Computational Geometry : Project

Charlotte Gullentops - 000463980
Eline Soetens - 000459750

07 December 2021

1 Presentation of the topic

The topic we will present here is how to find the rectangle with the maximum area inscribed in a convex polygon, taking into account that this rectangle must not necessarily be aligned with the x and y axis. There is a large sample of literature dedicated to finding, in a convex polygon, the largest area rectangle with edges parallel to the axis. For example, Alt et al. present a logarithmic algorithm to find it in O(log n) [1]. See a summary of the existing algorithms in Table 1. However the case of finding rectangles not aligned with the axis is less studied. The main article [2] we used for this topic presents an heuristic algorithm that can find rectangles with an area close to the maximum area possible :$area(R_{found}) \geq (1-\epsilon) \times area(R_{optimum})$ with $\epsilon > 0$.

Author	Year	Polygon	Orientation	Type	Computational cost
Fischer and Höffgen [3]	1994	Convex	Axis-aligned	Deterministic	$O(\log^2 n)$
Alt et al. [1]	1995	Convex	Axis-aligned	Deterministic	$O(\log n)$
Daniels et al. [4]	1997	Arbitrary	Axis-aligned	Deterministic	$O(n \log^2 n)$
Boland and Urrutia [5]	2001	Arbitrary	Axis-aligned	Deterministic	$O(n \log n)$
Knauer et al. [6]	2010	Convex	Arbitrary	Heuristic	$O(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log n)$
Knauer et al. [6]	2010	Arbitrary	Arbitrary	Heuristic	$O(\frac{1}{\epsilon} n^3 \log n)$
Molano et al. [7]	2010	Arbitrary	Arbitrary	Heuristic	$O(n^3)$

Table 1: Summary of computation costs of several variation of the largest inscribed rectangle (strongly inspired by [7])

As we focus on convex polygons, we based our algorithm on Knauer et al. [2], which is a more up-to-date version of the work presented in [6] (fifth in the table). It treats arbitrary orientation with an heuristic method. The original complexity of the algorithm we used is expressed in this theorem (the first in the article):

Theorem 1.1. *Let P be a convex polygon with n vertices. Suppose the vertices of the polygon are given in a clockwise order. Then, an inscribed rectangle in P with area of at least $(1-\epsilon)$ times the area of a largest inscribed rectangle can be computed*

- *with probability t in $O(\frac{1}{\epsilon} log(n))$ deterministic time for any constant $t < 1$;*
- *in $O(\frac{1}{\epsilon^2} log(n))$ deterministic time;*
- *in $O(\frac{1}{\epsilon} log(\frac{1}{\epsilon}) log(n) + \frac{1}{\epsilon^{28}})$ deterministic time.*

As we decided to let the user draw its own convex polygon, our algorithm's complexity must take Graham's scan complexity into account (see Section 2.1). Moreover, Knauer et al. used Alt et al (second in the table) to look for the largest rectangle, but we decided to implement Fischer and Höffgen (first in the table) for that part. Therefore, the $log(n)$ appearing in the complexity from Theorem 1.1 is a $log^2(n)$ in our algorithm.

2 Description of the algorithm

The main principle of the algorithm is to find potential directions for rectangles then compute the biggest area rectangle possible for those directions. The algorithm we follow is an heuristic one, meaning that we will not necessarily find the largest rectangle. However, this randomized algorithm has a probability t to find a rectangle inside the polygon which has at least (1-ϵ) times the optimum area.

The following pseudo-code presents the big steps of our algorithm to find the largest inscribed rectangle. Each step will be detail in the following subsections.

Algorithm 1 Find largest rectangle : Main function

```
1: areaMax ← 0
2: rectMax ← empty set
3: compute (NRPOINTS*NRPOINTS/ε) directions
4: for each possible direction uv do
5:    rotate polygon to match that direction
6:    rectangleAreaAndVertices ← largest rectangle in that direction
7:    if rectangleAreaAndVertices' area > areaMax then
8:       rectMax ← set of rectangleAreaAndVertices' vertices
9:       areaMax ← rectangleAreaAndVertices' area
10:      rotate back the polygon and the rectMax
11:   else
12:      rotate back the polygon
13:   end if
14: end for
15: global rectangle ← rectMax
```

Here are some remarks about the Algorithm 1; we compute respectively NRPOINTS for the U set of points (see Section 2.2) and NRPOINTS$*\frac{NRPOINTS}{\epsilon}$ for the V set. In the article, it is said that we need $\Theta(1)$ points in the first set and $\Theta(\frac{1}{\epsilon})$. We could then freely add that constant NRPOINTS to increase our number of directions.

