GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

TÉCNICAS METAHEUSRÍSTICAS PARA LA MAXIMIZACIÓN DE LA REGIÓN DE VORONOI

Autor: Javier Álvarez Martínez

Director: Santiago Canales Cano

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Técnicas metaheurísticas para la maximización de la Región de Voronoi

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2021/22 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Javier Álvarez Martínez Fecha: 05/ 07/ 2022

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Santiago Canales Cano Fecha: 05/ 07/ 2022

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

TÉCNICAS METAHEURÍSTICAS PARA LA MAXIMIZACIÓN DE LA REGIÓN DE VORONOI

Autor: Javier Álvarez Martínez

Director: Santiago Canales Cano

Madrid

Agradecimientos

Me gustaría mostrar mi agradecimiento a mi director de este proyecto, Santiago Canales Cano, por el apoyo, consejos, explicaciones y disponibilidad a lo largo de la realización del trabajo.

También me gustaría agradecer a mi familia por haberme dado la oportunidad de realizar la carrera de Ingeniería en Tecnologías de Telecomunicaciones en la Escuela Técnica Superior de Ingeniería (ICAI) y apoyarme en todo momento, incluso cuando las cosas no iban por el mejor camino. También quisiera agradecer a mis amigos el apoyo brindado en los momentos más complicados y estar siempre dispuestos a ayudar.

Por último, me gustaría agradecer a toda la Universidad Pontificia de Comillas y a los docentes, la oportunidad de formarme y darme los conocimientos necesarios para realizar este proyecto y sobre todo por formarme como ingeniero y como persona.

TÉCNICAS METAHEURÍSTICAS PARA LA MAXIMIZACIÓN DE LA REGIÓN DE VORONOI.

Autor: Álvarez Martínez, Javier.
Director: Canales Cano, Santiago.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

Este proyecto consiste en la creación de un software gráfico para la *Maximización de la Región de Voronoi* mediante la utilización de técnicas metaheurísticas, más concretamente la implementación mediante lenguaje Python de 3 técnicas heurísticas para solucionar el problema: *Random Search*, *Simulated Annealing* y *Ant Systems*.

Este software será capaz usando las técnicas enunciadas previamente de obtener de forma aproximada el punto cuya área de su *región de Voronoi* asociada sea máxima. Algunas de las aplicaciones que puede tener este proyecto pueden ser, por ejemplo, la correcta colocación de antenas de telefonía para delimitar el área del alcance que debe tener cada una o sus aplicaciones en el ámbito de la robótica, todo ello basándonos en los *Diagramas de Voronoi*.

Palabras clave: *Diagramas de Voronoi, maximización, región, región de Voronoi, técnicas metaheurísticas, random search, simulated annealing, ant systems.*

1. Introducción

La Geometría Computacional nace de la Geometría Clásica y la Geometría Informática. Consiste en la elaboración de herramientas y técnicas para resolver problemas de naturaleza geométrica centrándose en el diseño eficiente de algoritmos y estructuras de datos.

Los *diagramas de Voronoi* son una de las estructuras más conocidas dentro del mundo de la Geometría Computacional. Los podemos definir de la siguiente manera:

Definición: dada una región del plano *R* y un conjunto *N* de *n* puntos, podemos encontrar una subdivisión del espacio asociando a cada punto del conjunto *N* los puntos que se encuentren más cercanos a dicho punto que a cualquier otro. Esto es lo que conocemos como *región de Voronoi* y está muy relacionada con la *triangulación de Delaunay* [1]. En la Figura 1 podemos ver un ejemplo de lo que es un *diagrama de Voronoi de 15 puntos*.

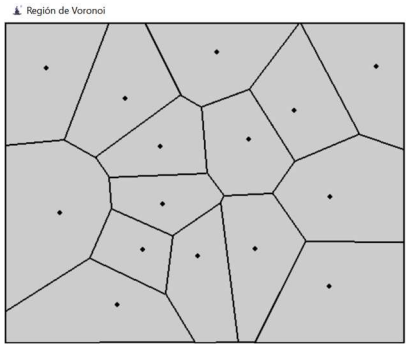


Figura 1. Ejemplo de Diagrama de Voronoi

El problema sin cotas algorítmicas eficientes ante el que nos encontramos actualmente es el de encontrar un nuevo punto *q* dentro de la región *R* tal que la región que le asociemos a dicho punto en el *diagrama de Voronoi* tenga área máxima.

La realización de este proyecto tiene como punto de partida y ha sido de gran utilidad la tesis: *"Métodos heurísticos en problemas geométricos. Visibilidad, iluminación y vigilancia"* [2] de Santiago Canales Cano.

2. Definición del proyecto

Este Proyecto tienen como objetivo la elaboración de un software gráfico e interactivo que permita resolver el problema previamente explicado, la *maximización de la región de Voronoi*, haciendo uso de diferentes técnicas metaheurísticas.

De forma sencilla, las técnicas metaheurísticas tienen como objetivo buscar una solución cercana a la óptima en un tiempo razonable. Este tipo de técnicas está teniendo un importante crecimiento dada la necesidad de dar soluciones concretas a problemas reales en el mundo de la Geometría Computacional en particular y de forma más general en el mundo de la Inteligencia Artificial.

Las tres técnicas en las que nos centraremos son: un algoritmo que hemos llamado *Random Search* y que consiste en una búsqueda aleatoria entre muchos puntos de aquel que consigue área máxima, un algoritmo basado en la técnica heursítica *Simulated Annealing,* que partiendo de una solución local es capaz de alcanzar otra solución de mayor calidad y por último. un algoritmo basasdo en la técnica heurística *Ant Systems*, o algoritmos de hormigas imitando la capacidad que tienen estos insectos de descubrir el mejor camino, o el camino más corto mediante la acumulación de la feromona que dejan al moverse.

3. Descripción del modelo/sistema/herramienta

El software del proyecto lo podemos dividir en varios archivos en función de la funcionalidad de cada uno de ellos. El esquema que hemos seguido lo podemos ver en la Figura 2 en la que tenemos un archivo principal *(Voronoi.py)* que es el encargado de la parte gráfica y de la comunicación con el resto de los programas, un archivo que contiene los algoritmos para la maximización de la región de Voronoi basados en geometría computacional *(funciones_gc.py),* un archivo que es capaz de abrir y guardar ficheros de texto que contienen los diagramas para que puedan ser representados en nuestra aplicación y que nos ha sido de gran utilidad para realizar el análisis de los algoritmos *(manejo_archivos.py)* y, por último, un archivo para el manejo de excepciones y mensajes en la aplicación para ayudar al usuario a la hora de usar la aplicación *(popup.py)*.

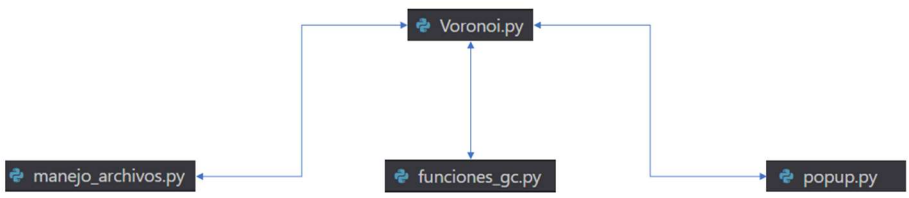
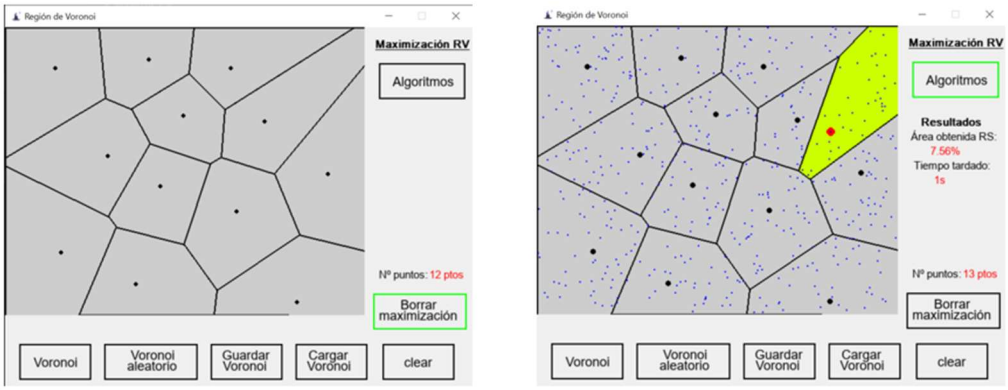


Figura 2. Arquitectura del proyecto

Como podemos entender, la aplicación la podemos dividir en dos partes principales. Una primera parte gráfica en la que nos encontramos con una aplicación capaz de representar diferentes regiones y permitir al usuario interactuar con la misma, integrada por los archivos *Voronoi.py, manejo_archivos.py y popup.py* y una segunda parte en la que se encuentra la parte de la geometría computacional encargada de realizar los procedimientos para encontrar las mejores soluciones a los problemas planteados que la podemos encontrar en el archivo *funciones_gc.py*.

4. Resultados

La aplicación resultante es capaz de, dada cualquier distribución de puntos introducida por el usuario obtener su *diagrama de Voronoi* y darle la posibilidad al usuario de obtener un nuevo punto que, al introducirlo en nuestro diagrama, tenga de forma aproximada la mayor área posible asociada. Un ejemplo de ejecución de la aplicación final la podemos ver en la Figura 3 donde se muestra la interfaz del usuario y un *diagrama de Voronoi* ya dibujado en (a) y en (b) podemos ver una solución al problema de la *maximización de la región de Voronoi* utilizando el algoritmo *Random Search* con 500 puntos aleatorios.



(a) *(b)*

Figura 3. Ejemplo final de la aplicación

5. Conclusiones

Con el objeto de realizar un estudio más en profundidad de los algoritmos se ha realizado una comparación de estos, hemos evaluado 50 diagramas diferentes generados aleatoriamente de 10, 50 y 100 puntos. En la Figura 4 podemos ver la comparativa entre el número de puntos inicial en el diagrama y el área media obtenida con cada uno de los algoritmos. Con el objetivo de ser lo más exactos posible y mantener una norma para el algoritmo *Random Search,* el número de puntos aleatorios que evalúa viene dado por la función 50*n siendo n el número de puntos iniciales del diagrama. Esto quiere decir que, si hay 10 puntos iniciales en el diagrama, el algoritmo *Random Search* habrá evaluado 500 puntos aleatorios.

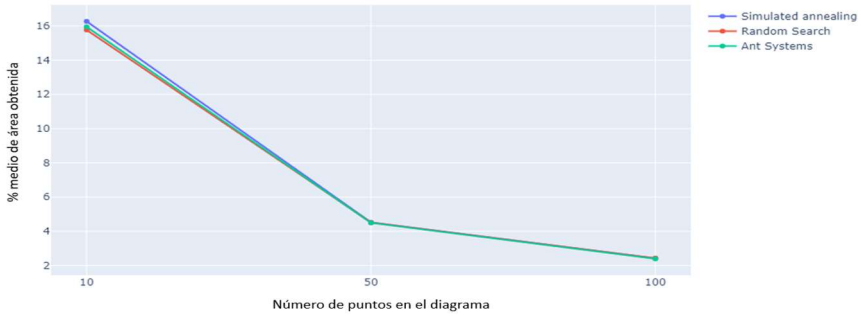


Figura 4. Comparativa de porcentaje medio de área obtenida por cada técnica

En la Figura 4 podemos observar cómo los valores del área obtenido por los 3 algoritmos son muy similares por lo que nos hace entender que es posible que estén encontrando la solución cercana al óptimo para la mayoría de los casos estudiados. Sin embargo, si es significativa la diferencia de tiempos medios obtenidos por cada uno de los algoritmos hasta dar con la solución óptima. Como podemos ver en las Tabla 1, Tabla 2 y Tabla 3, el porcentaje de área obtenido con los algoritmos es muy similar, sin embargo, aunque parezca leve el algoritmo *Simulated Annealing* devuelve en media un área de mayor calidad, aunque el coste de tiempo es significativamente más alto.
Se entiende que a la hora de utilizar estos algoritmos hay que tomar una decisión entre la calidad de la solución obtenida y el tiempo necesario hasta alcanzar dicha solución.

