INDEX

RESUMEN

Los diagramas de Voronoi tienen aplicaciones tanto prácticas como teóricas en muchos ámbitos, la mayoría relacionados con la ciencia y tecnología, aunque también se aplican en otros campos como el arte visual. El objetivo de mi proyecto es estudiar el problema inverso del diagrama de Voronoi y diseñar, comparar y analizar diferentes estrategias para su resolución. Dicho problema consiste en detectar si una partición dada es o no un diagrama de Voronoi, y en caso afirmativo, calcular las semillas que lo generan. Para particiones que no lo son, sería interesante encontrar el diagrama de Voronoi que mejor se aproxima. Para ello, necesitaremos algún método para poder conocer la calidad de una solución y poder comparar varias soluciones. Usaremos la diferencia simétrica para tal fin.

Nótese que el número de soluciones candidatas es infinito dado que se trata de un espacio de búsqueda continuo. Probar todas ellas y elegir la mejor es por tanto inviable. Por tanto, trataremos de resolver el problema mediante diferentes metaheurísticas, es decir, trataremos de buscar una solución lo suficientemente buena, no la mejor.

El esquema básico de todas las estrategias es el siguiente:

- Para cada datos de entrada, calcular un conjunto de puntos semilla a partir de los cuales comenzaremos la búsqueda.
- Desplazamos los puntos ligeramente según algún criterio y comprobamos si hemos conseguido mejorar nuestra solución, actualizando ésta en ese caso.
- Repetimos el paso anterior hasta que no podamos seguir mejorando o hasta que la solución obtenida se considere lo suficientemente buena.

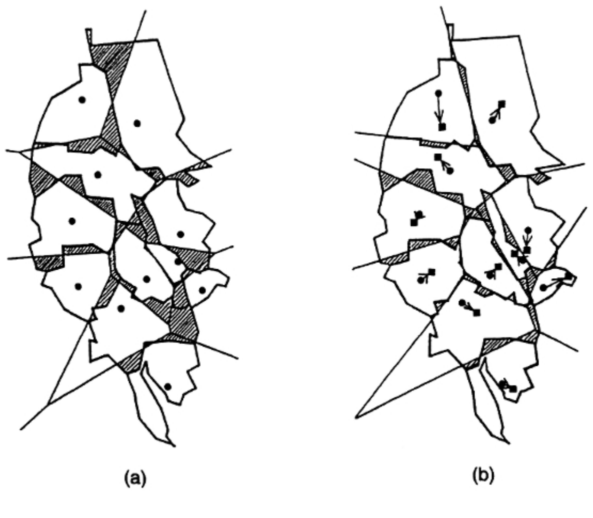


FIGURE 0.1. Distritos escolares en Tsukuba: (a) distritos escolares y ubicación actual de los colegios (círculos rellenos). Los estudiantes de las zonas sombreadas no pueden ir al colegios más cercano; (b) ubicación de los colegios (cuadrados rellenos) que minimizaría el área correspondiente a los estudiantes que no pueden ir al colegio más cercano. (Fuente: Suzuki and Iri, 1986a, Figure 14.)

El ajuste de una partición del plano mediante diagramas de Voronoi es útil para resolver problemas de optimización a la hora de colocar objetos con restricciones espaciales. Un ejemplo es el problema de colocar colegios atendiendo a los diferentes distritos escolares. La Figura 0.1(a) muestra los diferentes distritos escolares en Tsukuba (los círculos muestran la ubicación de los institutos). En Japón, los estudiantes de un distrito deben ir al colegio asignado a ese distrito. Como resultado, en algunas áreas los estudiantes tienen que ir a un colegio que no es el más cercano. Si asumimos que los estudiantes pueden llegar al colegio siguiendo el camino Euclidiano en la Figura 0.1, las zonas en las que los estudiantes no pueden ir al colegio más cercano aparecen sombreadas en la Figura 0.1(a), donde los polígonos se corresponden con el diagrama de Voronoi generado por la posición de los institutos.

El problema consiste en reubicar los colegios de forma que el número de estudiantes que no pueden ir al colegio más cercano es mínima. La Figura 0.1(b) muestra una colocación óptima local obtenida por Suzuki e Iri (1986a). La proporción de área de estudiantes que no pueden ir al instituto más cercano se reduce de un 20% a un 10%.

SUMMARY

Voronoi diagrams have practical and theoretical applications to a large number of fields, mainly in science, technology and visual art. The aim of my project is to study the inverse Voronoi diagram problem and design, compare and analyze different strategies for its resolution. The inverse Voronoi diagram problem consists on detecting whether a given plane tessellation is a Voronoi diagram and finding the seed points that would generate such a tessellation. For a tessellation which does not come from a Voronoi diagram, it would be interesting to find the best fitting Voronoi diagram. At this point, we need a way to measure how good a candidate solution is. We will be using the total symmetric difference between the two tessellations for that.

Note that despite the search space is bounded, it being continuous grants an infinite number of solution candidates. Therefore, we will try to solve the problem using different metaheuristics, which makes it impossible to tell whether the obtained solution is optimum.

The basic steps of all the strategies are:

- For each input, calculate a set of seed points from which we will start.
- Move each point slightly following some criteria and check if we improved the current best solution.
- Repeat last step until we cannot keep improving or we are satisfied with the result.

The method of fitting a Voronoi diagram to a given tesellation is also useful to solve the locational optimization of facilities whose use is spatially restricted. An example is the locational optimization of schools constrained by school districts. Figure 0.1(a) shows the districts of junior high schools in Tsukuba (the filled circles indicate the locations of the schools). In Japan, students in a school district are supposed to go to the school assigned to that district. As a result, students in some areas have to go to a school which is farther than the nearest school. If we assume that students can take the Euclidean path in Figure 0.1, the area in which students cannot go to their nearest schools is given by the hatched region in Figure 0.1(a), where the polygons are the Voronoi diagram generated by the school sites.

The locational optimization problem is to relocate the schools so that the number of students who cannot go to their nearest schools is minimized. Figure 0.1(b) shows the locally optimal locations obtained by Suzuki and Iri (1986a). The area in which students cannot go to their nearest schools reduces from 20% to 10% by this relocation.

Part 1. **INTRODUCTION**

Before we start, there are some concepts the reader should be familiar with in order to follow this document without problems. Those are:

- Partition of a set. Partition of the unit square.
- Voronoi diagram.
- Symmetric difference.

1. Partition of a set. Partition of the unit square

A partition of a set is a grouping of the set's elements into non-empty subsets, in such a way that every element is included in one and only one of the subsets.

For example, if we consider the set X = {1, 2, 3, 4, 5}, three different partitions could be:

- P1 = {1, 3}, {2, 4}, {5}
- P2 = {1, 2, 3}, {4, 5}
- P3 = {1}, {2}, {3}, {4}, {5}

However, the following would not be considered partitions of X:

- A = {1}, {1, 2, 3, 4, 5} - Not a partition because the element "1" belongs to more than one subset.
- B = {1, 2}, {4, 5} - Not a partition because the element "3" is not contained in any subset.

In this project, instead of sets, we will be working with tesellations of the unit square. That is, a set of polygons such that the union is the unit square and the intersection of two polygons is, at most, a segment. Note that there might be points contained in more than one polygon.

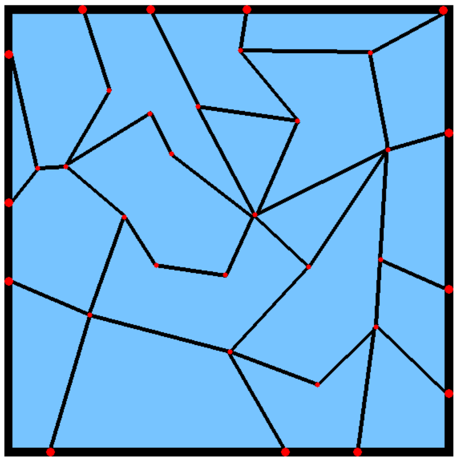


FIGURE 1.1. Tesellation of the unit square. As it can be seen, there are no overlaps or gaps between the polygons.

Mathematically, and in order to treat our tesellations as a true partition of the plane, we define the following sets:

- A = Interior points of every polygon in the tesellation.
- B = Points belonging to the line segments which devide each region, except both end points.
- C = End points of the line segments which devide each region.

Then, our partition T can be expressed as:

$$T = A \cup B \cup C$$

Figure 1.1 shows a partition of the unit square displaying each subset A, B, C in blue, black and red, respectively.

2. VORONOI DIAGRAMS

A Voronoi diagram is a partition of the plane into regions (called Voronoi cells) based on distance to points (called seeds or generators) in a specific subset of the plane.

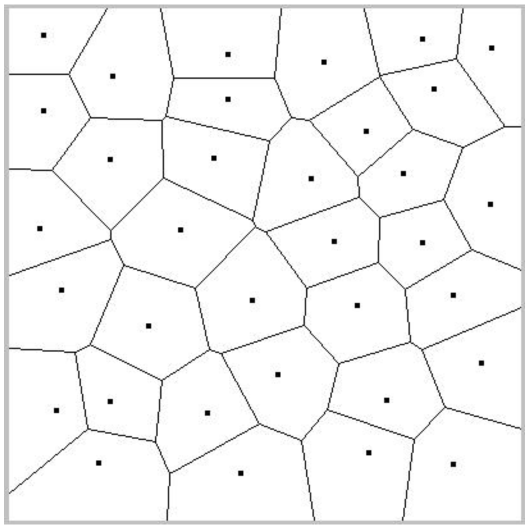


FIGURE 2.1. Visual representation of a Voronoi diagram inside the unit square. The black squares represent the seeds that generate the diagram.

In this project we will only be using the simplest case of Voronoi diagrams: the seeds are given as a finite set of points in the Euclidian plane.

Therefore, each region A_n can be described as the locus defined by the points P_k which are closer to some seed $S_n$than any other seed.

The lines that appear in a Voronoi diagram are the points of the plane that are equidistant to two or more of the nearest seeds.

3. Problem description

At this point, we can already state the problem we will try to solve.

Problem. Given a tesellation of the unit square consisting of convex polygons, find the Voronoi diagram that best fits it. Furthermore, if the given tesellation comes from a Voronoi diagram, we would like to arrive to that conclusion and obtain the seed locations.

More about how to determine how well a Voronoi diagram fits a tesellation can be read in the "Symmetric difference" section.

In order to solve this problem, we will use and compare different heuristic approaches. The general outline of the solving process of all the different approaches is the following:

(1) Calculate a starting Voronoi diagram V_1which will be our first approximation. We will be calculating this first solution from the given tesellation. More precisely, for each region of the tesellation, we will calculate a unique seed which will be used for generating V_1. As a consequence, every region of the tesellation will be related to a unique region of the Voronoi diagram, which means we will stablish a bijection between the given tesellation and our solutions regions.
(2) Apply different heuristic techniques to the current best solution, which generally involves slightly moving the seeds and checking if the new obtained Voronoi diagrams V_2, V_3, ... V_n fit better than the current best solution.
(3) Repeat this process until we cannot reach a better state. Return V_k, which will be the last obtained Voronoi diagram.

4. Symmetric difference

For measuring how good a given solution is, we will rely on the symmetric difference as an indicator. For two given sets A and B, the symmetric difference can be calculated as follows:

$$SD(A,B) = (A \cup B) \setminus (A \cap B)$$

For example, given the sets $C = \{1,2,3,4,5,6\}$and $D = \{2,4,6,8,10\}$,the symmetric difference of the two would be $SD(C,D) = \{1,3,5,8,10\}$. That is, the elements which appear in C or D, but not in both.

When applying this concept to two related polygons, it is clear that the symmetric difference would correspond to the area belonging to any polygon, but not both.

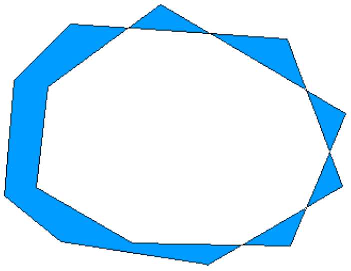


FIGURE 4.1. In blue, we can see a representation of the symmetric difference of two polygons. The highlighted area is the one that belongs to the symmetric difference. As we can see, it is the area that belongs to either polygon, but not both.

Applying the concept to two tessellations, we could calculate the symmetric difference as the sum of all the symmetric differences of each pair of related polygons divided by two (since we would be counting all twice).

Since in this project we will only be interested in working with convex polygons, we can use that to our advantage and calculate the symmetric difference in a different way: we will substract from the unit square area, the areas of all the intersections of each pair of related polygons. This will be explained more in detail later, in the development chapter.

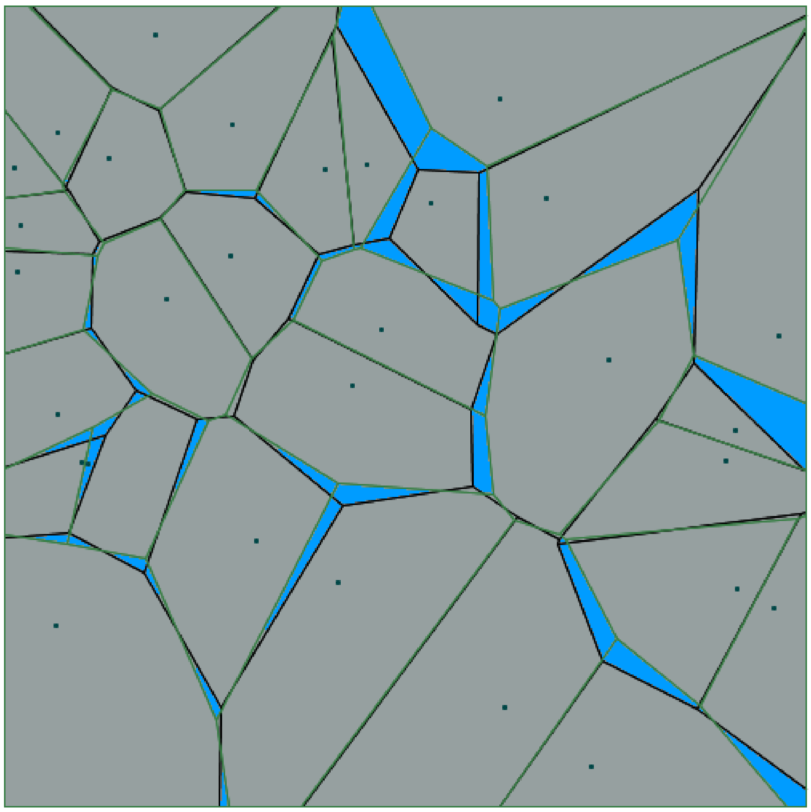


FIGURE 4.2. The new method would start with whole square painted in blue. We then subtract all the intersections of each pair of polygons. Graphically, the symmetric difference would be the total remaining blue area of the picture.

NOTE: In order to avoid issues with scaling, we will be treating the symmetric difference as a ratio with the total area of the unit square, so a 0% value will mean the two tessellations are exactly the same.

5. Inverse Voronoi problem

As mentioned earlier, the inverse Voronoi diagram problem consists on detecting whether a given plane tessellation is a Voronoi diagram and finding the seed points that would generate such a tessellation. From the mathematical point of view, it is interesting that, for tessellations that do come from a perfect Voronoi diagram, the problem can be solved in O(n) time, n being the number of seeds of such a diagram.

