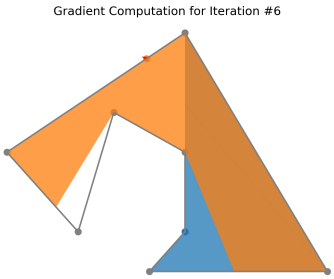
(a) All heuristics. The blue guard is on the reflex vertex.



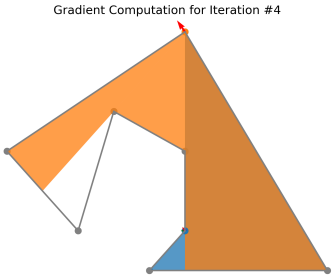
(b) All heuristics. The blue guard is still on the reflex vertex (in the reflex area).



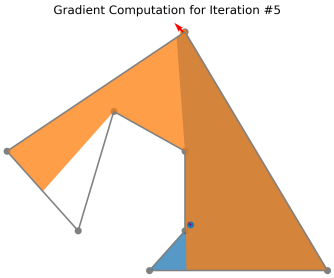
(c) All heuristics. The blue guard is still on the reflex vertex (in the reflex area).



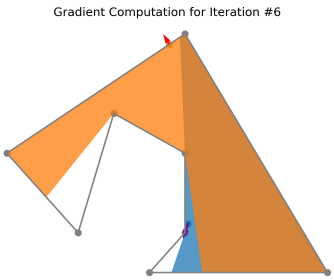
(d) All heuristics. The blue guard is still on the reflex vertex (in the reflex area).



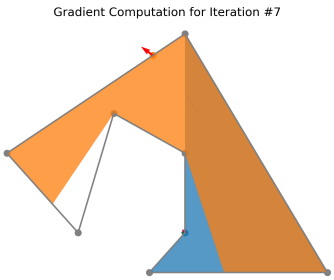
(e) No reflex area. The blue guard is on the reflex vertex.



(f) No reflex area. The blue guard moved away from the reflex vertex.



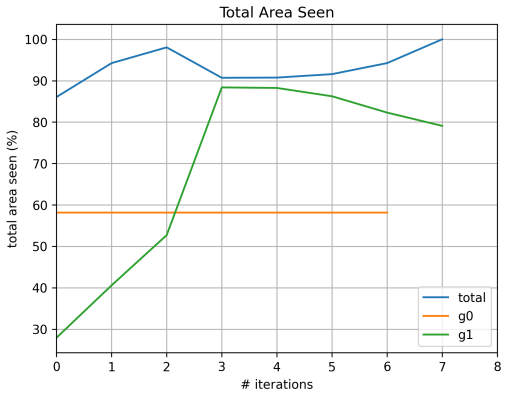
(g) No reflex area. The blue guard is drawn back to the same reflex vertex.



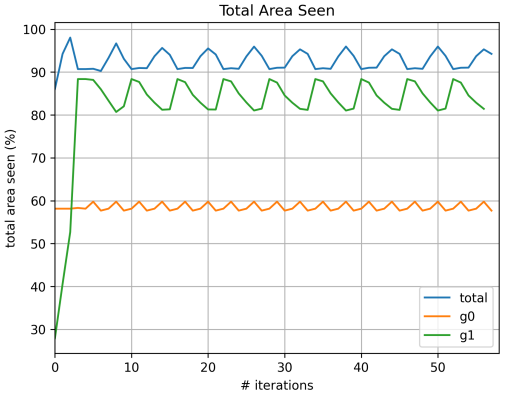
(h) No reflex area. The blue guard is back on the reflex vertex.

Figure 40: Comparison between using and not using the reflex area heuristic on an arbitrary polygon guarded by two guards.

Figure 40 displays the difference in the areas seen between the two approaches. When using all heuristics (Subfigure 41a), the algorithm terminates in 7 iterations. The continuous line clearly marks the guard who did not move away from the reflex vertex. When not using the reflex area heuristic (Subfigure 41b), the algorithm appears not to terminate. One of the guards loops away and back on the reflex vertex. The other guard loops so that it sees the area that is being seen and unseen by the other guard. In this way, they do not synchronise in order to see the polygon together at the same time. This is clearly shown in the repetitive plot.



(a) Area seen for all heuristics on another arbitrary polygon.



(b) Area seen for no reflex area heuristic on another arbitrary polygon.

Figure 41: Area comparison between using and not using the reflex area heuristic on an arbitrary polygon guarded by two guards.

Therefore, we believe that the reflex area heuristic is of great importance to the correct run of the algorithm. It addresses the issue of guards moving away and back on reflex vertices in a repetitive manner. This behaviour causes the other guards to move towards the area that becomes unseen by the first guard. The guards continue to move out of sync with each other, resulting in the polygon never being fully seen.