We keep the largest rectangle in a global variable. It allows us to draw it when the algorithm terminates.

2.1 Pre-treatment : create convex polygon

Convex hull algorithm from Eline's homework

To make our presentation more playful, we let the user create their own polygon. To do so they can click wherever they want in the window. Each time they mouse click, it creates a point. Then, they can announce that they finish their construction. Then, our algorithm applies Graham's scan to compute the convex hull from the set of points. Its complexity is $O(n\,log(n))$. For the rest of the algorithm, we can therefore consider to have the n vertices of our polygon in counter-clockwise order, similarly as in the article.

2.2 Finding ϵ-directions

Mainly Charlotte

The first step is to find directions close enough to the orientation of one of the sides of the optimal rectangle such that the area of our rectangle would almost be the largest possible. To find those directions, we draw two sets of random points inside the polygon and each direction is the combination

of one point from each set. In this set of direction, we should find what we will call $\epsilon-$close directions. We know that our optimal rectangle will be aligned with one of those direction with a good probability. To read about the following steps of the algorithm, jump to section 2.3. The rest of this part is the explanation of why choosing points at random works.

ϵ-direction is defined as follow : we consider that we know the optimal rectangle (R_{opt}) and we call the intersection of its diagonals s. The segment $\overline{ab}$ is one of its short side and the point d is the midpoint of $\overline{ab}$. The two triangles T_1 and T_2 are defined using vertices s, d and a third vertex $f_1 := d + \epsilon(b-d)$ or $f_2 := d - \epsilon(b-d)$. T_3 and T_4 are obtained using the same principle with the other short side. An $\epsilon-$close direction is a direction that intersects $\overline{f_1 f_2}$ while going through s. Intuitively, we can see that a rectangle closely aligned with the optimal rectangle will have a similar area. The complete proof is detailed in [2].

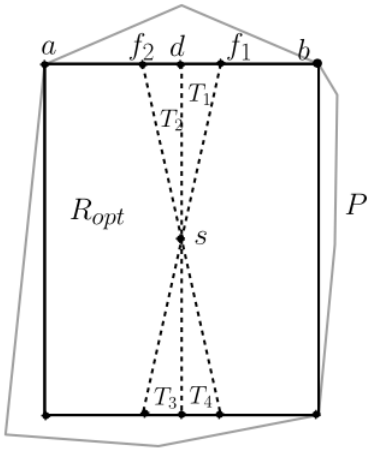

Figure 1: A largest rectangle R_{opt} in a convex polygon P. (from [2])

In practice, the idea to find a set of such $\epsilon-$close directions is to take a set U of $\Theta(1)$ points with a random uniform distribution, then take another set V of $\Theta(1/\epsilon)$ points. The quantity of points taken in the second set is determined by the ratio the user chose at the beginning. To find those points, Knauer et al [2] suggested several methods which can all be resumed by : take random points inside the polygon. Even though we did not exactly use their method (which consisted in choosing randomly a height between the two extremes y, then taking the largest width of the polygon at that height and finally taking one random point on this line), we did something similar: take a random y between the two y-extremes then take a random x between the x-extremes. If that point is outside the polygon, compute another x-coordinate and check again. We have a good probability of having $\epsilon-$close directions composed by a point in U and a point in V.

To find if the set U and V actually define some ϵ-directions, we could associate each point u of U with each point v of V and for each pair, compute its x and y deviation. Then, we would artificially create a point q whose coordinates are those of s, incremented by the newly computed deviations. Therefore, we could check if the segment $\overline{sq}$ or its projection will ever intersect $\overline{f_1 f_2}$. If yes, we would add the pair (u,v) to the set of ϵ-directions.

For the sake of pedagogy, we wrote a separated little program to prove that those random ϵ-directions are acceptable using the method described above. First, users can choose the ratio of points in U regarding the number of points in V, which indirectly defines the value of epsilon. Then, they can see the limits of f_1 and f_2, marked with green dots on the superior edge of the rectangle. When they click on "Generate points", one red dot appears as well a $1/\epsilon$ blue dots. By clicking on "Show

best directions", the acceptable directions (linking red and blue dots) will appear on green. If none appear, it simply means that we did not find any (which is possible as the algorithm is heuristic). In that case, the user can generate more points with the same ratio, or clear the points and start over.