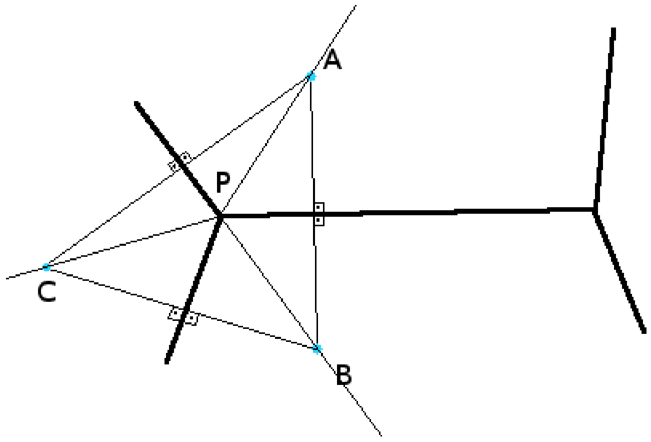


FIGURE 5.1. Region extracted from a Voronoi diagram. In blue, A, B and C are the seeds.

To prove it, we will start by assuming the tesellation comes from a perfect Voronoi diagram (all coordinates have infinite precision). For making the demonstration shorter, we will assume some point P of the diagram from which three edges come out has been found, even though the general demonstration follows the same logic. By definition, the distance from a pair of seeds to the edge they generate is always the same, as can be seen in Figure 5.1.

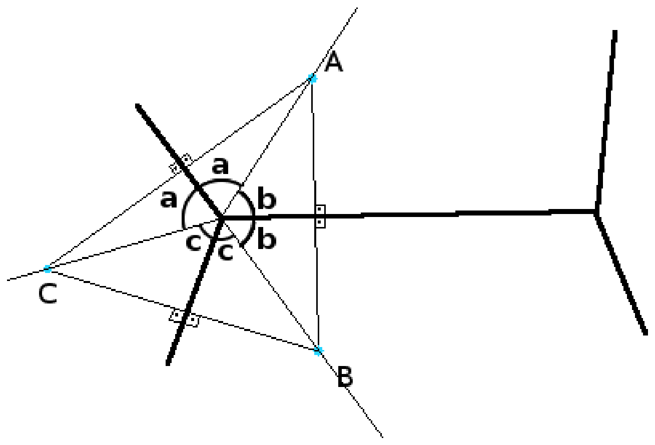


FIGURE 5.2. Angles formed by P, an edge and the lines that connect P with the corresponding two seeds of that edge.

Therefore, we know that the angles a, b and c formed by P, an edge, and the lines that connect P with the corresponding two seeds of that edge must be the same, as can be seen in Figure 5.2.

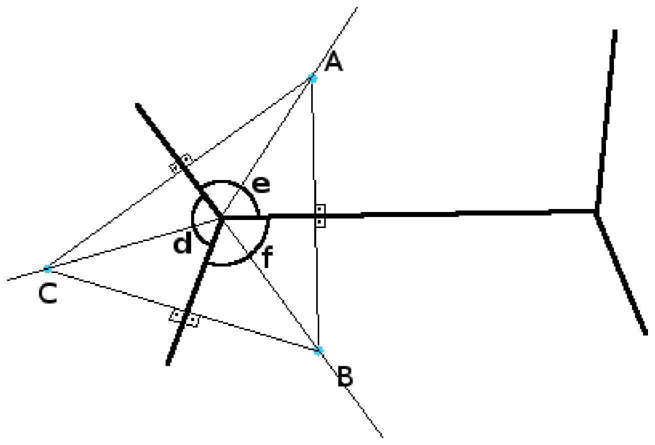


FIGURE 5.3. Angles between each pair of seeds.

Finally, let d, e and f be the angles between each pair of edges coming out from P, as seen in Figure 0.8. This values can be easily obtained knowing the points of the Voronoi diagram. With all these information we know the seeds A, B and C will have to lay on the lines with angles a, b and c to the corresponding edges. We can obtain the variables solving the following system of linear equations:

$$\begin{cases} d+e+f=2\pi \\ a+b=e \\ b+c=f \\ a+c=d \end{cases}$$

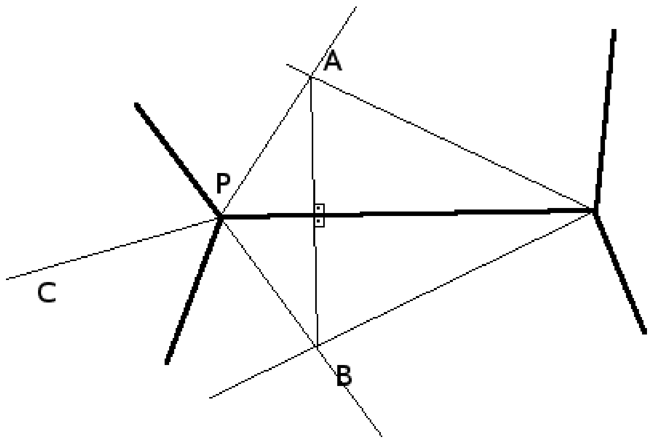


FIGURE 5.4. Seeds A and B have been located applying the explained method twice.

When we know the values for the angles a, b and c, we repeat the same process on a any vertex connected with P. As it can be seen in Figure 0.9, after knowing the angles for a neighbor vertex, the exact position of seeds A and B is obtained. From there, we can calculate any other seed by just mirroring the known seeds with the surrounding edges.

But what happens when the tessellation does not come from a Voronoi diagram? Even if it did, due to computers inaccuracy when representing floating-point numbers, the input data we have will not perfectly be the one that would be generated from a Voronoi diagram, but rather an approximation of it. If we applied this method, the location of the first seed would be off to its real value by certain offset ε. When calculating the rest of the seeds, on each iteration, that little offset would be multiplied by a certain constant, making it more inaccurate as the number of calculated seeds increases.

Therefore, we want to explore different methods for fitting an arbitrary tesellation, whether it comes from a Voronoi diagram or not.

6. Generalized inverse Voronoi problem

Another problem related to the ones exposed, but beyond the scope of this project, is the generalized inverse Voronoi problem. This problem consists on, given a plane tesellation, calculate a set of seeds such that the Voronoi diagram obtained from those seeds contains the original tesellation as a subset.

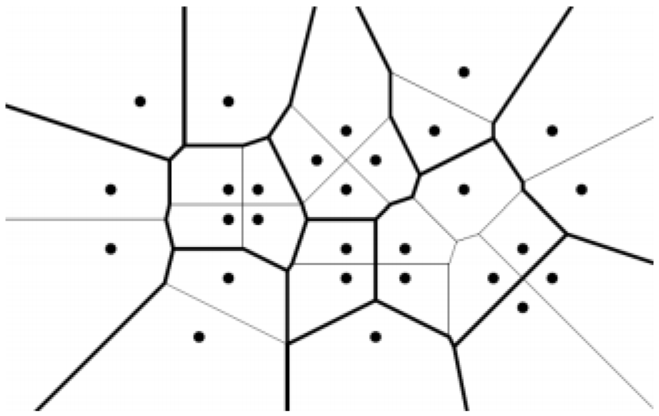


FIGURE 6.1. Representation of the generalized inverse Voronoi problem. Thick edges represent the original input tesellation.

Figure 6.1 shows how this problem would look like. The given input would be the tessellation represented by the thick edges. Then, as we can see, considering the dots as the seed placement, the obtained Voronoi diagram would be the union of the thick lines and the thin lines. As we can see, the original input is contained in the final Voronoi diagram.

7. Voronoi diagrams in real life

If one is familiar with Voronoi diagrams and has the chance to go for a walk in Madrid, they might find themselves in front of the “Teatros del Canal”. Something will attract their attention: some parts of the decoration of the building resemble those diagrams! In the “Results and Observations” part, we will adjust them and discuss how well they fit as a Voronoi diagram.

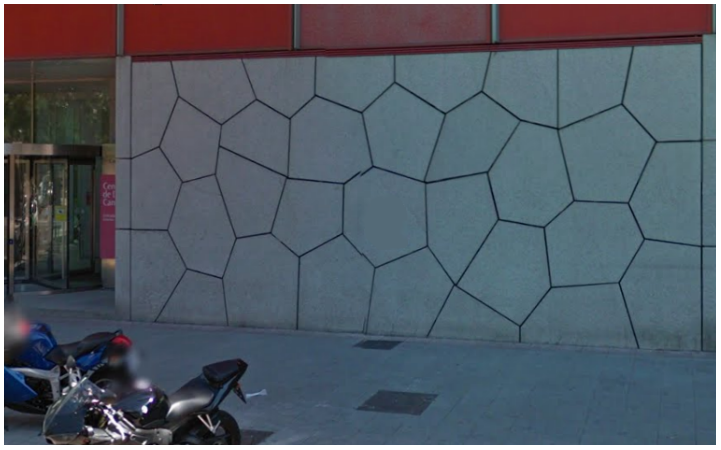


FIGURE 7.1. “Teatros del Canal” facade. It “clearly” resembles a Voronoi diagram.

In order to be able to work with that tessellation, we will need to convert it into our format. This will also be explained later, in the “Input data” section.

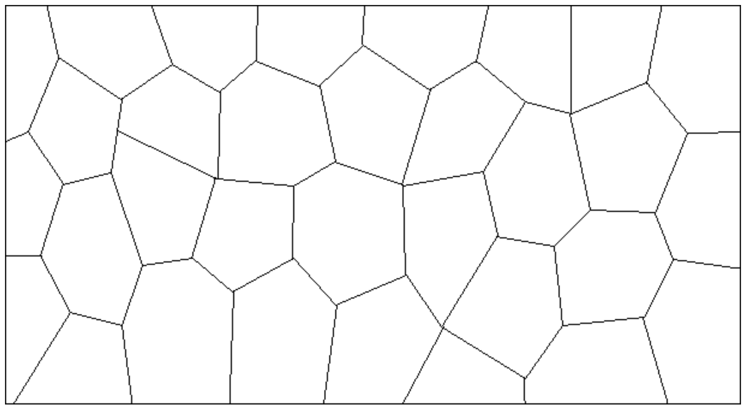


FIGURE 7.2. “Teatros del Canal” facade transcription. We will use this later to see and compare the results of the different fitting methods.

Voronoi diagrams have also been observed in nature. The mouthbreeder fish is a fish known for being highly predatory and extremely territorial. A research by Suzuki and Iri in 1986 showed that when located in a fishtank, the areas guarded by each fish fitted almost perfectly into a Voronoi diagram.

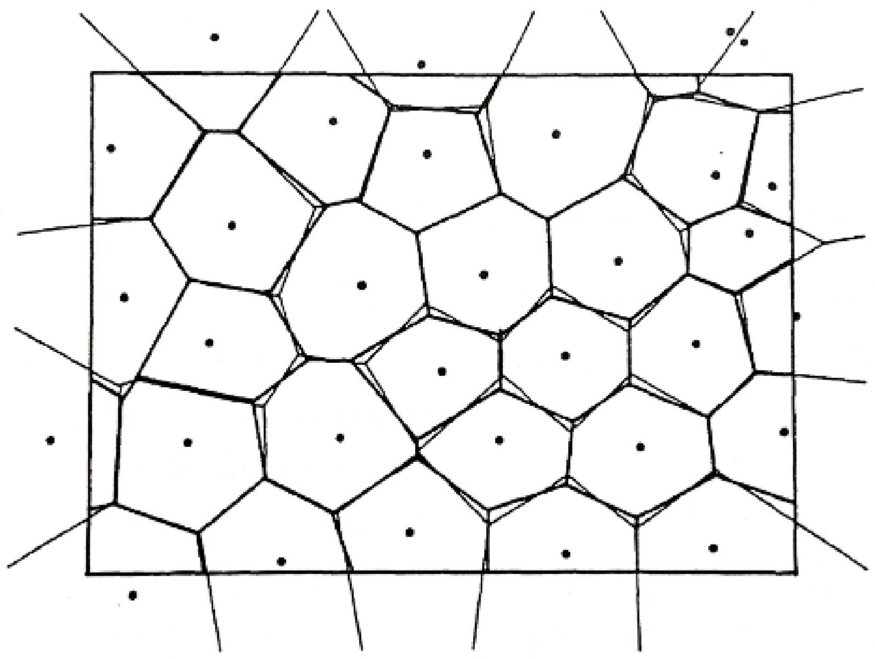


FIGURE 7.3. Fitting a Voronoi diagram to the territories of mouthbreeder fish. (Source: Suzuki and Iri, 1986a, Figure 13.)

Part 2. **OBJECTIVES**

The objectives of this project are:

- Fit a given tessellation of the unit square as best as possible through Voronoi diagrams. Therefore, the variable we want to minimize is the resulting symmetric difference between the input and the solution given. We want to implement different techniques and compare their results in a set of test cases.
- If the given tessellation is in fact a Voronoi diagram, we would like to arrive to that conclusion. That is, end with a symmetric difference close to 0.0. Even though this looks trivial, it will be very difficult to get there for non trivial input, since the number of local minimums that have to be avoided to find the seeds increases in a higher magnitude compared to the number of polygons.

Part 3. DEVELOPMENT

8. Why Processing: Pros and cons of Java

For developing the final application, I opted to go with Processing. Processing is an open source programming language and integrated development environment (IDE). The language builds on the Java programming language, but uses a simplified syntax and graphics programming model. Despite not being the most efficent language for crunching raw numbers, I decided to develop my program in Processing (Java) due to the following reasons:

- First of all, I had already used it in the past for a different project. One of the biggest advantages of Processing is the ease to display information in the screen through a GUI.
- The aim of this project is not to design an algorithm that can finish execution in the least amount of time, but rather, study and compare different techniques and their results. Therefore, we are not interested purely in execution times.
- It is extremely portable across the different operative systems.
- Java's data structures come in handy for our problem. ArrayList's are used constantly in the code to contain information about the tessellations and polygons.
- The number of regions we are going to be dealing with in our research will not be very high, in order to dedicate more time to analyzing the results than to optimizing the code.

9. Line clipping. Polygon clipping

For solving the inverse Voronoi diagram problem, there are two situations in which we have to find the intersection of two polygons: when clipping a polygon to the unit square and when clipping two polygons.

The first one is needed because, theoretically, there will always be some Voronoi region whose area will be infinite. In order to get rid of that issue, we must treat our Voronoi diagram in order to clip all regions into a bounded area, in our case, the unit square.

The second one is useful when we want to calculate the area of the intersection of two polygons for calculating the symmetric difference as explained earlier. However, note that this is doable since we will be working exclusively with convex polygons. An easy way to do it is calculate the actual intersection, and then, get the area of that polygon.

There is a simple, common operation in both tasks, to which these two problems can be reduced to: clip a polygon with a single line. If we have a solid function that can do this task, then we can rely on it to calculate any intersection of convex polygons (note that the unit square is a convex polygon too), repeating the process for all lines of one of the polygons against the other.

The algorithm works as follows:

Suppose we want to clip the polygon ABCDEFG with the blue line. Note that this polygon is not convex, since the angle in DEF is greater than 180^o, but for explanation purposes, we will apply the algorithm to it. Also note that the line has a direction, given by the blue arrow.