Puntos	*% Medio Área Región Voronoi*	*Tiempo [segundos]*
10	15,78	0,06
50	4,51	9,88
100	2,42	23,64

*Tabla 1. Resultados Random Search con m = 50*Puntos*

Puntos	*% Medio Área Región Voronoi*	*Tiempo [segundos]*
10	16,27	4,36
50	4,52	13,42
100	2,43	43,06

Tabla 2. Resultados Simulated Annealing con: $T_0 = 2*Puntos$, $T_f = 0{,}005$ *y* $T_i = \frac{T_0}{1+i}$

Puntos	*% Medio Área Región Voronoi*	*Tiempo [segundos]*
10	15,95	1,86
50	4,48	7,7
100	2,38	25,24

Tabla 3. Resultados Ant Systems con iteraciones $= \frac{Puntos}{2}$ $\forall\ Puntos > 20$ *o iteraciones = 2*Puntos* $\forall\ Puntos < 20$

6. Referencias

[1] Berg de, M.; Kreveld van, M.; Overmars, M.; Schwarzkopf, O.; "Computational Geometry. Algorithms and Applications", Springer, 1997.

[2] Canales Cano, Santiago. "Métodos heurísticos en problemas geométricos. Visibilidad, iluminación y vigilancia", Tesis doctoral, 2004.

METAHEURISTICS TECHNIQUES FOR THE MAXIMIZATION OF THE VORONOI REGION

Author: Álvarez Martínez, Javier.
Director: Canales Cano, Santiago.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

PROJECT SUMMARY

This project consists of the creation of a graphic software for the *Maximization of the Voronoi Region* using metaheuristics techniques, more specifically by the implementation by Python of 3 techniques heuristics for solving the problem: *Random Search, Simulated Annealing and Ant Systems.*

This software will be able to use the techniques stated above to obtain in an approximately the point whose area of its associated *Voronoi´s region* associated is maximum. Some of the applications this project may have can be, for example, the correct placement of telephony antennas, to delimit the area of range that each one or their applications in the robotic field, all based in the *Voronoi Diagrams.*

Key words: *Voronoi Diagrams, maximization, region, Voronoi region, metaheuristics techniques, random search, simulated annealing, ant systems.*

1. Introduction

Computational Geometry is born from Classical Geometry and Computer Geometry. It consists of the elaboration of tools and techniques for solving problems of geometric nature focusing on the efficient design of algorithms and data structures.

Voronoi Diagrams are one of the most known structures in the Computational Geometry world. They can be defined as follows:

Definition: given a region of the *plane R* and a set N of n points, a subdivision of space can be found associating each point of the set N the points the points that are closer to that point than to any other. This is what it is known as *Voronoi region,* and it is closely related to the *Delaunay triangulation* [1]. In the Figure 5, it can be observed an example of the *Voronoi diagram* of 15 points.

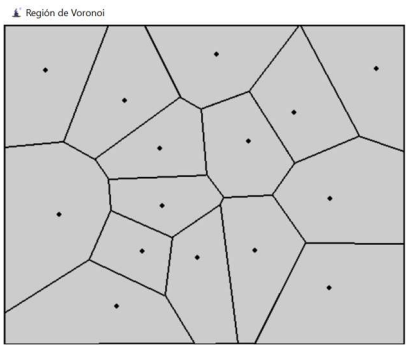


Figure 5. Voronoi Diagram Example

The problem without efficient algorithmic dimensions that are currently faced is finding a new point *q* inside the region *R* such the region associated to said point in the *Voronoi diagram* has a maximum area.

The realization of this project has as its starting point and has been of great utility the thesis: "*Métodos heurísticos en problemas geométricos. Visibilidad, iluminación y vigilancia*" [2] of Santiago Canales Cano.

2. Definition of the project

This project has as its goal the elaboration of a graphic and interactive software that allows solving the problem previously explained, the *maximization of the Voronoi region*, using different metaheuristic techniques.

In a simple way, the metaheuristic techniques have the goal of looking for a solution close to the optimal one in a reasonable time. This kind of techniques is having an important growth due to the need of giving concrete solutions to real problems particularly in the Computational Geometry world and in a more general way in the Artificial Intelligence world.

The three techniques that will be focused on will be: an algorithm called *Random Search* that consists of random search among many points of the one that achieves maximum area, an algorithm based on the heuristic technique *Simulated Annealing* that starting from a local solution is able of reaching another solution of higher quality and, lastly, an algorithm based in the heuristic technique *Ant Systems*, or algorithm of ants imitating the ability these insects have of discovering the best path, or the shortest way by the accumulation of the pheromone they emit while moving.

3. Description of the model/system/tool

The software of the project can be divided into multiple files depending on the functionality of each of them. The scheme followed can be observed in Figure 6 where the main file (*Voronoi.py*) is responsible of the graphic section and of the communication with the rest of the programs; a file that contains the algorithms for the maximization of the Voronoi region based on computational geometry (*funciones_gc.py*); a file capable of opening and saving text files that contain the diagrams to be represented in the app and that have been of great utility to perform the analysis of the algorithms (*manejo_archivos.py*); and, lastly, a file for handling the exceptions and messages of the app to help the user use the app (*popup.py)*.

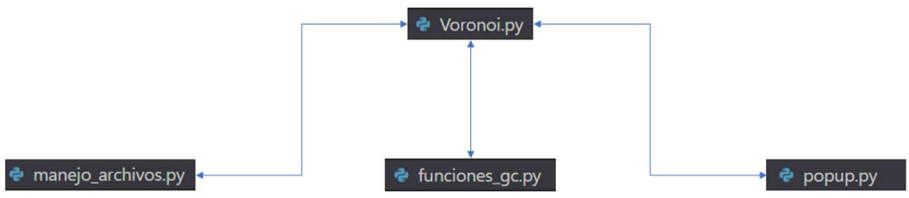


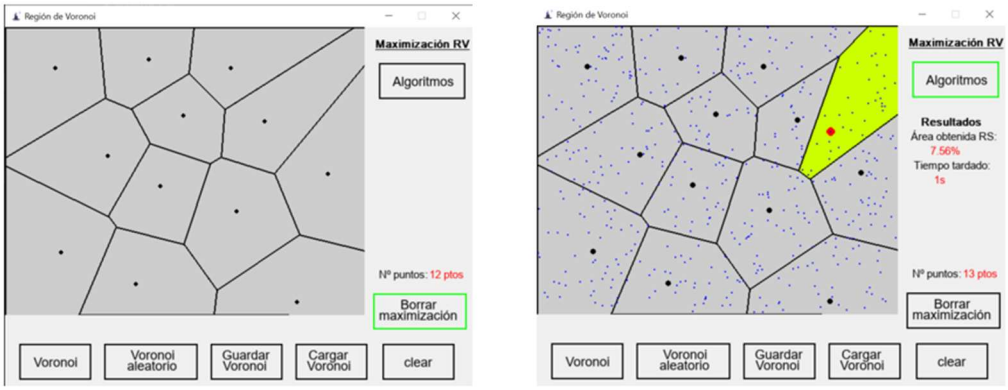
Figure 6. Architecture of the project

As can be understood, the app can be divided into two main parts. The first graphic part where it can be found an app that is able of representing different regions and allowing

the user to interact with it, integrated by the files *Voronoi.py, manejo_archivos.py and popup.py*. The second section is the one of the computational geometry in charge of performing the procedures to find the best solution to the problems raised, which can be found int *funciones_gc.py*.

4. Results

The final application is capable of, with a given distribution of points introduced by the user, obtaining the *Voronoi diagram* and giving the user the possibility of obtaining a new point that, when introduced into the diagram, has the approximately biggest associated area. An execution example of the final app can be seen in the Figure 7 where it is shown an interface of the user and a *Voronoi diagram* already drawn in *(a)* and in *(b)* the solution of the *maximization of the Voronoi region* using the algorithm *Random Search* with 500 random points is represented.



(a) *(b)*

Figure 7. Final example of the app

5. Conclusions

With the objective of conducting a deeper study of the algorithms, it has been created a comparation between them, evaluating 50 different diagrams randomly generated of 10, 50 and 100 points. In the Figure 8 is shown a comparation between the initial number of points in the diagram and the average area obtained with each algorithm. With the aim of being as exact as possible and maintaining a norm for the algorithm *Random Search*, the number of random points evaluated is given by the function 50*n, being n the number of points in the beginning of the diagram. This means that, if there are 10 starting points in the diagram, the algorithm *Random Search* will have evaluated 500 random points.

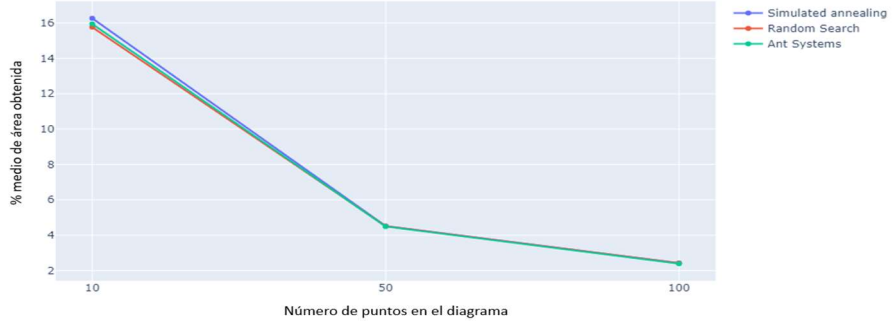


Figure 8. Comparation of the average percentage of area obtained by each technique

Figure 8 shows how the values of the area obtained by the 3 algorithms are very similar, what can lead to believe that it is possible that they are finding the solution near the ideal for most of the cases studied. Nevertheless, it is significative the difference of average times obtained for each of the algorithms before reaching the ideal solution. As can be seen in Table 4, Table 5 and Table 6, the percentage of area obtained with the algorithms is very similar, however, even though it may seem slight the algorithm *Simulated Annealing* gives back in average an area of higher quality, although the time cost is significantly higher.

It is understood that when using this algorithm, a decision between quality of the solution obtained and the time needed until reaching that solution must be made.

Points	*Average % of Voronoi Region area*	*Time [seconds]*
10	15,78	0,06
50	4,51	9,88
100	2,42	23,64

*Table 4. Results Random Search with m = 50*points*

Points	*Average % of Voronoi Region area*	*Time [seconds]*
10	16,27	4,36
50	4,52	13,42
100	2,43	43,06

Table 5. Results Simulated Annealing con: T0= Puntos, Tf = 0,005 and Ti = $\frac{T0}{(1+i)}$

Puntos	*Average % of Voronoi Region area*	*Time [seconds]*
10	15,95	1,86
50	4,48	7,7
100	2,38	25,24

Table 6. Results Ant Systems with iterations = $\frac{Points}{2}$ $\forall\ Points > 20$ *or iterations = 2*Points* $\forall\ Points < 20$

6. References

[1] Berg de, M.; Kreveld van, M.; Overmars, M.; Schwarzkopf, O.; "Computational Geometry. Algorithms and Applications", Springer, 1997.

[2] Canales Cano, Santiago. "Métodos heurísticos en problemas geométricos. Visibilidad, iluminación y vigilancia", Tesis doctoral, 2004.

Índice de la memoria

Índice de figuras

Índice de tablas

Capítulo 1. INTRODUCCIÓN

1.1 INTRODUCCIÓN A LA GEOMETRÍA COMPUTACIONAL

La Geometría Computacional nace de la conjunción de la Geometría Clásica y la Informática. Desarrolla herramientas y técnicas para resolver problemas principalmente de naturaleza geométrica haciendo más énfasis en el diseño eficiente de algoritmos y estructuras de datos.

La Geometría Computacional ha sido un área de estudio en la que incluso desde los años '50 ya existían herramientas que, pese a ser un poco rudimentarias, realizaban tareas de diseño por ordenador. Sin embargo, el principal impulso para el desarrollo de esta ciencia ha sido el avance en la computación gráfica, el diseño asistido por ordenador y principalmente el aumento en el número de áreas que usa esta tecnología, como puede ser la ingeniería, la robótica, el reconocimiento de patrones o el modelado geométrico. Aunque esta ciencia crece con el desarrollo de la tecnología muchos de los problemas a los que se enfrenta son de naturaleza clásica. Dentro de estos problemas "clásicos" nos podemos encontrar el cierre convexo, la triangulación de un polígono o la intersección de segmentos de recta.

1.2 LA GEOMETRÍA COMPUTACIONAL Y SUS ÁREAS DE APLICACIÓN

El gran número de aplicaciones que tiene la Geometría Computacional es uno de los aspectos más interesantes de la misma. Nos podemos encontrar diferentes ámbitos de aplicación como pueden ser:

- Cartografía. Las mallas cuadrangulares o triangulares son la forma de modelado más habitual para digitalizar los terrenos para hacer diferentes simulaciones, representaciones o simplificaciones de estos.

- Robótica. La planificación de movimientos, evitando colisiones es uno de los aplicativos que puede tener la Geometría Computacional en este campo ya que además de tener en cuenta el carácter geométrico de los obstáculos que nos podemos encontrar, también debemos ser capaces de buscar trayectorias lo más cortas posibles.
- Modelado geométrico. Esta rama trata de representar formas para poder visualizarlas, analizarlas o estudiar el comportamiento de las mismas, adaptándose a métodos de tomas de muestras o de medición.