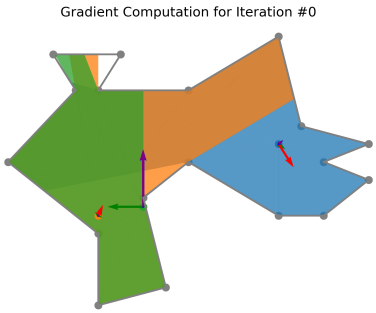
4.1.6 Without Angle Behind Reflex Vertex

In this section we will discuss the benefits of using the angle behind reflex vertices. Section 3.7 introduced this heuristic. The idea behind this technique is that guards should be drawn faster to larger unseen areas behind reflex vertices. In this way, we prioritise unseen areas based on their size, while still accounting for the very small unseen areas.

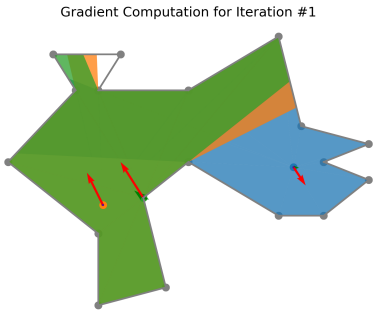
We will compare how the guards move when we are using all the heuristics to when we are not taking into account the angle behind the reflex vertex. We will use an arbitrary polygon as an example, that needs three guards to be fully seen.

Figure 42 shows a comparison between using and not using the angle behind the reflex vertex heuristic. Subfigures 42a and 42b display the first two iterations of the algorithm when all heuristics are used. Subfigures 42c and 42d focus on the first two iterations when not using the heuristic. We can observe a major difference in the way the final movement vector is computed for the orange guard in the two cases. When all heuristics are used, the movements of the guards are much smaller and smoother. For example, in Subfigure 42c, the orange guard has a large movement vector outside the polygon due to the unseen part in the upper pocket. When we also take into account the angles behind the reflex vertices in the pocket, its movement vectors become much

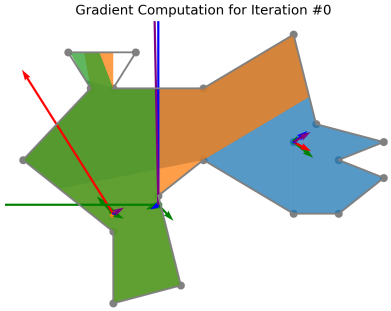
smaller (Subfigure 42a). In fact, the orange guard is also drawn slightly to the right part of the polygon, as it has a larger angle behind the reflex vertex. That part of the polygon is already seen by the blue guard, so it starts moving towards the upper pocket in Subfigure 42b.



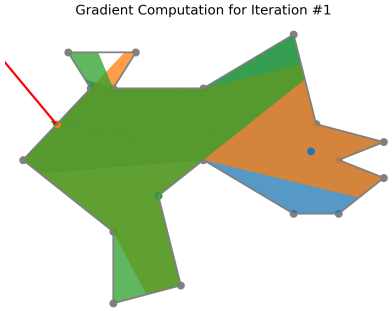
(a) All heuristics. The orange guard's movement is towards upper right.



(b) All heuristics. The orange guard's movement is towards upper left.



(c) No angle behind the reflex vertex. The orange guard's movement is all the way to the left boundary of the polygon.

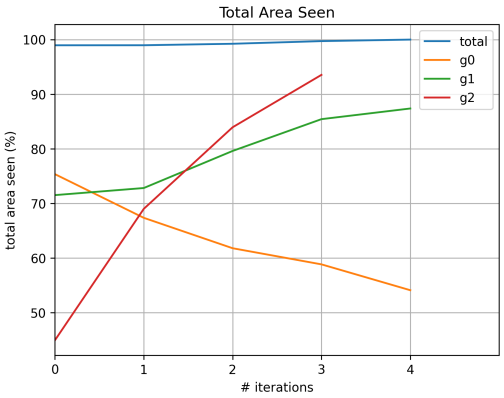


(d) No angle behind the reflex vertex. The orange guard's movement is towards the left-side outside the polygon.

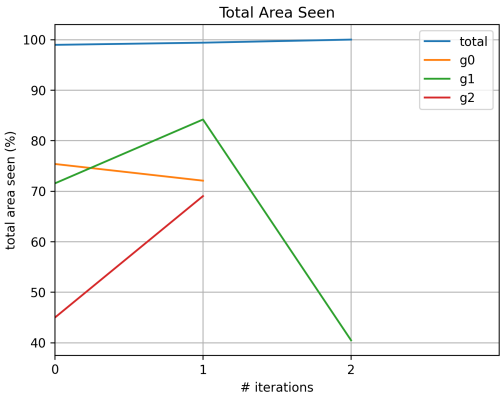
Figure 42: Example of different movements of guards with and without the angle behind the reflex vertex in an arbitrarily shaped polygon guarded by three guards.

In terms of efficiency, the optimal solution is achieved in more iterations when the angle heuristic is used. This can be observed in Figure 43. When using all heuristics, the optimal solution is achieved in 5 iterations (Subfigure 43a). The movement of the guards is smooth. Two of the guards have an increasing seen area, whereas the orange guard moves slowly towards the upper pocket of the polygon. Not using the angle heuristic allows us to achieve the same goal in only 3 iterations, in a less smooth manner (Subfigure 43b).