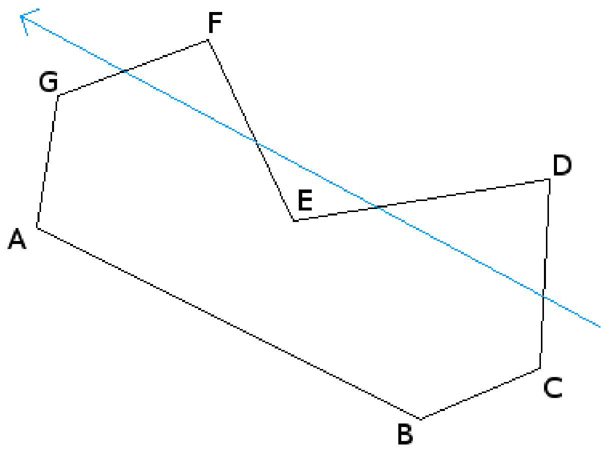


FIGURE 9.1. Starting point. We will clip the polygon with the given line (in blue). Starting point will be A.

The first step of the clipping algorithm is to set a starting vertex. We will set A as our starting point. We must check in which side of the blue line it lies. We can do this using the cross product of two points of the line and our point. Since in the implementation we will be working with positively oriented polygons, we are interested only in points which lie on the left of the line. The point A is lying on the left, so we add it to the solution and continue.

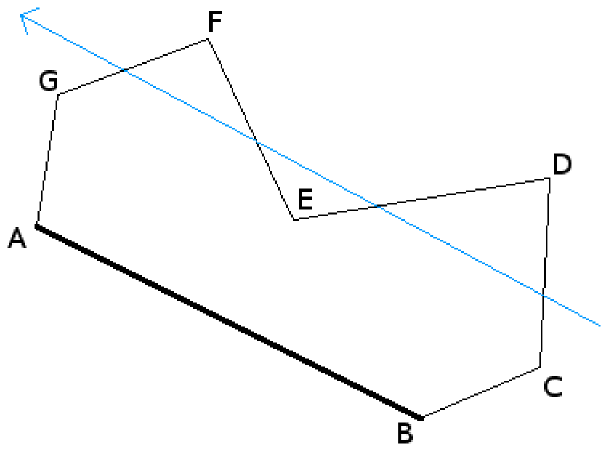


FIGURE 9.2. Since point B lies on the left of the line, add it to the result.

The next step is to select the next point and see whether it lies in the same side of the blue line. Since B also lies on the same side than A, we add it to the solution too and keep going.

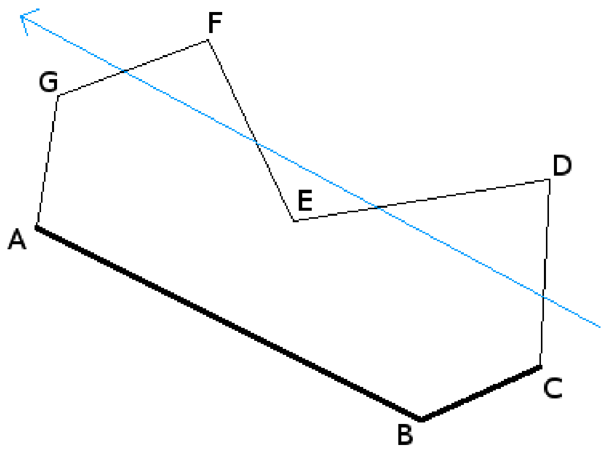


FIGURE 9.3. Since C is still on the left of the line, we add it to the final result too.

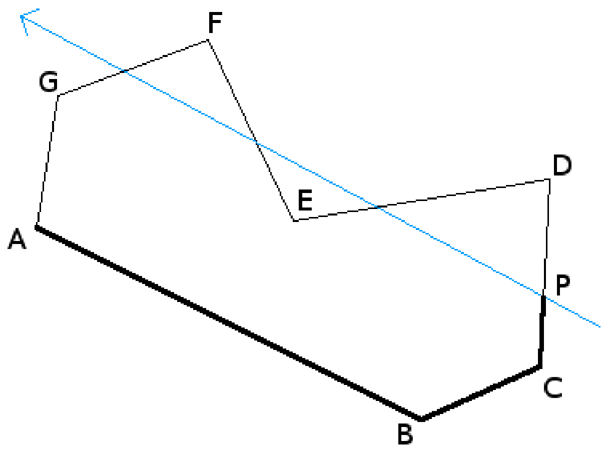


FIGURE 9.4. The next point, D, does not lie on the left of the blue line. Since last one did, it means an intersection point P has to exist. Add P to the final result.

When we check D, we see that it is lying on the right side of the line. Then, we know that this point will be clipped by the line. Therefore, we need to find point P, which is the intersection of CD with the blue line. We will be adding this point to the solution.

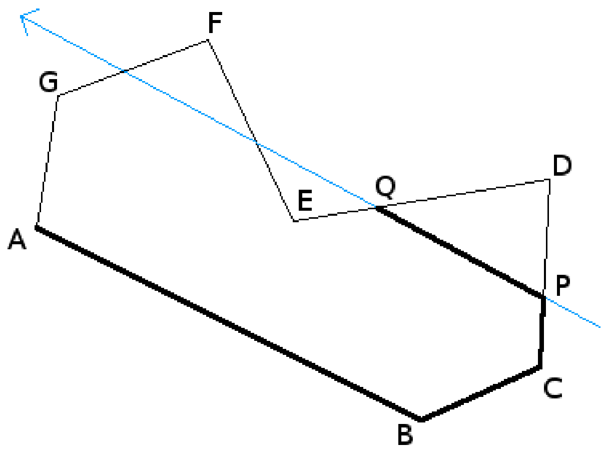


FIGURE 9.5. Current point D lies on the right of the line. Since E lies on the left, it means another intersection Q must exist. Add Q to the final result.

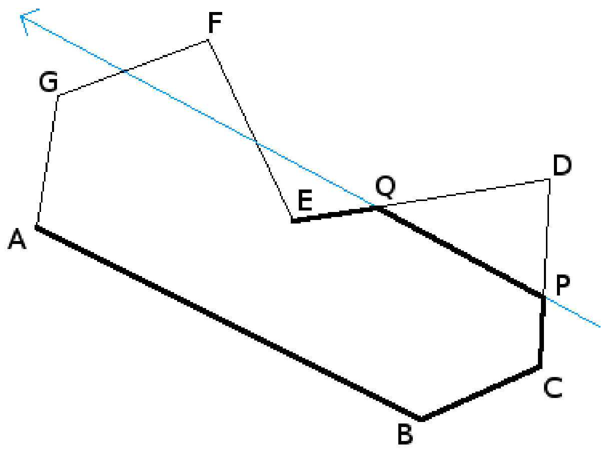


FIGURE 9.6. Point E lies on the left. Add it to the solution.

We keep repeating the same process until wee arrive to the starting point.

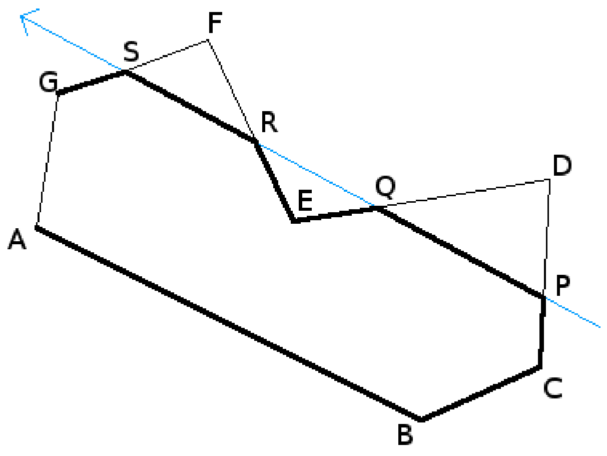


FIGURE 9.7. On the left again. Add G to the result.

When we arrive at the starting point, we have finished clipping the polygon with the line.

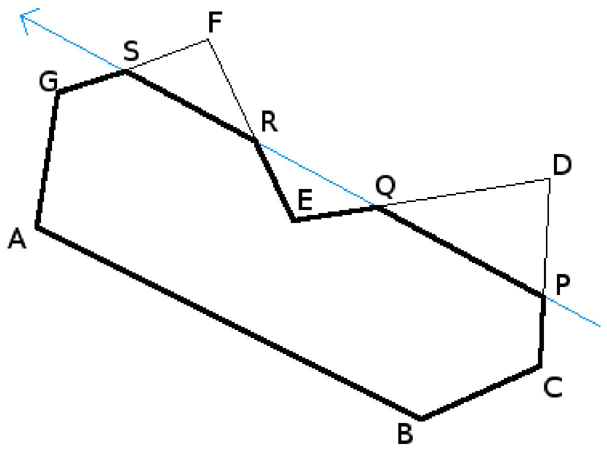


FIGURE 9.8. We arrived to A, our starting point.

For iterating a polygon A with another polygon B, given that both of them are correctly oriented, we can do it clipping A with all lines formed by B.

10. INPUT DATA

The input data for the problem is a tessellation of the unit square. That is, a set of polygons whose union is the unit square and, for every two polygons, its intersection is at most a straight line. More precisely, each polygon is given as a set of ordered 2D points. At first, we know we will be working only with convex polygons (that is, all internal angles are less or equal than 180^o). The input data will be read from a file defined by the user in which each line will contain a whole tessellation of the unit square.

If you are familiar with the Python programming language, you will see that the input follows the same syntax as Python code.

A Python tuple can be expressed as a list of comma separated values, with round brackets surrounding the whole tuple. For example:

$Fruits = (\text{"}Pear\text{"}, \text{"}Apple\text{"}, \text{"}Grape\text{"})$

A Python list can be expressed as a list of comma separated values, with square brackets surrounding the whole list. For example:

$Numbers = [1, 6, -464]$

We will define a point as a tuple of two floating point numbers, each of one representing a coordinate (X and Y respectively).

A polygon is a list of ordered points, following a positive or negative orientation.

A tessellation is a list of polygons (note that different polygons may have different orientations).

For example, lets say we want to adjust the following tesellation:

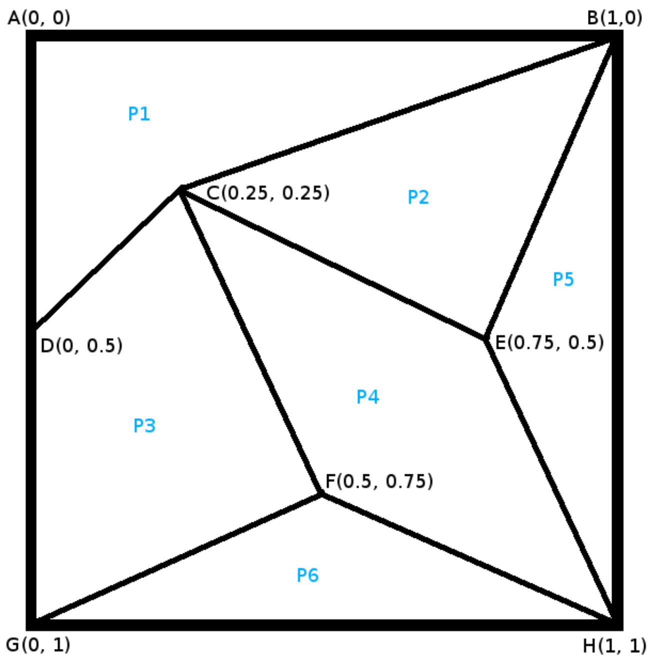


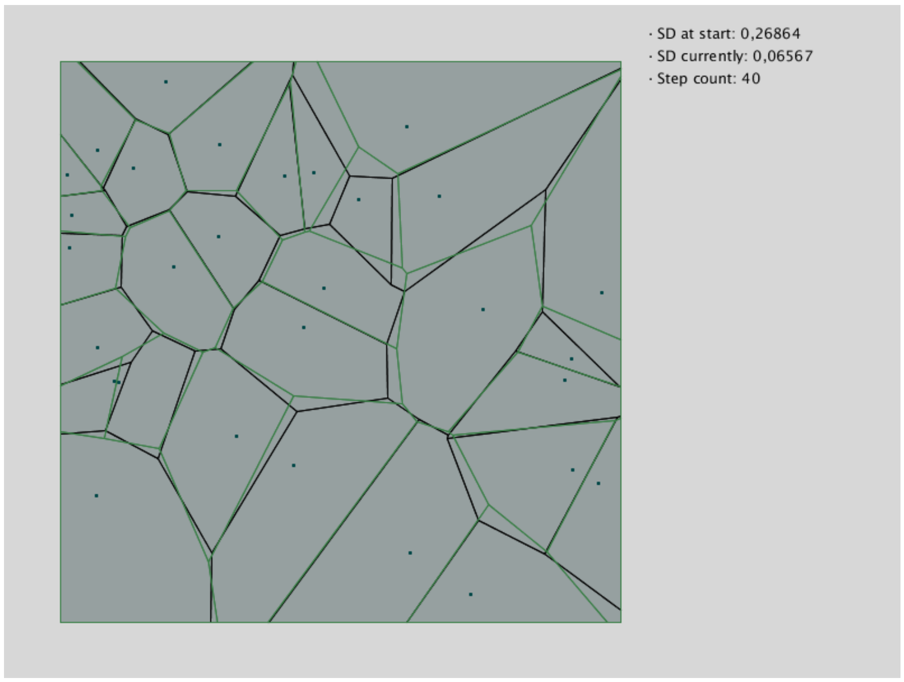
FIGURE 10.1. Tesellation of the plane we want to adjust.

There are 8 relevant points in the example:
$A = (0,0)$
$B = (1,0)$
$C = (0.25,0.25)$
$D = (0,0.5)$
$E = (0.75,0.5)$
$F = (0.5,0.75)$
$G = (0,1)$
$H = (1,1)$
As we can see, the tesellation is composed by 6 polygons, each of them formed by an arbitrary number of points:
Considerations of the input data:
$P1 = [A,B,C,D]$
$P2 = [C,B,E]$
$P3 = [D,C,F,G]$
$P4 = [C,F,H,E]$
$P5 = [E,H,B]$
$P6 = [F,G,H]$
Notice that for some polygons, the points are arranged with a positive orientation (anti-clockwise - i.e. P2) and for others such as P1, they are arranged with a negative orientation (clockwise - i.e. P1). It does not matter in which orientation the points are given for each polygon, since the program will reallocate them internally.
So our tesellation can be expressed as a list containing all the polygons:
$Tes = [P1,P2,P3,P4,P5,P6]$
Then we just have to keep substituting values until we get the following representation:
$Tes = [[A,B,C,D],[C,B,E],[D,C,F,G],[C,F,H,E],[E,H,B],[F,G,H]]$
Tes=[[(0,0),(1,0),(0.25,0.25),(0,0.5)],[(0.25,0.25),(1,0),(0.75,0.5)],[(0,0.5),(0.25,0.25),(0.5,0.75),(0,1)],[(0.25,0.25),(0.5,0.75),(1,1),(0.75,0.5)],[(0.75,0.5),(1,1),(1,0)],[(0.5,0.75),(0,1),(1,1)]]
Then, we will have to place it in some file and provide the program the file name and the line of that tessellation.