1.3 DIAGRAMAS DE VORONOI

Una de las estructuras más conocidas dentro de la Geometría Computacional son los *diagramas de Voronoi*. Dada una región del plano *R* y un conjunto *N* de *n* puntos, se puede encontrar una subdivisión del espacio asociando a cada punto del conjunto *N* los puntos de *R* que se encuentren más cercanos a dicho punto que a cualquier otro del conjunto *N*. Dicha subdivisión es lo que se conoce como *diagrama de Voronoi*.

Actualmente, uno de los problemas sin cotas algorítmicas eficientes es el problema de buscar un nuevo punto *q* en la mencionada región *R*, tal que la región que se le asocie a *q* en el nuevo *diagrama de Voronoi* tenga área máxima.

Este proyecto tiene como objetivo desarrollar un software basado en lenguaje Python que permita construir *diagramas de Voronoi* y buscar de forma heurística un punto *q* que maximice su *región de Voronoi*. Esta herramienta será útil para impartir la asignatura de Geometría Computacional en el nuevo grado en Ingeniería Matemática e Inteligencia Artificial (iMAT) de la Escuela Técnica Superior de Ingeniería (ICAI).

Para la realización de este proyecto ha sido de gran utilidad, la tesis (Métodos heurísticos en problemas geométricos. Visibilidad, iluminación y vigilancia) [1] de Santiago Canales Cano.

1.4 MOTIVACIÓN DEL PROYECTO

Los ingenieros y ciertamente todas las personas llevamos intrínsecamente en nosotros la necesidad de ser capaz de dar soluciones a los problemas que nos vamos encontrando, si no es posible encontrar la mejor solución, es útil encontrar una buena solución, la solución más cercana a la óptima que seamos capaces de encontrar y que dé respuesta a la necesidad o problema planteado.

Supongamos que disponemos de un mapa de España cuya división en provincias se ha realizado asignando a cada capital de provincia los puntos que se encuentran más cercanos a ella que a cualquier otra como se muestra en la Figura 9.

Figura 9. Divisón de provincias según Voronoi

Sobre esta división de provincias el problema que se quiere plantear es el de añadir una nueva capital de provincia que, al introducirla en el mapa, el número de puntos que estén dentro de esta región sea lo más grande posible.

Para este problema planteado no se ha conseguido encontrar un algoritmo óptimo que solucione este problema y tampoco se ha conseguido demostrar la naturaleza *NP-hard,* (problema para el que no se conoce ninguna solución algorítmica eficiente [16]).

La realización de este proyecto se sustenta en que no existe en la actualidad ningún software comercial que solucione el problema de buscar de forma aproximada un nuevo punto *q* de la región *R* que maximice el área de su *región de Voronoi.* Resulta de gran interés el desarrollo de un software gráfico e intuitivo que dé solución al problema explicado previamente y al que denotaremos comúnmente como *MAXA-P-VOR(N)* y sirva como herramienta para estudiantes e investigadores en el área de la Geometría Computacional.

Capítulo 2. ESTADO DE LA CUESTIÓN

Para entender los algoritmos de maximización de la *región de Voronoi* es importante entender los *diagramas de Voronoi*, por ello vamos a explicarlos previamente.

2.1 DIAGRAMAS DE VORONOI

Como ya se ha mencionado en el Capítulo 1 una de las estructuras más conocidas dentro de la Geometría Computacional son los *diagramas de Voronoi.* Dada una región *R* y un conjunto *N* de puntos podemos encontrar una subdivisión de dicha región asociando a cada punto del conjunto los puntos del plano que se encuentren más cerca de dicho punto que de cualquier otro del conjunto de puntos *N*. Esta subdivisión es lo que se conoce como *diagrama de Voronoi,* que denotaremos como *Vor(N,R)* y está muy relacionada con la *triangulación de Delaunay* [2]. (Ver Figura 10).

Como ya se ha mencionado previamente, tenemos un conjunto *N= {p1,p2,…,pn}* de *n* puntos en una región plana *R,* podemos asociar a cada punto *p* el conjunto de puntos de *R* que están más cerca de dicho punto *p* que de cualquier otro punto de la región, lo que denominaremos como la *región de Voronoi* asociada a dicho punto y se denotará como *Rv(pi, N, R).* El conjunto de todas las *regiones de Voronoi Rv(pi, N, R)* asociados a cada punto *p* del conjunto *N* es lo que forma el *diagrama de Voronoi (Vor(N,R))*. Formalmente, podemos definir la *región de Voronoi* de un punto p_i, y el *diagrama de Voronoi* de una nube de puntos de la siguiente forma:

$$R_V(p_i, N, R) = \{x \in \mathrm{R} \, |p_i - x| \leq |p_j - x| \; \forall p_j \in \mathrm{N} \; \mathrm{j} \neq \mathrm{i}\}$$

$$Vor(N, R) = \bigcup_{i=1}^{n} Rv(p_i, N, R)$$

A continuación, en la Figura 10, se adjunta una imagen de un *diagrama de Voronoi* para ejemplificar lo explicado previamente.

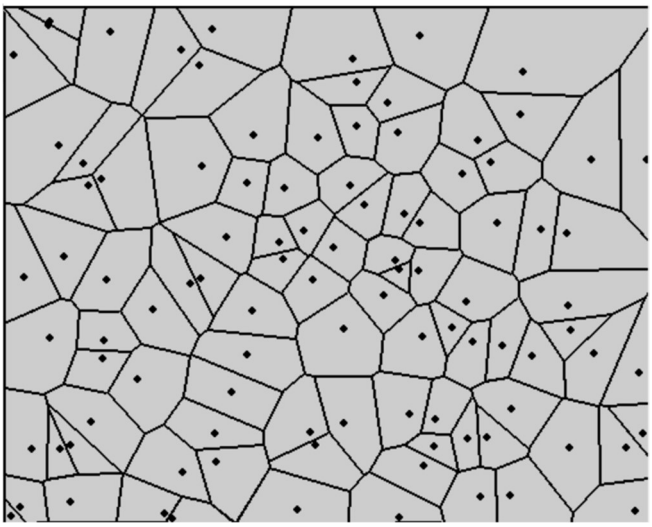


Figura 10. Diagrama de Voronoi de una región del plano R y un conjunto de 100 puntos

El problema que nos planteamos solucionar es el de buscar de forma aproximada un nuevo punto *q* de nuestra región *R* que maximice el área de su *región de Voronoi, Rv(q, N, R)*. Este problema que se puede denotar como *MAXA-P-VOR(N)* no ha sido referenciado en muchas ocasiones. Podemos mencionar referencias en Okabe y otros [3], Cheong y otros [4] y Ahn y otros [5]. En todas las menciones que aparecen no se explica ningún método que dé una solución al problema.

Desde un punto de vista algorítmico Dhene y otros [6], han tratado este tema para el caso en el que los vecinos del nuevo punto q estén en posición convexa y más recientemente Cheong y otros [7] han presentado nuevos algoritmos para calcular de forma aproximada la mejor *región de Voronoi*. Sin embargo, en ninguno de los dos casos anteriormente expuestos se ha tratado de solventar este problema desde el punto de vista heurístico.

A continuación, explicamos las tres técnicas heurísticas que abordaremos para tratar el problema *MAXA-P-VOR(N)* y que serán motivo de implementación en el presente proyecto. La primera llamada *Random Search – RS,* la segunda *Simulated Annealing – SA* y la tercera *Ant Systems – ANT*.

2.2 INTRODUCCIÓN A LAS TÉCNICAS HEURÍSTICAS

Los algoritmos heurísticos tienen como objetivo buscar una solución cercana a la óptima en un tiempo razonable. Estos procedimientos heurísticos están sufriendo un gran auge debido a la necesidad de disponer de herramientas que permitan ofrecer soluciones a problemas reales.

Dentro de estas técnicas podemos distinguir diversos métodos como: Métodos constructivos, de descomposición, de reducción, de manipulación del modelo y de búsqueda local. El interés que tenemos hoy en día es el de crear métodos generales que permitan resolver diferentes clases o categorías de problemas. Como estos métodos generales sirven para guiar la resolución de problemas específicos, se les ha dado el nombre de técnicas metaheurísticas.

Los profesores Osman y Kelly (1995) definen estas técnicas de la siguiente manera:

"Los procedimientos Metaheurísticas son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que las heurísticas clásicas no son ni efectivas ni eficientes. Las metaheurísticas proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de: inteligencia artificial, evolución biológica y mecanismos estadísticos."

Concretando en nuestro problema pasamos explicar las técnicas heurísticas que se implementarán en el presente proyecto para la maximización de la *región de Voronoi*.

2.3 SIMULATED ANNEALING – SA EN MAXA-P-VOR(N)

Para nuestro problema *MAXA-P-VOR(N)* partimos de un *diagrama de Voronoi* dado de un conjunto *N* de puntos y la salida es un punto *q* de nuestro plano, que en este caso maximice el área de su *región de Voronoi*.

Vamos a realizar una breve introducción a la heurística de Optimización Combinatoria que utilizaremos posteriormente conocido como S*imulated Annealing.* De una forma sencilla, este algoritmo usa una búsqueda local que, empezando desde una solución parcial, progrese hasta una solución de mayor calidad [11].

Los problemas de optimización combinatoria pueden ser planteados de la siguiente manera: dado un conjunto de soluciones finito $S = \{x1, ..., xm\}$ donde m es la dimensión del conjunto y dada una función de costes $C{:}S \rightarrow \mathbb{R}$, determinar $x^* \in S$, tal que $C(x*) \leq C(x)$ $\forall x \in S$.

El método *Simulated Annealing* depende fuertemente de la solución inicial elegida y por tanto esta solución puede quedarse atrapada en mínimo local y nunca llegar a ser un mínimo global. La solución a este problema es introducir una variable de control T que, con cierta probabilidad empeore la función objetivo y así salir del problema de quedarnos atrapados en un mínimo local.

La estrategia de este marco es empezar en una variable de control T alta, lo que proporciona una probabilidad alta de no mejora. En cada iteración, se va disminuyendo este valor y por tanto las probabilidades son cada vez más pequeñas conforme avanza el procedimiento y nos acercamos a la solución óptima.

El método previamente descrito se implementa de la siguiente manera:

- Primero adaptamos el problema con el conjunto S de configuraciones o soluciones factibles del problema, creamos la función de coste C, establecemos los criterios de vecindad y terminamos la configuración inicial.
- La estrategia que debemos seguir es situar la variable de control T inicial, disminuir esta variable en cada una de las iteraciones, especificar el número de iteraciones y establecer un criterio de parada.

El pseudocódigo que nos sirve como esquema general para esta heurística es el siguiente:

```
[1]   do
[2]         {do
[3]               {Generar solución y ∈ Vecindad(x) ⊂ S
[4]               Evalúa resultado ← C(x) – C(y)
[5]               if (resultado <0)  x ← y
[6]               else
[7]                     if ((resultado>0) Λ (U(0,1) < e^(−resultado/T)))  x ← y
[8]               n ← n+1
[9]               } while (n ≤ N(T)
[10]        Disminuye T
[11]        }while (parada == false)
```

El resultado final del algoritmo se puede ver en la Figura 11.

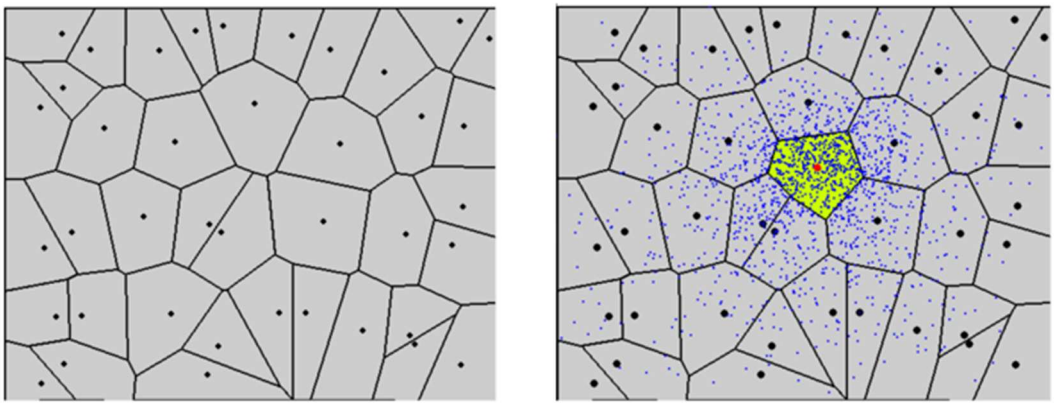


Figura 11. Solución del problema Max-p-Vor(n) con SA para 40 puntos

2.4 RANDOM SEARCH – RS EN MAXA-P-VOR(N)

Partiendo como entrada de un *diagrama de Voronoi*, este algoritmo va realizando una búsqueda aleatoria de los puntos interiores de la región *R* donde se encuentra la nube de puntos *N*. Generaremos una nube *T = {p1, ..., pm}* de puntos interiores a la región *R* y realizaremos una búsqueda del punto *p* tal que:

$$Área\big(Rv(p_i, N, \cup \{pi\}, R)\big) \geq Área\big(Rv(p_i, N, \cup \{pj\}, R)\big)\ \forall\, j \neq i \qquad i, j \in \{1, \ldots, m\}$$

El tamaño de *T* está relacionado con el número de puntos del *diagrama de Voronoi*, para los análisis de la eficacia de los algoritmos, sin embargo, con el objetivo de darle la posibilidad al usuario de poder decidir sobre el número de puntos aleatorios que quiera evaluar, en la aplicación, el tamaño de *T* no dependerá del número de puntos que hay en el *diagrama de Voronoi* de partida.

