Approximation of Minimum Convex Partitioning

Benjamin Kahl Abbas Mohammed Murrey Semjon Kerner
Konstantin Jaehne

March 2020

Abstract

A convex partition of a pointset consists of a planar subdivision of its convex hull so that all faces are empty and convex. Finding a partition with the minimum amount of edges (or faces), is still a problem of unknown complexity.

As part of the CG:SHOP 2020 competition we devised a set of four different algorithms to compute various convex partitions for differently arranged pointsets. Here we outline the implementation of each of them in tandem with our central findings on how they perform.

1 Introduction

1.1 Problem Description

The problem given by the CG:SHOP competition reads as follows:

n points in the plane are given. We must then compute a plane graph that partitions their convex hull into convex polygons. Notably the convex hull of the set of points is always included. Furthermore the goal is to minimize the resulting number of faces.[2]

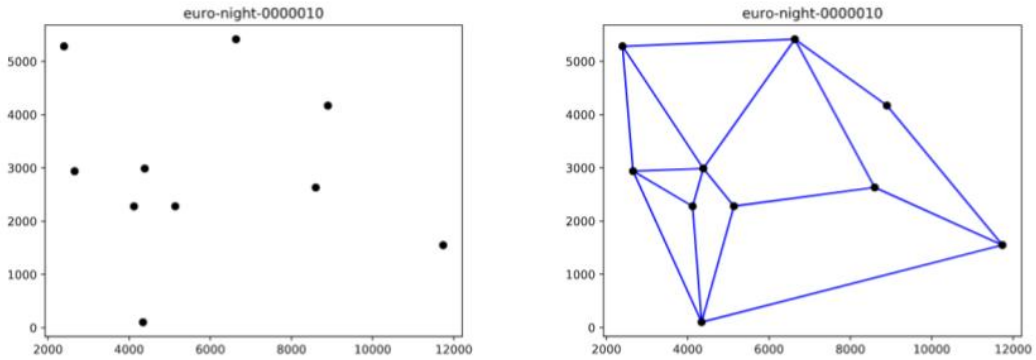

Figure 1: Example set of points with a valid (but sub-optimal) solution.[2]

The contest provided a total of 346 different instances, 247 of which were already published at the start of the competition. The instances can be classified as follows:

- Uniformly randomly spread points
- Based on some picture, random points on its edges
- Based on the brightness of some picture, random points
- Random points on a grid resulting in many collinear points

Each solution for an instance gets scored individually and the overall score aggregate yields our teams total score in the competition.
Note that the goals of minimizing the number of faces and of minimizing the number of

edges are interchangeable, as implied by Euler's formula. This defines a simple way to calculate scores based on the worst case (any triangulation) of an instance:

$$score = \frac{\#\text{edges of triangulation} - \#\text{edges of solution}}{\#\text{edges of triangulation}} \tag{1}$$

Any triangulation is actually a valid solution to the problem, albeit a rather bad one, resulting in a score of 0. The best case, a score of 1, cannot possibly be reached as it would require a solution consisting of zero edges. The goal, however, is to maximize the score over all instances.

1.2 Basic Functionality

1.2.1 Doubly-Connected Edge List

The data structure we used is the *Doubly-Connected Edge List* (or DCEL). Commonly used to represent plane graphs, it has the advantage of fast traversal and easy manipulation of its edges. However, as stated above, the problem can be formulated either in terms of minimizing faces or in terms of minimizing edges, hence in early stages of development we decided to use a simplified version that doesn't include faces.
Its component parts remain vertices and half-edges, where each edge is represented by a pair of half-edges and, in turn, each half-edge represents one direction of the edge. In detail this means:

- Each vertex v has a pointer to an outgoing edge called the Incident Edge of v
- Each half-edge e has pointers to:
  - its origin e.origin (a vertex)
  - its twin half-edge e.twin (the other half-edge of this edge)
  - the next half-edge on its incident face e.next
  - the previous half-edge on its incident face e.previous

See Figure 2: The interior half-edges of a face are arranged in counter-clockwise order, induced by the given e.next and e.prev pointers. For example, we get the target vertex of a given edge e by e.next.origin.[5]

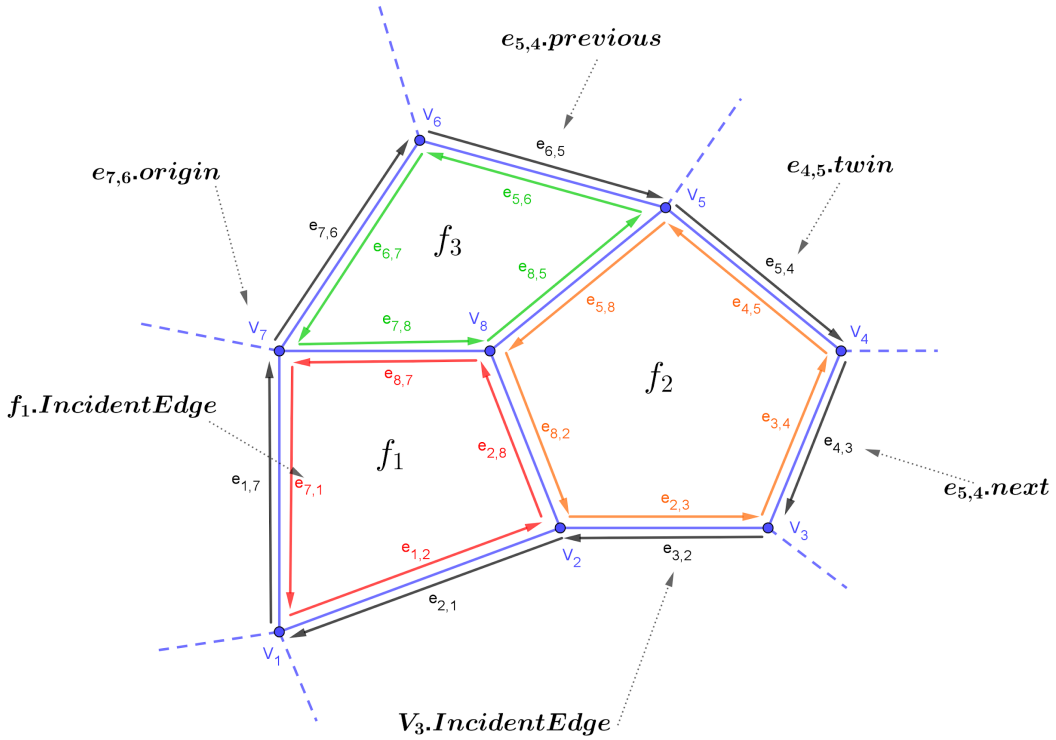

Figure 2: DCEL illustration

1.2.2 Orientation Test

Many of our programs functionalities are performed with the use of *orientation tests*. In essence, given a line and a point v in the plane, the orientation test equates to the side of the line the point lies on. To make this unambiguous, a line is defined to have a direction, given by the order of two distinct points a and b on the line. Thus we can compute whether the point is on the left of the line by considering the sign of

$$\begin{vmatrix} \overline{ab_x} & \overline{av_x} \\ \overline{ab_y} & \overline{av_y} \end{vmatrix}$$

If this number is greater than 0, it means the point is located on the left, and vice versa for the right. It is one of the core functions used by many of the other functions. For example, to determine whether two lines intersect, we simply test if the two points of one line are located on different sides of the other. The following pseudo code illustrates the basic functionality of the orientation test:

```
return ((b.x - a.x)*(v.y - a.y) - (b.y - a.y)*(v.x - a.x)) > 0
```

1.3 Team Organization

To commence, we developed two simple algorithms *single convex wave* and *nested hulls* which provided a baseline-score for our subsequent, more complex implementations to beat. We also settled Python as our programming language of choice, for various reasons:

- All our team members were comfortable with Python.
- Possible disadvantages in terms of runtime were negligible, because the competition didn't include a live performance; instead we merely submitted our solutions while the competition was ongoing, at any time of our choosing.
- The organizers of the contest provided a python library, consisting of a function that checks if a solution is valid, which is convenient.

Our team conducted regular, weekly meetings and communicated within a private *Slack* channel. The finished project is hosted on gitHub under
`https://github.com/SemjonKerner/convex_polygons` [1].

2 Nested Convex Hulls Approach

In this chapter we will give an overview on the functionality of the *nested hulls* algorithm that provides a simple way of partitioning a convex hull. Based on the arrangement of the instance points, this approach can bring good results in minimizing the number of generated faces as well.

2.1 The procedures

Figure 3.(a) shows an example of a set of points in plane, the usage of the *nested hulls* approach for partitioning this instance can be expressed in the following steps:

1. For a set of points S, iteratively keep computing convex hulls:
   (a) Compute the convex hull C of S
   (b) Subtract the data points of C from S
   (c) Repeat 1a and 1b until $S = \emptyset$, as in Figure 3.(b)
2. Connect each two sequential convex hulls in such a way that no edge would be added unless if we do not do that we violate the convexity conditions, as can be seen in figure 3.(c). Listing 1 shows a possible way to do so in code.
3. While in connecting step we ensure that no edge added can be deleted *"those marked orange in figure 3.(c)"*, however in step 1 we compute each convex hull separately, and we don't know if the edges added in this step are still needed after the connecting step. Thus, except for the most outer convex hull, for each convex hull computed in step 1 we need to iterate over all its edges, and check if any of them can be deleted. As can be seen in figure 3.(d) all doted edges are not needed and therefore can be removed.