11. OUTPUT DATA

The program generates various files after being executed. The files will be located in the "FittingVoronoi" folder, and will be the following:

- output.csv: CSV file in which each line represents each iteration of the algorithm. More specifically, each line will contain the information of where each seed point was at that stage and the symmetric difference in that precise step.
- before.png: Image that displays the first approximation of the Voronoi diagram.
- after.png: Image that displays the final approximation of the Voronoi diagram.



```
· SD at start: 0,26864
· SD currently: 0,06567
· Step count: 40
```

FIGURE 11.1. Output image generated by the program. It shows the final result, as well as some other information of the process.

12. Heuristics and general considerations

We will study three different heuristic techniques for solving the inverse Voronoi diagram problem. Those are:

- Gradient method.
- Gradient method with steps.
- Simulated annealing.

The underlying logic of all of them is similar:

- For each input, calculate a set of seed points from which we will start. Since we know all our polygons will be convex, for each of them, we will calculate the first Voronoi seed as the average of the coordinates of all of the points of that polygon. This guarantees that the point will lay on the inside of that polygon. When calculating the Voronoi diagram from the seeds, a relation will be stablished between each original polygon and the Voronoi cell obtained from the seed obtained from that polygon.
- On each iteration, move each point slightly following some criteria and check if we improved the current best solution. For each seed, we will compare the current solution against the solutions obtained from moving the seed in four directions (up, down, left and right) some distance (also called step).
- Repeat the previous step until we cannot keep improving our solution.

12.1. **Getting rid of the "repetition" problem: randomizing point order in each state.** Given the nature of the problem, it is easy to see that the order in which we will move our points will play a huge role in the final result. In order to avoid this cyclic effect, what do is move them in a pseudo-random mannner every iteration, regardless of the heuristic used.

13. Gradient method

The gradient method (also known as the hill climbing algorithm) is a simple heuristic used in optimization problems. It consists on, at any given state, compare neighbor solutions against the current best one. If they are better, mark it as the new best solution and repeat. If the minimum or maximum we are looking for exists, there will be a point at which we will not be able to continue improving. That will be our solution.

For example, consider you want to find the maximum value of a function. You start at a random point with a corresponding value. You will have to move to the left if the value at some distance to the left is higher, or to the right if the value at some distance to the right is higher. The problem with this is that we can easily get stuck in a local maximum.

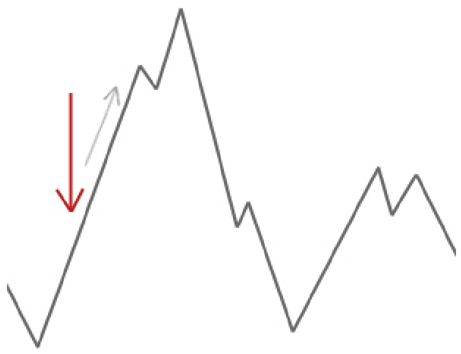


FIGURE 13.1. The starting point will be the left arrow. As we can see, this method would indicate us to go right in order to get to a better solution.

As it can be seen in Figure 12.1, for finding the maximum of that function, after comparing two points close to the red arrow, the gradient method tells us that we have to move to the right, since the value found there is higher than the current one and the one on the left.

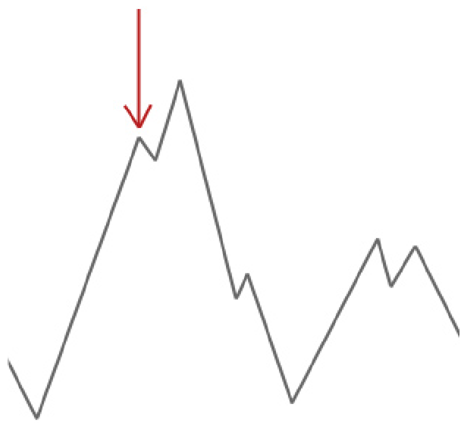


FIGURE 13.2. Ending point using a simple gradient method.

However, when reaching this point, it would check for its neighbors and see that whatever we do, we would get a worse solution. Therefore, the peak pointed by the red arrow would be our solution.

In this example, the value obtained is fairly close to the global maximum, but in more complex problems, we might get stuck even at our starting solution. We need some way to avoid (or at least try to) local maximum values.

14. Improved gradient method: implementing steps

One little improvement we can apply to the gradient method is to apply different iterations of it, but with a different step on each one. We start with a high value for the step, and when we cannot improve anymore for that step, we reduce it to some other value. When we have reached the lowest step, and we cannot keep improving the solution, the algorithm finishes and we return the solution found.

This way, we will adjust our solution more than with using the gradient method. We will also be able to get further away from the initial solution. However, lots of solutions are lost and this method is greatly dependent on the step values chosen. Also, we have the problem of deciding which values to use for the step. There is no combination which is guaranteed to work on every single case. Therefore, intuition and testing will be key in order to find the desired set of step values.

15. Simulated annealing

Simulated annealing copies a phenomenon in nature (the annealing of solids) to optimize a complex system. Annealing involves heating and cooling a material to alter its physical properties due to the changes in its internal structure. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties.

In simulated annealing we keep a temperature variable to simulate this heating process. We initially set it high and then allow it to slowly cool as the algorithm runs. While this temperature variable is high, the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local optimums it finds itself in early on in execution. As the temperature is reduced so is the chance of accepting worse solutions, therefore allowing the algorithm to gradually focus in on a area of the search space in which hopefully, a close to optimum solution can be found.

At any given point during the process, a neighbor solution will be calculated. The logic to whether accept it as the next solution or not is the following:

- If the neighbor solution is better than the current one, we accept it unconditionally as the new solution.
- If it is worse, we need to consider the following factors:
  - How much worse the neighbor solution is.
  - What is the temperature at that particular state of the algorithm. When the temperature is higher (near the start of the algorithm) the chances of accepting a worse solution will be high. When the temperature is lower, so will be the chances.

The implementation of this is very simple. On each step, if the neighbor solution is worse, we calculate the following value, which is called the **Acceptance Function**:

$$p = e^{(s-n)/T}$$

where:
e is Euler's number (2.71828 approximately)
s is the current best solution value
n is the neighbor solution value
T is the temperature. Every problem will have to start with a temperature relative to it, so some research should be done in order to find an acceptable value for this variable. The current implementation of the algorithm uses the current step as the temperature.

Then, we generate a random value in the range of (0, 1) and check if p is greater than that value. If it is, mark the neighbor solution as the current one, and keep searching from there.

The pseudo-code of the simulated annealing algorithm would look like this:

- Generate a initial solution S. Mark this as the best solution B. Mark it as the current solution C.
- Get a neighbor solution N from the current solution C.
  - If the N is better than the current solution: Mark it as the current solution. If it is better than the best solution B, mark the best to be the current solution.
  - If N is worse, but we accepted it according to the acceptance probability, mark it as the current solution.
- Repeat last step until the solution does not improve any more.
- Reduce the temperature T.
- Repeat until steps until we cannot improve any more. Return B.

Part 4. **RESULTS AND OBSERVATIONS**

The following sets of steps have been used during the study of the gradient method with steps and the simulated annealing technique.

- $A = \{0.05, 0.04, 0.01, 0.005, 0.001\}$

Set A is very precise. Its highest step is just 5% of the side of the unit square. We should expect mostly good results from this one, since when increasing the number of seeds, they will tend to be closer to each other.

- $B = \{0.05, 0.03, 0.01\}$

Set B is similar to set A, even though not being as precise. Since the biggest difference between the highest and the lowest step is 4%, it should only work well with a certain set of tessellations.

- $C = \{0.1, 0.01, 0.005\}$

Set C tries to jump out of local minimums by starting with a 10% of the unit square side step. Then, it focuses on that result and approximates the best result possible with steps of 1% and 0.5%.

- $D = \{0.2, 0.16, 0.09, 0.05, 0.03\}$

Set D focuses on big steps and a quick decline of the temperature. It should converge to a solution relatively fast, even though it should not yield the best solution when compared to the other sets.

The following pages will contain the extracted information about the tests realized. For each of them, there will also be two images, showing the final result of the best and worst configuration for that particular test. At the end of all tests, there will be a deeper and more general conclusion of the study.

16. Tessellation example 1

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.21728	0.09680	5
Grad.	0.01	0.21728	0.06632	13
Grad.	0.005	0.21728	0.05689	27
Grad. steps	Set A	0.21728	0.04463	63
Grad. steps	Set B	0.21728	0.05221	17
Grad. steps	Set C	0.21728	0.03407	29
Grad. steps	Set D	0.21728	0.06526	17
Sim. Annealing	Set A	0.21728	0.04463	65
Sim. Annealing	Set B	0.21728	0.05221	17
Sim. Annealing	Set C	0.21728	0.03407	29
Sim. Annealing	Set D	0.21728	0.06526	17

TABLE 1. Results obtained for lineNumber = 1. This tessellation is made up of 20 polygons.

For the first case, it can be seen that the worst obtained solution is around 10% (0.09680). This converged fast. In just 5 steps we managed to half the starting solution. The best solution is obtained when using set C, whether applying simulated annealing or the gradient method with steps. The step count for both is 29. It is also remarkable that when using set A (the most precise), we got a solution a bit worse and it took more than twice the steps.

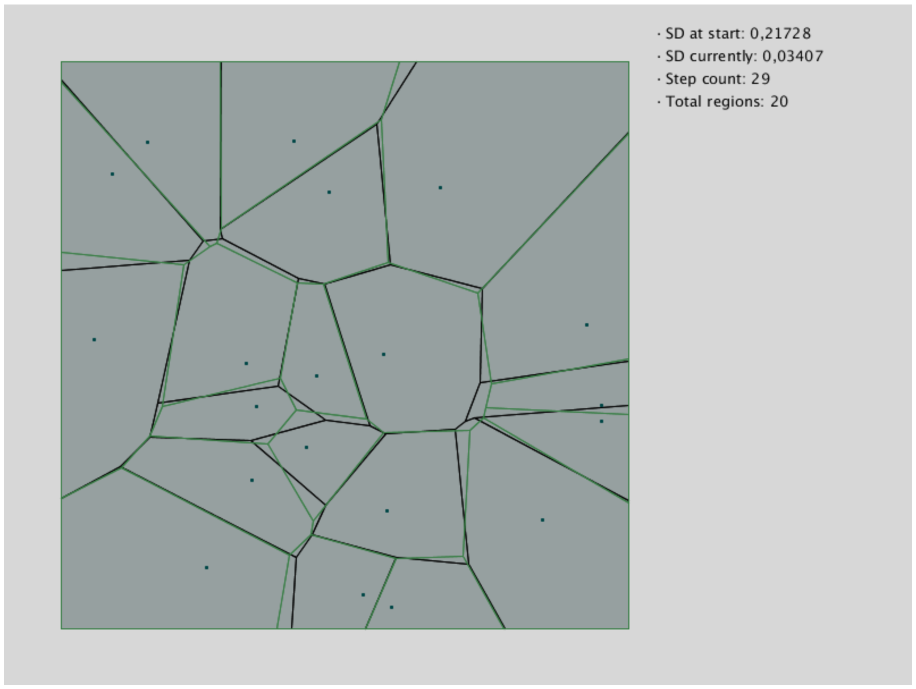

· SD at start: 0,21728
· SD currently: 0,03407
· Step count: 29
· Total regions: 20

FIGURE 16.1. Result of the best tested configuration for lineNumber = 1.

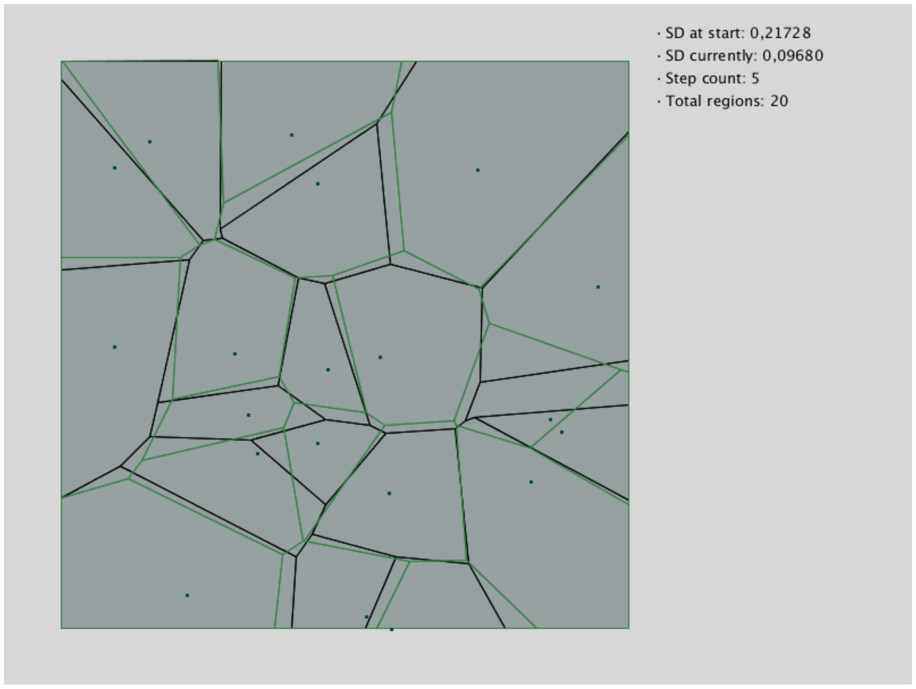

· SD at start: 0,21728
· SD currently: 0,09680
· Step count: 5
· Total regions: 20

FIGURE 16.2. Result of the worst tested configuration for lineNumber = 1.

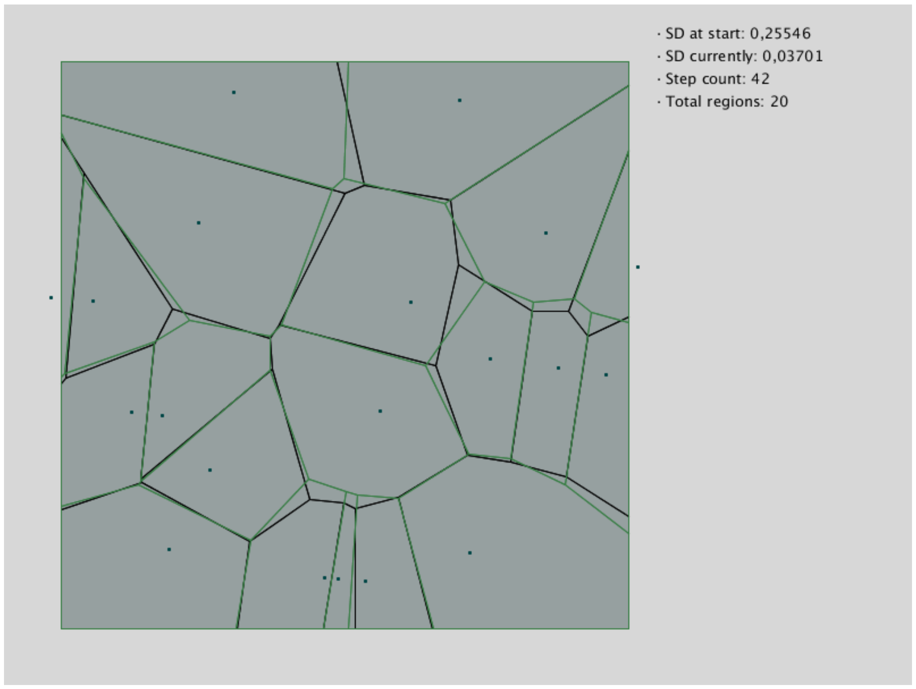
17. Tessellation example 2

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.25546	0.08712	6
Grad.	0.01	0.25546	0.06859	15
Grad.	0.005	0.25546	0.06485	25
Grad. steps	Set A	0.25546	0.03746	33
Grad. steps	Set B	0.25546	0.05474	14
Grad. steps	Set C	0.25546	0.06172	27
Grad. steps	Set D	0.25546	0.09467	17
Sim. Annealing	Set A	0.25546	0.03701	42
Sim. Annealing	Set B	0.25546	0.05474	14
Sim. Annealing	Set C	0.25546	0.06172	27
Sim. Annealing	Set D	0.25546	0.09467	20

TABLE 2. Results obtained for lineNumber = 2. This tessellation is made up of 20 polygons.