Therefore, we believe that computing the angle behind the reflex vertex is a heuristic that allows us to fine-tune and smoothen the movement of the guards. In this way, we can focus on moving fast towards the bigger unseen areas, while not neglecting the smaller ones. We could say that this heuristic offers a trade-off between the number of iterations and path smoothness. For this reason, the performance of the algorithm is influenced by the type of polygons it is applied to: polygons with sharper, narrower turns could benefit the most from this heuristic.



(a) Seen area for all heuristics.



(b) Seen area without the angle behind the reflex vertices.

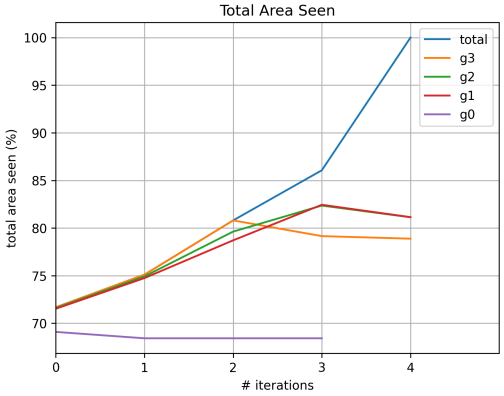
Figure 43: Total and individual areas seen per iteration for an arbitrarily shaped polygon guarded by three guards.

4.1.7 No Hidden Movement

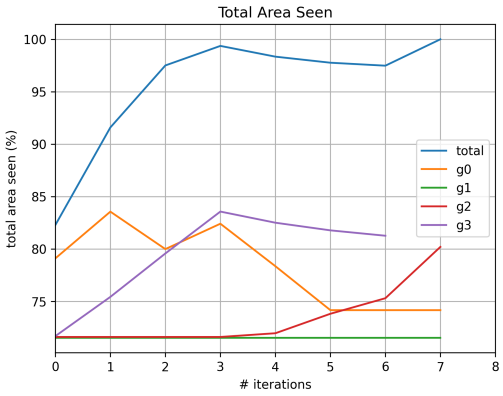
In this section we will discuss the importance of using the hidden movement heuristic. Section 3.8 introduced it. The idea is based on the fact that we want guards to make progress, no matter how little. If a guard's visibility region is already seen by other guards (so its movement vector is zero), it is still unlikely that its position is optimal. Thus, we want the guard to still progress.

We will compare how the guards move when we are using all the heuristics to when we are not using the hidden movement heuristic. We will use the comb polygon with four teeth as an example.

Figure 44 displays a comparison between using and not using the hidden movement in our algorithm. In Subfigure 44b we can notice that the execution of the algorithm takes twice as many iterations than in Subfigure 44a. The reason behind this is the fact that for half of the iterations two of the guards do not move when no hidden gradient is used. Additionally, the total area seen in that case fluctuates. This is in contrast with the case where the hidden gradient heuristic is used. In that case, the total area seen is continuously increasing, and all guards who can make progress move.



(a) All heuristics.



(b) No hidden gradient.

Figure 44: Seen area for the comb polygon with four teeth guarded by four guards.

Therefore, we believe that the hidden gradient heuristic results in a significant improvement to our algorithm in terms of efficiency. In this way, all guards are ensured to make some progress towards their optimum and are not stalled.

4.1.8 Greedy Initialisation

In this section we will discuss the benefits of using a greedy initialisation technique for our algorithm. Section 3.9 introduces it as beneficial for giving a head start to our algorithm. In our experiments we will greedily initialise the guards in the middle of the leftmost segment of each of their visibility regions. The reason behind this choice is that CGAL did not offer a quick way to pick a randomised position inside the visibility area. Due to the time constraints of this thesis, we decided to use a deterministic positioning.

So far we have not used greedy initialisation in any of our examples. The reason is that this technique can already solve some polygons at guard placement time. Naturally, this would not allow us to show the benefits of using any of the other heuristics. An example in this case is the comb polygon with four teeth in Figure 45.

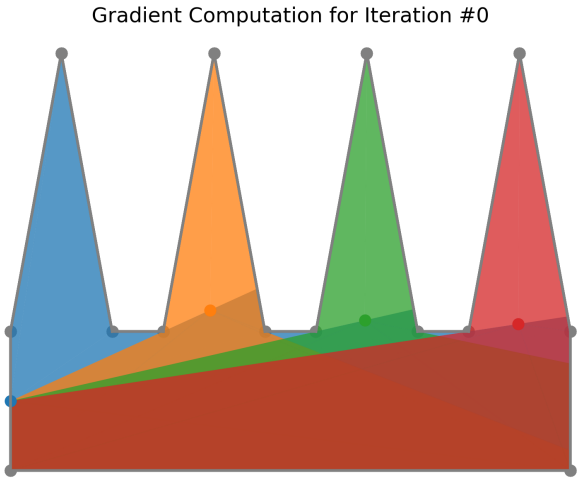


Figure 45: The comb polygon with four teeth is already completely seen at greedy initialisation time.

