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. Tratar 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.

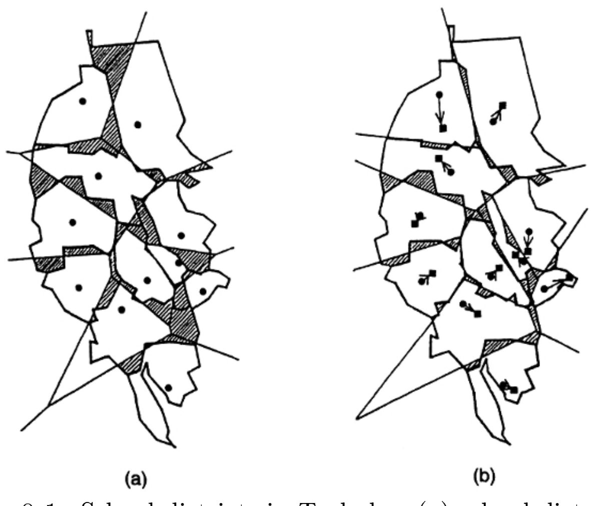


FIGURE 0.1. School districts in Tsukuba: (a) school districts and the present sites of schools (filled circles), the students in the shaded regions cannot go to their nearest schools; (b) school sites (filled squares) that minimize the area in which students cannot go to their nearest schools. (Source: Suzuki and Iri, 1986a, Figure 14.)

El ajuste de una partición del plano mediante diagramas de Voronoi también 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. La Figura 0.1(a) muestra los diferentes distritos de institutos en Tsukuba (los círculos muestran la ubicación de los institutos). En Japón, los estudiantes de un distrito deben ir al instituto asignado a ese distrito. Como resultado, en algunas áreas los estudiantes tienen que ir a un instituto que no es el más cercano. Si asumimos que los estudiantes pueden llegar al instituto siguiendo el camino Euclidiano en la Figura 0.1, las zonas en las que los estudiantes no pueden ir al instituto 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 solving. The inverse Voronoi diagram problem consists on detecting whether a given plane partition is a Voronoi diagram and finding the seed points that would generate such a partition. For a partition 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 partitions 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.

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 diagrams.
- Symmetric difference.

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 document, 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.

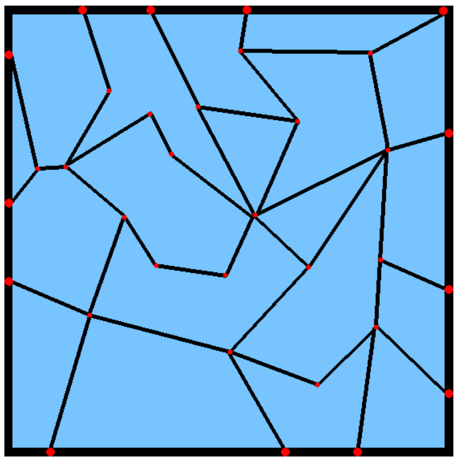


Figure 0.2. Tesellation of the unit square. As it can be seen, there are no overlaps or gaps between the polygons.

Mathematecally, 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 tesellation T can be expressed as:

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

Figure 0.2 shows a tesellation of the unit square displaying each subset A, B, C in blue, black and red, respectively.

Voronoi diagrams

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

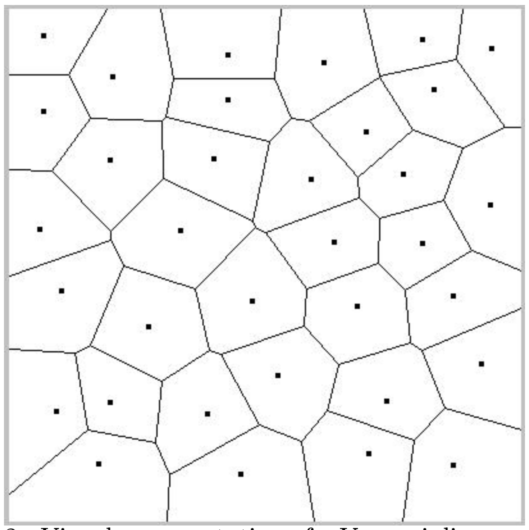


Figure 0.3. Visual representation of a Voronoi diagram inside the unit square. The black squares represent the seeds that generate the diagram.

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 a locus defined by the points P_k which are closer to some seed S_n.

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.

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 seeds.

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 $\mathrm{V_1}$which 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 $\mathrm{V_1}$. 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 Voronoi diagrams $\mathrm{V_2}$, $\mathrm{V_3}$, ... $\mathrm{V_n}$ fit better than the current best solution.

(3) Repeat this process until we cannot reach a better state. Return $\mathrm{V_k}$, which will be the last obtained Voronoi diagram.

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.

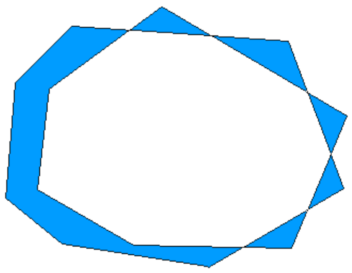


Figure 0.4. In blue, we can see a representation of the symmetric difference of two polygons. The highlighted area is the one that belongs to the symmetric difference. As we can see, it is the area that belongs to either polygon, but not both.

Applying the concept to two partitions, 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.