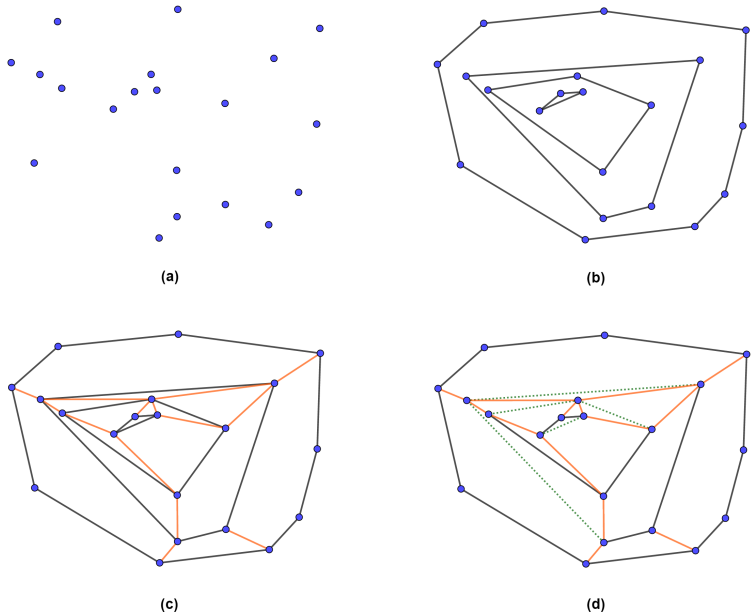

Figure 3: Nested convex hulls example

A possible pseudo-code of connecting two nested convex hulls could look like this:

Listing 1 ConnectingTwoNestedConvexHulls

```
1:  Initialization:
2:      in_f ← the most right point of the inner hull
3:      in_l ← next point to in_f in clockwise direction
4:      out_f ← the most bottom right point to in_f from outer hull
5:      out_l ← next point to out_f in clockwise direction
6:  procedure CONNECTING
7:      while in_f, in_l and in_l.next are collinear do
8:          in_l ← in_l.next // in clockwise direction
9:      if out_f is on left of line going from in_f to in_l then
10:         while out_l is on left of line going from in_f to in_l do
11:             out_f ← out_l
12:             out_l ← out_l.next // in clockwise direction
13:         if in_l and out_f are already connected then
14:             terminate!
15:         else
16:             connect in_l to out_f
17:             in_f ← in_l
18:             in_l ← in_l.next // in clockwise direction
19:     else
20:         if in_f and out_l are already connected then
21:             terminate!
22:         else
23:             connect in_f to out_l
24:             out_f ← out_l
25:             out_l ← out_l.next // in clockwise direction
26:     goto 7
```

Some exceptional cases of the pseudo-code proposed in listing 1, would arise when the inner hull is incomplete.

In this case, when only two points are left, the same procedure can handle this case by replacing lines 17 and 18 by just swapping between *in_f* & *in_l*. And in the case of only one point, we need just to add at least two and at most three edges in order to connect one point inside a convex hull to its boundary points and still preserving the convexity conditions.

2.2 Analysing produced edges

Let V be the number of given points in plane, and E be the number of edges that could be added by the algorithm when partitioning, then we have the following observations:

- From step 1 when computing the convex hulls iteratively, we add at least $V-1$ and at most V edges.
- And when connecting each two sequential nested convex hulls in step 2, each point of the inner hull could be connected to at least none and at most two points of the outer hull, except for the case when the most inner hull is not complete. If the most inner hull is of size one, then that point need to be connected to at least two and at most three points of the outer hull, and in the case when only two points were left as the most inner hull, each one of these two points needs to be connected to at least one and at most two points of the outer hull. So the number of edges that could be added at most in this step would be in the case when the most outer hull is of size three and the most inner hull is of size one, then we need to add at most $2(V-3)+1$ edges.
- In step 3 when deleting the unneeded edges of each nested convex hull separately, we were not able to analyze exactly how many edges are going to be removed. Thus, we naively assume that no edges would be eligible to be deleted in this step.

Therefore, the overall number of edges that could be added at most is then $E = V + 2(V-3) + 1 = 3V - 5$ edges. In fact this is also the number of edges in a simple triangulated mesh.
Although practically we were not able to formulate an example for which the algorithm returns a simple triangulation when a better solution is possible, we also were not able to prove theoretically that such a case cannot happen.
However in practice and for the all instances that we got for the competition and also based on the structure of distribution of the data points in plane, the algorithm performed producing roughly $E = 2V - \sim 20\% \approx \frac{8}{5}V$ edges.

2.3 Runtime complexity

Based on the implementation, each part of the procedures in section 2.1 could have a different category of time complexity. And in our implementation they are categorized as follow:

1. **Iteratively computing convex hulls:** We first begin with sorting the set S by x-coordinates and subsequently by y-coordinates using a built-in sorting algorithm. Based on its documentation it has an amortized worst case of $O(n \log n)$. Then, for each single convex hull, we iterate over all points in S twice by computing the upper half and the lower half of the convex hull, which then takes $T(n) = 2n$ for each single convex hull. The smallest convex hull can be of size three, meaning the worst case would be to decrease the size of S by only three points in each iteration. This would take $T(n) = T(n-3) + 2n$ to compute all nested convex hulls. Assuming that $|S|\%3 = 0$, it can be shown[1] that the previous recursion will take $T(n) = \frac{1}{3}n^2 + n \approx O(n^2)$ until we get $S = \emptyset$.
2. **Connecting each pair of sequential convex hulls:** For each convex hull C_i we iterate over all its data points twice, once when connecting it with C_{i-1} and a second time when connecting C_{i+1} to it. For the most outer and the most inner hulls, we iterate over their points only once. Assuming that the most outer and the most inner hulls are of size three, then the connecting step will take $T(n) = 2n - 6 \approx O(n)$.

[1]

$$
\begin{aligned}
T(n) &= 2n + \overset{1}{2(n-3)} + \overset{2}{2(n-6)} + \overset{3}{2(n-9)} + \ldots + \overset{\frac{n}{3}}{2(n-(n-3))} \\
&= 2n + 2n - 2*3 + 2n - 2*6 + 2n - 2*9 + \ldots + 2n - 2(n-3) \\
T(n) &= \tfrac{n}{3} * 2n - 2*3*\overset{1}{1} - 2*3*\overset{2}{2} - 2*3*\overset{3}{3} - 2*3*\overset{4}{4} - \ldots - 2*3*\overset{(\frac{n}{3}-1)}{(\tfrac{n}{3}-1)} \\
&= \tfrac{2}{3}n^2 - 2*3(1+2+3+4+\ldots..+(\tfrac{n}{3}-1)) \\
&= \tfrac{2}{3}n^2 - 6\textstyle\sum_{i=1}^{\frac{n}{3}-1} i \text{ , replacing } n \text{ by } (\tfrac{n}{3}-1) \text{ in summation formula } \textstyle\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \text{ we get:} \\
T(n) &= \tfrac{2}{3}n^2 - 6\tfrac{(\frac{n}{3}-1)(\frac{n}{3}-1+1)}{2} = \tfrac{1}{3}n^2 + n
\end{aligned}
$$

3. **Removing unneeded edges:** Except for the most outer convex hull, we examine all edges of each hull C_i, assuming that the most outer hull is of size three and the most inner hull is complete, then we have at most $|S| - 3$ edges to examine, which also takes $O(n)$.

From the all above three procedures we can conclude that the overall time complexity is then bounded by $O(n^2)$ when computing nested convex hulls iteratively.

3 Convex Waves

In this chapter we give insight into the *single convex wave* algorithm. This first approach we developed at the beginning of the competition is a distance-based sweep algorithm that propagates in the form of an outward-growing circle, starting from a given location.

3.1 Algorithm

In this Algorithm each iteration proceeds in accordance with the following loop-invariant:

Given a starting point s, pointset P and loop-index i, all points p_j in P with $|p_j - s| \leq |p_i - s|$ are convexly partitioned.

In other words, all vertices within the current propagation radius are correctly partitioned and those outside are not.

As can be observed in Listing 2, the convex hull of the partition is maintained as a separate list (in this case H) at all times. This allows us to efficiently ascertain the exact edge-sequence visible to an exterior point. These edges are the only ones that the integration of said point may impact.