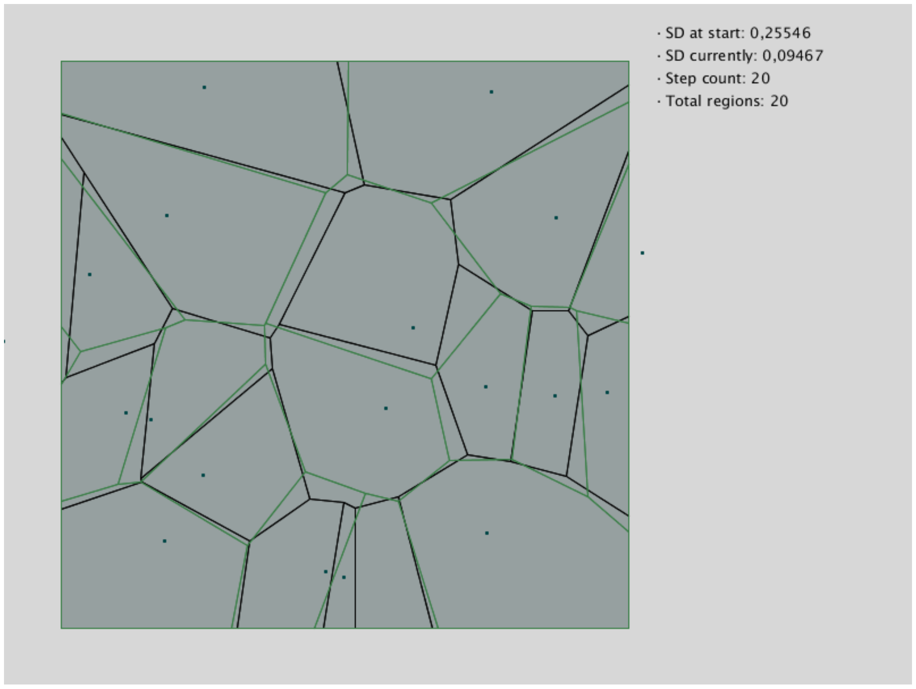
In this case, all methods performed relatively well. There are various things we must note about this example:

- The results obtained with the gradient method for step 0.05 are surprisingly good. In just 6 steps, it managed to reduce the symmetric difference from 25% down to 8%.
- When using set D, whether on simulated annealing or gradient method with steps, we got worst results than with any step on the gradient method. This indicates that not always a complex solution will be better than a simple one.
- When using set A, the difference between the gradient method with steps and the simulated annealing technique is only 0.04% at the cost of almost 30% more steps.



· SD at start: 0,25546
· SD currently: 0,03701
· Step count: 42
· Total regions: 20

FIGURE 17.1. Result of the best tested configuration for lineNumber = 2.



· SD at start: 0,25546
· SD currently: 0,09467
· Step count: 20
· Total regions: 20

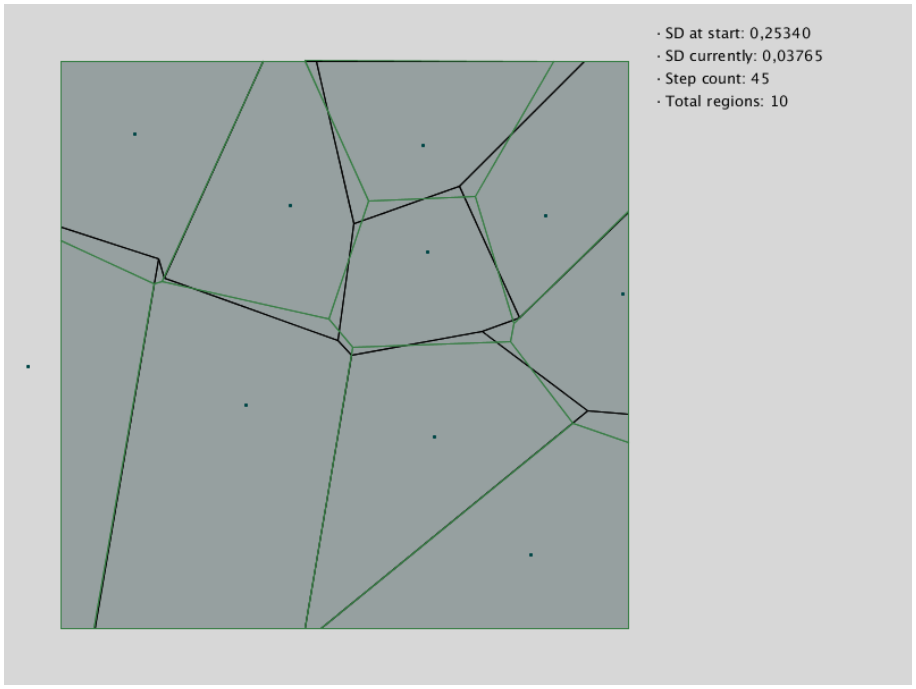
FIGURE 17.2. Result of the worst tested configuration for lineNumber = 2.

18. Tessellation example 3

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.25340	0.06583	6
Grad.	0.01	0.25340	0.04352	29
Grad.	0.005	0.25340	0.06344	34
Grad. steps	Set A	0.25340	0.03765	45
Grad. steps	Set B	0.25340	0.04568	13
Grad. steps	Set C	0.25340	0.04356	24
Grad. steps	Set D	0.25340	0.07275	19
Sim. Annealing	Set A	0.25340	0.03768	57
Sim. Annealing	Set B	0.25340	0.04568	13
Sim. Annealing	Set C	0.25340	0.04356	24
Sim. Annealing	Set D	0.25340	0.07275	19

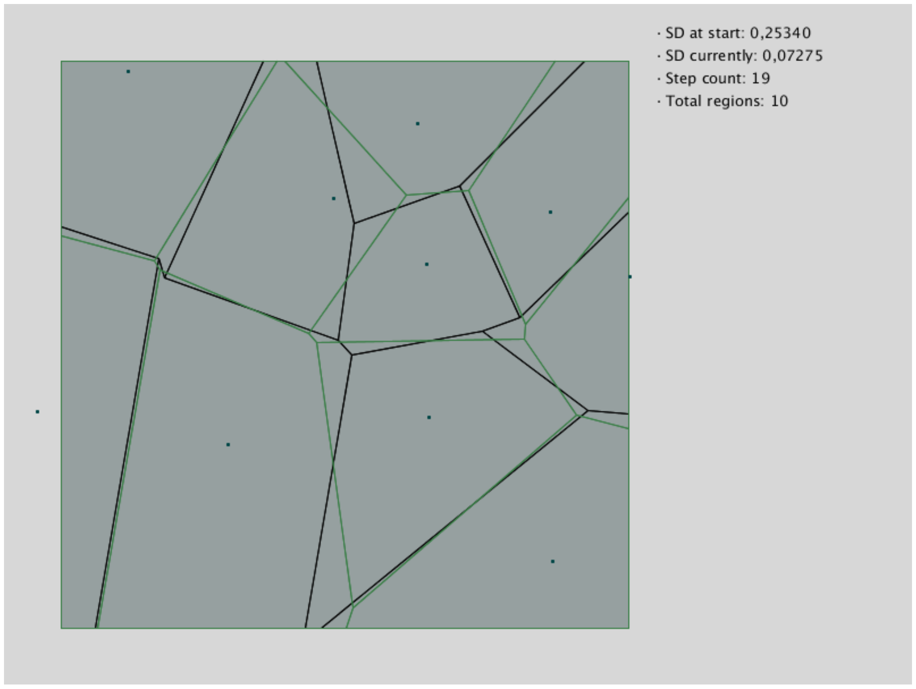
TABLE 3. Results obtained for lineNumber = 3. This tessellation is made up of 10 polygons.

As we can see, by adjusting too much from the beginning using the gradient method, we failed to get better results with step 0.005 than with 0.01. This is because the algorithm fell in a local minimum early with step 0.005, but managed to avoid it with step 0.01.



· SD at start: 0,25340
· SD currently: 0,03765
· Step count: 45
· Total regions: 10

FIGURE 18.1. Result of the best tested configuration for lineNumber = 3.



· SD at start: 0,25340
· SD currently: 0,07275
· Step count: 19
· Total regions: 10

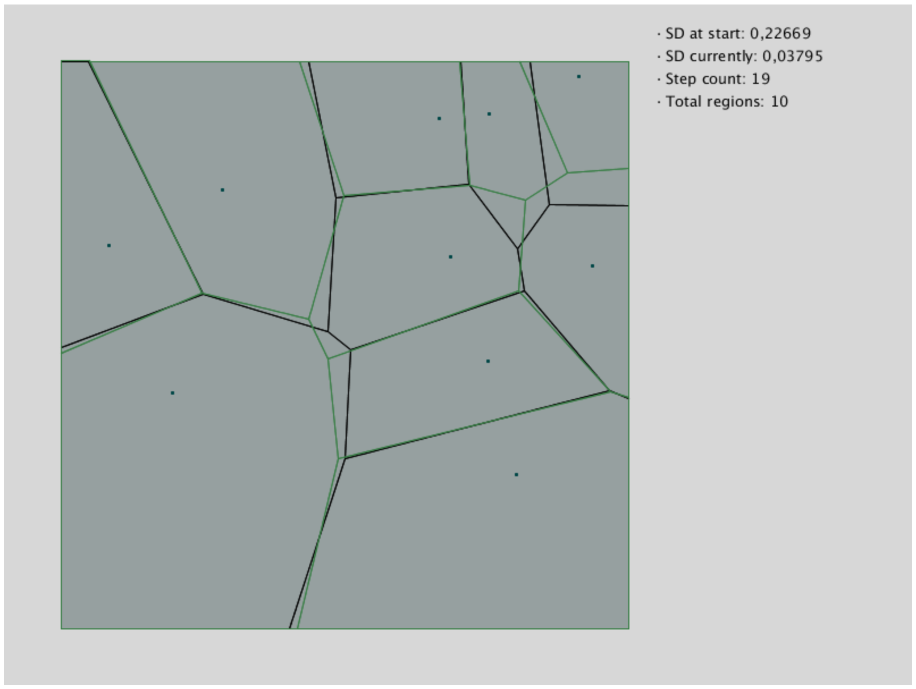
FIGURE 18.2. Result of the worst tested configuration for lineNumber = 3.

19. Tessellation example 4

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.22669	0.06117	6
Grad.	0.01	0.22669	0.05093	20
Grad.	0.005	0.22669	0.05588	24
Grad. steps	Set A	0.22669	0.03988	22
Grad. steps	Set B	0.22669	0.04204	14
Grad. steps	Set C	0.22669	0.03795	19
Grad. steps	Set D	0.22669	0.06035	14
Sim. Annealing	Set A	0.22669	0.03988	22
Sim. Annealing	Set B	0.22669	0.04204	14
Sim. Annealing	Set C	0.22669	0.03795	19
Sim. Annealing	Set D	0.22669	0.06035	14

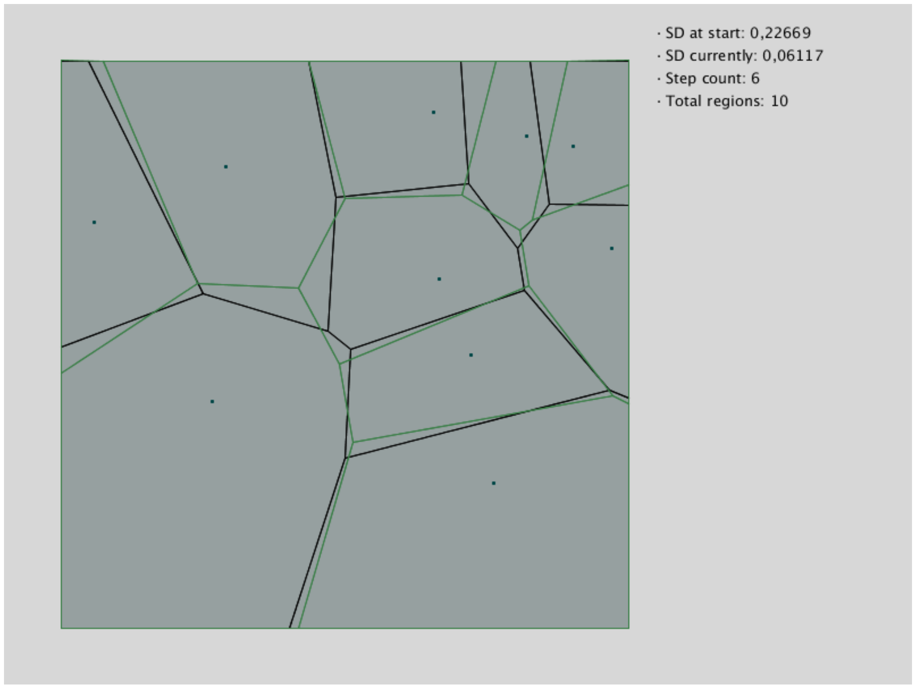
TABLE 4. Results obtained for lineNumber = 4. This tessellation is made up of 10 polygons.

Being the first step of set C double the first step of set A, in this example set C managed to avoid an earlier local minimum and got a better result than set A in less steps, despite set A containing more precise steps. Also, it managed to do it in less steps. It is worth noting too that all the solutions differ 3% at maximum.



· SD at start: 0,22669
· SD currently: 0,03795
· Step count: 19
· Total regions: 10

FIGURE 19.1. Result of the best tested configuration for lineNumber = 4.



· SD at start: 0,22669
· SD currently: 0,06117
· Step count: 6
· Total regions: 10

FIGURE 19.2. Result of the worst tested configuration for lineNumber = 4.