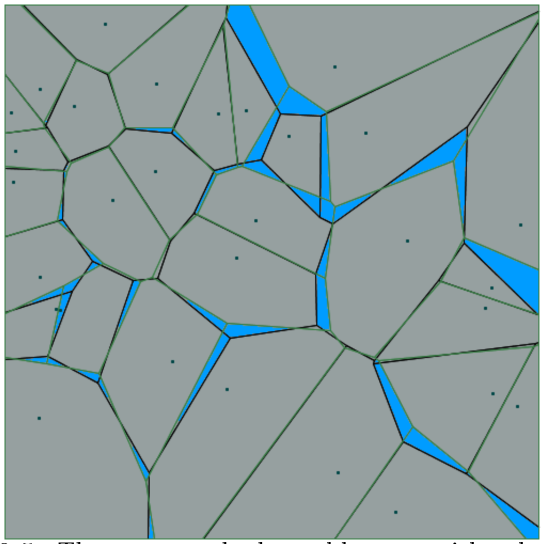


FIGURE 0.5. The new method would start with whole square painted in blue. We then subtract all the intersections of each pair of polygons. Graphically, the symmetric difference would be the total blue area of the picture in proportion to the area of the whole square.

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 partitions are exactly the same.

Inverse Voronoi problem

As mentioned earlier, the inverse Voronoi diagram problem consists on detecting whether a given plane partition is a Voronoi diagram and finding the seed points that would generate such a partition. From the mathematical point of view, it is interesting that, for partitions 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.

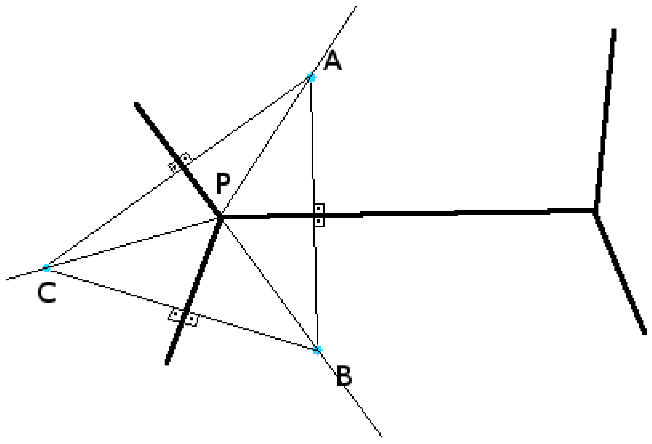


Figure 0.6. Region extracted from a Voronoi diagram. In blue, A, B and C are the seeds.

To prove it, we will start by knowing the tesellation comes from a perfect Voronoi diagram. 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 0.6.

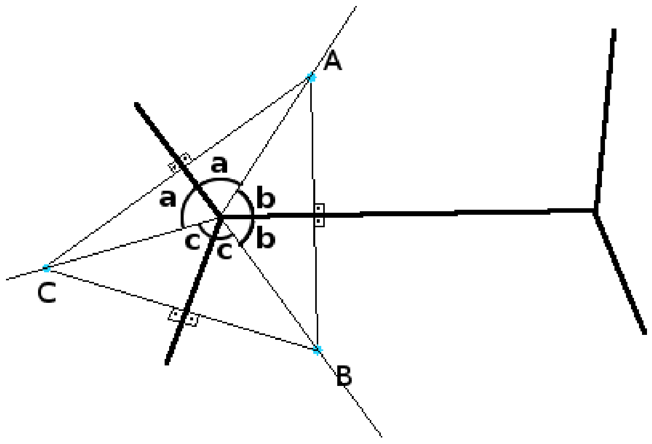


Figure 0.7. Angles formed by P, an edge and the lines that connect P with the corresponding two seeds of that edge.

Therefore, we know that the angles 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 0.7.

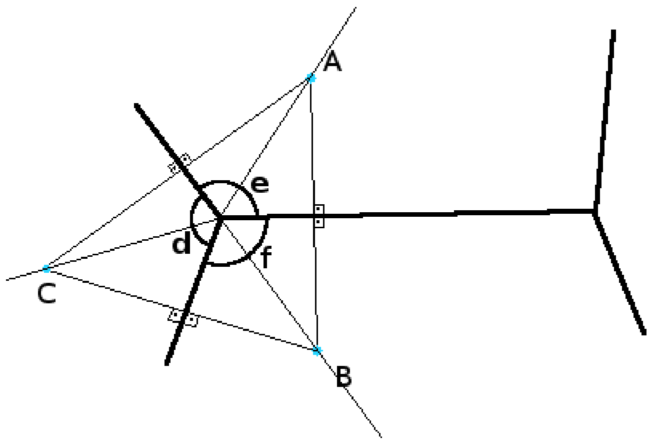


FIGURE 0.8. Angles between each pair of seeds.

Finally, let d, e and f 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, the unknown location of the seeds A, B and C can be obtained by solving the following system of linear equations:

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

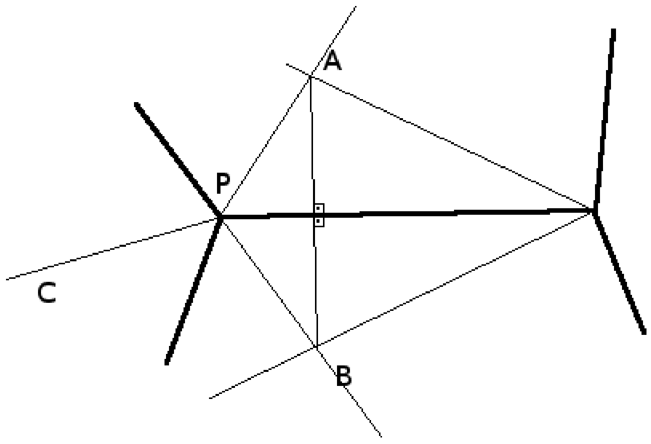


FIGURE 0.9. Seeds A and B have been located applying the explained method twice.

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 neighbour vertex, the exact position of seeds A and B is known. From there, we can calculate any other seed by just mirroring the known seeds with the surrounding edges.