Listing 2 Convex Wave with pointset P and starting point s

Initialization:
  $Q \leftarrow$ Sort P by euclidean distance to s
  $H \leftarrow$ Arrange the first three points in Q into a triangle
Partitioning:
  For each point q_i in $\{q_4, ...q_n\}$:
    a) $h_l \leftarrow$ Calculate the leftmost point in H from q_i
    b) $h_r \leftarrow$ Calculate the rightmost point in H from q_i with $r > l$
    c) Connect q_i to all points h_x with $l \leq x \leq r$
    d) Remove any redundant edges (h_x, h_{x+1}) with $l \leq x < r$
    e) Remove any redundant edges (h_x, q_i) with $l \leq x \leq r$
    f) $H \leftarrow$ Replace all points h_x with $l < x < r$ in H with q_i

The vertices are processed in order of their distance to the chosen starting point, hence the radial loop-invariant.

A new vertex is integrated into the partition by connecting it to all vertices on the hull that are visible to it (two vertices are mutually visible if no edges lie in-between them) (steps a-c). Subsequently, the hulls edges are tested for redundancy, then removed or kept respectively (step d). Additionally, the presence of collinear points on the hull may permit the removal of some edges created in step c (step e). Lastly, the separately maintained convex hull is updated to accommodate the newly annexed vertex (step f).

A concrete example of these steps is illustrated in figure 4.

3.2 Complexity

Visible bounds on a convex polygon (steps a and b) can be computed in logarithmic time, thus, the resulting algorithm provides a convex partition of exclusively non-redundant edges with a worst-case complexity of $O(n \log n)$.

If the input set consists of n *randomly* chosen points, then the expected size of their convex hull equates to $n^{\frac{1}{3}}$ [4], implying an expected runtime of $O(n \log n^{\frac{1}{3}})$.

In practice, the considerably low complexity of this algorithm manifested itself in the form of minuscule computation times, even when running the competitions largest problems (n=1000000) on consumer-grade hardware.

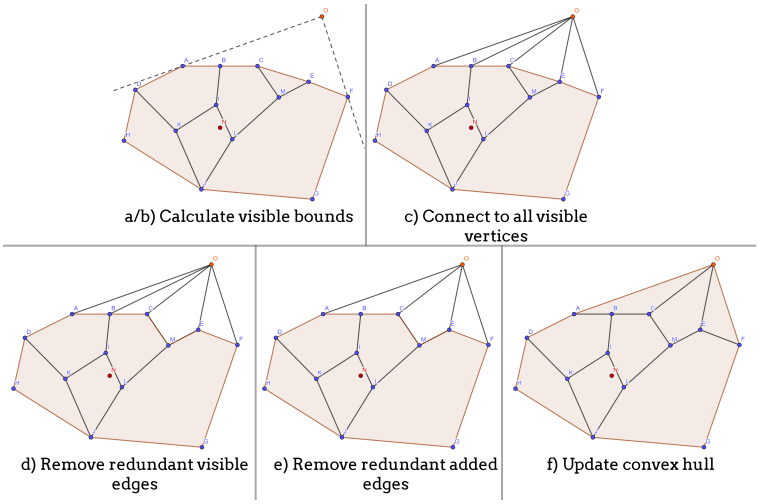

Figure 4: Convex wave iteration procedure as described in Listing 2 with starting point N and new vertex O. Note that vertices A, B and C are collinear.

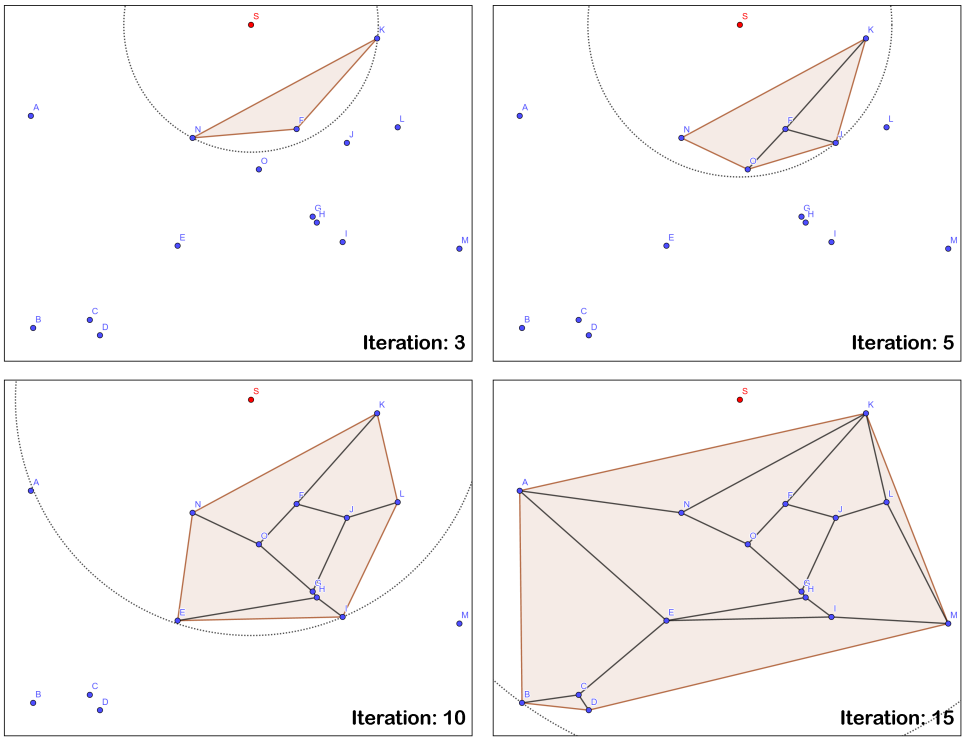

Figure 5: Convex wave on the euro-night-15 instance with chosen starting point S.

3.3 Parellization / Multiple runs

For large inputs, the results provided by *single convex wave* are typically sub-optimal, but can vary in correlation of the chosen starting point. This trait enabled us to rerun the algorithm on a single input with varying, randomly chosen starting points, whilst only preserving the best solution.

Based on our results, we found that after a few passes, the algorithm converged with the smallest achievable amount of generated edges, which was only a slight improvement compared to the very first solution.

In comparison to the *nested hulls* approach, a single convex wave would occasionally perform better and usually yield a mildly improved result after multiple runs. In sets of many collinear points *nested hulls* consistently outperformed *single convex wave*.

3.4 Drawbacks

Figure 6 portrays the result provided by *single convex wave* for a set of 500 points with the start-location chosen at the top-center.

Notably so, the immediate vicinity of the starting-location produces quite uniform faces that are stretched out evenly. On the other hand, faces that are further away from that location have a tendency to be stretched in a single direction. This pattern of arranging longer edges to be in parallel to the starting point becomes more pronounced in relation to the increasing distance.

We identified this trait as a primary weakness of the convex wave algorithm and set about to ameliorate its effects in the next step.

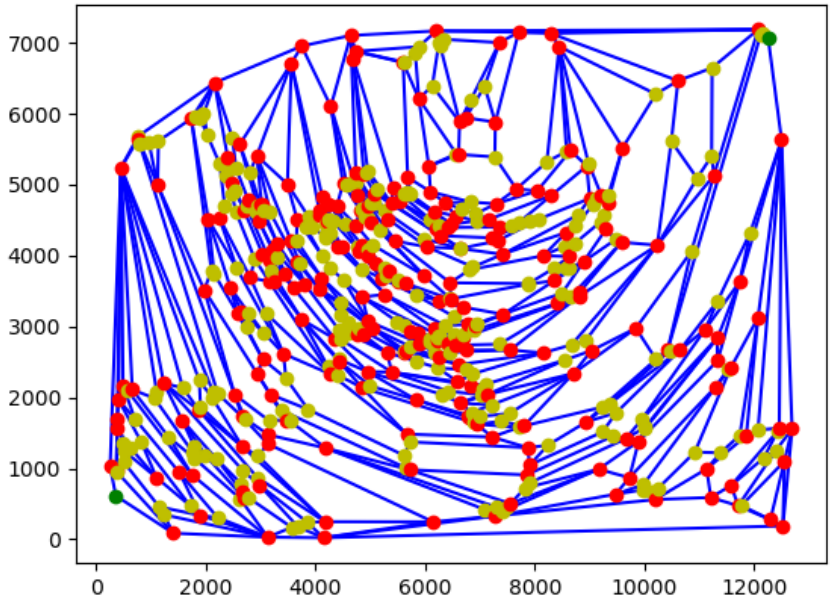

Figure 6: Convex wave result of the euro-night-500 instance. The vertices' color indicates their degree.

4 Merged Convex Waves

In an attempt to break up the inauspicious pattern produced by *single convex wave*, we devised a variant consisting of multiple wave-instances running in tandem. Whenever two of these collide, the instances are merged into one.

The intent was to maximize the desirable results *single convex wave* produced in the vicinity of their starting points whilst curtailing the aforementioned circular expansion.

In this chapter we illustrate this approach and discuss the surprisingly weak results.

4.1 Merging algorithm

Our applied procedure to merge two *single convex wave* instances conforms to following steps (see figure 7):

- **Calculate visible bounds:** Compute the outermost, mutually visible vertices on each hull.
- **Query for intermediate points:** Using the foregoing visible bounds, find all vertices that lie in-between both hulls which remain unclaimed by either.
- **Break up occluding hulls:** If any of these vertices are occupied by other wave-instances, they are cleared of all their connected edges.
- **Repair broken instances:** If any vertices were cleared in the last step, recalculate the convex hulls of the instances they belonged to and triangulate ruptures caused by the previous step.