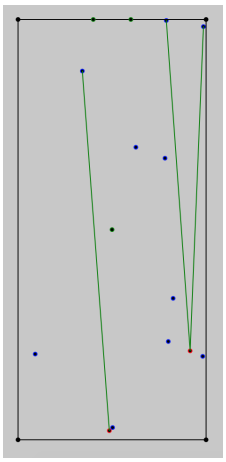

Figure 2: Usage of *Find epsilon-direction*,the two green point of the upper edge define the segment $\overline{f_1 f_2}$, the green lines show ϵ-directions

NB: the chosen ratio can not be changed during the execution.

The ratio and the ϵ are intimately related: $ratio = \frac{1}{\epsilon}$. The ratio is the number of points put within V for each point added to the set U. But ϵ also appears in the formula to find f_1 and f_2. Intuitively, we can see that if our condition is for the direction to intersect the segment $\overline{f_1 f_2}$ and that we decrease the length of that segment (ϵ decreases), we will need more points in V to have the same probability to have valid ϵ-directions (ratio increases).

2.3 Additional part : rotate our polygon accordingly to our direction

Team work

Once we found all these possible directions, the main part remains; we must compute the biggest area rectangle in that direction. We will do it using Fisher and Höffgen method [3]. In the original article of C. Knauer et al [2], they use H. Alt's work [1] which computes the largest rectangle in logarithmic time. As Alt's article is based on our [3], we took the liberty to apply the earliest one. Nevertheless, it only finds the largest inscribed rectangle in $O(log^2 n)$ so the total complexity of our algorithm is not the same as the one announced in our first article [2]. As those algorithms compute the largest axis-aligned rectangle, we first need to rotate our polygon accordingly to the $\epsilon-$close direction being tested so the algorithm aligns the rectangle with that particular direction and not with the axis.

Therefore, for each ϵ-direction, we will rotate our polygon such that our direction is considered as the new x-axis. The equation system of our rotation is the following :

$$\begin{cases} x' = x\ cos\ \theta + y\ sin\ \theta \\ y' = -x\ sin\ \theta + y\ cos\ \theta \end{cases}$$

.

where $\theta = arctan(\frac{delta_y}{delta_x})$. See a pseudo-code of that part below.

Algorithm 2 Rotate polygon (and eventual rectangle)

Require: a direction (via two points u and v), backForth which defines if it is the forth or backward rotation and the points of the rectangle (if any)
Ensure: to rotate the polygon and the possible rectangle
1: compute the direction inclination with regard to axe x (θ)
2: get the extreme points of the polygon (minX, maxX, minY, maxY)
3: compute the center coordinates (tx, ty) of the square composed by those extremes
4: call actualRotation(tx, ty, theta, backForth, polygon vertices list);
5: **if** rectangle is defined && backForth = back **then**
6: call actualRotation(tx, ty, theta, backForth, rectangle vertices list);
7: **end if**

Algorithm 3 Actual rotation

Require: coordinates of polygon's center (tx, ty), angle of rotation θ, backForth and the points of the rectangle (if any)
Ensure: to modify the coordinates of the vertices list according to θ and backForth
1: **for** each point of the list **do**
2: $x' = (x - tx) * cos(backForth * \theta) + (y - ty) * sin(backForth * \theta)$
3: $y' = -(x - tx) * sin(backForth * \theta) + (y - ty) * cos(backForth * \theta)$
4: **end for**

NB: for the actual rotation part (Algorithm 3), we needed to store the previous x and y values as the change to x' and y' can not be executed at the exact same time and they both depend on the old values x and y. We did not expressively write it in the pseudo-code to keep it clear but this needed to be addressed.

The backForth parameter is equals to 1 if we are doing the first rotation, and equals to -1 if we want to rotate back to the initial x-y system. By multiplying θ by it, we ensure the right rotation direction.

Concerning the complexity of our rotation, we can see that computing θ is O(1) as it is only a call to an arc-tangent function. Then, looking for the extremes of the polygon takes O(n) as we look at all our vertices. Then we call actualRotation which is also O(n) because it needs to apply simple operations on every point. If we have a rectangle to rotate, it takes O(4) as the rectangle is only composed of four vertices. The total complexity of our rotation is O(n).

NB: once we found the biggest rectangle in our rotated polygon, we have to rotate back to our original axis system so we can check the other directions. The area of the rectangle is preserved by the rotation but not its corners' coordinates so we also rotate those.