But what happens when the partition 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. Thus, we want to explore different methods of fitting an arbitrary tesellation, whether it comes from a Voronoi diagram or not.

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 of, 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.

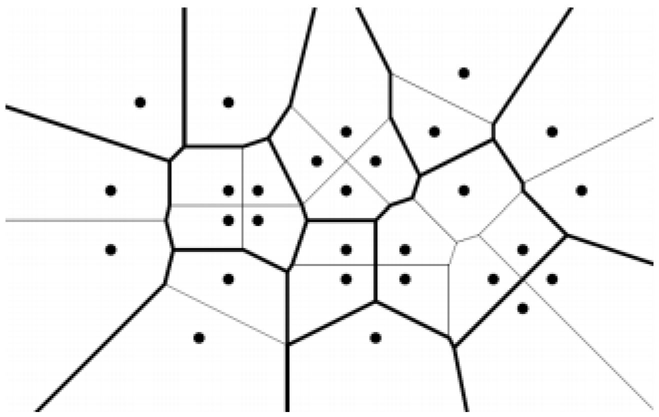


Figure 0.10. Representation of the generalized inverse Voronoi problem. Thick edges represent the original input tesellation.

Voronoi diagrams in real life

If one has the chance to go for a walk in Madrid, he/she might find itself in front of the "Teatros del Canal". If he is familiar with Voronoi diagrams, something will catch his eye: some parts of the decoration of that building resemble those diagrams! Further in this project, we will adjust them and see how well they fit as a Voronoi diagram.

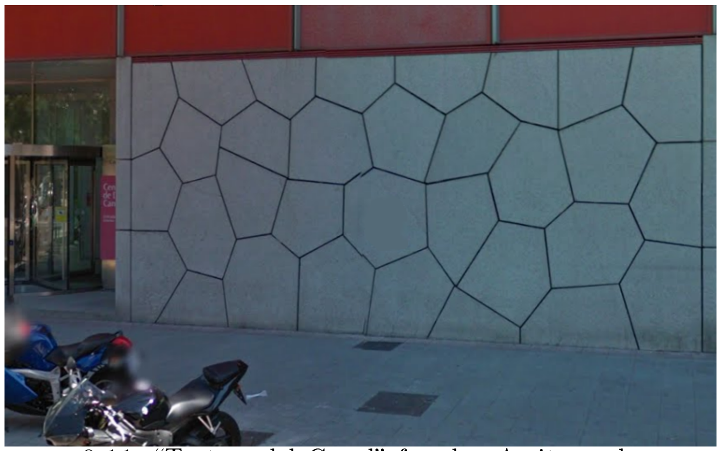


Figure 0.11. "Teatros del Canal" facade. As it can be seen, it looks very similar to a Voronoi diagram.

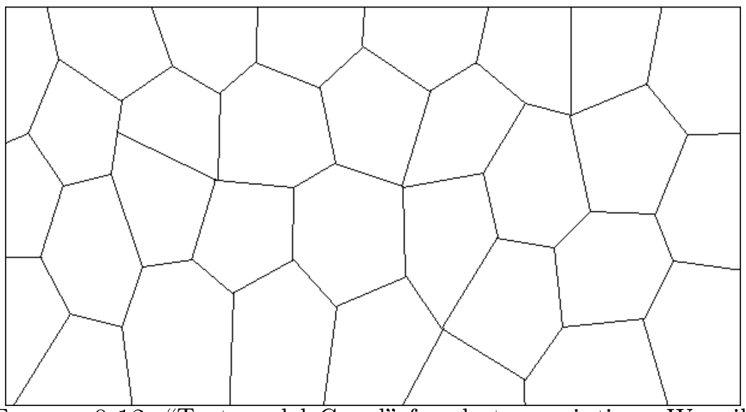


Figure 0.12. "Teatros del Canal" facade transcription. We will later use this to see and compare the results of the different fitting methods.

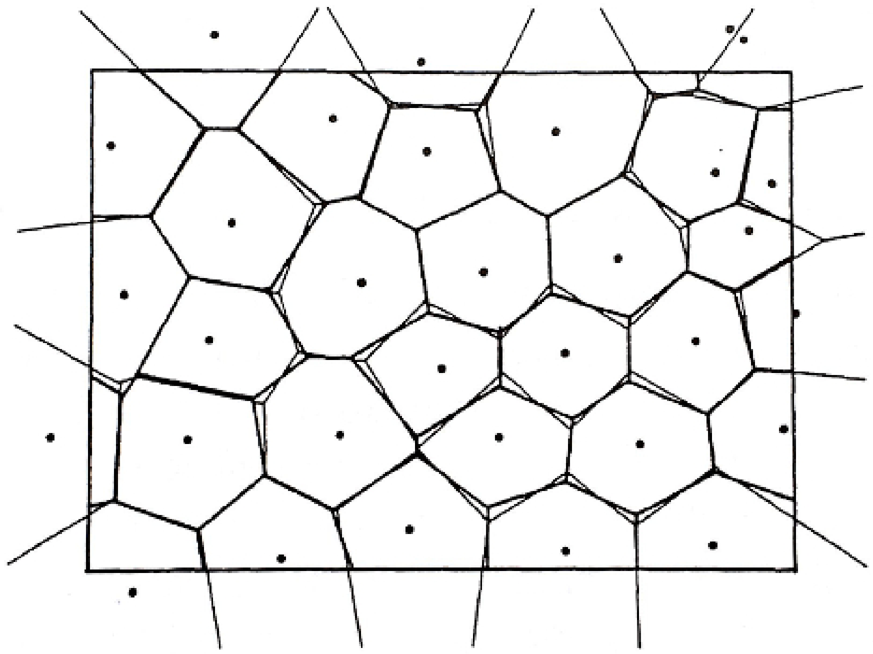


FIGURE 0.13. Fitting a Voronoi diagram to the territories of mouthbreeder fish. (Source: Suzuki and Iri, 1986a, Figure 13.)