El pseudocódigo que podemos usar como esquema para este algoritmo es el siguiente:

```
[1] área = 0
[2] ptos = Generar aleatorios (R,50*n)
[3] Calcula Diagrama de Voronoi Vor(N,R)
[4] for (pi ∈ ptos)
[5]   { Calcula región de Voronoi con el nuevo punto Rv(pi, N ∪ {pi},R)
[6]   if (área < Área(Rv(pi, N ∪ {pi},R)))
[7]   área = Área(Rv(pi, N ∪ {pi},R))
[8]   p = pi
[9]   } return p
```

Un ejemplo con el resultado de este algoritmo lo podemos ver en la Figura 12.

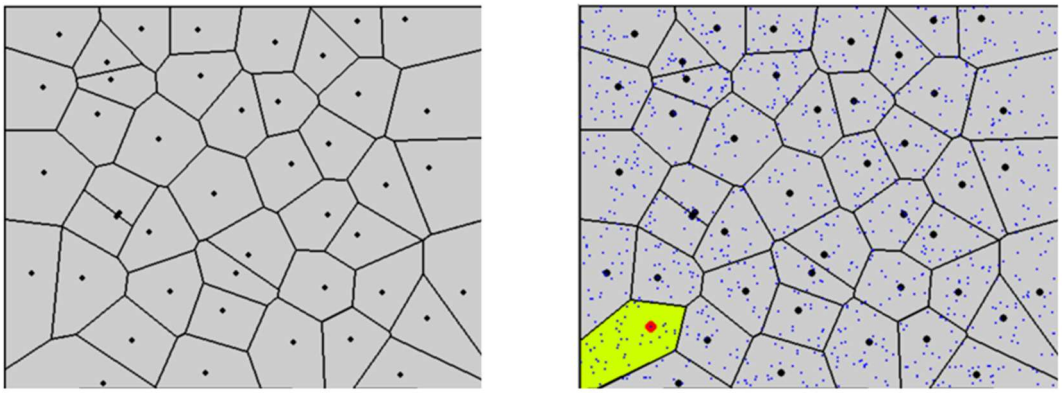


Figura 12.Solución del problema Max-p-Vor(n) con RS para 44 puntos iniciales y 1000 puntos aleatorios evaluados

2.5 ANT SYSTEMS – ANT EN MAXA-P-VOR(N)

Partimos que la entrada de nuestro problema es un *diagrama de Voronoi* ya dado. Este algoritmo simula el comportamiento real de una colonia de hormigas y es el primer algoritmo desarrollado en esta área [14]. Para entenderlo correctamente debemos entender su inspiración biológica y la razón de ser de este algoritmo.

2.5.1 LAS HORMIGAS EN LA VIDA REAL, INSPIRACIÓN BIOLÓGICA

Las hormigas son capaces de encontrar el camino más corto desde su hormiguero hasta un punto donde haya comida sin utilizar la vista. Son capaces de adaptarse ya que si encontraran un obstáculo en la ruta serían capaces de encontrar el camino más corto para rodearlo. Esto lo logran mediante un rastro, es decir, rastreando la feromona que dejan el resto de las hormigas al desplazarse. De una manera visual lo podemos observar en la Figura 13 que como tras pasar un tiempo todas las hormigas siguen el mismo camino. Esto se debe a que la acumulación de la feromona es mayor por el lado más corto ya que hay más cantidad de feromona en menos distancia por lo que la hormiga irá por donde note que la concentración de este rastro es mayor [15].

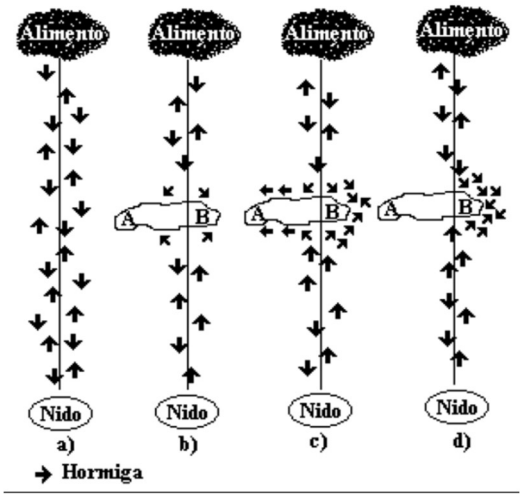


Figura 13. Comportamiento de las hormigas frente a un obstáculo

2.5.2 ALGORIMTO ANT SYSTEMS

El comportamiento del algoritmo es similar al de la conducta de las hormigas. En este caso nuestro rastro serán una lista de los mejores puntos que consiga la hormiga y se irá actualizando con el resto de las hormigas en cada una de las iteraciones. Los mejores puntos serán aquellos que devuelvan el mejor porcentaje de área de todos los evaluados. De esta forma iremos acotando el lugar donde pueden ir observando nuestros agentes en busca de los mejores puntos hasta dar con el de mejor valor.

El pseudocódigo que podemos usar como esquema para este algoritmo es el siguiente:

```
[1] for hormiga = 0 to max //Número máximo de hormigas que generamos
[2]   {for puntos_visitados = 0 to maxi// n° ptos que visita cada hormiga
[3]         { La hormiga debe realizar el recorrido aleatorio por el mapa
[4]           e ir quedándose con los 10 puntos que al introducirlos
[5]           generen el mayor área posible.
[6]   } Reducimos el área en el que se generan los puntos visitados
[7] } devolvemos el punto que genera el mayor área
[9] Fin
```

De una forma visual podemos ver el resultado del algoritmo en la Figura 14.

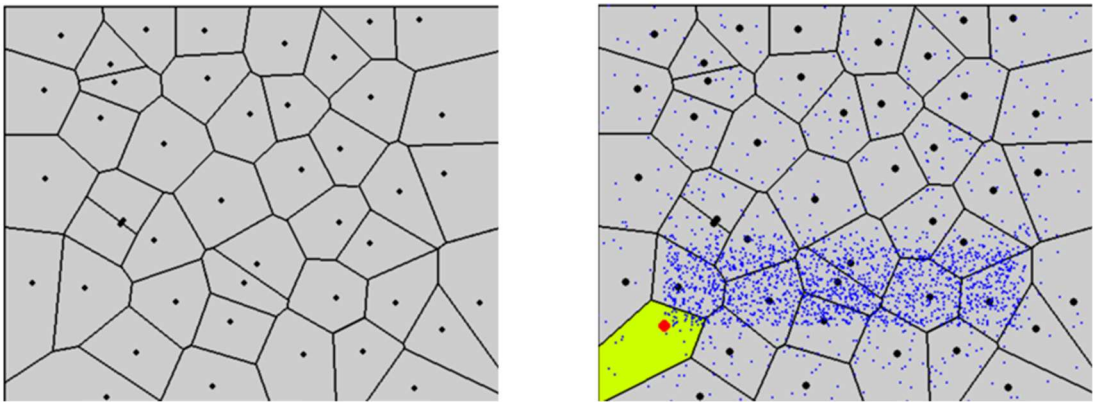


Figura 14.Solución del problema Max-p-Vor(n) con ANT para 44 puntos iniciales

Capítulo 3. DEFINICIÓN DEL TRABAJO

3.1 JUSTIFICACIÓN Y CASOS DE USO

Como se ha explicado con anterioridad los *diagramas de Voronoi* son subdivisiones de una región más grande a la que podemos llamar *R* en base a unos puntos iniciales sobres los que se forma dicho diagrama. A su vez, la maximización de la *región de Voronoi* consiste en la búsqueda de un nuevo punto dentro de los límites de nuestra región R cuya área asociada sea lo más grande posible.

Muchos de los problemas que nos encontramos hoy en día, relacionados con ubicaciones, áreas afectadas o emplazamientos no disponen de aplicaciones con herramientas matemáticas eficaces capaces de resolver este tipo de problemas. Es por eso por lo que la elaboración de este software tiene sentido, para dar una ayuda a todos aquellos que necesiten una herramienta para la elaboración de *diagramas de Voronoi* y la capacidad para añadir un nuevo punto que maximice la *región de Voronoi.*

Podemos encontrarnos con diferentes casos de uso donde la aplicación desarrollada podría ser útil, vamos a exponer algunos ejemplos a continuación.

3.1.1 DIAGRAMAS DE VORONOI Y EL BROTE DE CÓLERA EN LONDRES DE 1854

El cólera [8] es una enfermedad bacteriana que puede provocar nauseas, diarreas o deshidratación, en el caso de que no sea tratado puede provocar la muerte, aunque hoy en día, el tratamiento es sencillo y muy eficiente. Actualmente esta enfermedad ha sido prácticamente erradicada de muchos países desarrollados, sin embargo, aquellos lugares del mundo donde las aguas residuales o el tratamiento moderno para el agua no está muy desarrollado o es inexistente, como puede ser África o Haití, sigue siendo un problema.

En 1854, en el barrio de Soho en Londres apareció un brote de cólera. En menos de una semana la cifra de fallecidos había alcanzado las 700 personas y no se conseguía descubrir

cuál era la causa del contagio de dicha enfermedad. Algunas personas pensaban que el viento era el culpable de la transmisión y otros pensaban que era el contagio directo con las personas contagiadas. Sin embargo, John Snow, médico inglés al que hoy en día se le conoce como el héroe del cólera [9] creía que el principal problema eran las bombas de agua repartidas por la ciudad las causantes de la enfermedad.

La idea de Snow era que la acumulación de fallecidos se daba en la zona donde había una bomba de agua en mal estado. Para ellos localizó los domicilios de las personas fallecidas y calculó la distancia con la bomba de agua más cercana. Una vez terminó de analizar los resultados se pudo concluir como el problema venía principalmente de la bomba situada en la calle Broad Street. Como se puede ver en la Figura 15, la *región de Voronoi* en la que estaba la bomba era el lugar donde se concentraban los fallecidos por la enfermedad del cólera.

Una vez se cerró la bomba se pudo observar como la incidencia de la enfermedad se redujo, al igual que la mortalidad. A John Snow se le conoce como una de las primeras personas que utilizaron métodos geográficos y una de las primeras personas que puso en práctica el uso de los *diagramas de Voronoi.*

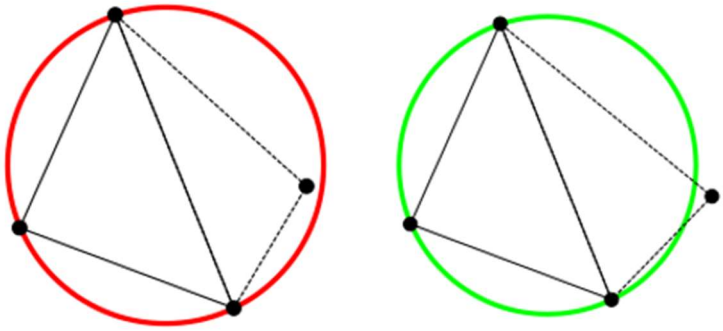
Figura 15. Mapa elaborado por John Snow donde se pueden ver los domicilios de los fallecidos (puntos) y las bombas de agua (cruces). Superpuesto se puede ver un diagrama de Voronoi con las regiones asociadas a cada bomba de agua.

3.1.2 DIAGRAMAS DE VORONOI Y LA TELEFONÍA FIJA

Uno de los problemas a los que ha sido complicado encontrarles una solución ha sido al enrutamiento de llamadas de telefonía fija, es decir, la ruta que tienen que seguir los paquetes de telefonía para dar con la mejor ruta hasta el destino.

Una de las soluciones más simples es la de aplicar los *diagramas de Voronoi* junto con la *triangulación de Delaunay*. Ya se ha explicado previamente lo que es un *diagrama de Voronoi*, sin embargo, *la triangulación de Delaunay* aún no ha sido explicada.