- **Allocate intermediate points:** The points in-between both instances are divided into two domains by a line and correspondingly allocated to the respective instance.
- **Integrate intermediate points:** In accordance of the foregoing allocation, each intermediate vertex is integrated into their respective hull using the regular convex wave iterative procedure.
- **Connect instances:** By advancing along the mutually visible bounds, both hulls are connected.

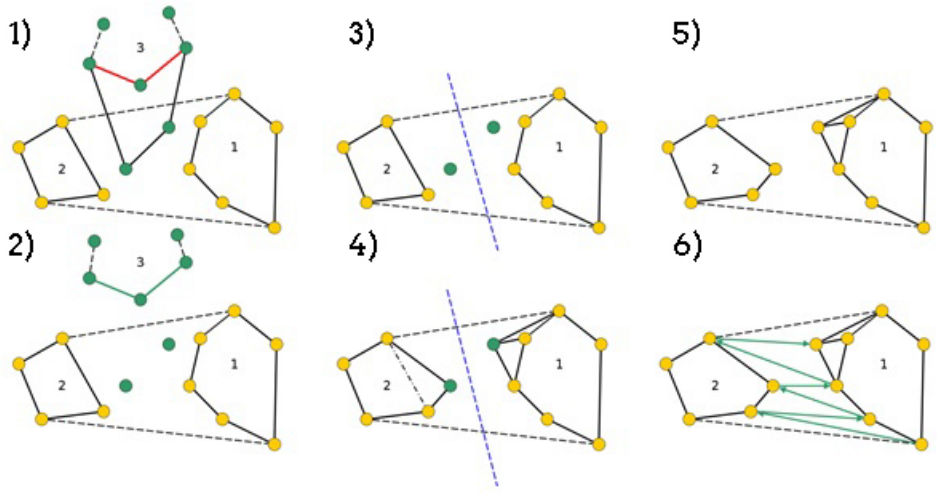

Figure 7: Steps to merge two convex wave instances

In practice, the implementation of a fully functional merging algorithm proved more challenging than anticipated. Two or more instances can be arranged in a plethora of unusual predicaments that are left unhandled by the above given steps. (For instance, an intermediate, occluding instance may span across both visible boundaries, leading to it being split into two parts.)

Such edge cases had to be dealt with on an individual basis. A few still remain unhandled in our final program, because early testing revealed that the *merged convex waves* approach was not worthy of any further pursuit.

4.2 Drawbacks

As the previous paragraph may suggest, the drawbacks of this particular algorithm are indeed manifold. Our central findings were that it produced on average 10% more edges than the best solutions computed by a *single convex wave*. In addition, optimization possibilities were scarce, making it not only our worst performing but also our slowest algorithm.

The causes of this lousy performance can be observed in figure 8. The merging steps produce non-circular instances. As a result, the loop invariant of the convex wave is no longer true and the effective radius of the instance is stiltedly stretched, putting subsequently integrated points into a disadvantageous predicament.

Our final program allows the execution of this algorithm, although none of our submitted solutions were computed by it.

5 Pass based Algorithm

In light of the foregoing, we made the decision to avert the complications of a merging step by opting instead for a *pass based* variant.

This approach performs a series of sequential procedures where the output of a previous step is used as input for the next. In order for this pipeline to function correctly, the input and output of each pass must conform to a strict specification, which we accomplish by employing several intermediate passes.

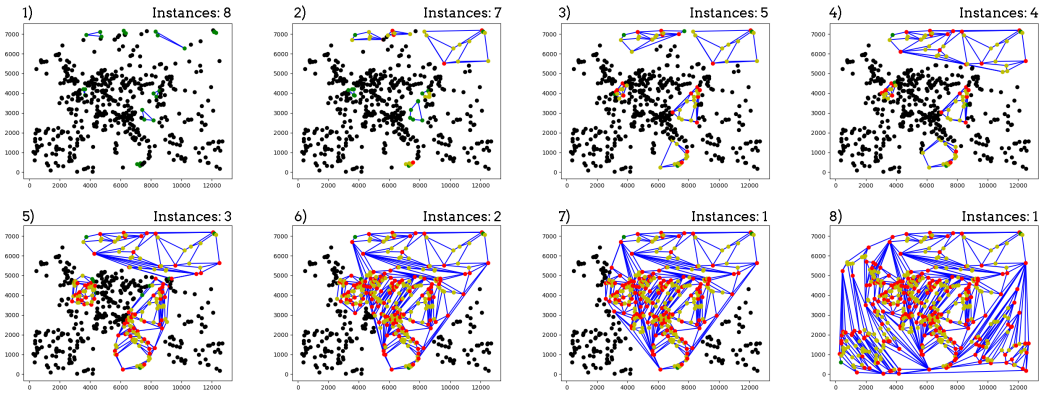

Figure 8: Iterations of the *merged convex waves* algorithm on a 500 point instance.

5.1 Pass description and specification

5.1.1 First Pass: Secure largest faces

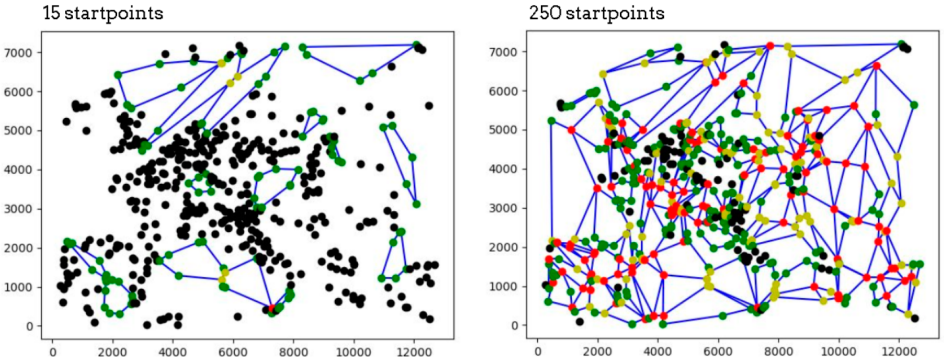

Figure 9: First pass after processing 15 (left) and 250 (right) starting points.

Input: *Set of vertices without any edges and a set of starting points*
Output: *Amalgamate of stray and connected vertices*

The first pass will attempt to build a simple, convex polygon around each starting point. Each polygon is grown as large as possible without causing polygon-intersections and having no stray vertices on their interior. (Example: No vertices can be added to the polygons in figure 9 (left) without causing intersections or leaving vertices on the polygons interior.)

The expansion of the polygons operates in much the same way as a regular *single convex wave* iteration. Although in this case only a convex hull is maintained and points which are occluded by other polygons or can see more than one edge are skipped.

Since the starting points are processed in the order they are provided, it is advised to sort them in the order of their favorability.

5.1.2 Second Pass: Gather stray points

Input: *Amalgamate of stray and connected vertices*
Output: *Convex hull of the pointset with its interior partitioned exclusively into simple polygons (without any intersecting edges or holes). Stray points are allowed in small numbers*

The bulk of the second pass performs exactly the same steps as the first one, but rather than a list of explicitly given starting points, polygons are grown for all vertices that are not yet connected to anything. This step is meant to gather any remaining stray points into smaller polygons and incorporate them into a single network.

In rare occasions, where the pass fails to create a triangle at a starting-vertex due to potentially intersecting edges, some stray points can still remain. These are typically quite few and will be processed separately at a later step.

There is a natural balance between the first and second pass: The more polygons are formed in the first, the fewer the second will have to create and vice-versa. Given that the

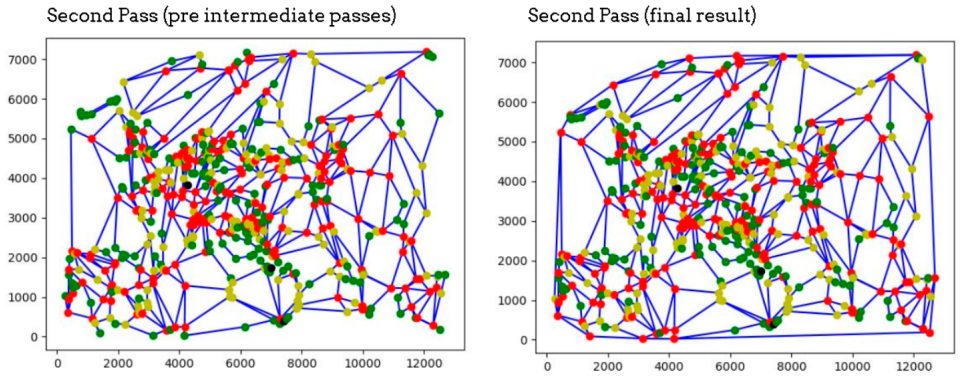

Figure 10: Second pass after gathering stray points (left) as well as incorporating the systems convex hull and integrating islands (right). The integration of an island can be observed in the top left (2000 | 6000).

first pass operates on a set of heuristically chosen starting points, it can generally be assumed that these will produce a better yield than leaving the first pass empty. However, having an excess of starting points in the first pass would lead to large amount of wasted computation time, as most of them would not even be able to provide a commencing triangle.