2.4 Finding the largest rectangle in a particular direction

Mainly Eline

Once the vertices of our polygon have been rotated according to the ϵ-direction, we can search the axis-aligned rectangle with the biggest area. In their paper, Fischer and Höffgen [3] present deterministic algorithms for two different cases. Indeed the rectangle with the biggest area can have either two or three corners on the border of the polygon. If the rectangle has only two corners on the border then those corners are on a diagonal of the rectangle. In the code we implemented as demonstration, we only implemented the case of looking for a rectangle with three corners on the border.

The first step of the algorithm is to identify the vertices of the polygon with the minimum and maximum x- and y-coordinates. We use those vertices to define the *southwest*, *southeast*, *northeast* and *northwest* - called SW, SE, NE and NW - parts of the boundary of the polygon.

The next part of the algorithm is described for the SW part of the boundary but must be applied to the other parts as well in order to find the rectangle with the biggest area. We begin by identifying x_l, the minimum x-coordinate vertex of the polygon. Then we identify x_{SW} and x_{NW} the maximum x-coordinate vertices on SW and NW. We define then $x_r = \min\{x_{SW}, x_{NW}\}$.

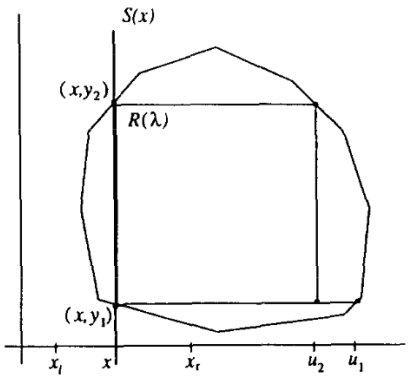

Figure 3: $R(x)$ is the rectangle associated with the x x-coordinate on SW(from [3])

Now, for any point (x, y_1) on the SW part of the boundary where $x \in [x_l, x_r]$, we can find the rectangle with the biggest area with three corners on the boundary. To do so, we begin by finding the point (x, y_2) that sits on the NW part of the border. Then we can find $u_1(u_2)$ which is the x-coordinate of the horizontal projection of $y_1(y_2)$ onto $NE \cup SE$. We keep $x' = \min\{u_1, u_2\}$ and we obtain that the three other corners of the biggest area rectangle at (x, y_1) are $(x, y_2), (x', y_2), (x', y_1)$.

The function we implemented in order to find the corner (and area) of a rectangle defined by an x-coordinate as well as the region in which this x-coordinate is situated is described in the Algorithm 4 using pseudo code. We only wrote down the case of a point belonging to the SW or NW parts. For a point in the *eastern* region the same principle can be applied.

Algorithm 4 Area Rectangle from X and Region - west part

Require: an x value and which region we want to run the algorithm
Ensure: to return the area of the rectangle defined as well as the coordinates of its corners
1: verify if the x is in the interval defined by the region
2: $v_1 \leftarrow$ BinarySearchX(x,SW)
3: $v_2 \leftarrow$ BinarySearchX(x,NW)
4: $y_1 \leftarrow$ y-coordinate of the point on edge v_1 with coordinate x
5: $y_2 \leftarrow$ y-coordinate of the point on edge v_2 with coordinate x
6: $v_3 \leftarrow$ BinarySearchY(y_1, SE $\cup$ NE)
7: $v_4 \leftarrow$ BinarySearchY(y_2, SE $\cup$ NE)
8: $u_1 \leftarrow$ x-coordinate of the point on edge v_3 with coordinate y_1
9: $u_2 \leftarrow$ x-coordinate of the point on edge v_4 with coordinate y_2
10: $u \leftarrow$ minimum(u_1, u_2)
11: RectangleCorners $\leftarrow [(x, y_1), (x, y_2), (u, y_2), (u, y_1)]$
12: Area $\leftarrow (u - x) * (y_1 - y_2)$
13: **return** Area and RectangleCorners

BinarySearchX (resp. Y) is a binary search used on a region in order to find on which edge the point with x-coordinate (y-coordinate) is located.

For the time complexity, verifying if x is in the interval of the region is simply comparing x with the x-coordinate of the first and last vertex of the region which is done in O(1). The binary search has a complexity of O(log n) since the function used during the comparison is only doing a comparison between x-coordinates (resp. y-coordinate). Finding the y-coordinate (resp. x-coordinate) of a point on an edge is in O(1). Computing the minimum between two values, the list of corners of the rectangle and its area are all operation with a cost of O(1). The total cost for computing the rectangle and its area for an x value and a region is then O(log n)

Fischer and Höffgen [3] then prove that the area of this rectangle as a function of $x \in [x_l, x_r]$ is continuous and strictly increasing on a first part then strictly decreasing on a second part. It follows that the x corresponding to the maximum area rectangle, and thus the area of this rectangle, can be found using a binary search.