La *triangulación de Delaunay* es un conjunto de triángulos conexos y convexos que cumplen la condición de que la circunferencia circunscrita de cada triángulo no puede contener algún vértice de otro triángulo. Como podemos ver en la Figura 16 en (a), hay un vértice dentro de la circunferencia por lo que no se cumple la condición previamente explicada, sin embargo, en (b) podemos ver como si se cumple la condición de *Delaunay.*



(a) *(b)*

Figura 16. Triangulación de Delaunay

Una vez entendido que es la *triangulación de Delaunay,* vamos a explicar cómo se puede aplicar a la telefonía fija. Supongamos que los puntos representan nodos de telefonía fija. Como se puede ver en la Figura 17, realizando el *diagrama de Voronoi* a partir de estos puntos, obtenemos la región de influencia de las antenas, es decir, el área en el que los puntos que estén dentro de la región se conectarán a esa antena.

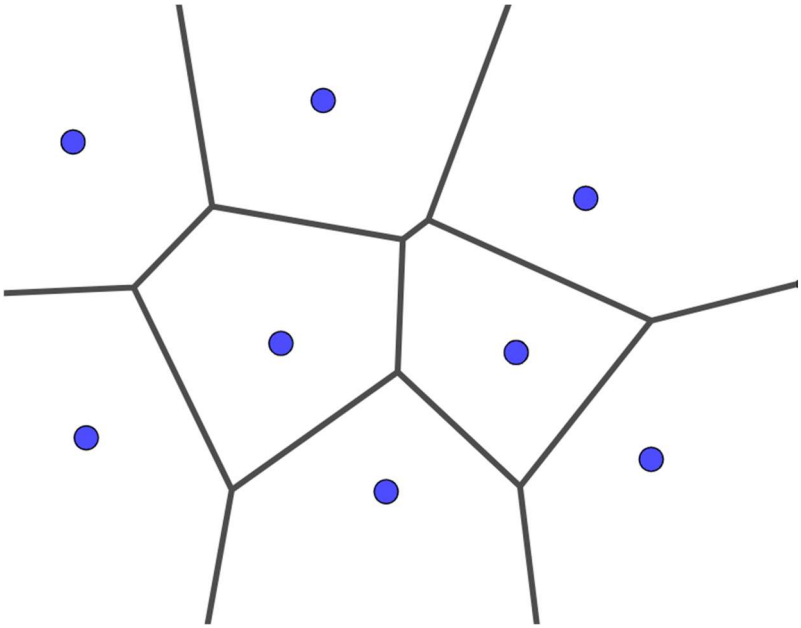


Figura 17. Diagrama de Voronoi y antenas de telefonía (puntos)

La *triangulación de Delaunay* y *los diagramas de Voronoi* se pueden relacionar sabiendo que, si dos puntos del *diagrama de Voronoi* comparten una arista de dicho diagrama, estos puntos serán unidos con una arista en la *triangulación de Delaunay.*

Sabemos que cualquier punto que esté en el interior de dicha región está más cerca de la antena que forma la *región de Voronoi* que de cualquier otra antena del mapa. Como se puede observar en la Figura 18 si a continuación calculamos la *triangulación de Delaunay* sobre el *diagrama de Voronoi* y aplicamos, por ejemplo, el algoritmo de Dijkstra, ya sabemos cuál debe ser la ruta más corta que debe seguir la llamada hasta llegar a su destino.

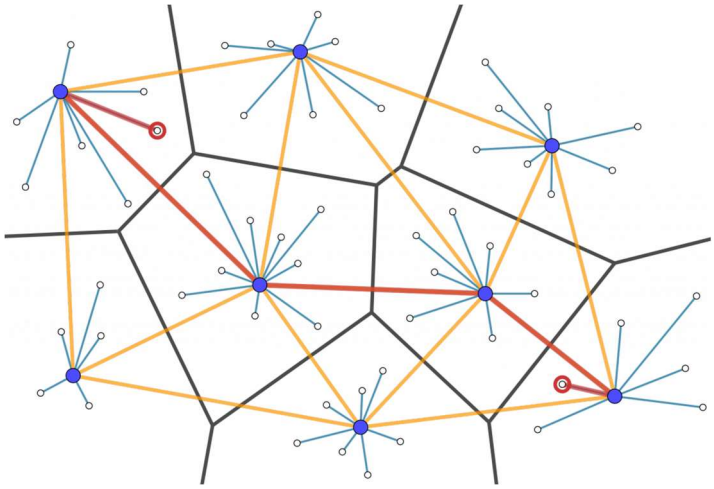


Figura 18. Ruta óptima para la llamada

3.1.3 DIAGRAMAS DE VORONOI Y ROBÓTICA

Otro de los problemas que nos podemos encontrar es en la de darle a un robot la capacidad de moverse de forma autónoma por un espacio en el que hay obstáculos. Si creamos el *diagrama de Voronoi* de tal forma que los obstáculos sean el origen de este diagrama, podremos hacer que el robot se desplace sobre las líneas que forman el diagrama, de esta forma estaría en todo momento entre dos obstáculos y no se llegaría a chocar con ninguno de ellos. En la Figura 19 podemos ver un ejemplo en el que se han representado los obstáculos como si fuera puntos y la trayectoria que podría seguir el robot son las líneas azules, así mientras el robot camine sobre alguna de esas líneas no llegará a chocarse.

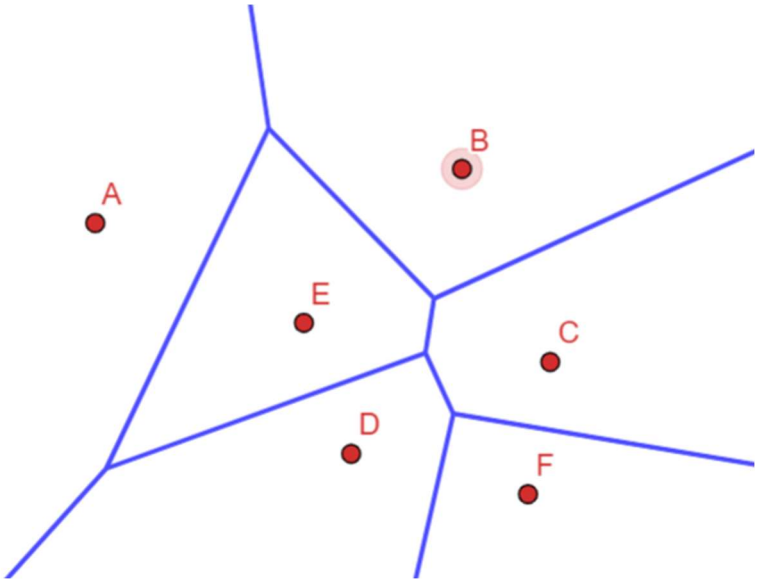


Figura 19. Ruta para evitar las colisiones de un robot al desplazarse

3.1.4 DIAGRAMA DE VORONOI Y COLOCACIÓN DE ANTENAS

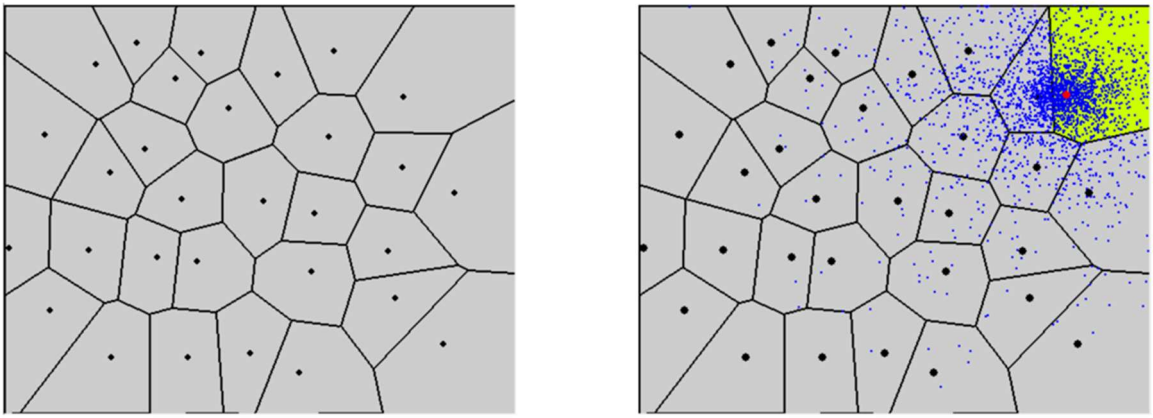
Otro de los problemas que nos podemos encontrar es el de la colocación de un nuevo recurso una vez tenemos unos recursos originales colocados en una región y además queremos que el área que sea capaz de cubrir sea máxima.

En nuestro caso vamos a simular que nuestros recursos son antenas. Partimos inicialmente de una red de antenas ya dispuesta sobre una región geográfica y es necesario colocar una nueva. Esta nueva antena tiene más capacidad y puede soportar un mayor número de conexiones por lo que nos es interesante que abarque una región que sea lo más grande posible.

Utilizando la geometría computacional este problema lo podríamos resolver de una manera sencilla, utilizando los *diagramas de Voronoi* sobre las antenas que ya están colocadas para posteriormente poder aplicar la *maximización de la región de Voronoi* y encontrar el punto en el que poder instalar la nueva antena y que cumpla con los requisitos pedidos.

En la Figura 20 podemos ver en (a) la red de antenas de telefonía que hay previamente y en (b) se encuentra el nuevo punto y el área que abarcaría. Esta solución se ha obtenido con el

algoritmo *Simulated Annealing* para una región de 30 antenas siendo la antena número 31 la que se desea descubrir el mejor lugar para su colocación.



(a) *(b)*

Figura 20. Solución para la colocación de una nueva antena con el algoritmo de Simulated Annealing

3.2 OBJETIVOS

El proyecto consiste en la implementación de un software interactivo que permita la construcción de *diagramas de Voronoi* e implemente las técnicas heurísticas para la *maximización de la región de Voronoi*.

Hemos dividido los objetivos entre didácticos y técnicos.

- *Didácticos*
  - i. Estudiar los *diagramas de Voronoi*, sus diferentes métodos de aplicación, las propiedades más importantes y darlo a conocer.
  - ii. Estudiar técnicas metaheurísticas para la *maximización de la región de Voronoi.*
  - iii. Mostar posibles aplicaciones de los *diagramas de Voronoi.*
- *Técnicos*
  - i. Diseño de un software gráfico para implementar *diagramas de Voronoi*.

ii. Implementar algoritmos heurísticos como *random search, simulated annealing y ant systems*

iii. Analizar y comparar el rendimiento de los diferentes algoritmos heurísticos utilizados

3.3 METODOLOGÍA

El proyecto se ha realizado siguiendo una metodología Agile. Pese a no incorporar reuniones diarias, se trataba de implementar reuniones semanalmente para comentar los siguientes pasos a realizar y convocar las siguientes reuniones para presentar el trabajo realizado hasta la fecha, así como nuevas ideas y problemas que nos hemos ido encontrando a lo largo de la realización del proyecto.

Este proyecto ha sido dividido en 5 etapas principales:

I. Diseño del front de la aplicación e investigación. Consistió en elaborar la parte visual de nuestra aplicación, decidiendo tanto la estructura como el estilo que le queramos dar y el número de funcionalidades que queramos implementar, pudiendo variar a lo largo de la elaboración del software. Además, a la vez que realizábamos el diseño de la aplicación, íbamos realizando una investigación y estudio de lo que son los *diagramas de Voronoi*, de los diferentes métodos para su construcción y de las diferentes técnicas heurísticas para la *maximización de la región de Voronoi.*

II. Programación de algoritmo para la creación del *diagrama de Voronoi.* En esta parte programaremos un algoritmo incremental, que nos permita, dada una nube de puntos introducida por el usuario construir su *diagrama de Voronoi*.

III. Programación de técnicas heurísticas para la maximización de la región de Voronoi. Esta etapa es el centro del proyecto, la implementación y adaptación de las heurísticas *Simulated Annealing, Random Search y Ant Systems* en nuestro programa que nos permita, una vez creado nuestro *diagrama de Voronoi*, maximizar la región de Voronoi *maximizar la región de Voronoi* asociada a un nuevo punto del plano.

IV. Análisis de resultados. En esta etapa hemos querido centrarnos en el análisis y comparativa de las técnicas implementadas. Se han medido fundamentalmente dos variables: tiempo empleado por cada técnica y porcentaje de área asignado al punto solución.

V. Elaboración de la memoria del proyecto. Desarrollaremos tanto la memoria del proyecto como la presentación para su defensa.

3.4 PLANIFICACIÓN Y ESTIMACIÓN ECONÓMICA

A continuación, se adjunta la planificación seguida para la elaboración de este proyecto, donde se especifican las tareas realizadas y las fechas para cada una de las tareas.