Figure 6.3 illustrates an actual application carried by Suzuki and Iri (1986b) who fit a Voronoi diagram to the territories of mouthbreeder fish, a fish known for being highly predatory and extremely territorial.

OBJECTIVES

The objectives of this project are:

- Fit a given partition of the unit square as much as we can 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 partition is in fact a Voronoi diagram, we would like to arrive to that conclusion. 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.

DEVELOPMENT

Why Processing: Pros and cons of Java

Despite not being the most efficent language for crunching raw numbers, I decided to develop my program in Java due to the following reasons:

- First of all, I had already used it in the past for a different project, and one of the biggest advantages of it is the ease to display information in the screen.
- 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.
- 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 partitions and polygons.
- The number of regions we are going to be treating in our research will be 200 maximum, a reasonable number for any modern programming language.

Line clipping. Polygon clipping

There are two situations in the problem in which we have to calculate 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. This is doable since we will be working 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, but for explanation purposes, we will apply the algorithm to it. Also note that the line has a direction, given by the blue arrow.

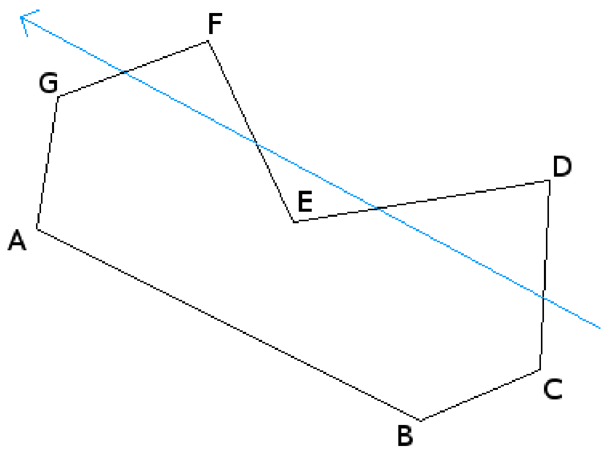


Figure 0.14. Starting point. We will clip the polygon with the given line (in blue). Starting point will be A.

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.

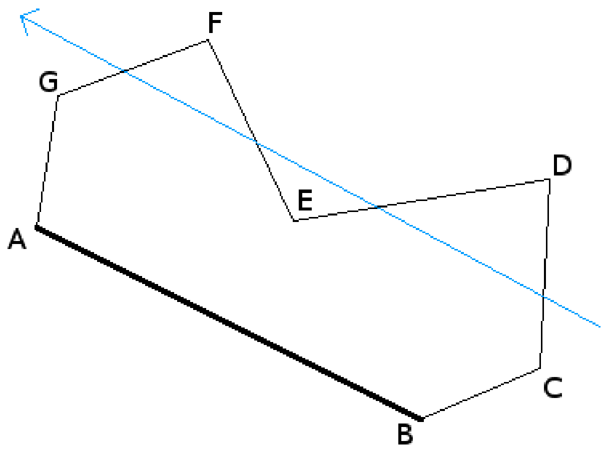


Figure 0.15. Since point B lies on the left of the line, add it to the result.

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.

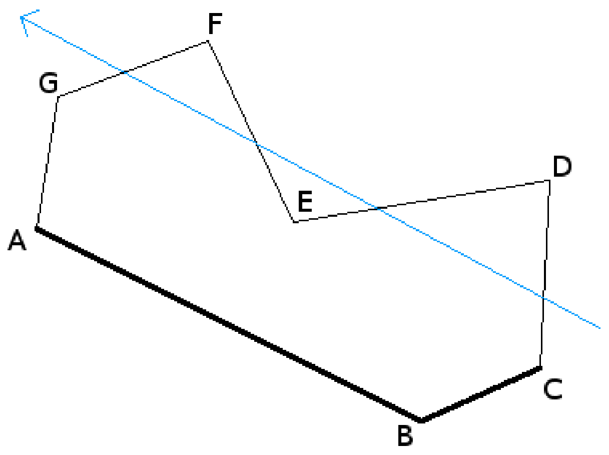


FIGURE 0.16. Since C is still on the left of the line, we add it to the final result too.

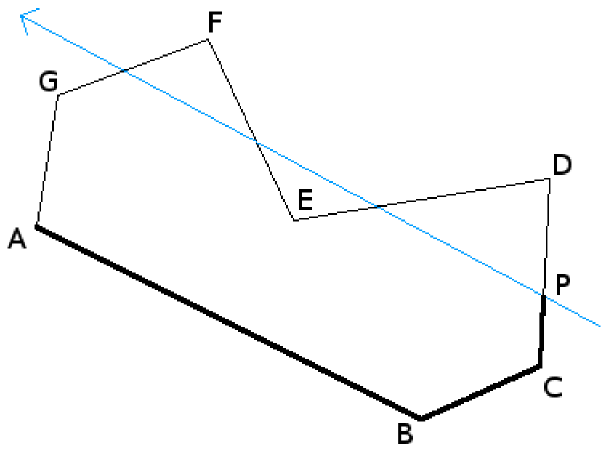


FIGURE 0.17. The next point, D, does not lie on the left of the blue line. Since last one did, it means an intersection point P has to exist. Add P to the final result.

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.