Through trial-and-error we assessed that utilizing roughly 10%-33% of available starting points (equating to an amount of approximately 25%-90% of n) gave the best results, though values closer to 10% also ran significantly faster.

After gathering the stray points, the resulting vertex network may still contain nested polygons (holes/islands) and does not necessarily span the whole of the convex hull (see figure 10). To rectify these issues in accordance to the output specification, we employ a series of two intermediate passes:

Intermediate Pass: Convex Hull

To ensure a fully encapsuled system, we incorporate the sets convex hull by performing a regular Graham scan and instantiating the respective edges.

Intermediate Pass: Integrate Islands

To finalize the second pass, we must take care of any holes - or islands - present in the current partition. These are detected by performing a rudimentary DFS traversal on any point lying on the convex hull which marks all vertices connected to it. Any non-stray vertices that remain unmarked must be part of an island and are thus integrated into their surrounding face.

To identify the face an island is located in, we employ a *ray casting algorithm* (also known as the *even-odd rule*): If a ray originating from a vertex on the island intersects a polygon an uneven number of times, the vertex, and by extension the island, is contained by that polygon.

Since we do not require directional rays for this task, we simply aver a horizontal ray each time, which massively simplifies the task of computing intersection points. Given an island-vertex v and a potentially containing polygon P, then E_1 is the set of edges in P that cross a horizontal line through v:

$$E_1 = \{e \in P \quad | \quad (e.origin.y < v.y) \neq (e.next.origin.y < v.y)\} \tag{2}$$

A regular orientation test then yields the set E_2 of edges in P that cross a horizontal line through v *and* lie to the left of v:

$$E_2 = \{e \in E_1 \quad | \quad (e.origin.y < v.y \wedge v \text{ is to the } \textbf{right} \text{ of } e) \vee (e.next.origin.y < v.y \wedge v \text{ is to the } \textbf{left} \text{ of } e)\} \tag{3}$$

E_2 now corresponds to the set of edges in P that are intersected by a horizontal, left-bound ray originating from v. Consequently, if the size of E_2 is not divisible by 2, then v is fully contained by P.

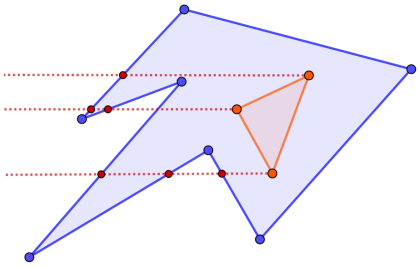

Figure 11: The *Even-Odd rule:* If a ray originating from a vertex intersects a polygon an uneven amount of times, the polygon surrounds that vertex (and by extension island).

Once the corresponding face to each island has been found, we iterate over its boundaries and the islands respective boundaries until two mutually visible vertices are found, which are subsequently connected, concluding the integration.

Once this intermediate pass is concluded, the solution consists exclusively of simple, non-intersecting polygons which are all connected by a network. To conclude the partitioning, any remaining non-convex vertices (henceforth *inflex points*) need to be processed in the next pass.

5.1.3 Third Pass: Resolve inflexes

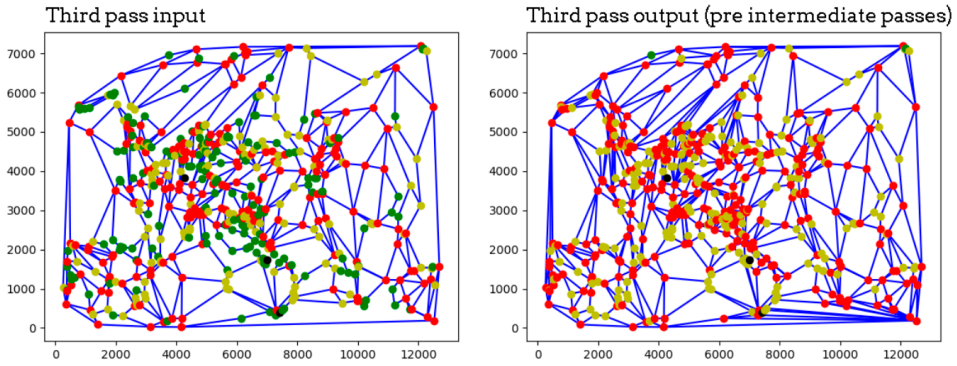

Figure 12: Left: Third pass input (output of second pass). Right: Third pass after resolving inflex points, leaving only convex faces and the stray points from the input.

Input: *Convexly enclosed vertex network consisting exclusively of simple, non-intersecting polygons. (In some cases: a few remaining stray points)*
Output: *Fully realized convex partition of the underlying pointset with some redundant edges.*

Inflex points are resolved on a face-by-face basis. However, since our basic DCEL data structure does not actively maintain a list of faces, these have to be aggregated manually at this point in the algorithm. We do this by simply traversing the bounding edges of each face individually (by means of the 'edge.next' pointer), saving them as a list and marking them. This process is repeated until no more unmarked edges remain.

Once a set of all faces has been established, we resolve the respective inflex points by performing a raycast along its bisection line. Whichever edge the ray collides with is classified as the *opposing edge AB*.

Figure 13 portrays examples of the cases handled in order of their priority:

- a) If either A or B is visible from the inflex vertex and lies inside of the bisection-angle, then the inflex is resolved by simply connecting to that respective vertex.
- b) If A and B are both visible, but do lie outside of the bisection angle, the inflex vertex is connected to both.
- c) If either A or B is obstructed, then there are still other unresolved inflexes in the currently processed face. We append this inflex to the back of the queue and resolve all other inflexes first, before attempting to resolve this one again.

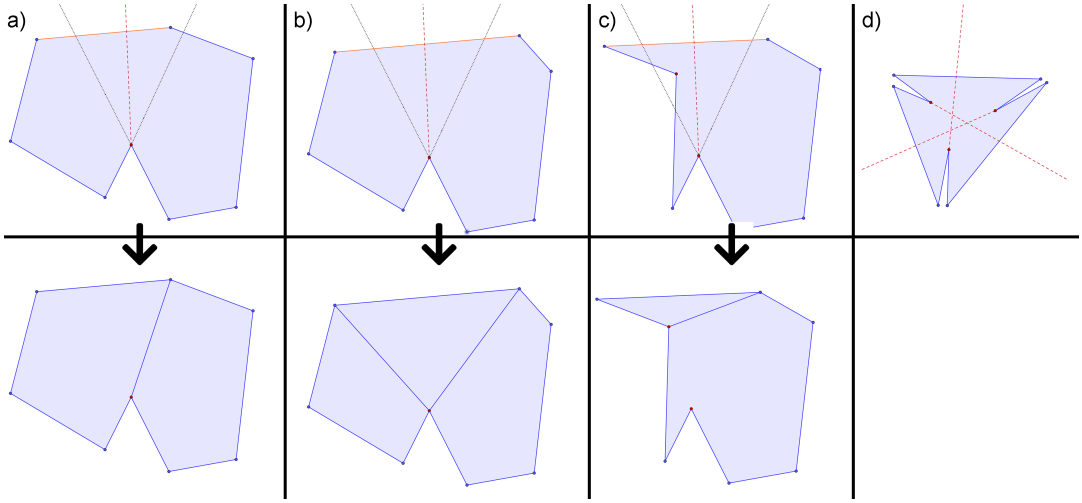

Figure 13: Possibilities of an inflex-bisection in order of their beneficiality (left-to-right).

- d) Almost all inflexes fall into one of the previous three categories. However, there is the possibility of a *deadlock scenario* (see figure 13 d). Albeit quite rare, here a number of inflex vertices will continuously queue up waiting for each other to be resolved. To break up this deadlock we simply connect any of the inflex vertices to *any* other vertex on the face that is visible, effectively dividing the polygon into two, which are processed individually.

Once the inflex resolution has concluded, only convex faces will remain. This significantly simplifies the task of integrating any stray points that may have endured from the first two passes:

Intermediate Pass: Integrate stray points

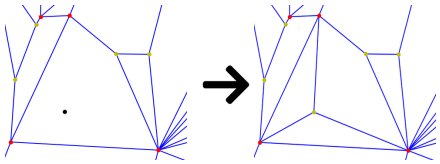

Figure 14: Close-up example of a stray point integrated into its surrounding face.

Working under the assumption that all remaining faces are convex makes the task of finding a surrounding face to a stray point trivial. Similarly, the integration procedure can also be performed in a single edge-loop, connecting only to the furthest vertices within a 180 degree angle of the previous.

