AN HEURISTIC FOR MINIMUM/MAXIMUM AREA POLYGONALIZATION PROBLEM

João Fontes Gonçalves
Final project - INF562
École Polytechnique
joao.fontes-goncalves@polytechnique.edu

April 5, 2020

1 Introduction

Given a set S of n points in the plane, we want to compute a simple polygonalization of S (a simple polygon whose vertex set is precisely the set S) that has maximum or minimum area among all polygonalizations of S. Every set S of n points in the plane has at least one polygonalization. The number of different polygonalizations is finite (though possibly exponentially large) for any finite points set S. The optimization problems, both for maximization or for minimization, are known to be NP-hard. Here, we implement a greedy heuristic to find an approximation for it.

2 Method

The algorithm for the construction of a polygonization is based in the heuristic **SteadyGrowth**, proposed in (Auer and Held, 1996) for the generation of random polygons. The pseudo-code for it is shown in figure 1, taken from (Taranilla et al, 2011).

Given a set S of points in the plane, in the initial phase the algorithm selects three points $p_i p_j, p_k \in S$, such that no other point of S lies within the triangle formed by p_i, p_j, p_k. After that, in each phase, the algorithm builds a polygon P_k, adding one feasible point $p \in S$. A point p is feasible if no remaining points of $S - p$ lie in the interior of the convex hull $CH(P_k \cup p)$ and there is at least one edge of P_k completely visible from p. In order to add the feasible point to the solution, an edge (p_j, p_k) from the set of visible edges must be selected and replaced with the edges (p_j, p) and (p, p_k).

In order to minimize/maximize the polygonalization area we first select the initial triangle with minimum/maximum area. Then, at each step we select the feasible point that provides the minimum/maximum increment of area.

3 Details

Code was written in C++. The geometric operations were all implemented from scratch. To get convex hulls, I implemented Grahan's scan algorithm, which calculates it with time complexity $O(n\ logn)$.

The check for points inside a convex polygon of size s was done in $O(s)$ by comparing the areas obtained when triangulating it from a vertex of the polygon and from the point that we want to test.

The check for the visible edges of a simple polygon of size n from a given point was done in $O(n)$ by testing the intersection of a segment that goes from the point we are analyzing and the midpoint of the edge that we want to check visibility and the other edges in the polygon.

The search for the initial empty triangle in the the initial set with n points was done in $O(n^4)$ (time to go through all triangles and check for emptiness).

Algorithm 1 ***BuildPolygonization***
$P_k \leftarrow$ *BuildInitialTriangle*(S)
S $\leftarrow$ S - {p / p $\in P_k$}
while (S $\neq \emptyset$) **do**
  FP $\leftarrow$ *FeasiblePoints*(S)
  p $\leftarrow$ *SelectFeasiblePoint*(FP)
  VE $\leftarrow$ *VisibleEdges*(p, P_k)
  (p_j, p_k) $\leftarrow$ *SelectEdge*(VE)
  AddPointSolution(p, (p_j,p_k), P_k)
  S $\leftarrow$ S - {p}
end while

The main components of the algorithm *BuildPolygonization* are described:

- *BuildInitialTriangle*(S): returns a triangle formed by p_i,p_j,$p_k \in S$.
- *FeasiblePoints*(S): returns the set of feasible points $p \in S$.
- *SelectFeasiblePoint*(FP): returns a feasible point $p \in S$.
- *VisibleEdges*(p,P_k): returns the set of edges of P_k completely visible from p.
- *SelectEdge*(VE): returns an edge (p_j,p_k)$\in VE$.

Figure 1: Quantity of points versus time (in seconds) for maximum search

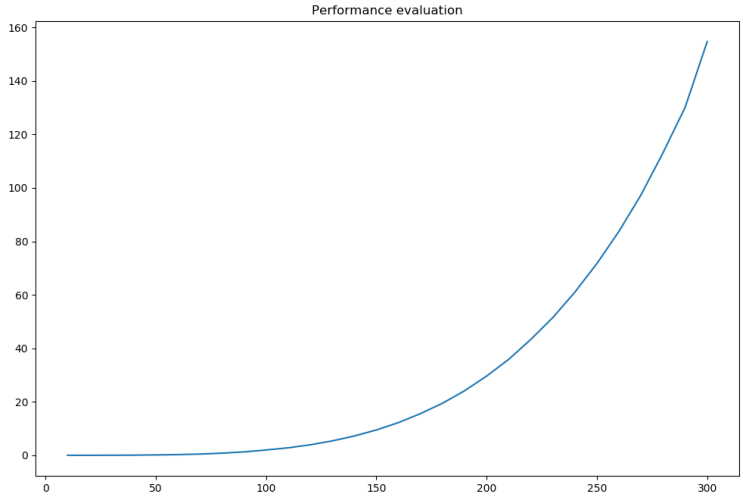

Figure 2: Quantity of points versus time (in seconds) for maximum search

If we are in a stage that we got a solution of size s we can expand it to a solution of size $s+1$ in time $O(n^2 s^2)$. Then, the total time until get the final solution is $O\left(n^2 \sum_{i=1}^{n} i^2\right) = O(n^5)$.

The final code has 624 lines.

4 Tests

In order to test running time, we generated random points around a variable radius circle (to avoid three or more collinear points). We test the running time on point sets with sizes in the set $\{10, 20, 30, 40, ..., 300\}$ and plotted a graph using a C++ Python's matplolib wrapper. The results are shown in the figures 2 and 3.