In the case of polygons who are not completely seen at initialisation time, it is unclear whether the greedy placement improves the overall progress. How well the algorithm behaves is highly dependent on the initial placement of the guards. Namely, the algorithm often does not finish with different starting positions (it is stuck in local optima). Because of this drawback, we cannot create extensive experiments with different starting guard positions.

Therefore, we believe that the greedy initialisation technique would be most beneficial with a deterministic placement tailored to the shape of the polygon. An arbitrary placement would be most convenient for generalising the performance of the technique. However, due to time constraints, this heuristic is to be explored with a more robust and arbitrary implementation.

4.2 Scaling for the Comb Polygon

In this section we will observe how the algorithm scales on the comb polygon. We will take the comb polygons with 2, 3, ..., 10, 15, 20, 50, 100 teeth. We will note how the algorithm progresses in terms of the total area seen per iteration. As such, we will run the algorithm with all heuristics for all the previously mentioned comb polygons. All guards will start at the same y-coordinate, 0.1 units apart from each other. An example of such a starting point can be found in Figure 46. In this way, we can ensure that the performance of the algorithm can be compared using the same starting position of the guards. We will deem a timeout of one hour for declaring an algorithm unfeasible for larger polygons. We will then analyse how the algorithm performed under each of the circumstances. Lastly, we will offer some points of improvement.

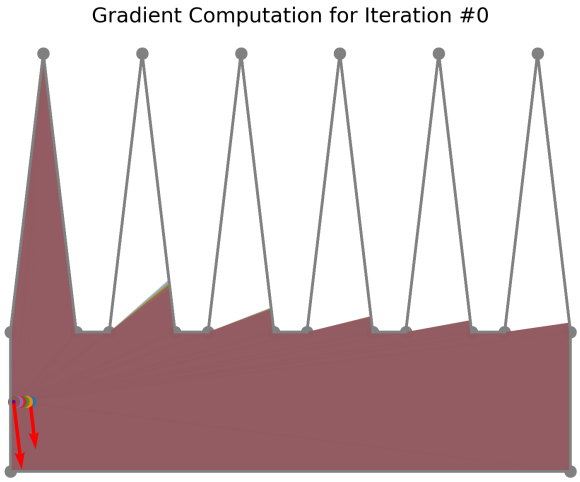


Figure 46: In the comb polygon with 6 teeth, all guards start at the same y-coordinate, with 0.1 units in between them.

We expect that the number of iterations and the execution time needed would scale with the number of teeth of the polygon. We expect that the time limit would already be achieved with less than 20 comb teeth.

Figure 47 display the progression of our algorithm for the comb polygons with 2, 3, ..., 10, 15, 20 teeth within one hour. The comb polygons with 50 and 100 teeth did not complete any iteration within that time. For comb polygons with 5 teeth and more than 6 teeth, the timeout was not enough to find a feasible solution. Nonetheless, it is worth discussing their behaviour, as it highlights the different issues the algorithm faces. For comb polygons with 7, 9 and 10 teeth, the guards appear to be stuck in a local plateau that they did not manage to escape. This can be traced to an edge case where the guards are stuck in their position because their movement vector does not improve the total area. Another reason why the guards do not move is that they are still trying to maximise their local area in the case where the total area seen cannot be increased within that step. This could result in them all moving to the same part of the polygon. Since that part of the polygon maximises their local area, they will not move anymore. For the comb polygon with 5 teeth, the guards appear to be stuck in a local maxima that they manage to escape. Unfortunately, they later return to it and restart the cycle. It is unclear whether the comb polygon with 8 teeth

would display a similar looping behaviour. Given the timeout, the guards trying to solve the comb polygon with 8 teeth display a less predictable behaviour. We believe that hyperparameter tuning could address this issue.