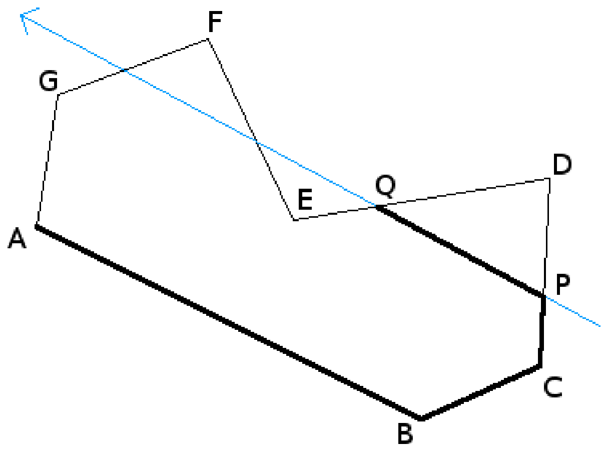


FIGURE 0.18. Current point D lies on the right of the line. Since E lies on the left, it means another intersection Q must exist. Add Q to the final result.

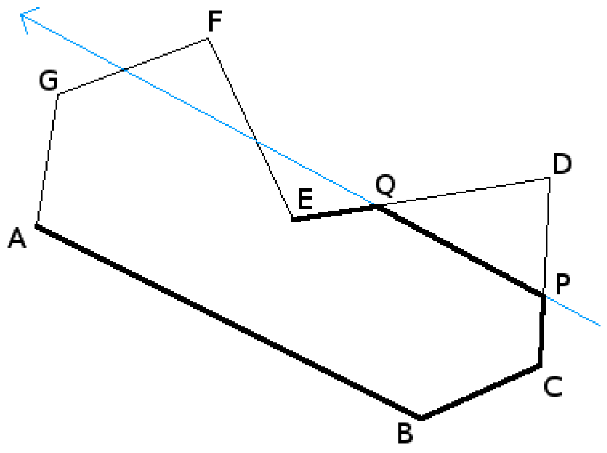


FIGURE 0.19. Point E lies on the left. Add it to the solution.

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

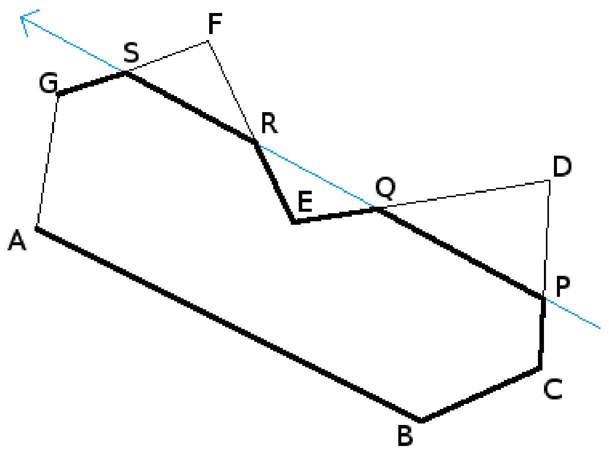


Figure 0.20. On the left again. Add G to the result.

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

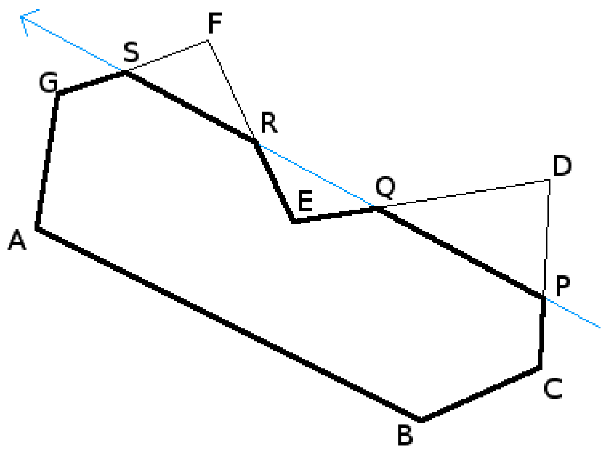


Figure 0.21. We arrived to A, our starting point.

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.

Input data

The input data for the problem is a partition 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 degrees). The input data will be read from a *.txt file in which each line will contain a whole partition.

If you are familiar with the Python programming language, you will easily detect 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:

$Fruits = ("Pear", "Apple", "Grape")$

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

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

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

A tuple is a list of two floating point numbers, which represents both coordinates.

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

A partition is a list of polygons.

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

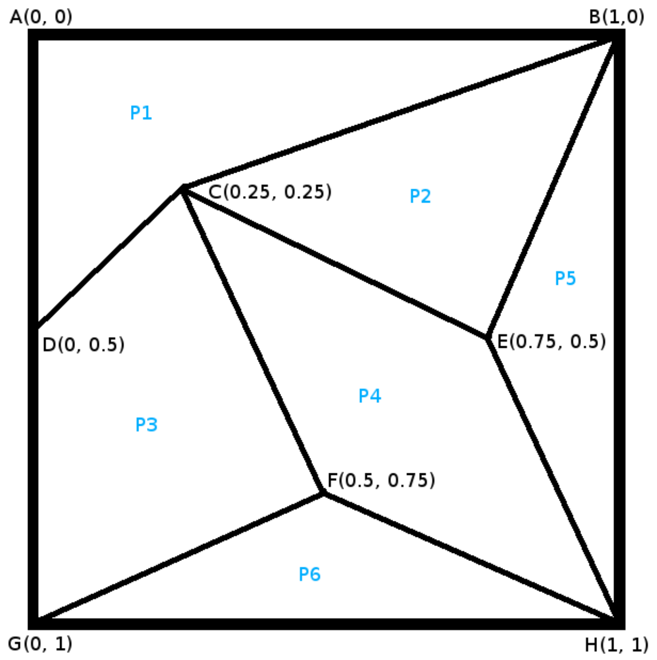


Figure 0.22. Tesellation of the plane we want to adjust.