We can then follow the same process for the other regions in order to identify the biggest area rectangle in each of those, then only keep the largest one of all.

Algorithm 5 Search largest rectangle

Ensure: to return the biggest area rectangle found with the polygon in that specific orientation
1: compute polygon's extremes points and store their index (i_minX, i_maxX, i_minY, i_maxY)
2: SW ← part of the points list from i_minX to i_maxY
3: SE ← part of the points list from i_maxY to i_maxX
4: NE ← part of the points list from i_maxX to i_minY
5: NW ← part of the points list from i_minY to i_minX
6: use those regions to assign values to x_min_w (minimum X coordinate on the west), x_max_w, x_min_e and x_max_e which will be useful to compute the rectangle
7: **if** there is only one vertex in one of the west parts **then**
8: sol_e ← 0, west_solution ← 0
9: **else**
10: sol_w ← binarySearch(AreaofRectanglefromXandRegion, SW, "SW", x_min_w, x_max_w)
11: west_solution ← AreaofRectanglefromXandRegion(sol_w, "SW");
12: **end if**
13: **if** there is only one vertex in one of the east parts **then**
14: sol_e ← 0, east_solution ← 0
15: **else**
16: sol_e ← binarySearch(AreaofRectanglefromXandRegion, SE, "SE", x_min_e, x_max_e)
17: east_solution ← AreaofRectanglefromXandRegion(sol_e, "SE");
18: **end if**
19: **if** west_solution or east_solution != 0 **then**
20: keep the biggest as solution (max_areaX ← sol, max_reg ← "region name")
21: **else**
22: no biggest rectangle found
23: **end if**
24: **return** [max_area, max_rectangle]

In the Algorithm 5, we make the difference between *sol_...* which is the index of the vertex i which will give the biggest area and *..._solution* which is the actual area of that biggest rectangle (that has i as one of its corner).

At line 7 and 13, we check if one of the regions only has one vertex. If it is the case, we can not do a binary search to find on which edge our vertex i giving the biggest area is, so we put our *sol_...* and *..._solution* at zero. Otherwise, we look by binary search where the vertex i should be and we use that information to call AreaofRectanglefromXandRegion, which is a function described in the Algorithm 4.

At line 19, we compare the best solution of western and eastern regions. If neither of those succeeded in computing a solution, there is no largest rectangle we can find with that orientation such that it has three of its corners on the boundary of the polygon.

Since the vertex of the polygon are stocked in a counter-clockwise order, we could find the polygon's extreme vertices by using a binary search which has a cost of O(log n). However, note that, as it was not the main focus of our algorithm, we used a brute force algorithm to find the extremes and this part is actually done in O(n).

Finding the values of x_min_w, x_max_w, x_min_e and x_max_e is done in O(1).

The binary search done at line 10 and 16 have O(log m) steps, where m is $x\_max\_w - x\_min\_w$ (resp. $x\_max\_e - x\_max\_e$). At each of these steps, we must compute the rectangle corresponding to this x-coordinate which has a cost of O(log n). We can then deduce that the total cost of our implementation of the search for the largest rectangle parallel to the axis is O(log m log n). The algorithm described in the article [3] use binary search with O(log n) steps where we have O(log m) steps. This brings the complexity of their version to $O(\log^2 n)$

Restrictions

The way we divide the boundary into region give rise to a major inconvenient in certain case. Indeed, we can have polygons where some of the regions contains only one vertex.

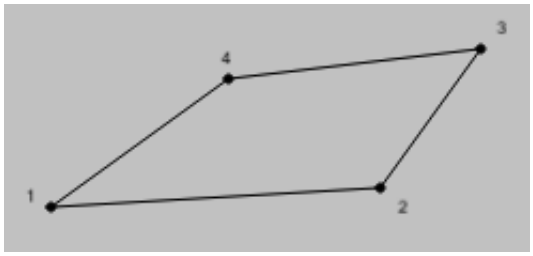

Figure 4: Polygon were the SW and NE region are composed of only one vertex

In the figure 4, the SW region is only composed of the vertex 1, the SE region is composed of vertices $1, 2, 3$, the NE region is only composed of vertex 3 and the NW region is composed of vertices $3, 4, 1$. The algorithm, as we described it above, cannot be applied on a region containing only one vertex. Indeed, we can see on the figure that we can not find two points with the same x-coordinates that sit on the border SW and NW (resp. SE and NE). This means that for certain orientation, we can not find rectangle with three corners on the boundary of this polygon. However, if the polygon rotates, we can end up in a situation where all regions have at least two vertices, i.e the vertices with the minimum and maximum x- and y-coordinates are all different. In these cases, we can apply the algorithm. Since we will test a large variety of directions (and thus rotations) for our polygon, we should be able to find some directions where we can compute the largest area rectangle.