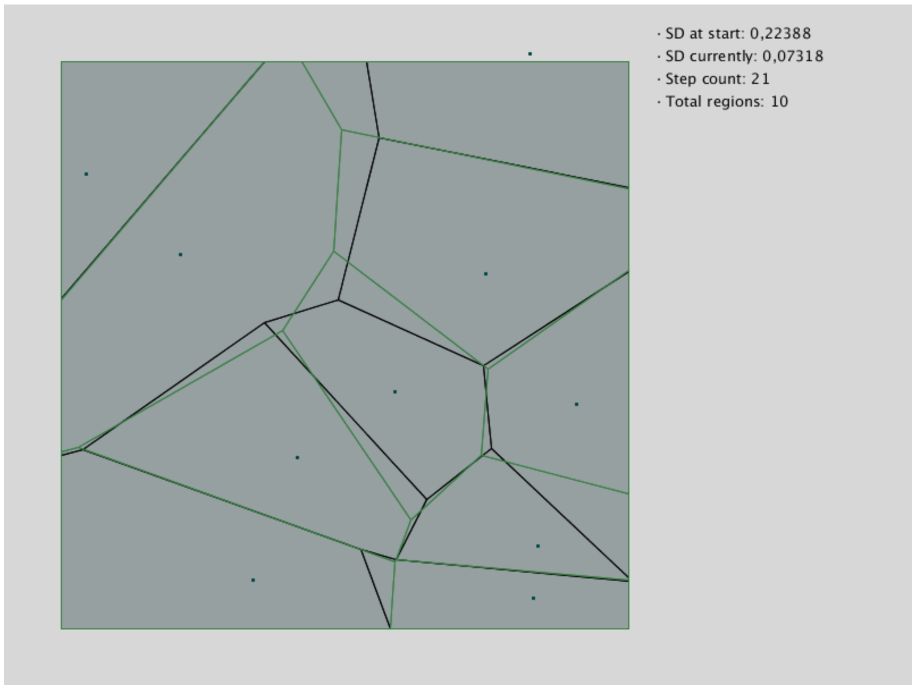
20. Tessellation example 5

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.22388	0.09622	5
Grad.	0.01	0.22388	0.09248	13
Grad.	0.005	0.22388	0.09191	21
Grad. steps	Set A	0.22388	0.07318	21
Grad. steps	Set B	0.22388	0.07967	10
Grad. steps	Set C	0.22388	0.09538	16
Grad. steps	Set D	0.22388	0.10418	15
Sim. Annealing	Set A	0.22388	0.07318	21
Sim. Annealing	Set B	0.22388	0.07967	10
Sim. Annealing	Set C	0.22388	0.09538	16
Sim. Annealing	Set D	0.22388	0.10418	15

TABLE 5. Results obtained for lineNumber = 5. This tessellation is made up of 10 polygons.

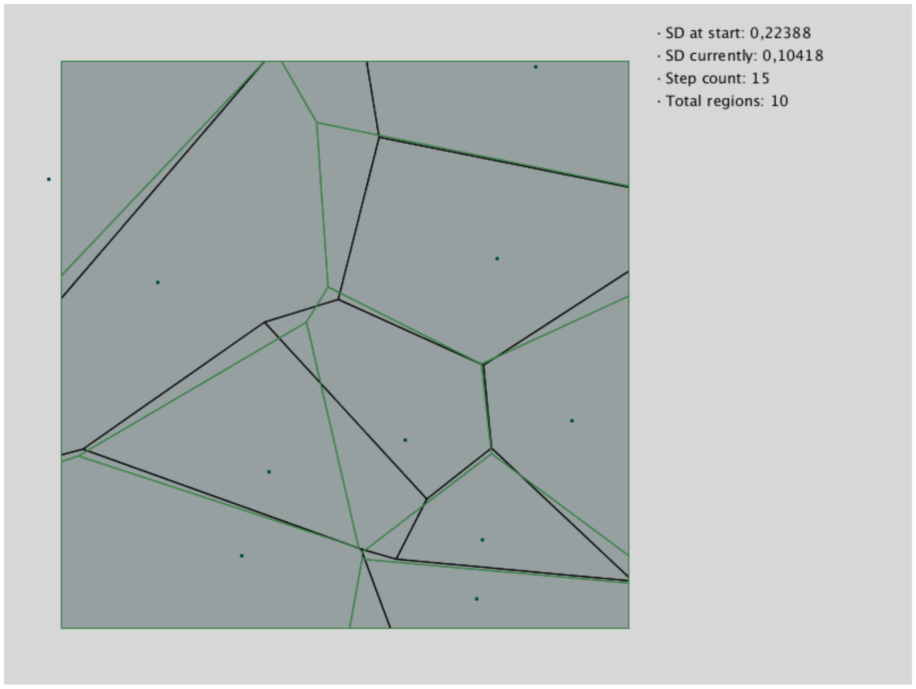
In this example, it is notable that the results obtained with the gradient method for all the steps are very similar.

Also, it is remarkable that applying the gradient method with steps for sets C and D, we get worse results than applying the basic gradient method. Keep in mind that in other examples, set C manages to consistently get the best or second best results.



```
· SD at start: 0,22388
· SD currently: 0,07318
· Step count: 21
· Total regions: 10
```

FIGURE 20.1. Result of the best tested configuration for lineNumber = 5.



```
· SD at start: 0,22388
· SD currently: 0,10418
· Step count: 15
· Total regions: 10
```

FIGURE 20.2. Result of the worst tested configuration for lineNumber = 5.

21. Tessellation example 6

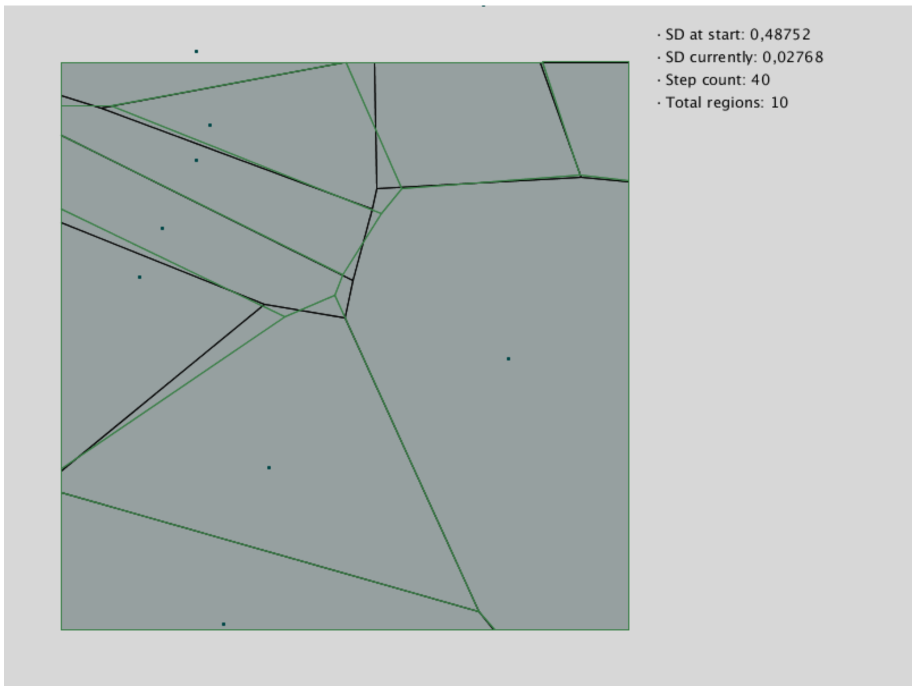
	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.48752	0.08620	9
Grad.	0.01	0.48752	0.03954	36
Grad.	0.005	0.48752	0.02882	73
Grad. steps	Set A	0.48752	0.02873	50
Grad. steps	Set B	0.48752	0.05171	24
Grad. steps	Set C	0.48752	0.03429	45
Grad. steps	Set D	0.48752	0.06475	21
Sim. Annealing	Set A	0.48752	0.02768	40
Sim. Annealing	Set B	0.48752	0.04765	27
Sim. Annealing	Set C	0.48752	0.03881	45
Sim. Annealing	Set D	0.48752	0.06475	21

TABLE 6. Results obtained for lineNumber = 6. This tessellation is made up of 10 polygons.

It is remarkable that when first calculating the initial seeds, the symmetric difference is almost 50%.

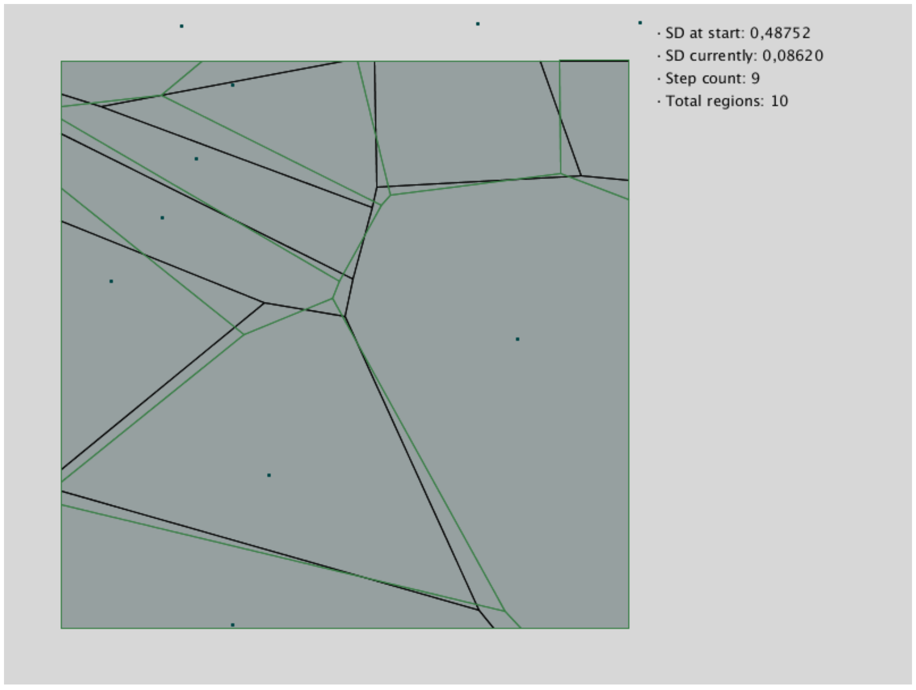
When applying the simulated annealing technique, this is one of the cases in which we got a better result for set A in less steps than the one obtained through the gradient method with steps.

In the other hand, for set C, we got worse results in the same number of steps, and for set D, the same results were obtained for both methods.



```
· SD at start: 0,48752
· SD currently: 0,02768
· Step count: 40
· Total regions: 10
```

FIGURE 21.1. Result of the best tested configuration for lineNumber = 6.



```
· SD at start: 0,48752
· SD currently: 0,08620
· Step count: 9
· Total regions: 10
```

FIGURE 21.2. Result of the worst tested configuration for lineNumber = 6.

22. Tessellation example 7

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.22892	0.11978	6
Grad.	0.01	0.22892	0.05093	16
Grad.	0.005	0.22892	0.04286	31
Grad. steps	Set A	0.22892	0.03667	85
Grad. steps	Set B	0.22892	0.05497	20
Grad. steps	Set C	0.22892	0.05138	39
Grad. steps	Set D	0.22892	0.08754	17
Sim. Annealing	Set A	0.22892	0.04006	73
Sim. Annealing	Set B	0.22892	0.06551	18
Sim. Annealing	Set C	0.22892	0.05138	39
Sim. Annealing	Set D	0.22892	0.08754	17

TABLE 7. Results obtained for lineNumber = 7. This tessellation is made up of 30 polygons.

In this example, we can see that the results obtained with the simulated annealing technique are worse for both sets A and B than its gradient method with steps counterpart. Also, note that despite getting very close results (a difference of 1.5%), the number of steps obtained with set A is quite high.

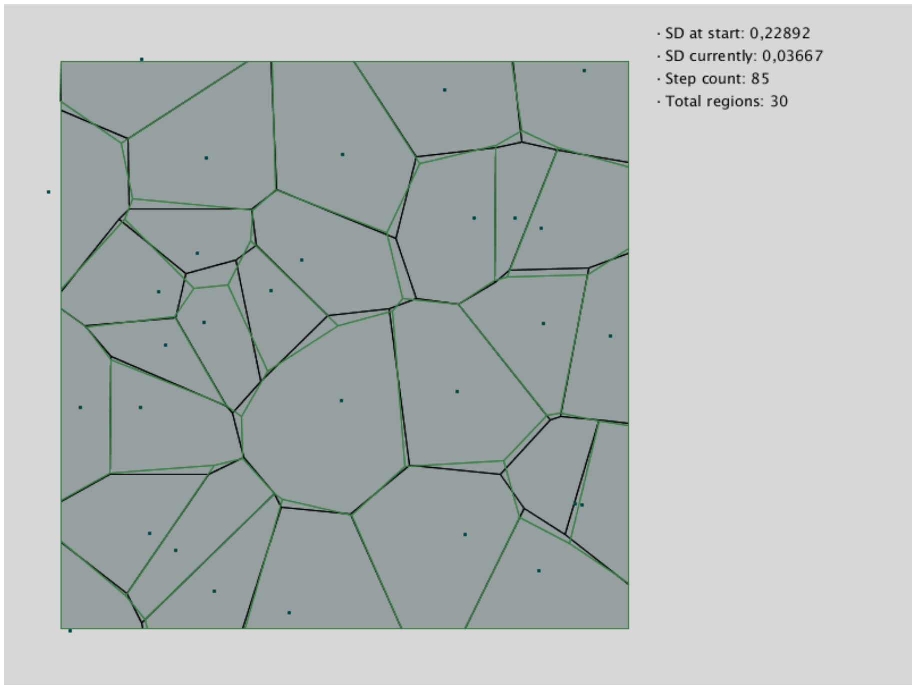

· SD at start: 0,22892
· SD currently: 0,03667
· Step count: 85
· Total regions: 30

FIGURE 22.1. Result of the best tested configuration for lineNumber = 7.