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 (anticlockwise - 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:
$Tes = [[A, B, C, D], [C, B, E], [D, C, F, G], [C, F, H, E], [E, H, B], [F, G, H]]$

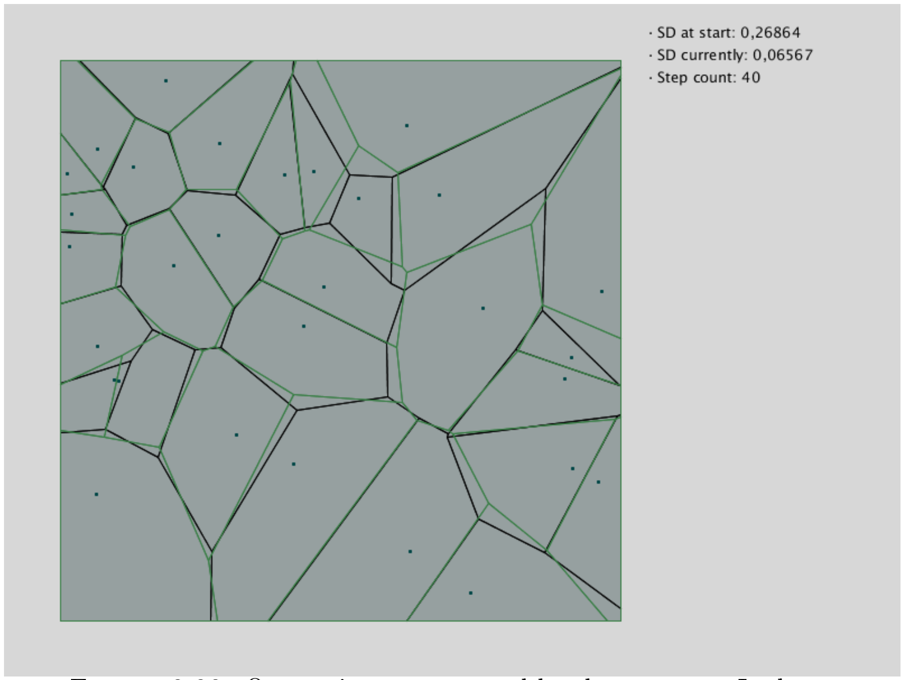
Tes=[[(0,0),(1,0),(0.25,0.25),(0,0.5)],[(0.25,0.25),(1,0),(0.75,0.5)],[(0,0.5),(0.25,0.25),(0.5,0.75),(0,1)],[(0.25,0.25),(0.5,0.75),(1,1),(0.75,0.5)],[(0.75,0.5),(1,1),(1,0)],[(0.5,0.75),(0,1),(1,1)]]

Then, we will have to place it in some file and provide the program the file name and the line of that partition.

Output data

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

- output.csv: CSV file in which each line represents each iteration of the algorithm. Each line contains 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 0.23. Output image generated by the program. It shows the final result, as well as some other information of the process.

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 neighbour 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 is bounded, 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.

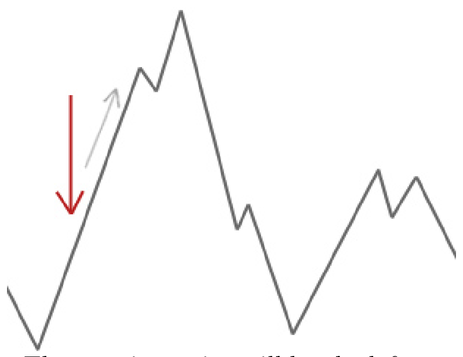


Figure 0.24. The starting point will be the left arrow. As we can see, this method would indicate us to go right in order to get to a better solution.

As it can be seen in Figure 0.24, 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.

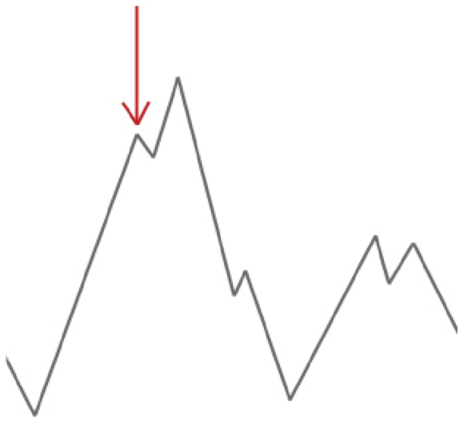


Figure 0.25. Ending point using a simple gradient method.

However, when reaching this point, it would check for its neighbours 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.

Improved gradient method: implementing steps

One little improvement we can apply to the gradient method is the gradient method with variable step. It consists on iterating different gradient methods, but varying the step size we are taking 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.

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.

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.

Although the gradient method (specially with steps) can be surprisingly effective at finding a good solution, it also has a tendency to get stuck in local optimums. The simulated annealing algorithm is excellent at avoiding this problem and is much better on average at finding an approximate global optimum.

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

- If the neighbour 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 neighbour 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 low, so will be the chances.

The implementation of this is very simple. On each step, if the neighbour 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 neighbour 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.

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 neighbour 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 neighbour solution N from the current solution.
  - If the N is better than the current solution: Mark it as the current solution. If it is better than the best solution, mark the best to the current solution.
  - If N is worse, but we accepted it according to the acceptance probability, mark it as the current solution.
- Reduce the temperature T.
- Repeat until steps until a certain stop condition is met.

The stop condition will generally be a number of X steps to check at maximum.

Getting rid of the "repetition" problem: randomizing point order in each state. Given the nature of our 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. This is done simply by reordering the indexes of the array each iteration.