5.1.4 Fourth Pass: Cleaning Pass

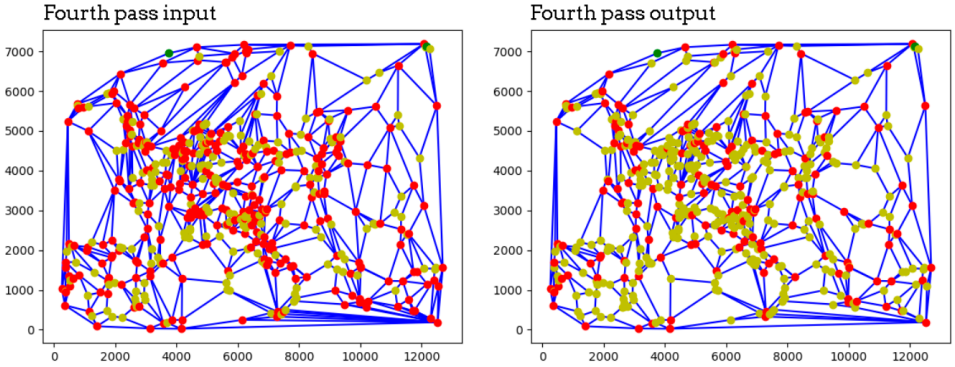

Figure 15: Fourth pass input and output (final result). Note in particular the high amount of red vertices turned yellow (vertices of degree 4 or higher to degree 3).

Input: *Convex partition*
Output: *Convex partition with no redundant edges*

In order to remove any unnecessary edges produced by the previous passes we iterate once over each edge and test it for redundancy.

This step allows for the introduction of a back-tracking algorithm to maximize the amount of removed edges, yet from our assessment this would strongly impact runtime and only produce a low yield of improvement. Thus we process the edges once in the arbitrary order they are stored in.

5.2 Performance

Our *pass based* algorithm follows a complex set of steps and procedures, making it significantly slower than *single convex wave* or *nested hulls*. Yet, due to its promising results we applied a plethora of optimizations that vastly improved its effective runtime.

These include optimizations to datastructures, heuristic information like circle-intersections in the first two passes, arbitrary cut-off points and some multi-threading. This dramatically reduced computation time by several orders of magnitude, allowing us to run the algorithm for all instances of the competition within a reasonable timeframe. (Example: a 5000 large instance would drop from 2 hours to under 2 minutes of runtime on the same hardware)

In terms of produced edges, it significantly outperformed both *single convex wave* and *nested hulls* on most instances also producing the most evenly spread out partitions of any of our algorithms.

6 Generating better starting points

In this chapter we will first discuss the generation of starting points to initialize the previously discussed algorithms.

We have implemented five of the most promising approaches that we considered for comparison. We will outline two of these which were excluded at an early stage. Subsequently, we will discuss the functionality of the more elaborate approaches. Finally, we give an overview of the effectiveness of each approach using the above described *pass based* algorithm.

6.1 Can specific starting points improve the solutions?

The emergence of the possibility to start our algorithms at various locations gave rise to the question of whether if it is advantageous to start at specific points.

For this chapter we distinguish between instance points, which describe the points given for an instance and are part of the solution, and the starting points, which are generated in a separate step before the actual algorithm. These starting points are used to initialize the *merged convex wave* and *pass based* algorithms, but are not part of the solutions.

6.2 Dismissed Approaches

We initially considered two ideas. For one, starting points in large open spaces may be preferred, and for another, starting points within clusters of instance points might provide better results. To explore these ideas we explored the following algorithms to improve our understanding of the starting point distribution.

6.2.1 Clustering

In a clustering approach, polygons would be created within clusters first instead of large areas. A general estimate on the number of edges of a polygon is difficult to make. Most likely, the number of edges per starting polygon will not be minimized by this method. However, this could be useful when using a merge function as in *merged convex waves*, since the boundaries of neighboring polygons are reached quickly and thus have to be merged earlier, which should be advantageous for our implementation of the merge. This estimation is speculative as we have not done any detailed studies using *merged convex waves*.

We have researched various known algorithms for clustering points, including k-means. This algorithm had the advantage of existing as a library function and of converging on its own, thus being executable unattended on many instances.

Unfortunately, k-means always converged rapidly and generated less than ten starting points on instances with several thousand instance points. Even after tweaking the parameters unreasonably the problem persisted.

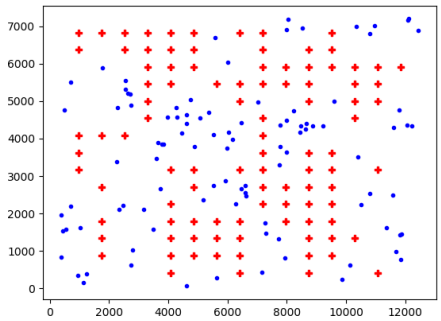

Figure 16: Too many starting points in heatgrid

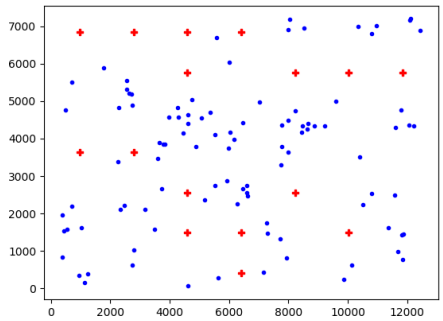

Figure 17: Not enough starting points in heatgrid

6.2.2 Heat-grid

The generation in open spaces should have the advantage that large polygons are generated first, both in area and in the number of edges. However, our initial experiments in *merged convex wave* have shown that merging large polygons has a large administrative overhead and leads to poor results. Consequently, we originally assumed that this approach is preferable for the *pass based* algorithm.

In order to find large free areas and create starting points in them we experimented with heat-grids by creating these in different ways. The area covered by the instance points is initialized with starting points equally spaced horizontally and vertically at the intersections of an imaginary grid. The algorithm deletes those starting points from the grid which are too close to an instance point.

Unfortunately, it turned out immediately that both parameters, distance of grid intersections and distance to instance points, are strongly dependent on the instance. After several attempts we could not find a good method to calculate these parameters automatically. Resulting problems are illustrated in figures 16 and 17. Moreover, the results for very large instances could no longer be evaluated with plots, which led to a certain skepticism about practical results based on the experience gained on small instances.

6.3 Triangulation

In our more advanced approaches, we based the algorithms on a *Delaunay triangulation*, which we computed using a publicly available library function that produced deterministic results, easing their comparison. The computation of a Delaunay triangulation is doable in a runtime of $O(n \log n)$. Below we provide three possibilities to generate the starting points using this triangulation. Afterwards we will compare these methods in runtime and distribution of the starting points.

The first two algorithms place starting points in the centroids of the triangles obtained from the Delaunay triangulation. These two algorithms differ only in the metrics used to prioritize the starting points. The prioritization results in an order of execution in which supposedly better starting points are used first. The centroid is calculated by the arithmetic mean of the vectors to the vertices of each triangle. This function was compiled by a *just in time* compiler and could be accelerated significantly. The used compiler, *Numba*, optimizes the Python bytecode right before the first execution of the function and compiles it into machine code. Each subsequent execution of this function is then executed with this compiled version. It is very efficient on frequently used math functions, but does not provide support for complex datatypes, hence why it is only sparsely used in our project.

6.3.1 Triangle Area

The triangle area is a metric where the starting points in the centroids of the triangles are prioritized by area of the triangle.

Since the triangles are generated by the library function of the Delaunay triangulation, this algorithm does not require significant additional effort. The area of each triangle is calculated by half of the determinant of two edges of a triangle. This is again a simple

mathematical function on the triangle's coordinates, which could be optimized with the just in time compiler. Finally, the starting points were sorted according to the corresponding triangle area.

This method produces a good distribution of the starting points in open spaces. Clusters are given low priority, but this method still allows for a sparse distribution within crowded areas. As mentioned before, in this way larger polygons by area and number of edges should be generated first.

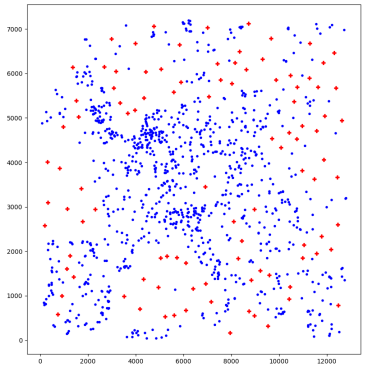

Figure 18: Starting point distribution by triangle area

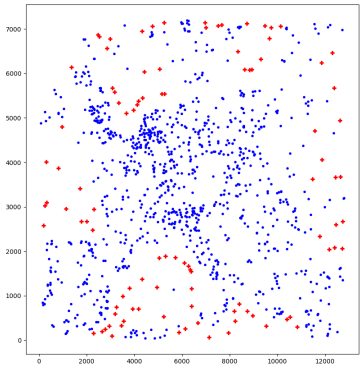

Figure 19: Starting point distribution by triangle span

6.3.2 Triangle Spanning

The span of a triangle is a metric where the longest altitude on the sides of the triangle is preferred. The idea is that this will avoid elongated border areas between polygons (resp. convex graphs) by first creating polygons (resp. convex graphs) within these areas. From the experience with our *merged convex wave* algorithms we know that these areas give worse results when the starting point to a convex graph is wither far away or merging becomes necessary, because in both cases many elongated edges are generated.