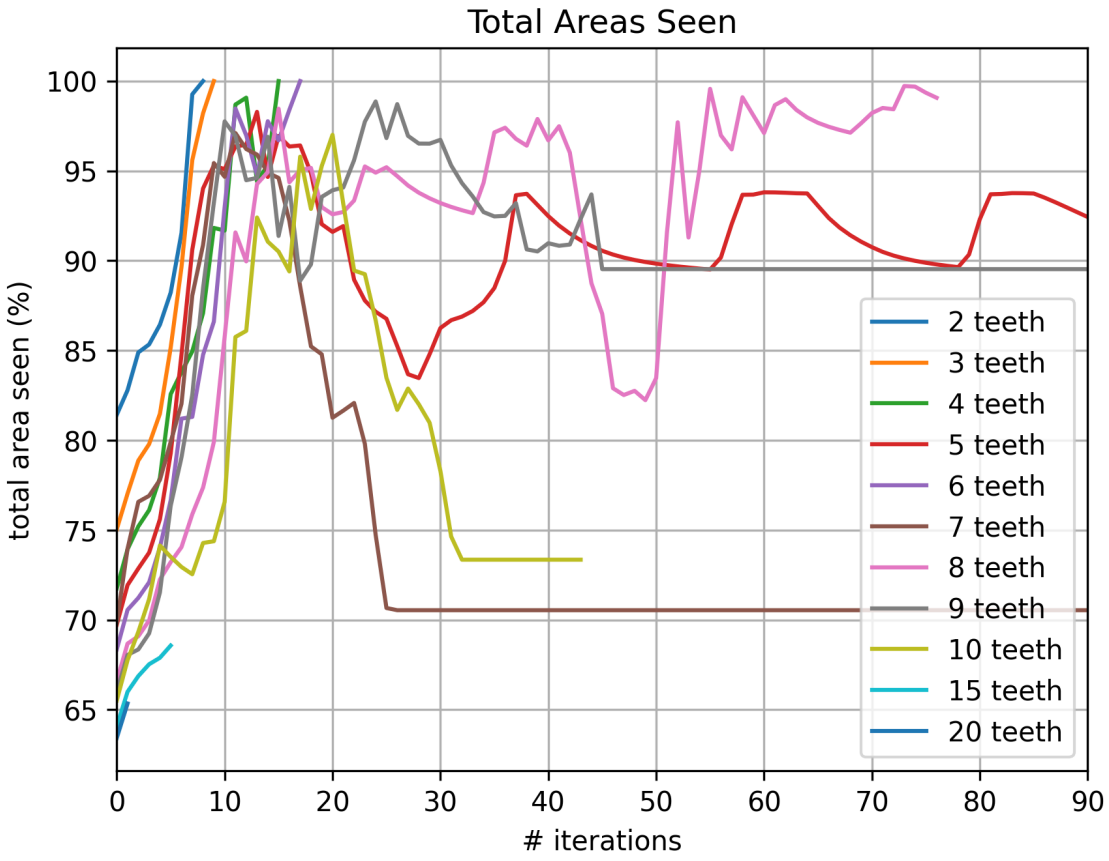


Figure 47: The percentage of the area seen in terms of iterations for comb polygons with 2, 3, ..., 10, 15, and 20 teeth.

Conversely, it is also crucial to address in Figure 48 the comb polygons which the algorithm manage to solve: 2, 3, 4 and 6 teeth. In this case, we can observe a clear scaling: the more teeth a comb polygon has, the longer it takes to be solved. What is more, the solution also scales with the number of teeth. The comb polygon with 4 and 6 teeth take twice as many iterations to be solved than the ones with 2 and 3 teeth, respectively. We would expect a similar behaviour for comb polygons with more teeth, if the algorithms would finish.

It is also worth discussing how the initial guard placement and comb polygon shapes could influence the performance of the algorithm. Firstly, it is important to note the initial placement strategy. Guards are placed in the same lower left part of the polygon. However, the larger a polygon, the lower the initially seen area. For example, we can observe how the comb polygon with 2 teeth starts at more than 80% seen area. On the other hand, the comb polygon with 20 teeth has a less than 65% seen area at start. We could argue that this type of placement gives larger polygons a start disadvantage. A way to mitigate this issue could be to find a way of placement that scales, while still keeping the initial similar positioning among different combs. Moreover, the algorithm is sensitive to the shape of the polygons. The bigger a comb polygon grows, the sharper and narrower its teeth are. A good example in this case is Figure 49, which displays a