Results and conclusions

The following sets have been used during the study of the different methods.
$A = \{0.05, 0.04, 0.01, 0.005, 0.001\}$
$B = 0.05, 0.03, 0.01$
$C = 0.1, 0.01, 0.005$

Partition example 8

	Parameters	Starting SD	Ending SD	Total adjustment	Steps	Adj/step
Grad.	0.05	0.26864	0.12694	0.14170	6	0.02116
Grad.	0.01	0.26864	0.10517	0.16347	20	0.00526
Grad.	0.005	0.26864	0.10109	0.16755	32	0.00316
Grad. steps	Set A	0.26864	0.06518	0.20346	29	0.00225
Grad. steps	Set B	0.26864	0.07564	0.19300	15	0.00504
Grad. steps	Set C	0.26864	0.06018	0.20847	37	0.00163
Sim. Annealing	Set A	0.26864	0.06449	0.20416	31	0.00208
Sim. Annealing	Set B	0.26864	0.07564	0.19300	15	0.00504
Sim. Annealing	Set C	0.26864	0.06788	0.20076	27	0.00251

Table 1. Results obtained for lineNumber = 8. This partition is made up of 30 polygons.

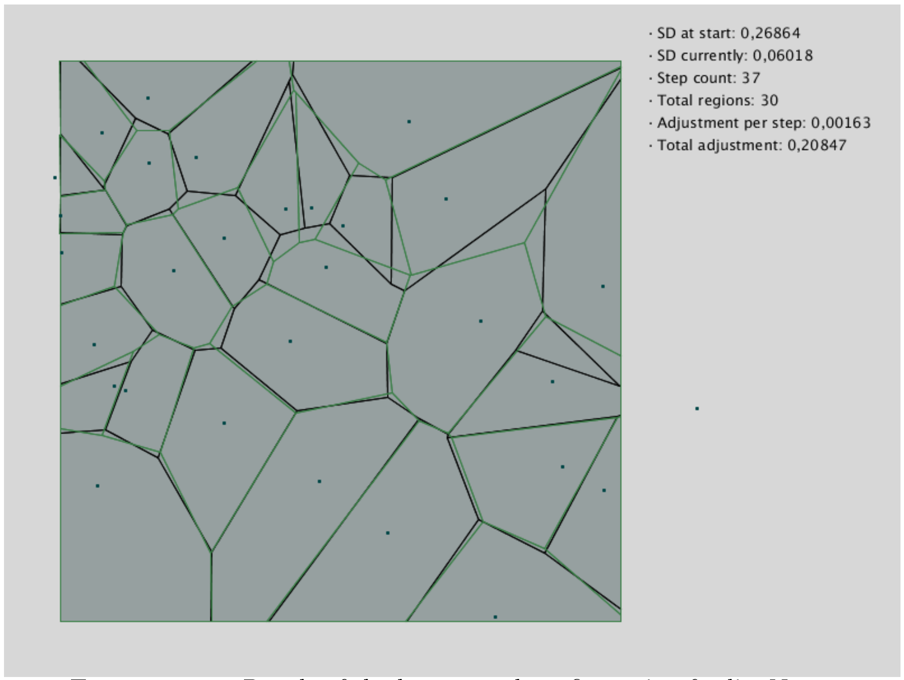
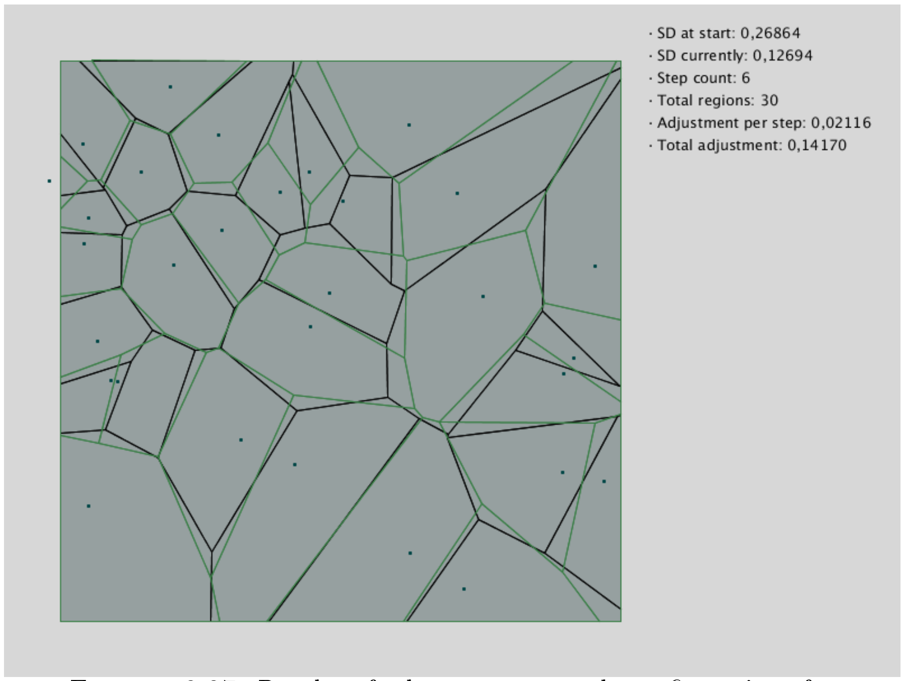

· SD at start: 0,26864
· SD currently: 0,06018
· Step count: 37
· Total regions: 30
· Adjustment per step: 0,00163
· Total adjustment: 0,20847

Figure 0.26. Result of the best tested configuration for lineNumber = 8.