Tareas	SEPTIEMBRE				OCTUBRE				NOVIEMBRE				DICIEMBRE			
	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
ESTUDIO DE LOS DIAGRAMAS DE VORONOI																
ELABORACIÓN FRONT DE LA APLICACIÓN																
ESTUDIO TÉCNICAS HEURÍSTICAS																
PROGRAMACIÓN ALGORITMO DE DIAGRAMAS DE VORONOI																
PROGRAMACIÓN ALGORITMO RANDOM SEARCH																
PROGRAMACIÓN ALGORITMO SIMULATED ANNEALING																
PROGRAMACIÓN ALGORITMO ANT SYSTEMS																
COMPARATIVA DE LOS ALGORITMOS																
MEMORIA Y DEFENSA DEL PROYECTO																

Tabla 7. Planificación de septiembre hasta noviembre

Tareas	ENERO				FEBRERO				MARZO				ABRIL			
	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
ESTUDIO DE LOS DIAGRAMAS DE VORONOI																
ELABORACIÓN FRONT DE LA APLICACIÓN																
ESTUDIO TÉCNICAS HEURÍSTICAS																
PROGRAMACIÓN ALGORITMO DE DIAGRAMAS DE VORONOI																
PROGRAMACIÓN ALGORITMO RANDOM SEARCH																
PROGRAMACIÓN ALGORITMO SIMULATED ANNEALING																
PROGRAMACIÓN ALGORITMO ANT SYSTEMS																
COMPARATIVA DE LOS ALGORITMOS																
MEMORIA Y DEFENSA DEL PROYECTO																

Tabla 8. Planificación de enero hasta abril`

Tareas	MAYO				JUNIO				JULIO
	S1	S2	S3	S4	S1	S2	S3	S4	S1
ESTUDIO DE LOS DIAGRAMAS DE VORONOI									
ELABORACIÓN FRONT DE LA APLICACIÓN									
ESTUDIO TÉCNICAS HEURÍSTICAS									
PROGRAMACIÓN ALGORITMO DE DIAGRAMAS DE VORONOI									
PROGRAMACIÓN ALGORITMO RANDOM SEARCH									
PROGRAMACIÓN ALGORITMO SIMULATED ANNEALING									
PROGRAMACIÓN ALGORITMO ANT SYSTEMS									
COMPARATIVA DE LOS ALGORITMOS									
MEMORIA Y DEFENSA DEL PROYECTO									

Tabla 9. Planificación de mayo hasta julio

Capítulo 4. SISTEMA/MODELO DESARROLLADO

4.1 ELABORACIÓN DE UN DIAGRAMA DE VORONOI

Un *diagrama de Voronoi* es una partición del espacio en la que las regiones en las que se divide el espacio se incluyen los puntos más cercanos al punto sobre el que está construida cada una de las regiones que a cualquier otro punto sobre el que se han construido las otras particiones.

Existen diferentes algoritmos para la creación de un *diagrama de Voronoi,* algunos de los más comunes son: algoritmo incremental, algoritmo divide y vencerás, algoritmo de Fortune o el algoritmo de intersección de semiplanos. Para la realización de este trabajo se pensó en realizar la construcción de los diagramas mediante el algoritmo divide y vencerás o el algoritmo incremental, siendo este último el algoritmo elegido para la construcción de los *diagramas de Voronoi.*

Pese a no hacer uso del algoritmo divide y vencerás sí que es importante analizar su funcionamiento para poder explicar posteriormente el algoritmo incremental y la razón de habernos decidido por este último.

4.1.1 ALGORITMO DIVIDE Y VENCERÁS

Para entender el funcionamiento de este algoritmo, vamos a ir paso a paso explicando, en orden, los pasos que hay que seguir para crear un *diagrama de Voronoi* mediante este método.

1. El objetivo de este algoritmo es dividir el problema en distintos subproblemas de menor tamaño, por lo que lo primero que habrá que hacer será realizar una subdivisión en dos mitades, de el mismo tamaño o muy parecido. De forma gráfica lo podemos ver en la Figura 21.

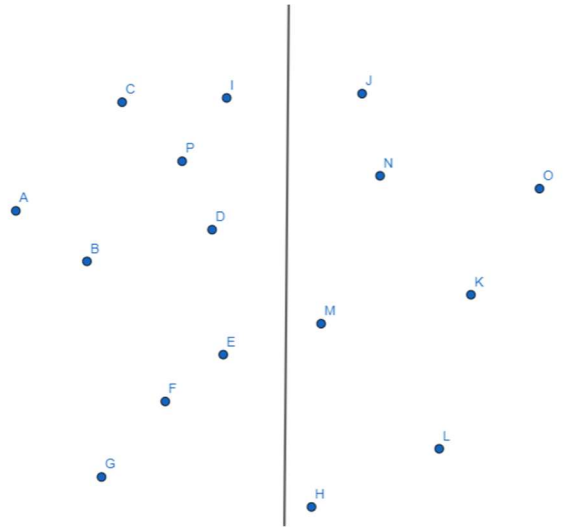
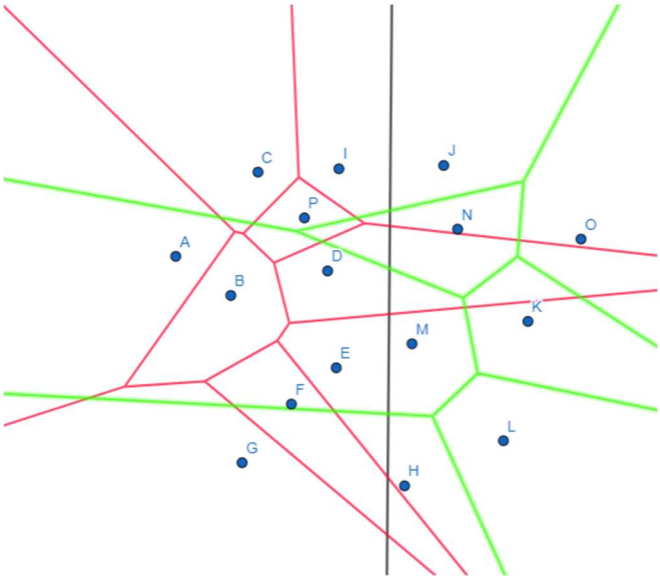


Figura 21. Primer paso del algoritmo divide y vencerás

2. A continuación, se ha de resolver cada una de las mitades por separado, es decir, debemos calcular el *diagrama de Voronoi* de cada una de las dos partes de forma independiente. En la Figura 22 podemos observar gráficamente este paso.



Voronoi 1 *Voronoi 2*

Figura 22. Diagramas de Voronoi independientes

3. A continuación, debemos calcular la línea divisoria entre las dos mitades independientes. Esta parte es la clave de este algoritmo pues es la antesala para obtener nuestro *diagrama de Voronoi* completo. Debemos ir realizando las

bisectrices entre los puntos que forman las regiones de Voronoi que nos vamos encontrando. Lo podemos observar visualmente en las Figuras 23 y 24, en las que observamos dos iteraciones para poder entender el funcionamiento para realizar la división.

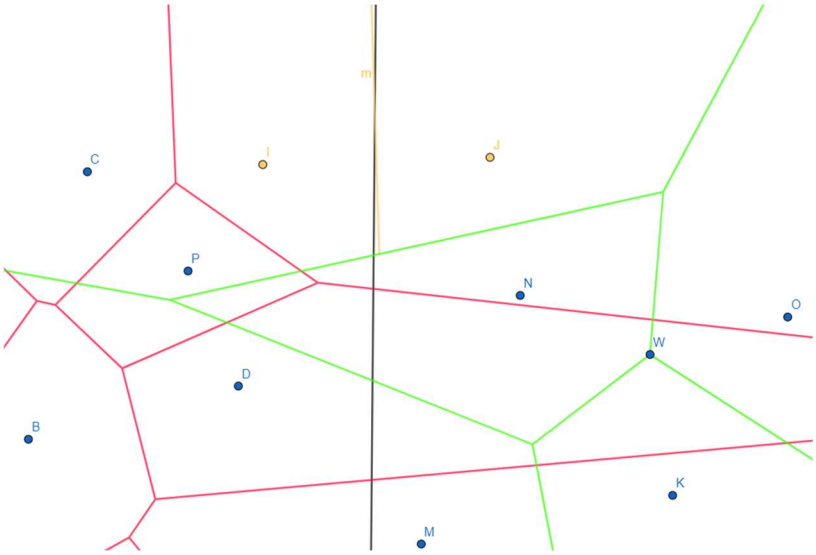


Figura 23. Primera iteración para realizar la línea de división.

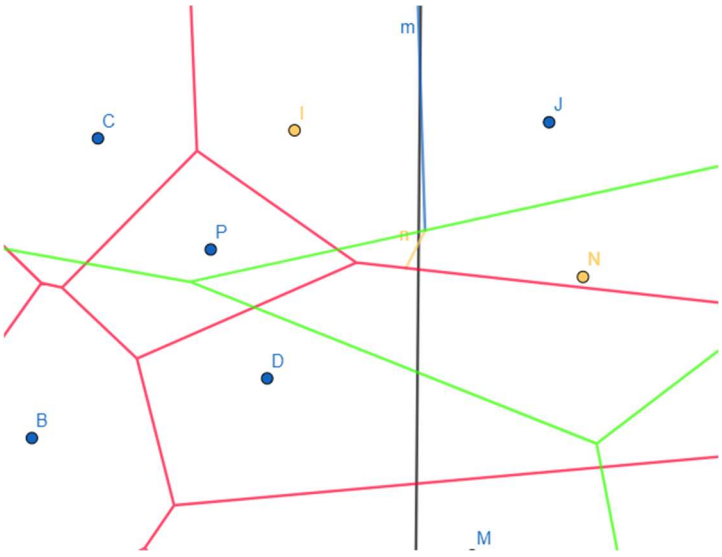


Figura 24. Segunda iteración para realizar la línea de división

Como se puede observar lo primero que se realiza es la línea mediatriz entre los dos primeros puntos, los más cercanos a la línea intermedia de división. Al realizar esta

recta, nos encontramos con una línea del diagrama de Voronoi 2 por lo que tenemos que tomar como referencia para realizar la siguiente mediatriz los puntos I y N como observamos en la Figura 24. Al realizar este segmento nos encontramos con una línea del diagrama de Voronoi 1 por lo que la siguiente mediatriz se realizaría con los puntos N y D. La línea de división final la podemos observar en la Figura 25.

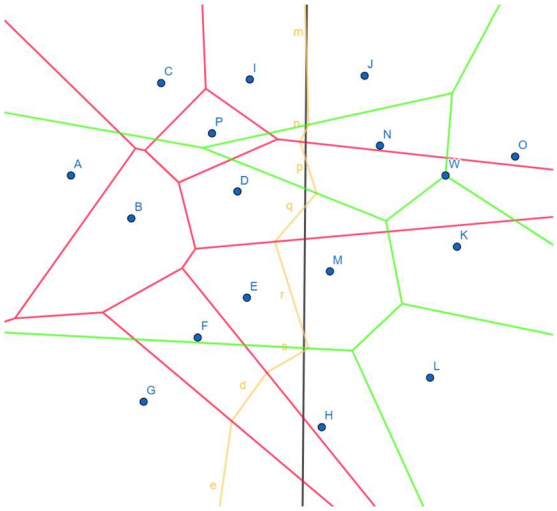


Figura 25. Línea de división final

4. El último paso para la realización de este algoritmo es unir el diagrama final mediante los dos diagramas de Voronoi diferentes y la línea de división creada. El resultado final lo podemos ver en la figura 26.

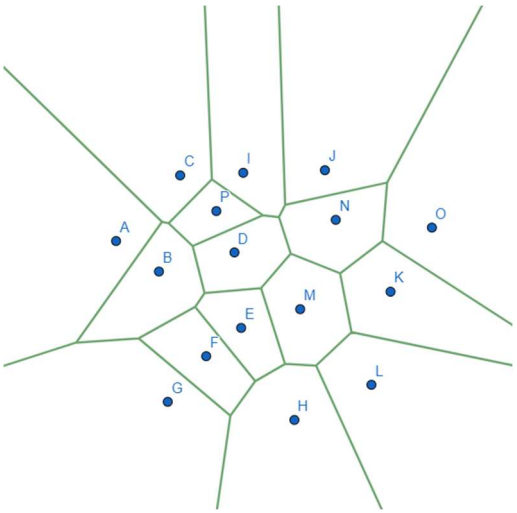


Figura 26. diagrama de Voronoi obtenido con algoritmo divide y vencerás

La complejidad de este algoritmo es de O(n), y la principal ventaja que tiene es la capacidad para reducir el problema inicial en problemas más pequeños sin tener que aumentar la complejidad del algoritmo.