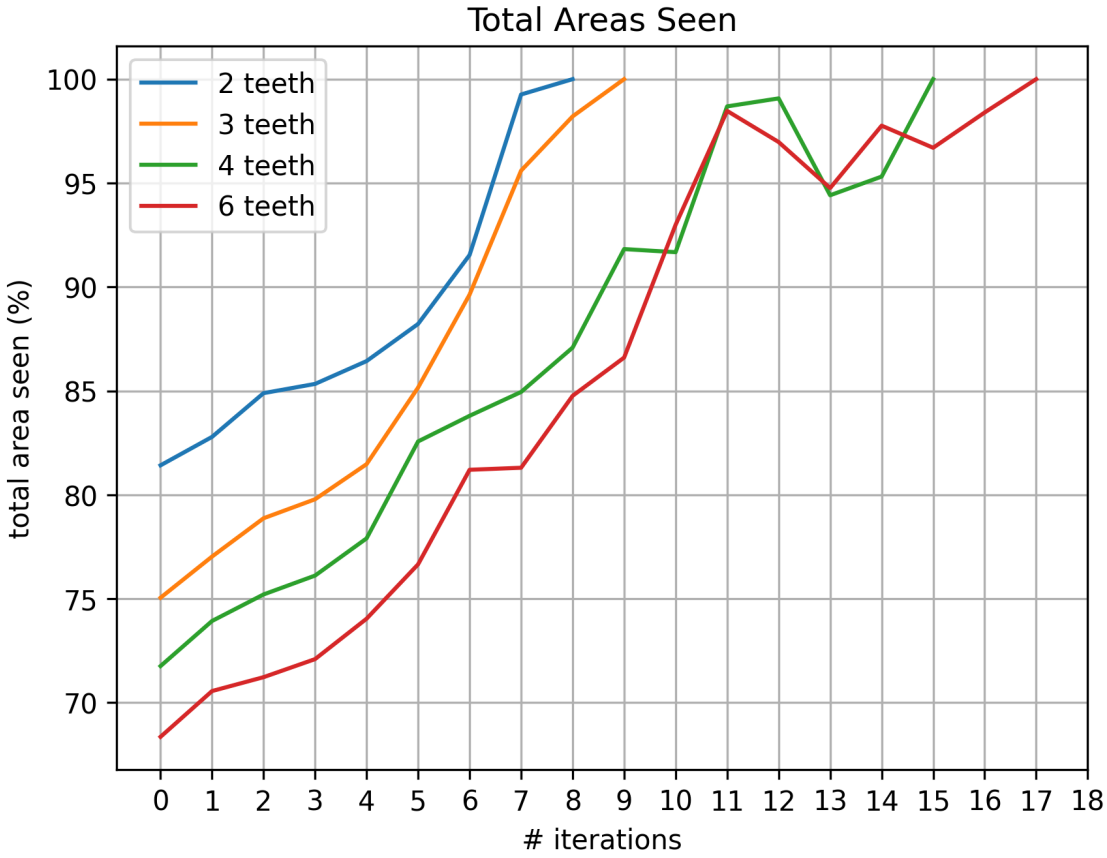


Figure 48: The percentage of the area seen in terms of iterations for comb polygons with 2, 3, 4, and 6 teeth.

comb polygon with 20 teeth. Clearly, the teeth are much narrower than in the case of a comb with 4 teeth. This raises the question about the performance of the algorithm given different angles in the polygon boundary. A possible solution to this aspect would be to scale up the polygon. However, a similar question would remain: does the algorithm necessarily perform better or worse because of the polygon shape? What would be the optimal such shape?

4.3 Hyperparameter Sensitivity

The algorithm is highly sensitive to the hyperparameter choices, polygons shapes and sizes, and guard starting points. We have already emphasised how the hyperparameter values have been chosen: other values than the ones used would result in the program crashing or not terminating. Similarly, in the context of the comb polygon, we have discussed that the polygon shape can have a crucial effect on the performance of the algorithm.

In this section we will additionally explain how the guard starting points influence the outcome of the program. We will use the irrational guards polygon [1] as an example. The polygon can be guarded by 3 guards with irrational coordinates, or 4 guards with rational coordinates. We will thus compare these placements. We will position 3 guards at an approximation of the optimal irrational coordinates, and 4 guards at arbitrary rational coordinates. The approximated irrational guards will have coordinates $(2, 0.59), (19, 1.71), (10.57, 2.12)$. The arbitrary guards will have coordinates

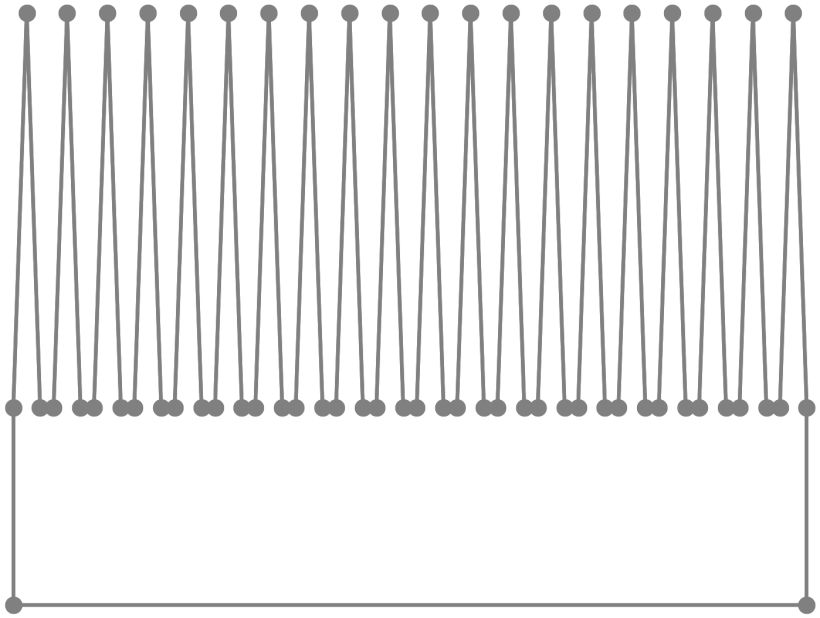


Figure 49: Comb polygon with 20 teeth.

$(5, 0), (5.5, 0), (6, 0), (6.5, 0)$.

Figure 50 displays the difference in runs between the previously mentioned guard placements. The orange line displays the algorithm run for the guard irrational approximation coordinates. As expected, the algorithm deems that the whole polygon is seen within the first iteration. On the other side, the blue line shows the algorithm performance for the guards with rational coordinates. Unfortunately, the program in this case crashes after the 8^{th} iteration. The error was unclear and the time constraints of this thesis did not allow us to solve it.