To avoid the additional computational effort to determine the altitude vector, we initially chose to approximate it from the known centroid to the most distant triangle vertex. After examining the results, however, we found this method to be poorly adjustable and rejected it. Nevertheless, we were able to develop the following algorithm based on these experiences:

6.3.3 Topology Starting Points

Conclusively, the final algorithm with which we calculated the starting points for all instances, is also based on the same Delaunay triangulation, in this called *instance point triangulation* (IPT). In this metric the starting points are located at the midpoint of each edge of the IPT.

These starting points – at the midpoints of the IPT – are then prioritized as follows: A second Delaunay triangulation is generated between those starting points, in this called *starting point triangulation* (SPT) (see figure 20), which consequently connects the midpoints of neighbouring edges of the IPT. The SPT is then transformed into a directional graph in which the edges are directed from the corresponding longer edge to the shorter edge of the IPT. Then the starting points at the centers of the edges of the IPT are sorted by the degree of outgoing edges in the directional topology graph, so that the starting points with the most outgoing edges are preferred. Meaning, that midpoints of longer edges are preferred as starting points over midpoints on neighbouring, shorter edges in the IPT. This is illustrated by an example in figure 23, showing a decent distribution both in open spaces as well as inside of clusters.

We decided on the naming of this algorithm from the fact that the outgoing edges can be represented in the third dimension (as in figure 21 and 22), thus spanning a three-dimensional topological curve in which the locally preferred starting points can be identified as peaks.

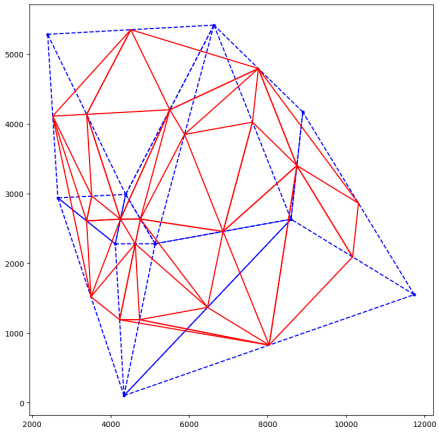

Figure 20: Instance point triangulation (blue) and starting point triangulation (red)

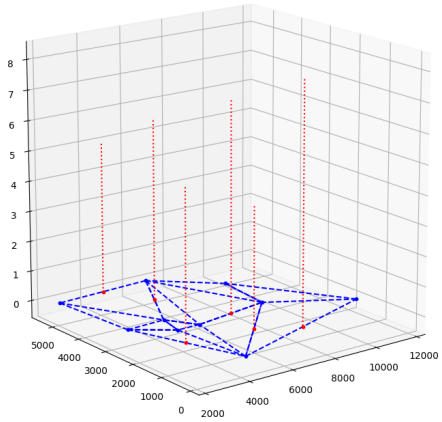

Figure 21: Best 6 starting points by edge degree (on z axis) in starting point triangulation

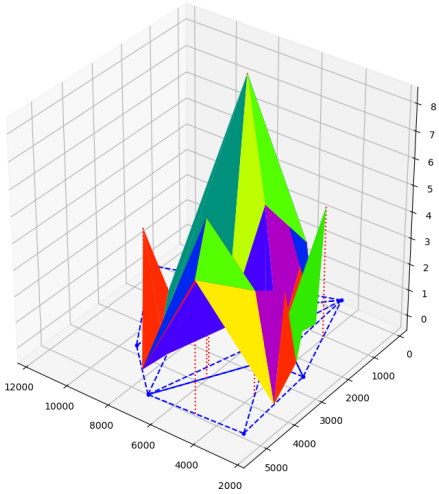

Figure 22: Topological curve based on starting point triangulation

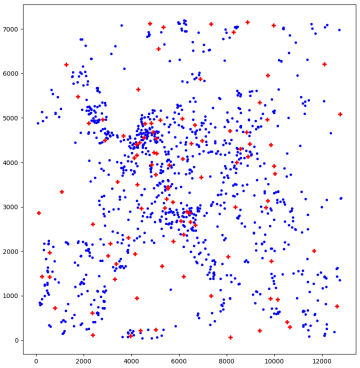

Figure 23: Starting point distribution by *topo start*

6.4 Results for pass based algorithm

The intentions of the experiments with starting points had been carefully considered, therefore we expected a noticeable improvement by these algorithms. Unfortunately, our high expectations were tempered by a strong measure of disappointment.

Comparing the results from running the *pass based* algorithm with our complex topology against using just four randomly chosen starting points, only a slight improvement was observable, and only if less than a third of the available starting points were utilized. This can be explained by the *pass based* algorithm itself, since in the second pass (see 5.1.2) – when there are still a lot of points unconnected, but no more available starting points – the remaining instance points are used in the given sort order to start further polygons at this point. It appears that this procedure was already quite efficient and only a few polygons can be calculated more efficiently with well selected starting points. A comparison with the area and span metrics did not show any noticeable improvements.

In this respect, the significant, preceding computational overhead in conjunction with the *pass based* algorithm can be rated as very cost-intensive.

7 Result comparison

The charts in figure 24 portray the number of instances solved by each of the three algorithms in our final submission. Note that in the case of two equally good solutions, the left one is kept. As a result, the amount of solutions provided by *single convex wave* may be somewhat over-represented.

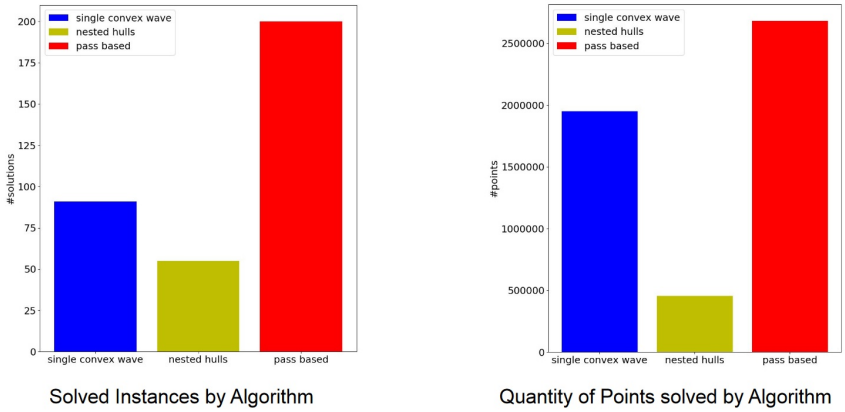

Figure 24: Solutions by algorithms compared by number of instances and by number of instance points. The peak in *single convex wave* in instance points is explained by it solving the 1,000,000 points instance best.

7.1 Strengths and Weaknesses

7.1.1 Random Pointsets

When confronted with inputs based on isotropically spread points as well as image- or brightness-based, our *pass based* algorithm proved to be an indispensable benefit to our overall performance. It vastly outperformed both other algorithms, albeit at a significantly higher computational cost.

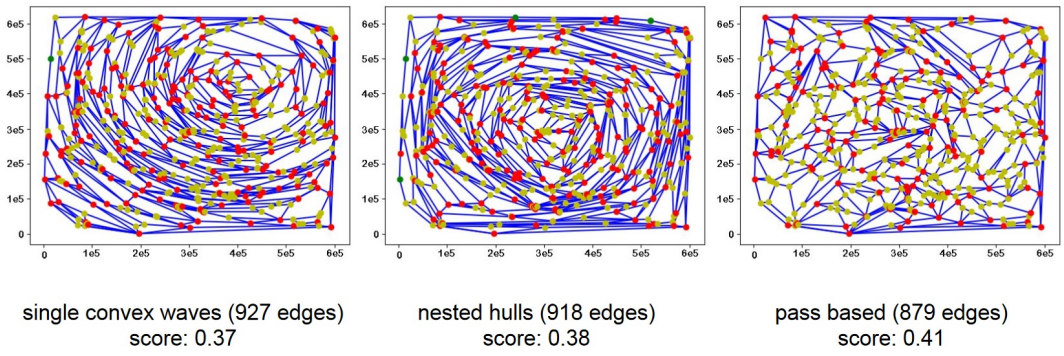

Figure 25: Example results for an instance of 500 randomly spread points (1480 triangulation edges)

7.1.2 Orthogonal Pointsets

For artificially assembled instances of mostly collinear points, the *nested hulls* algorithm proved highly invaluable. To the naked eye, figure 26 may initially appear as a victory for passed-based, but this is mostly due to the areas being stretched out more evenly. The *nested hulls* approach produces longer squares by combining multiple vertices into a single chain.