4.1.2 ALGORITMO INCREMENTAL

El algoritmo incremental es quizás el algoritmo más usado para elaborar *diagramas de Voronoi.* Como se puede deducir del propio nombre del algoritmo, consiste en ir realizando el diagrama punto a punto. Es decir, cada vez que añadamos un punto, recalculamos el diagrama de Voronoi y lo actualizamos. Vamos a realizar el ejemplo para un diagrama de Voronoi formado por 3 puntos, los podemos ver en la Figura 27. Los pasos son los siguientes:

Figura 27. Situación inicial para calcular el diagrama de Voronoi con algoritmo incremental

1) Calculamos el *diagrama de Voronoi* del punto inicial, como es el único punto de la región, su *región de Voronoi* será la región del espacio completa, como lo podemos ver en la Figura 28.

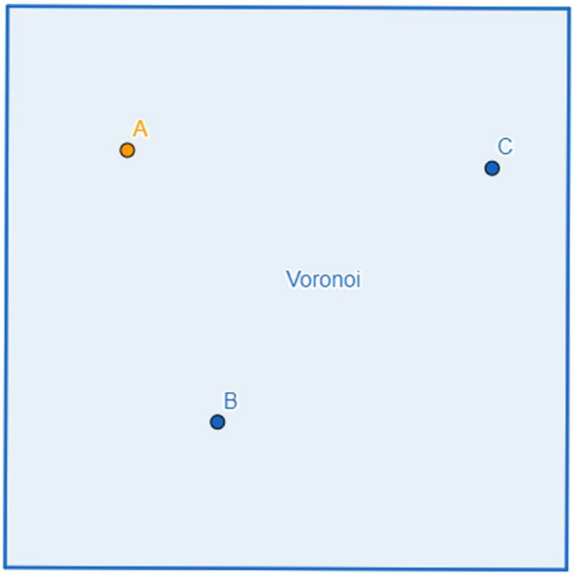


Figura 28. Iteración 1 del algoritmo incremental

2) El siguiente paso es introducir el siguiente punto y calcular la mediatriz entre los dos puntos de tal forma que se realice la división entre las dos regiones. El diagrama lo podemos observar en la Figura 29.

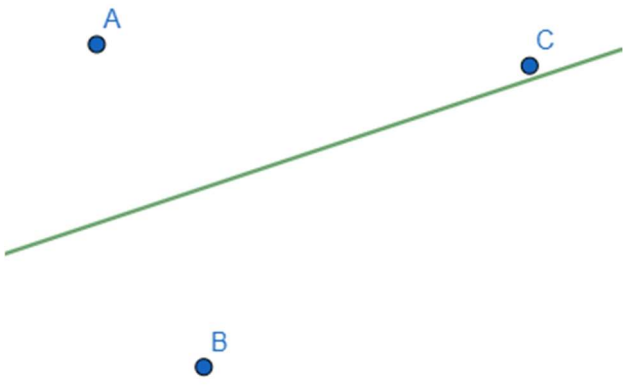


Figura 29. Iteración 2 del algoritmo incremental.

3) Por último, el siguiente paso es añadir el punto C. En este caso se puede ver que está situado en la *región de Voronoi* del punto A por lo que tenemos que trazar la mediatriz con dicho punto, como se ve en la Figura 30.

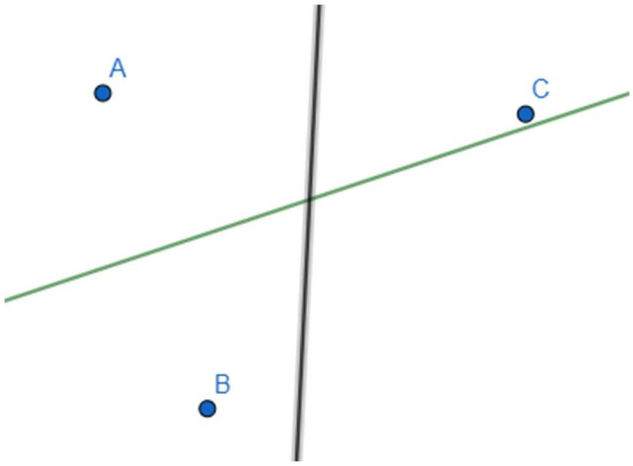


Figura 30. Iteración 3 del algoritmo incremental. Mediatriz entre A y C.

Como se puede ver en la Figura 30, la mediatriz choca con la línea que separa las *regiones de Voronoi* de A y B por lo que también es necesario calcular la mediatriz entre B y C, obtendríamos el diagrama de la Figura 31.

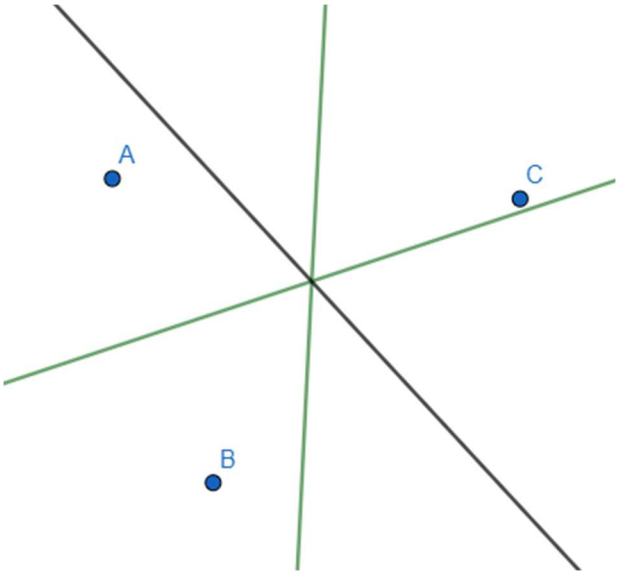


Figura 31. Iteración 3 del algoritmo incremental. Mediatriz entre B y C

Por último, como no hay más regiones ni puntos en el plano, únicamente tenemos que recortar las partes que no nos interesan de las mediatrices obtenidas y conseguimos el *diagrama de Voronoi* final utilizando el algoritmo incremental. La solución se puede

observar en la Figura 32. Si tuviésemos más puntos inicialmente habría que repetir los pasos del punto 3 con cada punto que introdujéramos en el diagrama.

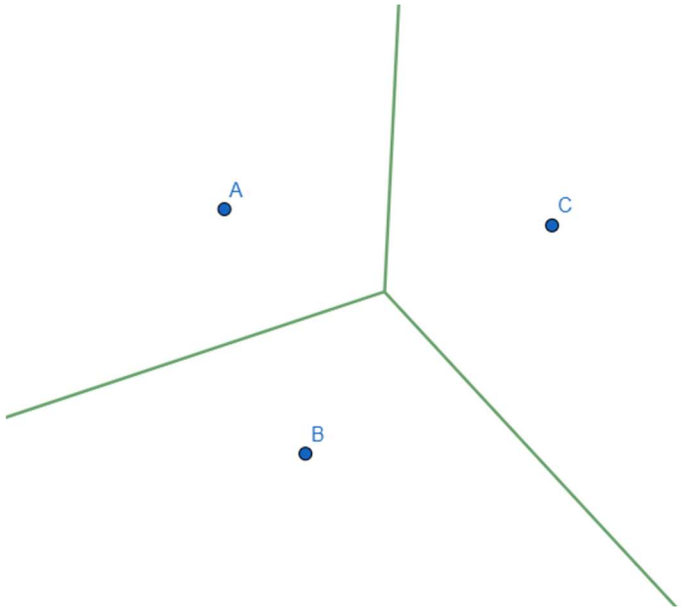


Figura 32. Resultado final de un diagrama de Voronoi obtenido con el algoritmo incremental.

Este algoritmo tiene una complejidad de O(n*log(n)) y es el más utilizado ya que es el algoritmo más sencillo para la realización de los diagramas de Voronoi.

En este proyecto se ha decidido utilizar el algoritmo incremental porque se quería utilizar el algoritmo que menor número de complicaciones pudiera dar ya que la parte en la que se centra este proyecto es en la capacidad de encontrar un nuevo punto que maximice la *región de Voronoi* y no tanto en la creación de los diagramas. Además, la definición del problema al que nos vamos a enfrentar, *MAXA-P-VOR(N)* introduce su propio algoritmo incremental al tener que ir buscando el nuevo punto que maximice la *región de Voronoi.*

4.1.3 PROPIEDADES DE UN DIAGRAMA DE VORONOI

A continuación, se van a explicar una serie de propiedades de los diagramas de Voronoi para profundizar en el entendimiento de estos y hacer más sencillo el proceso para llegar a comprender el funcionamiento de los algoritmos para *maximizar la región de Voronoi.*

I. Las *regiones de Voronoi* son conjuntos convexos, es decir: cualquier segmento que una dos puntos cualesquiera de una *región de Voronoi,* siempre queda dentro de dicha región, en ningún momento atraviesa la línea que lo separa de otra *región de Voronoi*. Podemos observar un ejemplo en la Figura 33.

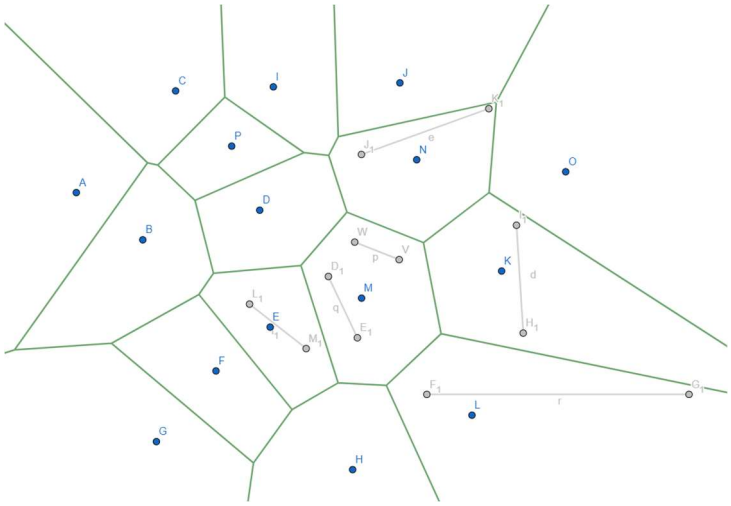


Figura 33.Conjunto convexo

II. Cada punto de una arista es equidistante de sus 2 puntos vecinos más cercanos. Se puede observar en la figura 34.

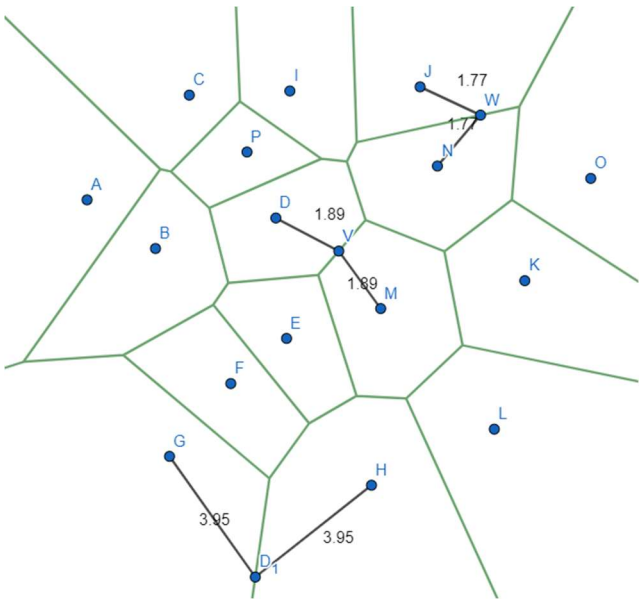


Figura 34. Puntos de una arista equidistante a los vecinos más cercanos

III. Los *diagramas de Voronoi* son conjuntos únicos y distintos entre sí. Para un conjunto de puntos dados, el *diagrama de Voronoi* generado es único, no hay un conjunto de puntos distintos que genere el mismo diagrama.

IV. Un vértice en el cuál 3 regiones de Voronoi se intersecan es equidistante de los puntos que forman dichas regiones, como se puede ver en la Figura 35.