Therefore, we believe that the irrational guards polygon is a suggestive example to the sensitivity of the algorithm to initial guard placements. The starting position of the guards can determine whether the algorithm is able to escape local minima or not crash. Nonetheless, we are not able to provide a generalisation of this statement. Namely, we were not able to determine how the starting position of the guards influences the progress of the algorithm, while it is also dependent on the polygon shape.

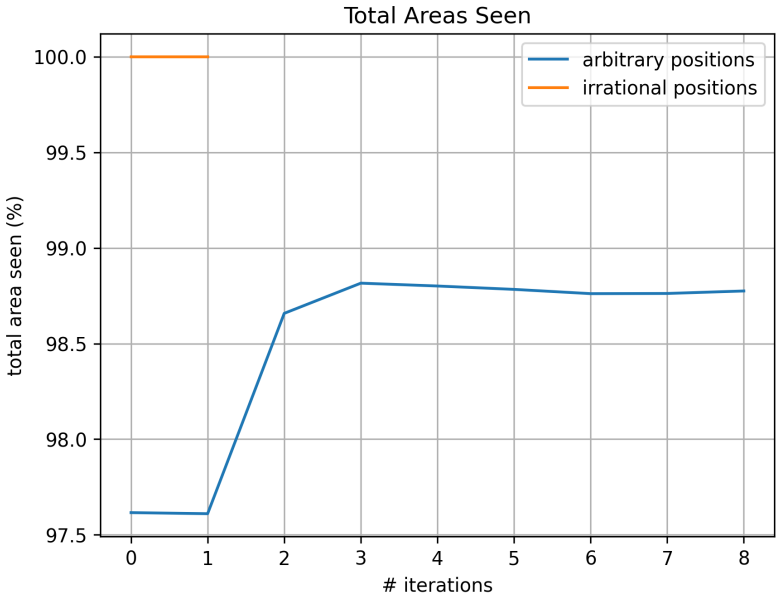


Figure 50: Comparison for the irrational guards polygon between guards with approximately irrational coordinates and rational guard coordinates.

5 Problems Encountered

The development of this algorithm has encountered a number of important problems that affected its progress. This section will emphasise the most influential such issues and, if known, suggest a solution for resolving them.

To begin with, some edge-cases remained unaddressed. For example, Figure 51 shows a frequently appearing such edge-case: the blue guard is stuck on the vertex because its movement direction cannot be projected inside the polygon. So, it cannot move downwards and is instead stuck on the vertex.

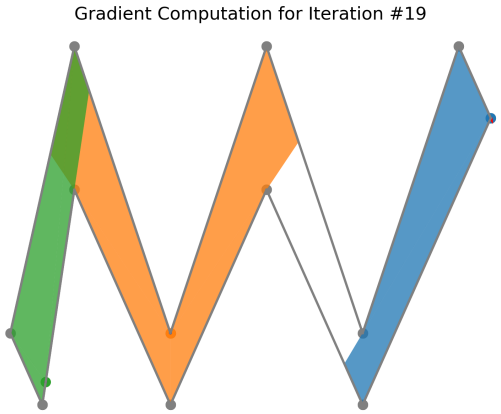


Figure 51: Edge-case for when the blue guard cannot move because its movement direction cannot be projected towards the inside the polygon.

Moreover, numerous issues were posed by the CGAL library itself. These problems mainly related to the library's non-detailed errors and brief documentation. Having to reverse engineer and delve into the source-code of CGAL slowed down the debugging process. Because some errors were not explicit at all, some crashes remained unsolved (for example, the program crashes for certain starting positions for different polygons).

Some issues were posed by converting between CGAL's number types and the native C ++ types (`double`). This was necessary for speeding up the computations. Using only CGAL's types resulted in program runs longer than five minutes even for the smallest test cases. Approximation to `double`s allows us to reduce this time to near-instant results for the smallest test cases. Nonetheless, such conversion resulted in approximation errors. For instance, some guards "close enough" to reflex vertices were wrongly considered to be on the reflex vertex. Conversely, comparing point coordinates would not always work. Sometimes guards with the same coordinates would not be considered to be the same.

Additionally, we were especially interested in the irrational guards polygon [1]. However, our program can only solve the irrational guards polygon if the guards are already very close to the optimum. For other starting positions, the program crashes with a mysterious CGAL error. Because of the vagueness of the error and the time constraints of the thesis, we could not fix this issue.

Lastly, we also wanted to test the program on polygons from the APGlib library [6]. The APGlib library offers an extensive testbed for polygons. Unfortunately, the same type of mysterious error

happened with APGlib polygons with 20 vertices. Interestingly, the program only got stuck in a local minimum for one of the polygons with 20 vertices (polygon 3). Nonetheless, the fact that we could not address the CGAL error in this thesis' time constraints was quite dissatisfying.