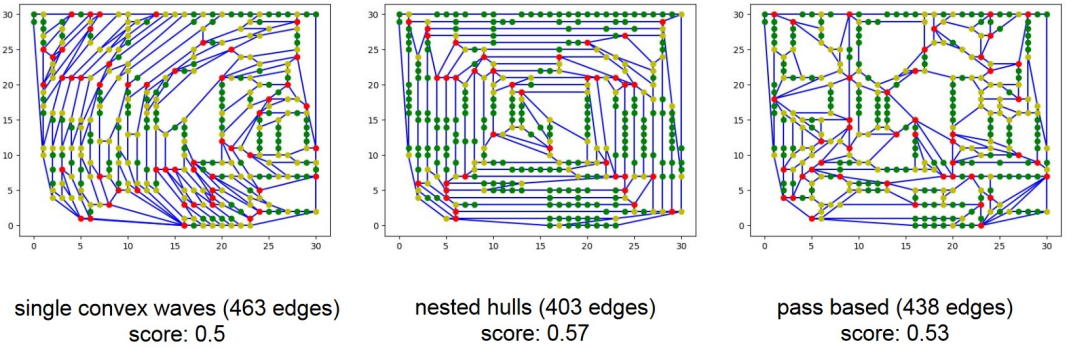

Figure 26: Example results for an instance of 326 points arranged on orthogonal lines (932 triangulation edges)

7.1.3 Computation Time

The *single convex wave* algorithm was the fastest and acted as a useful fallback for the few cases that *pass based* was unable to provide a solution for. The growth of computation time can be observed in figure 27.

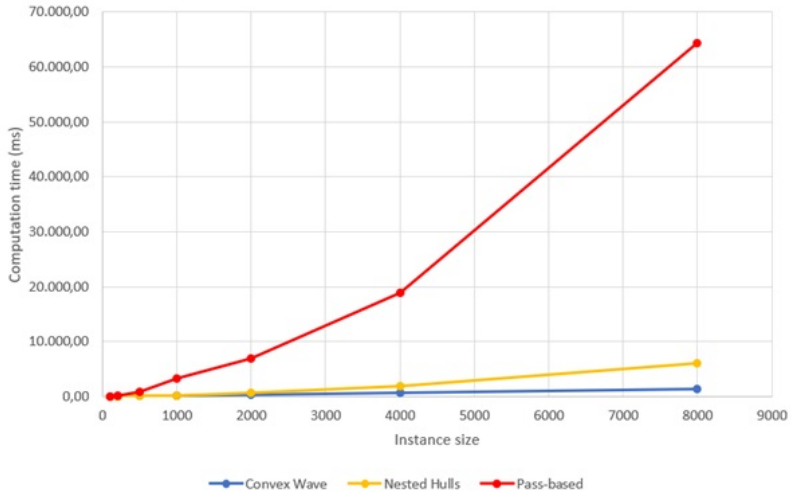

Figure 27: Computation time for euro-night instances of various sizes. (All performed on an AMD R1700)

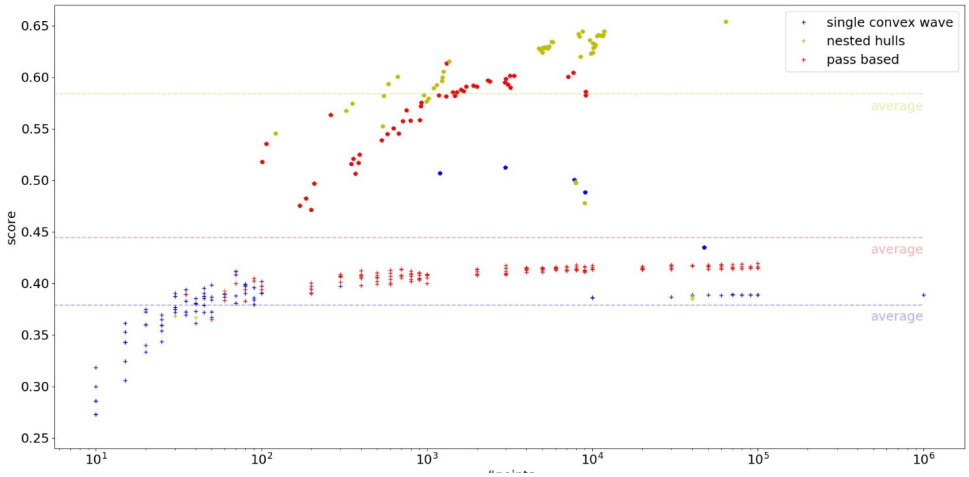

Figure 28: Our solutions to all orthogonal (diamond shape) and regular instances (plus shape)

Figure 28 shows our overall distribution of solutions. As can be observed, the *pass based* algorithm dominates throughout most of the larger instances. When it comes to instances with many collinear points (at the top) *nested hulls* and *pass based* are mostly at an impasse, with *nested hulls* scoring the majority of solutions.

7.2 Further Considerations

In addition to the four algorithms presented in this document, we also gave consideration to a number of alternative methods. Amongst these were:

- **Flipping and deleting edges from a triangulation:** On paper we developed a flipping edges algorithm utilizing depth first search and backtracking, which is able to find optimal solutions. However, since the number of possible triangulations is $O(59^n n^{-6})$ [3], backtracking is not better than brute force and consequently we did not consider flipping and deleting algorithms at all. In the end of this project we explored another flipping edges algorithm which starts at a random triangle and then searches for the best local option by deleting the most edges from this triangle and all adjacent triangles. Given that we found this algorithm near the end of the contest, we did not implement this approach.

- **Game Theory:** We considered game theory for a flipping edges algorithm as well and discussed following approach: Players are assigned a polygon of a triangulation and they are allowed some actions flipping, placing or deleting edges that belong to this polygon. The player would then move to an adjacent polygon, until all Polygons have been tried by at least one player once. A polygons visited state is reset when it is changed without a player standing in it (by changing an adjacent polygon). Introducing resources per round and specific costs to actions could allow further tweaking of a players strategy. Players intend must be to delete as many edges as possible, though by placing edges it might enable other players to delete more edges than got placed. However, inventing a set of rules to generate a reasonable competition between players seemed too extensive to further explore this approach.

- **Competing Neural Networks:** Due to their novelty and well provided python-implementation, we considered using neural networks (NN) to tackle the previous problem.

  In our approach a NN would take the actions and rules as explained in the last chapter. That way different strategies would be developed automatically and could be studied by examining the behaviour of the NNs.

- **Attacking the checker:** Since we had insight into both the C++ and python libraries' source code, which were used for the checker software, we considered attacking the checker itself. In this case we would try and find and exploit a flaw or bug in the checker to achieve extreme good results with malicious solutions. This idea rose after we found, fixed and reported a bug in the checker software. However, even though we expect bugs of some usable nature in the checker we did not follow up on this idea. We only discussed possible procedures and decided it makes sense to hand in such malicious solutions only minutes before the end of the contest to avoid the jury finding and fixing any exploits with the help of these solutions. Anyway, we expected this approach not to be accepted by the jury and abandoned the idea.

7.3 Verdict

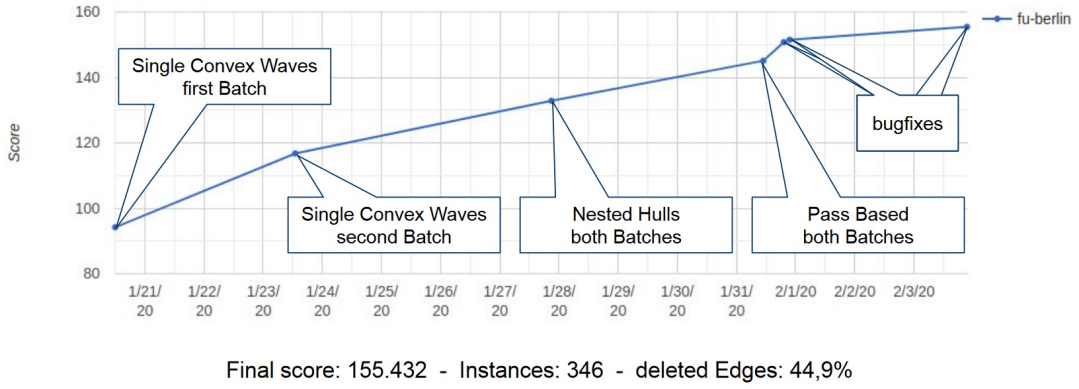

Figure 29: Milestones of the project and corresponding score

Overall our group achieved place 15 of 25 in the competition. With a score of 156 and the best teams having 175. Compared to other teams our results on instances with many collinear points were significantly better than those with uniformly distributed instance points.

Evidently better approaches are available when the goal is only to minimize edges. Unfortunately we have no further information on the runtimes of other teams algorithms compared to ours.

References

[1] B. KAHL, S. KERNER, K. JAEHNE, A. MURREY, *Github repository of implementation*, 2019/20. URL: `https://github.com/SemjonKerner/convex_polygons.git`.

[2] ERIK DEMAINE, SÁNDOR FEKETE, PHILLIP KELDENICH, DOMINIK KRUPKE, JOE MITCHELL, *Competition website*, 2019/20. URL: `https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2020/`.

[3] R. S. FRANCISCO SANTOS, *A better upper bound on the number of triangulations of a planar point set*, N/A, 102, Issue 1 (2003), pp. 186–193. URL: `https://www.sciencedirect.com/science/article/pii/S0097316503000025`.

[4] S. HAR-PELED, *On the expected complexity of random convex hulls*, N/A, (2011). URL: `https://arxiv.org/abs/1111.5340`.

[5] M. V. K. M. O. MARK DE BERG, OTFRIED CHEONG, *Computational Geometry: Algorithms and Applications*, vol. 3rd edition, Springer, 04 2008.