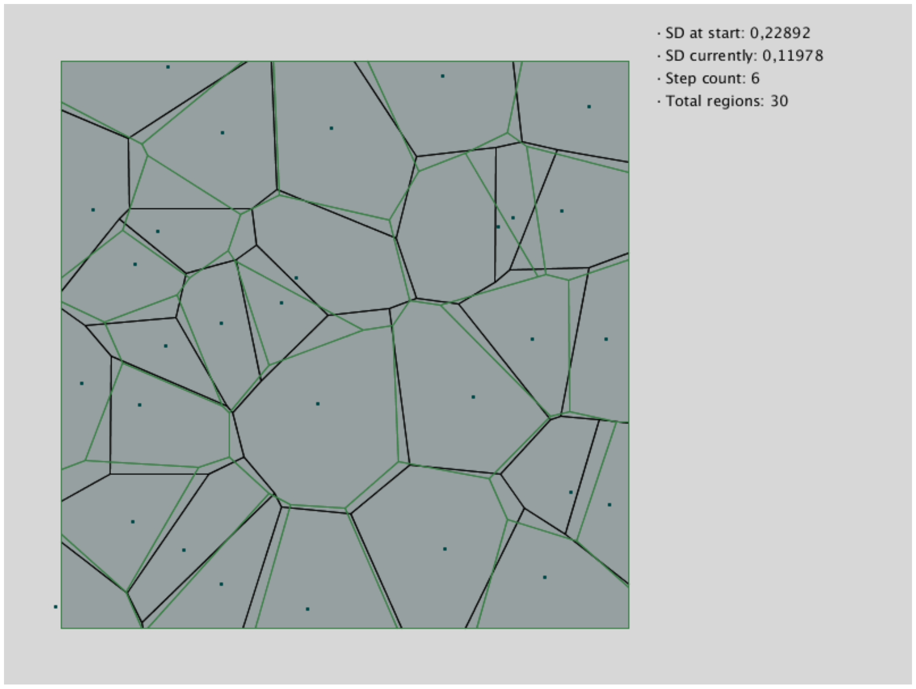

· SD at start: 0,22892
· SD currently: 0,11978
· Step count: 6
· Total regions: 30

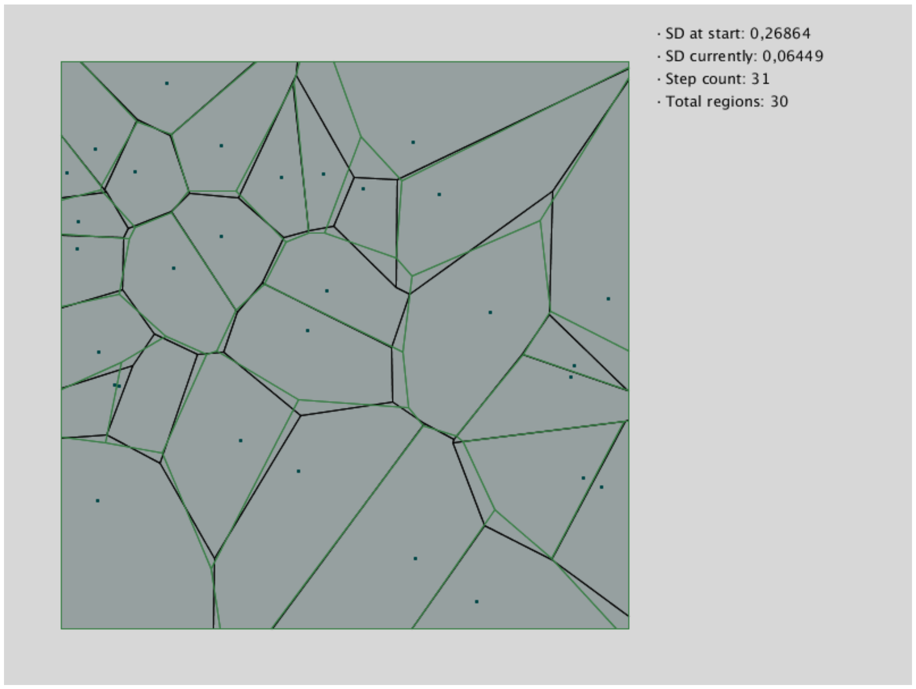
FIGURE 22.2. Result of the worst tested configuration for lineNumber = 7.

23. Tessellation example 8

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.26864	0.12694	6
Grad.	0.01	0.26864	0.10517	20
Grad.	0.005	0.26864	0.10109	32
Grad. steps	Set A	0.26864	0.06518	29
Grad. steps	Set B	0.26864	0.07564	15
Grad. steps	Set C	0.26864	0.06018	37
Grad. steps	Set D	0.26864	0.09161	15
Sim. Annealing	Set A	0.26864	0.06449	31
Sim. Annealing	Set B	0.26864	0.07564	15
Sim. Annealing	Set C	0.26864	0.06788	27
Sim. Annealing	Set D	0.26864	0.09123	17

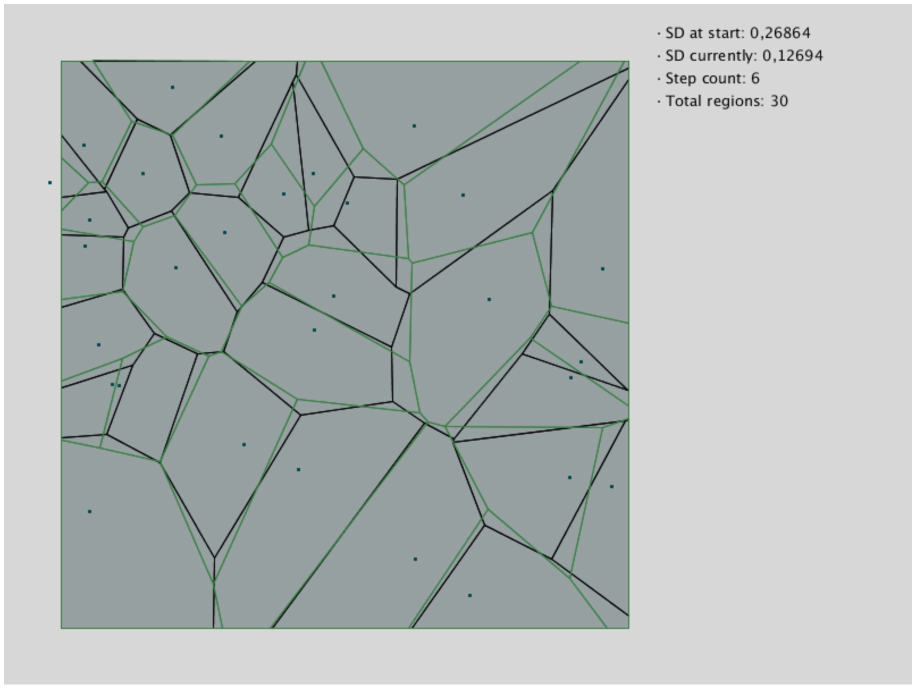
TABLE 8. Results obtained for lineNumber = 8. This tessellation is made up of 30 polygons.

In this example, it is remarkable that the method which used the most steps was the gradient method with step 0.005.



```
· SD at start: 0,26864
· SD currently: 0,06449
· Step count: 31
· Total regions: 30
```

FIGURE 23.1. Result of the best tested configuration for lineNumber = 8.



```
· SD at start: 0,26864
· SD currently: 0,12694
· Step count: 6
· Total regions: 30
```

FIGURE 23.2. Result of the worst tested configuration for lineNumber = 8.

24. Tessellation example 9

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.16570	0.10682	3
Grad.	0.01	0.16570	0.04358	27
Grad.	0.005	0.16570	0.05268	40
Grad. steps	Set A	0.16570	0.03862	20
Grad. steps	Set B	0.16570	0.04230	16
Grad. steps	Set C	0.16570	0.03701	14
Grad. steps	Set D	0.16570	0.06586	13
Sim. Annealing	Set A	0.16570	0.03805	29
Sim. Annealing	Set B	0.16570	0.04131	19
Sim. Annealing	Set C	0.16570	0.03701	14
Sim. Annealing	Set D	0.16570	0.06586	13

TABLE 9. Results obtained for lineNumber = 9. This tessellation is made up of 5 polygons.

Here again, we found the same problem that in some other tests: due to starting with a very precise step, we end up getting worse results (and with more steps) in the gradient method with step 0.005 when compared to the gradient method with step 0.01. We can also see that when using the simulated annealing technique for set A, we just managed to improve our solution about 0.06% at the cost of 30% more steps.

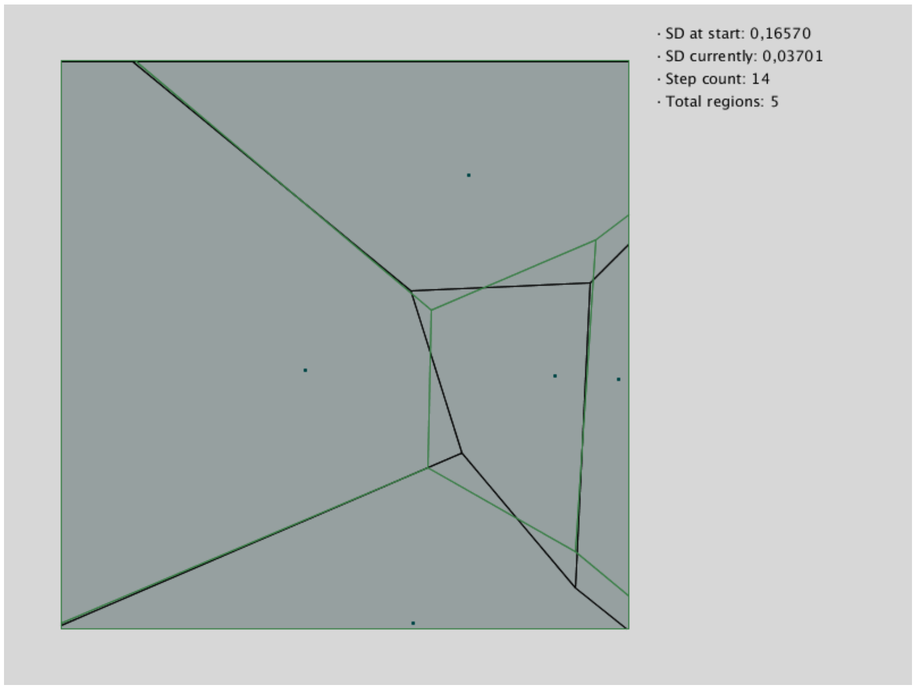

```
· SD at start: 0,16570
· SD currently: 0,03701
· Step count: 14
· Total regions: 5
```

FIGURE 24.1. Result of the best tested configuration for lineNumber = 9.

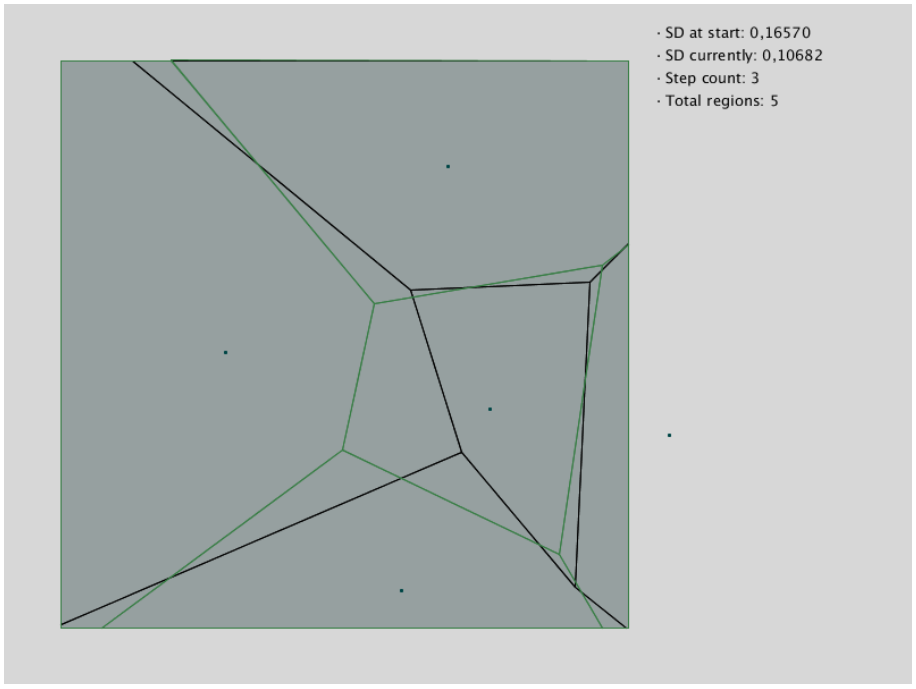

```
· SD at start: 0,16570
· SD currently: 0,10682
· Step count: 3
· Total regions: 5
```

FIGURE 24.2. Result of the worst tested configuration for lineNumber = 9.

25. Tessellation example 10

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.18799	0.04343	9
Grad.	0.01	0.18799	0.08146	13
Grad.	0.005	0.18799	0.03667	51
Grad. steps	Set A	0.18799	0.02918	25
Grad. steps	Set B	0.18799	0.03495	14
Grad. steps	Set C	0.18799	0.02409	18
Grad. steps	Set D	0.18799	0.06564	14
Sim. Annealing	Set A	0.18799	0.03050	30
Sim. Annealing	Set B	0.18799	0.03495	14
Sim. Annealing	Set C	0.18799	0.02336	26
Sim. Annealing	Set D	0.18799	0.06564	14

TABLE 10. Results obtained for lineNumber = 10. This tessellation is made up of 5 polygons.

Something interesting happens in this example. If we take a look at the results obtained from applying the gradient method, we can see that the symmetric differences obtained with steps 0.05 and 0.005 are both around 0.04. However, the one obtained with step 0.01, which is in between the two others, is around 0.08.

Again, the results obtained for the gradient method are better for set C than for set A.

Yet again, some results obtained with the simulated annealing are better than the ones obtained with the gradient method with steps, and some others are worse.

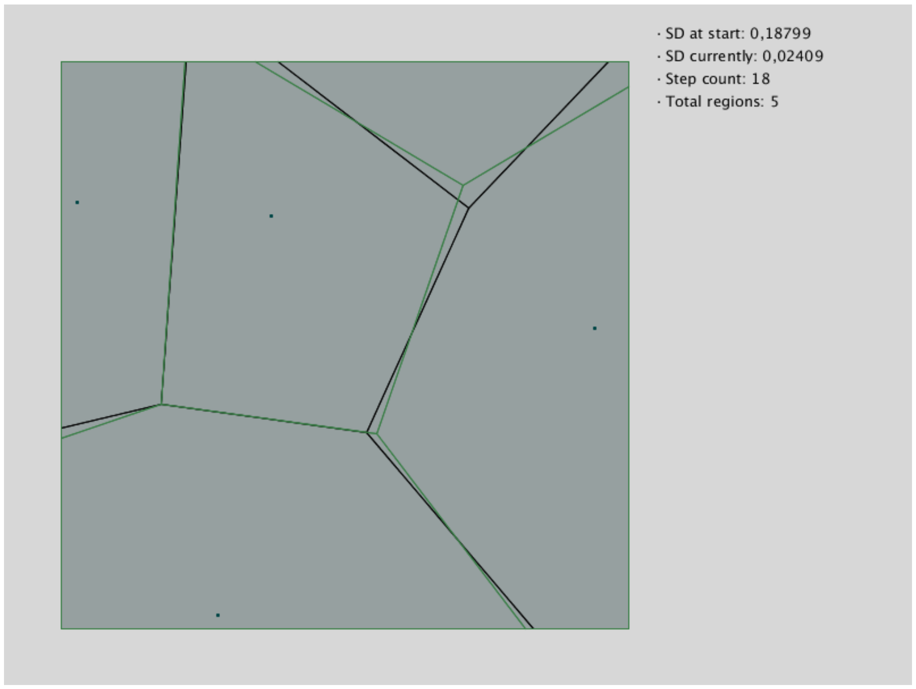


FIGURE 25.1. Result of the best tested configuration for lineNumber = 10.