Another problem worth mentioning is scalability. The program does not scale. As mentioned in Section 4, for comb polygons with more than 6 teeth, the waiting time already exceeds an hour to finish. We believe that this waiting time is inadequate for the size and number of guards of the polygon. Some of the largest bottlenecks that work against scalability are the visibility area and the Hidden Gradient computations. Currently, the visibility area of each guard is updated at every iteration. We are using the Triangular Expansion Visibility [5] which runs in $O(g)$ time, with g the number of guards. Thus, the visibility computation per iteration runs in $O(g^2)$ time. In terms of the Hidden Gradient computation, each guard has its gradient recomputed at most g times. This happens in the case when only one guard out of the g guards has a non-zero gradient, and thus it gets removed from the set. The remaining $g-1$ guards have their gradient recomputed. Again, in the worst case only one guard has a non-zero gradient. Thus, a guard gets its gradient recomputed in $O(g^2)$ time. Other poorly scalable parts of the algorithm are based on the number of reflex vertices. The gradient computation depends on the number of reflex vertices r. If $r >> g$, then the algorithm will perform poorly for polygons with significantly fewer guards than reflex vertices.

For these reasons, the algorithm is sensitive to the heuristics used, the shape of the polygons and the values of the hyperparameters. In the Section 4 we have mentioned some shapes of polygons that the algorithm can solve, and with which hyperparameters initial guard positions. We believe that if these edge-cases and errors get solved, the program should perform correctly and with no crashes for any input polygons. We would still expect that the program would get stuck in local minima. However, it should be possible to escape them with case-specific hyperparameter tuning and heuristic choice.

Acknowledgements

I would like to thank Till Miltzow and Frank Staals for supervising the thesis and always offering creative and constructive feedback. I would also like to thank Simon Hengeveld for the helpful advice and explanation of his implementation for solving the Art Gallery Problem. Lastly, I would like to thank my partners Tim and Teun for always supporting me during both the rough and the happy times of this thesis.

References

[1] Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. "Irrational Guards are Sometimes Needed". In: *CoRR* abs/1701.05475 (2017). arXiv: 1701.05475. URL: http://arxiv.org/abs/1701.05475.

[2] Tetsuo Asano. "An efficient algorithm for finding the visibility polygon for a polygonal region with holes". In: *IEICE TRANSACTIONS (1976-1990)* 68.9 (1985), pp. 557–559.

[3] Pritam Bhattacharya, Subir Kumar Ghosh, and Bodhayan Roy. "Approximability of guarding weak visibility polygons". In: *Discrete Applied Mathematics* 228 (2017). CALDAM 2015, pp. 109–129. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2016.12.015. URL: https://www.sciencedirect.com/science/article/pii/S0166218X16306333.

[4] Édouard Bonnet and Tillmann Miltzow. "An Approximation Algorithm for the Art Gallery Problem". In: *CoRR* abs/1607.05527 (2016). arXiv: 1607.05527. URL: http://arxiv.org/abs/1607.05527.

[5] Francisc Bungiu et al. "Efficient Computation of Visibility Polygons". In: *CoRR* abs/1403.3905 (2014). arXiv: 1403.3905. URL: http://arxiv.org/abs/1403.3905.

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

[7] Boris Delaunay et al. "Sur la sphere vide". In: *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk* 7.793-800 (1934), pp. 1–2.

[8] Subir Kumar Ghosh. "Approximation algorithms for Art Gallery Problems in polygons". In: *Discrete Applied Mathematics* 158.6 (2010), pp. 718–722. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2009.12.004. URL: https://www.sciencedirect.com/science/article/pii/S0166218X09004855.

[9] I Goodfelow, Y Bengio, and A Courville. "Deep learning (adaptive computation and machine learning series)". In: (2016), p. 296.

[10] Simon B. Hengeveld and Tillmann Miltzow. "A Practical Algorithm with Performance Guarantees for the Art Gallery Problem". In: *CoRR, SoCG* abs/2007.06920 (2020). arXiv: 2007.06920. URL: https://arxiv.org/abs/2007.06920.

[11] Barry Joe and Richard B Simpson. "Corrections to Lee's visibility polygon algorithm". In: *BIT Numerical Mathematics* 27.4 (1987), pp. 458–473.

[12] D. Lee and A. Lin. "Computational complexity of Art Gallery Problem". In: *IEEE Transactions on Information Theory* 32.2 (1986), pp. 276–282. DOI: 10.1109/TIT.1986.1057165.

[13] Shiva Maleki and Ali Mohades. "Implementation of polygon guarding algorithms for art gallery problems". In: *CoRR* abs/2201.03535 (2022). arXiv: 2201.03535. URL: https://arxiv.org/abs/2201.03535.

[14] Joseph O'rourke et al. *Art gallery theorems and algorithms*. Vol. 57. Oxford University Press Oxford, 1987.

[15] WH Swann. "A survey of non-linear optimization techniques". In: *FEBS letters* 2.S1 (1969), S39–S55.