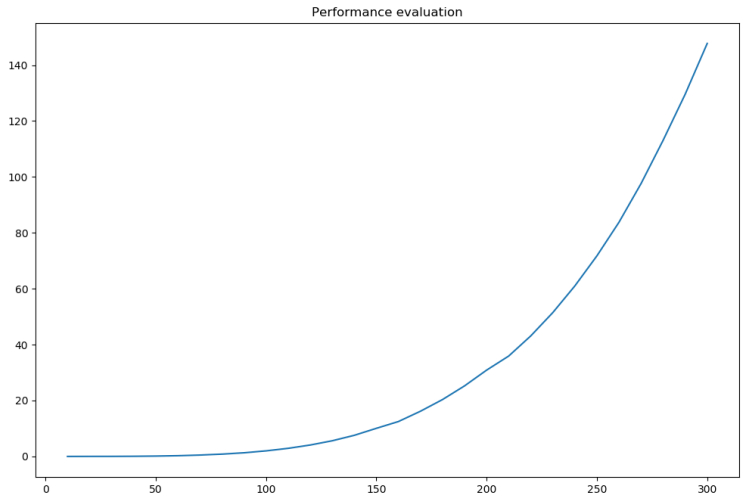

Figure 3: Quantity of points versus time (in seconds) for minimum search

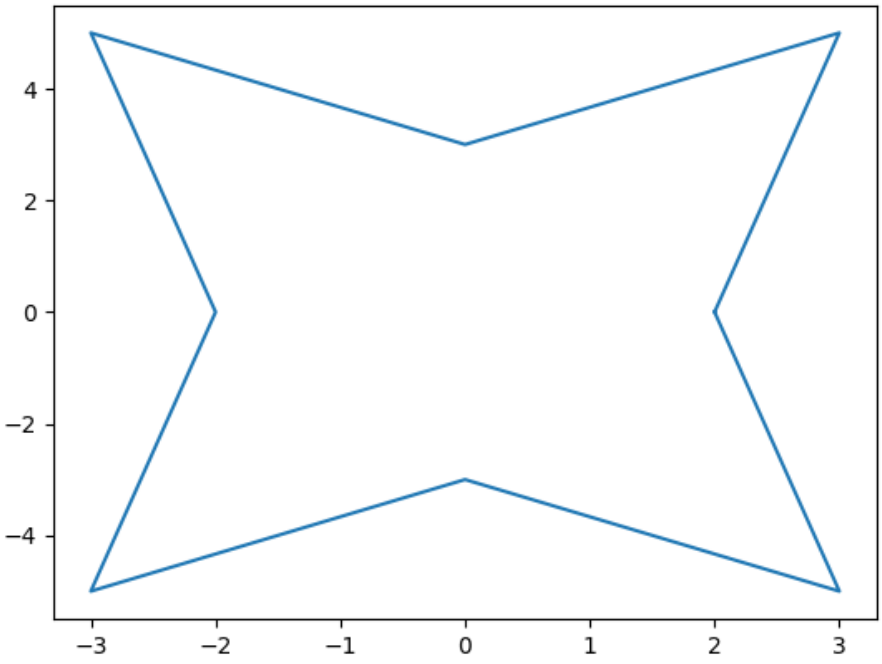

Figure 4: Solution for the maximum search

For illustrative purposes, we used an example with eight points. We can notice that the minimum and maximum solutions coincided with the optimal ones for this particular case.

The polygons are shown in figures 4 and 5.

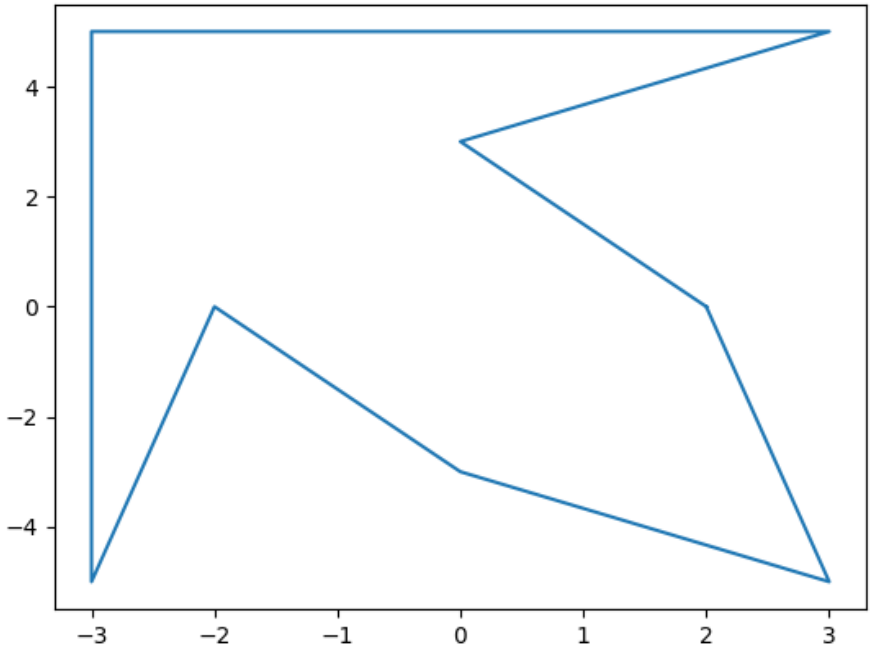

Figure 5: Solution for the minimum search