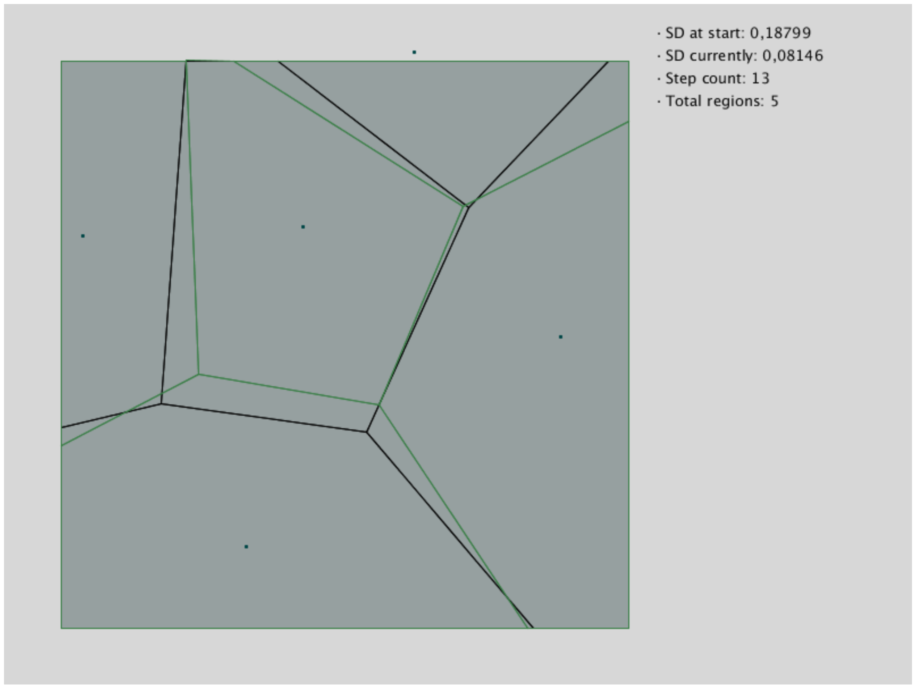


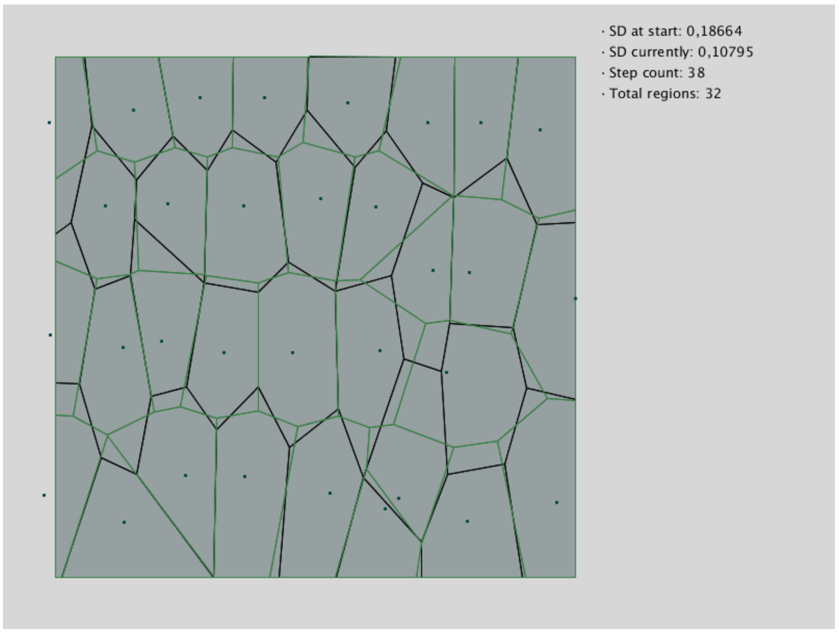
FIGURE 25.2. Result of the worst tested configuration for lineNumber = 10.

26. Teatros del Canal facade

	Parameters	Starting SD	Ending SD	Steps
Grad.	0.05	0.18664	0.15619	4
Grad.	0.01	0.18664	0.11353	11
Grad.	0.005	0.18664	0.11055	23
Grad. steps	Set A	0.18664	0.10795	38
Grad. steps	Set B	0.18664	0.11692	13
Grad. steps	Set C	0.18664	0.11205	20
Grad. steps	Set D	0.18664	0.13561	32
Sim. Annealing	Set A	0.18664	0.10795	38
Sim. Annealing	Set B	0.18664	0.11692	13
Sim. Annealing	Set C	0.18664	0.11205	21
Sim. Annealing	Set D	0.18664	0.13561	12

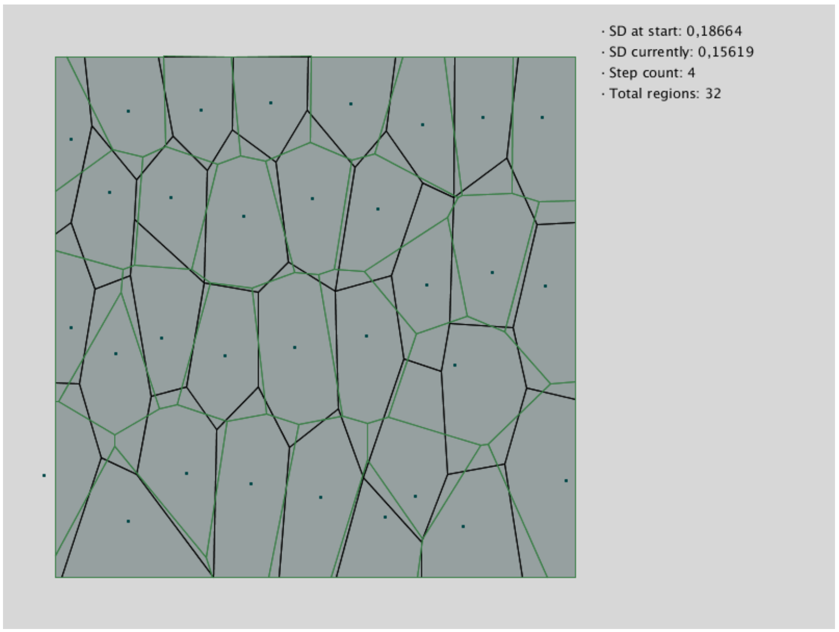
TABLE 11. Results obtained for the "Teatros del Canal" facade. This tessellation is made up of 32 polygons.

Despite resembling a Voronoi diagram, the results clearly indicate it does not come from one. No tested configuration is capable of breaking the 10% symmetric difference. If we take a closer look at the best obtained result in the next page (Figure 25.1), we can see that most vertical corners are a bit more pointy than what they should in order to be a perfect Voronoi diagram. Also, some other polygons (mostly those around the right hand side of the tessellation) do not fit very well. Whether or not the architect was inspired by Voronoi diagrams to design the facade will remain a mistery.



```
· SD at start: 0,18664
· SD currently: 0,10795
· Step count: 38
· Total regions: 32
```

FIGURE 26.1. Result of the best tested configuration for the “Teatros del Canal” facade.



```
· SD at start: 0,18664
· SD currently: 0,15619
· Step count: 4
· Total regions: 32
```

FIGURE 26.2. Result of the worst tested configuration for the “Teatros del Canal” facade.

From all these cases, I draw the following conclusion:

- First of all, there is no best method for all circumstances. The final solution we will get will depend heavily on each input. This implies that for each tessellation we want to adjust, we will have to spend some time testing which method with which parameters works better.
- Specific conclusions and thoughts about each method:
  - Gradient method: In most of the tests, it obtained the worst results. However, it is the one with the least average number of steps. It might be worth using this method when we are satisfied with results whose offset is around 10%.
  - Gradient method with steps: Despite being a simple improvement over the gradient method, it has reported fantastic results, very similar to the ones obtained with the simulated annealing technique.
  - Simulated annealing: Before testing and seeing the results, I thought that the simulated annealing results would be so much better than the ones obtained with any other method. However, they are very similar to the ones obtained with the gradient method with steps.
- From within each method, we cannot guarantee that a set of steps will work better than other one under all circumstances.
- Specific conclusions and thoughts about the different sets of steps:
  - Set A and Set C: These two sets are the ones that got consistently better results. They both provide relatively huge steps at the beginning and a small step for the last iteration.
  - Set B: Set B aimed to be a hybrid between sets A and C. It has managed to get similar results to both of them and in most cases, using a signifant less amount of steps than A or C (an average of around 40% less steps).
  - Set D: This set aimed to provide accurate enough fittings through big step changes. Its approach did not work as I expected. In many tests, it was beaten by the simple gradient method, in both the final symmetric difference and in the number of steps taken.

Part 5. USER GUIDE

In order to run the application in your computer, you will need to:

- Have Processing installed.
- Download the lastest version of the source code.

27. Installing Processing

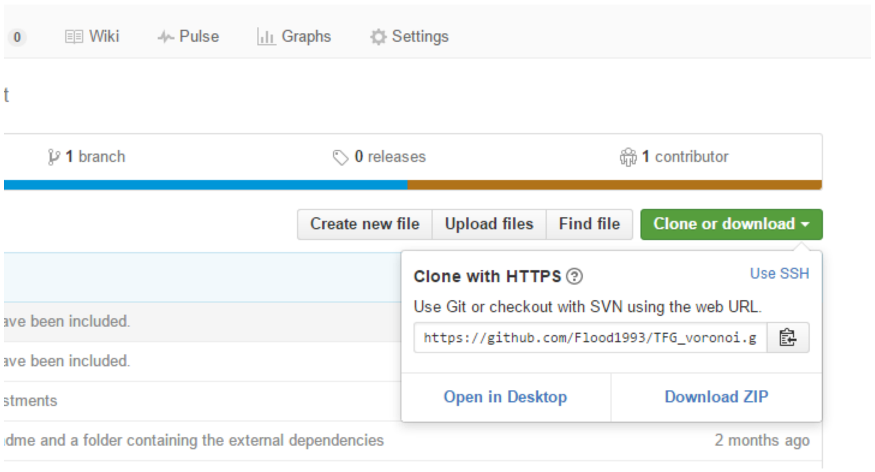
You can download the lastest version of Processing from its website: https://processing.org/

Make sure you download the version corresponding to your operative system and processor architecture. When the download has finished, proceed to install it.

28. Downloading the lastest version of the source code and installing the application

You can download the lastest version of the application from my the following GitHub repository: https://github.com/Flood1993/TFG_Voronoi

Open that link in your web browser of choice and click the "Clone or download" button. Then click on "Download ZIP".



Use Git or checkout with SVN using the web URL.

https://github.com/Flood1993/TFG_voronoi.g

ave been included.

ave been included.

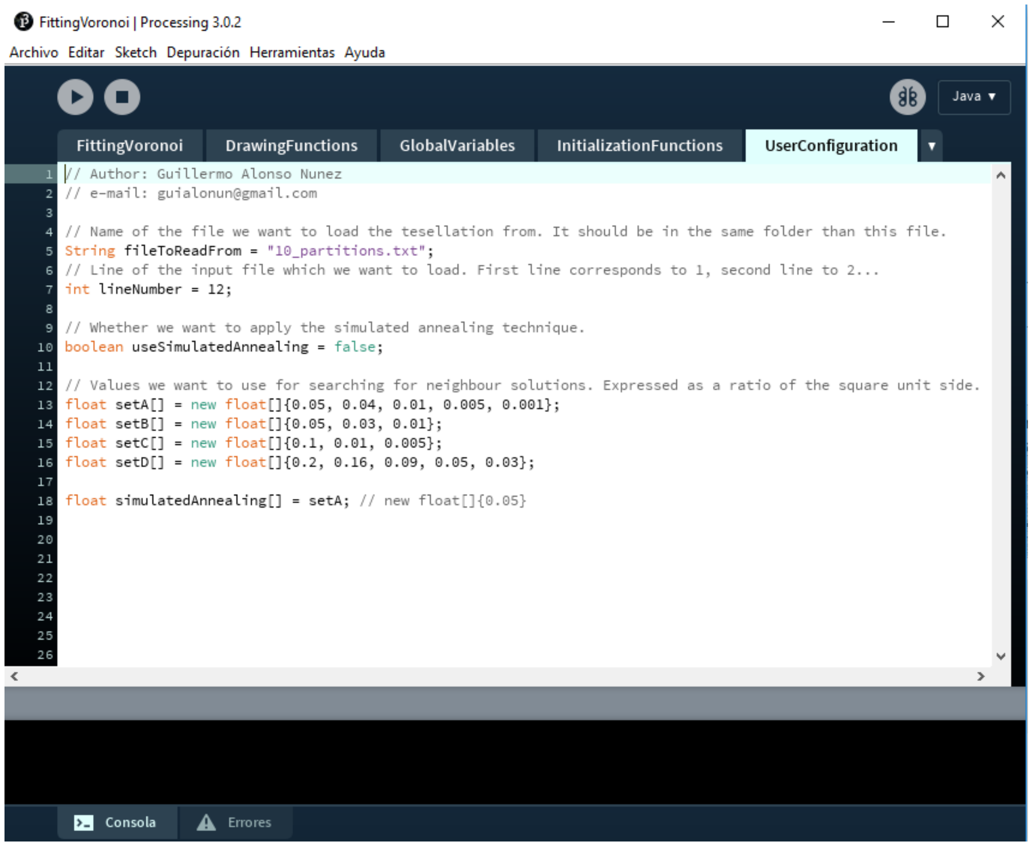
stments

dme and a folder containing the external dependencies

FIGURE 28.1. Screenshot of the GitHub repository. Click on the green button and then on "Download ZIP".

When the download has finished, you will need to extract the "libraries/" folder into your "Processing/" folder. This will copy the mesh library, used for calculating the Voronoi diagrams from the seeds. Then, extract the "FittingVoronoi" folder to a folder of your choice.

After you have done this, open Processing and open any .pde file from the "FittingVoronoi" folder. That should load the whole sketch. Then click on the UserConfiguration tab.



```
// Author: Guillermo Alonso Nunez
// e-mail: guialonun@gmail.com

// Name of the file we want to load the tesellation from. It should be in the same folder than this file.
String fileToReadFrom = "10_partitions.txt";
// Line of the input file which we want to load. First line corresponds to 1, second line to 2...
int lineNumber = 12;

// Whether we want to apply the simulated annealing technique.
boolean useSimulatedAnnealing = false;

// Values we want to use for searching for neighbour solutions. Expressed as a ratio of the square unit side.
float setA[] = new float[]{0.05, 0.04, 0.01, 0.005, 0.001};
float setB[] = new float[]{0.05, 0.03, 0.01};
float setC[] = new float[]{0.1, 0.01, 0.005};
float setD[] = new float[]{0.2, 0.16, 0.09, 0.05, 0.03};

float simulatedAnnealing[] = setA; // new float[]{0.05}
```

FIGURE 28.2. Screenshot of the Processing user interface.

That tab contains everything you should change for using the application.

- **fileToReadFrom**: Modify the filename between the quotation marks for the file you want to use. Note that the file should be placed in the "FittingVoronoi" folder.
- **lineNumber**: Modify this value to match the line you want to load from the file.
- **useSimulatedAnnealing**: This value should be "true" or "false". Set to "true" if you want to accept worse solutions according to the simulated annealing technique. "false" otherwise.
- **simulatedAnnealing[]**: This value can either be one of the predefined sets, or a custom one. The syntax for creating a set can be seen in the set definition above, from set A, for example. Copy that structure for your own one.

When you are ready, save your changes and click on the button with the play icon (a triangle) located on the top left corner.

Execution of the program will begin. When it has finished, simply close the pop up window or hit the stop button (right to the play button).

Part 6. **BIBLIOGRAPHY**

- Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, Sung Nok Chiu, 1992, “Spatial tessellations: Concepts and Applications of Voronoi Diagrams (Second Edition)”
- Greg Aloupis, Hebert Pérez-Rosés, Guillermo Pineda-Villavicenco, Perouz Taslakian, Dannier Trinchet-Almaguer, 2013, “Fitting Voronoi Diagrams to Planar Tesselations”
- https://processing.org – Open source programming language and IDE in which the whole project is coded
- http://leebyron.com/mesh/ - External processing library used for calculating the Voronoi diagrams
- http://www.lyx.org/ - Document processor used for writing this thesis
- https://github.com/Flood1993/TFG_voronoi - Git repository containing everything about this project