3 Examples

See at Figures 5 to 7 the execution of the steps of our algorithm on an arbitrary set of points.

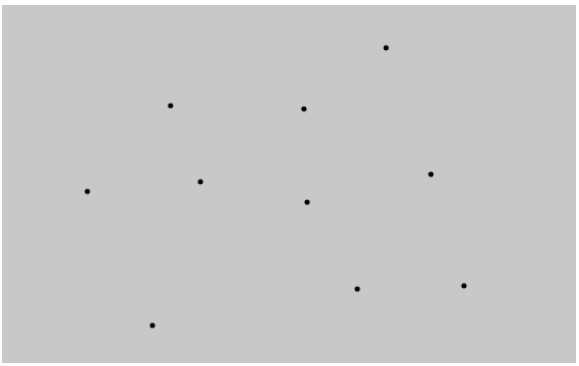

Figure 5: First step : draw arbitrary points

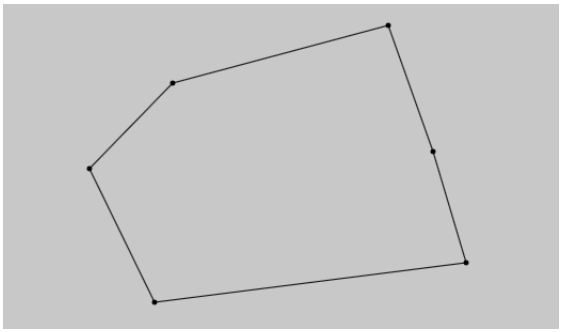

Figure 6: Second step : compute their convex hull

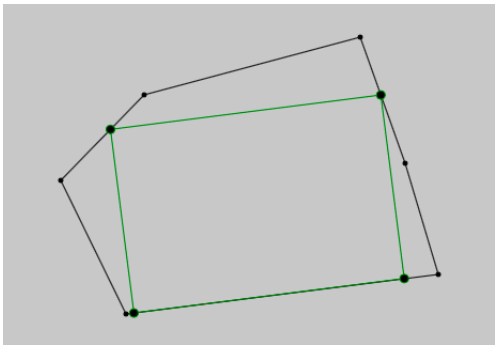

Figure 7: Last step : compute largest rectangle

4 Resources

- Here is the link of our Github repository : https://github.com/ElineSoetens/BiggestAreaRectangle.
- You can also visit our GitHub page and play with the code : https://elinesoetens.github.io/BiggestAreaRectangle/

References

[1] J. Snoeyink H. Alt, D. Hsu. Computing the largest inscribed isothetic rectangle. *Proceedings of 1995 Canadian Conference on Computational Geometry*, pages 67–72, 1995.

[2] J. M. Schmidt H. R. Tiwary C. Knauer, L. Schlipf. Largest inscribed rectangles in convex polygons. *J. Discrete Algorithms*, 13:78–85, 2012.

[3] K.-U. Höffgen P. Fischer. Computing a maximum axis-aligned rectangle in a convex polygon. *Inf. Process. Lett.*, 51:189–193, 1994.

[4] D. Roth K. Daniels, V. Milenkovic. Finding the largest area axis-parallel rectangle in a polygon. *Comput. Geo. Theor. Appl.*, 7:125–148, 1997.

[5] J. Urrutia R.P. Boland. Finding the largest axis-aligned rectangle in a polygon in o(nlog n) time. *Proc. 13th Canad. Conf. Comput. Geom*, pages 41–44, 2001.

[6] J.M. Schmidt H.R. Tiwary C. Knauer, L. Schlipf. Largest inscribed rectangles in convex polygons. *Proceedings of the 26th European Workshop on Computational Geometry (EuroCG'10)*, 2010.

[7] A. Caro M. L. Durán R. Molano, P. G. Rodríguez. Finding the largest area rectangle of arbitrary orientation in a closed contour. *Applied Mathematics and Computation*, 218(19:9866–9874, 2012.