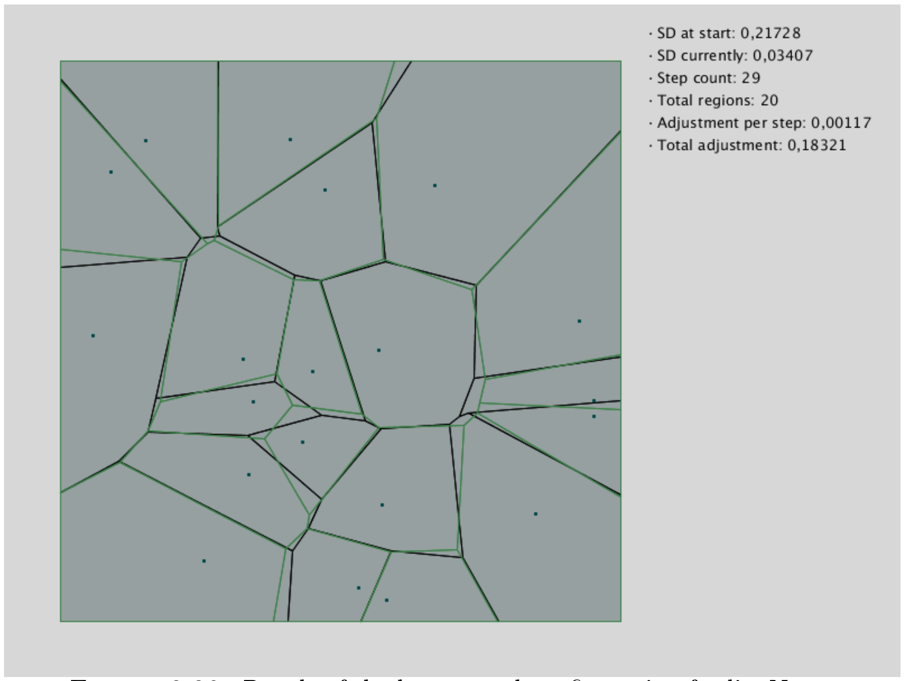
· SD at start: 0,26864
· SD currently: 0,12694
· Step count: 6
· Total regions: 30
· Adjustment per step: 0,02116
· Total adjustment: 0,14170

FIGURE 0.27. Result of the worst tested configuration for lineNumber = 8.

PARTITION EXAMPLE 1

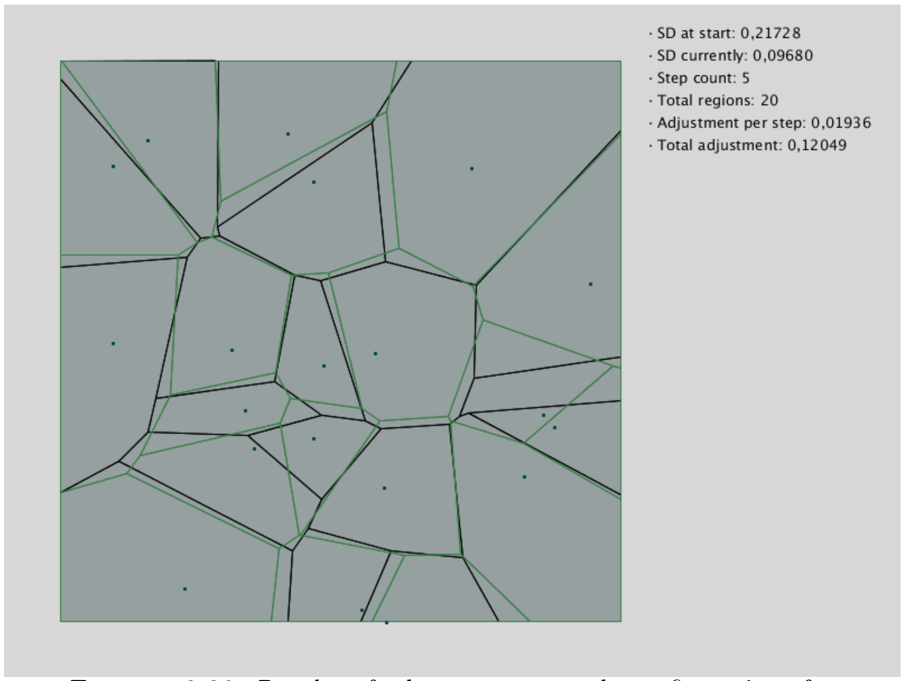
	Parameters	Starting SD	Ending SD	Total adjustment	Steps	Adj/step
Grad.	0.05	0.21728	0.09680	0.12049	5	0.01936
Grad.	0.01	0.21728	0.06632	0.15097	13	0.00510
Grad.	0.005	0.21728	0.05689	0.16039	27	0.00211
Grad. steps	Set A	0.21728	0.04463	0.17625	63	0.00071
Grad. steps	Set B	0.21728	0.05221	0.16507	17	0.00307
Grad. steps	Set C	0.21728	0.03407	0.18321	29	0.00117
Sim. Annealing	Set A	0.21728	0.04463	0.17625	65	0.00069
Sim. Annealing	Set B	0.21728	0.05221	0.16507	17	0.00307
Sim. Annealing	Set C	0.21728	0.03407	0.18321	29	0.00117

TABLE 2. Results obtained for lineNumber = 1. This partition is made up of 20 polygons.



· SD at start: 0,21728
· SD currently: 0,03407
· Step count: 29
· Total regions: 20
· Adjustment per step: 0,00117
· Total adjustment: 0,18321

FIGURE 0.28. Result of the best tested configuration for lineNumber = 1.



· SD at start: 0,21728
· SD currently: 0,09680
· Step count: 5
· Total regions: 20
· Adjustment per step: 0,01936
· Total adjustment: 0,12049

FIGURE 0.29. Result of the worst tested configuration for lineNumber = 1.

Bibliografía

- Spatial tessellations: Concepts and Applications of Voronoi Diagrams (Second Edition) - Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, Sung Nok Chiu
- 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
- http://apmonitor.com/me575/index.php/Main/SimulatedAnnealing - Simulated annealing
- http://www.exploringbinary.com/why-0-point-1-does-not-exist-in-floating-point/ - Inaccuracy of computers when dealing with floating-point numbers