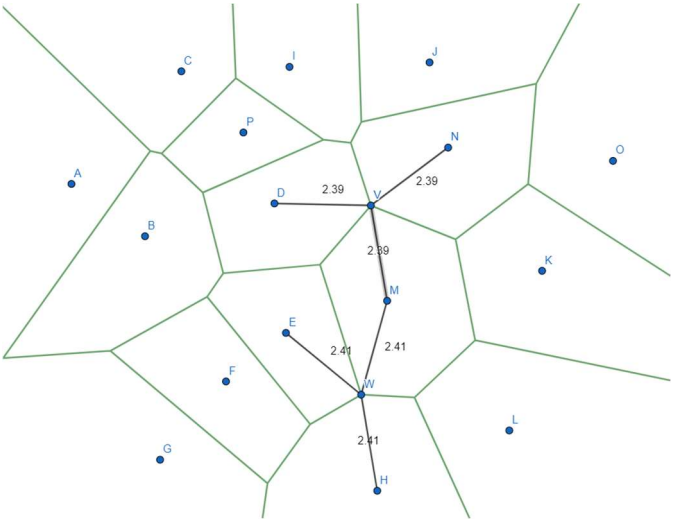


Figura 35. Vértice equidistante a los 3 puntos

V. El círculo que tiene como centro un vértice de una *región de Voronoi* y que pasa por 3 puntos que generan las regiones, no puede tener en su interior otro punto que genere otra *región de Voronoi,* se puede ver un ejemplo en la Figura 36.

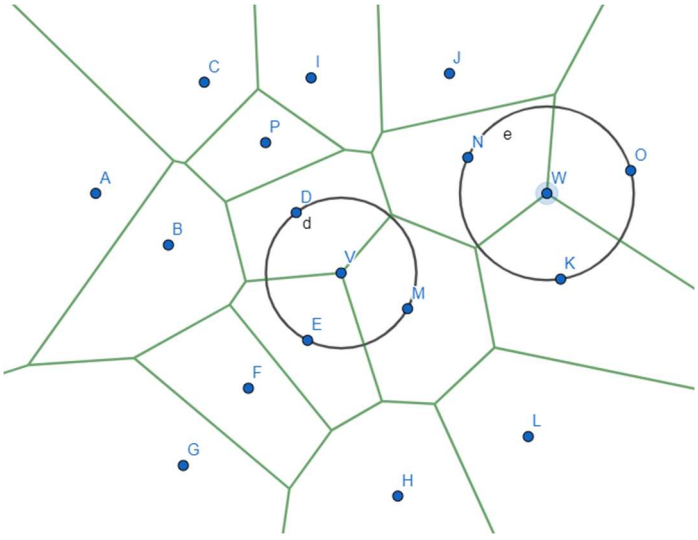


Figura 36. Círculos con vértice y 3 puntos que generan regiones de Voronoi

4.2 HEURÍSTICA RANDOM SEARCH

En esta sección se va a comentar la primera técnica heurística para calcular la *maximización de la región de Voronoi*, el algoritmo Random Search. Para facilitar la comprensión del funcionamiento de esta técnica se adjunta el pseudocódigo correspondiente.

```
[1] área = 0
[2] ptos = Generar aleatorios (R,50*n)
[3] Calcula Diagrama de Voronoi Vor(N,R)
[4] for (pi ∈ ptos)
[5]   { Calcula región de Voronoi con el nuevo punto Rv(pi, N ∪ {pi},R)
[6]   if (área < Área(Rv(pi, N ∪ {pi},R)))
[7]   área = Área(Rv(pi, N ∪ {pi},R))
[8]   p = pi
[9]   } return p
```

Este algoritmo está basado en una búsqueda aleatoria sobre el conjunto de puntos que son interiores a la región *R*, donde encontramos la nube de puntos *N* y el *diagrama de Voronoi* que forma dicha nube. Además, este diagrama es la entrada para nuestro problema de maximización. En este algoritmo generaremos una nube aleatoria $A = \{p_1, \ldots, p_m\}$ de m puntos interiores a nuestra región *R*, realizando una búsqueda del punto $p_i \in A$ y verifica:

$$Área\left(Rv(p_i, A \cup \{p_i\}, R)\right) \geq Área\left(Rv\left(p_j, A \cup \{p_j\}, R\right)\right) \quad \forall j \neq i \quad i,j \in \{1, \ldots, m\}$$

La función que genera la nube aleatoria de puntos es la siguiente:

```
#Función para generar puntos aleatorios
def aleatorio(numero):
    m =[]
    #bucle para generar numeros aleatorios y crear puntos
    for i in range (numero):
        a = numpy.random.uniform(2,498)
        b = numpy.random.uniform(2,398)
        nuevo = (a,b)
        m.append(nuevo)

    #bucle para mostar los puntos aleatorios por pantalla
    for k in range(len(m)):
        pygame.draw.circle(ventana,(0,0,255),m[k],1)
        pygame.display.flip()
    return m
```

Esta función recibe como argumento el número de puntos aleatorios que quiere generar el usuario, es decir, que el tamaño de la nube de puntos que genera es variable, a elección del usuario. Los valores que representan a cada punto *(x,y)* son generados aleatoriamente ayudándonos de la librería de Python numpy y el rango en el que le permitimos generar valores es entre 2 y 498 para el valor x y 2 y 398 para el valor y ya que las dimensiones de nuestro lienzo son de 500x400. Finalmente se crea el punto (x,y) y se añade a nuestro vector.

En esta función se ha añadido también la capacidad de mostrar estos puntos aleatorios en el lienzo, para ello nos apoyamos en la librería pygame de Python que nos permite dibujar en nuestro lienzo.

A continuación, se muestra el propio código del algoritmo Random Search:

```
#Funcion que implementa el algoritmo de random search
def random_search_funcion(m,RS,lista):
    posicion = 0
    area = 0
    if(lista == []):
        RS=[]
    #bucle para ir añadiendo a nuestra lista de puntos dibujada cada punto nuevo
    for k in range(len(m)):
        lista.append(m[k])
        RS.append(voronoiRegion(lista,len(lista)-1))
        area_nueva = Area2(RS[len(lista)-1]) #calculamos el area de cada una de
las regiones que formarian los nuevos puntos aleatorios
        if(area_nueva > area):
            #si dicho area es nayor que el area que ya teniamos lo guardamos y
nos quedamos con la posicion de dicho punto aleatorio
            area = area_nueva
            posicion = k
        RS.pop()
        lista.pop() #eliminamos el último elemento de nuestra lista de puntos
        print(k)
    return posicion
```

En el código se puede ver como únicamente vamos recorriendo la nube aleatoria de puntos y cómo vamos añadiendo cada uno de los puntos al diagrama de Voronoi y calculando el área para compararlo con el valor anterior y si es mejor actualizamos los valores, vamos realizando este proceso hasta terminar con todos los puntos aleatorios y devolvemos la posición del punto que genera la mayor área de toda nuestra nube de puntos aleatoria.

En la Figura 37 se pude ver un ejemplo de salida para este algoritmo con un *diagrama de Voronoi* inicial de 20 puntos.

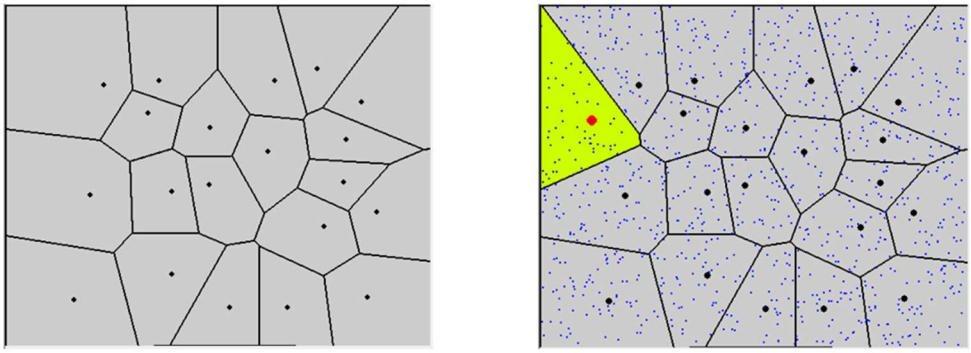


Figura 37. Ejemplo de salida de Random Search con m = 1000 puntos

Los resultados obtenidos con este algoritmo y para este ejemplo son los siguientes:

Puntos aleatorios: 1000

Porcentaje de área obtenida: 6,59%

Tiempo tardado: 2s

4.3 HEURÍSTICA SIMULATED ANNEALING

La técnica heurística que se va a tratar en esta sección es la del *Simulated Annealing* o *Recocido Simulado.* Para facilitar la comprensión del algoritmo se muestra el pseudocódigo de la técnica:

```
[1]   do
[2]         {do
[3]               {Generar solución y ∈ Vecindad(x) ⊂ S
[4]               Evalúa resultado ← C(x) – C(y)
[5]               if (resultado <0)  x ← y
[6]               else
[7]                     if ((resultado>0) Λ (U(0,1) < e^(−resultado/T)))  x ← y
[8]               n ← n+1
[9]               } while (n ≤ N(T)
[10]        Disminuye T
[11]        }while (parada == false)
```

Como se observa en el pseudocódigo expuesto anteriormente hay que definir una serie de parámetros para que el algoritmo funcione correctamente: la *configuración inicial*, una *función de coste*, la *función vecindad*, las estrategias para el templado del algoritmo y el *criterio de parada* utilizado.

La *configuración inicial*, como se ha explicado en la anterior técnica heurística es el *diagrama de Voronoi* de los puntos *N* de nuestra región *R*. Además, el conjunto de puntos que se puede evaluar son todos aquellos que están en el interior de la región *R*.

La *función de coste* que vamos a utilizar en este caso es el área que genera el punto evaluado en cada momento. De tal forma que la función de coste que vamos a utilizar queda definida de la siguiente manera:

$$C(p_i) = Área\left(Rv(p_i, A \cup \{p_i\}, \mathrm{R})\right)$$

La *función de vecindad* debe encontrar un nuevo punto p'_i para analizar después de haber analizado el punto p_i. Esta función suma a las coordenadas del punto p_i un valor aleatorio que siga una distribución normal *N(0,1)*. Podemos expresar esta función de la siguiente forma:

$$\begin{cases} x'_i = x_i + N(0,1) \\ y_i = y + N(0,1) \end{cases}$$

El código utilizado para la función de vecindad se muestra a continuación:

```
#Funcion que nos genera un punto vecino para el algoritmo de simulated annealing
def generar_vecindad(punto):
    global div

    while True:
        #generamos dos numeros aleatorios entre 0 y 1
        aleatorio1 = numpy.random.uniform(0,1)
        aleatorio2 = numpy.random.uniform(0,1)

        #Calculamos los sumandos para los nuevos puntos x e y
        sumando1 = (numpy.sqrt(-
2*numpy.log(aleatorio1))*numpy.sin(2*numpy.pi*aleatorio2))
        sumando2 = (numpy.sqrt(-
2*numpy.log(aleatorio1))*numpy.cos(2*numpy.pi*aleatorio2))

        #Calculamos las coordenadas de nuestro punto vecino
```

```
        x = punto[0]/500 + sumando1/div
        y = punto[1]/400 + sumando2/div

        if((x>=0)and(y>=0)and(x<=1)and(y<=1)):#repetimos este bucle mientras que
nos salgamos del cuadrado unidad
            break

    #Volvemos a coordenadas de pantalla (pixels)
    x = x*500
    y = y*400
    vecino = [x,y] #Creamos el punto vecino y lo devolvemos
    return vecino
```

Cabe destacar que la función *N(0,1)* genera puntos aleatorios en el cuadrado unidad por lo que, en nuestro caso al trabajar en píxeles tenemos que hacer una transformación de unidades para poder trabajar de esta forma. Los valores que vamos sumando están divididos por un factor *"div"*, esto es debido a que con cada iteración de nuestro algoritmo queremos ir haciendo más pequeño el área por donde vamos explorando los puntos, este valor es controlado por la función principal como explicaremos a continuación.

Una vez calculado el nuevo punto a evaluar, el vecino volvemos a coordenadas de pantalla y devolvemos el punto.

Las estrategias de templado son el conjunto de técnicas que hacen que nuestro algoritmo converja. Como se ha explicado previamente, el problema principal que tiene este algoritmo es que quede atrapado en un máximo local y no llegue a convertirse nunca en un máximo global. Para solucionar este problema se introduce una variable a la que llamaremos *Temperatura* y denotaremos con *T*. Esta variable nos permite con una cierta probabilidad explorar zonas peores de nuestra región *R* para así poder aumentar el espacio de búsqueda. Donde radica la complejidad de este algoritmo es en elegir correctamente estos valores para que la convergencia del mismo sea correcta y el espacio analizado sea lo suficientemente grande para poder analizar todas las zonas de nuestra región.

Para el problema MAXA-P-VOR(N) hemos elegido los siguientes valores: en primer lugar, la *temperatura inicial,* $T_0 = 2*n$ siendo *n* el número de puntos que hay en el *diagrama de Voronoi* inicial. El objetivo de esta temperatura es que vaya disminuyendo a medida que se va iterando para que la probabilidad de analizar un nuevo punto que, pese a que nuestra

función de coste diga que es peor, lo elijamos candidato para generar un nuevo espacio de búsqueda y conforme vaya convergiendo el algoritmo la probabilidad de cambiar a un punto que es peor sea menor. En este caso la disminución que se utiliza es lineal y fue propuesta por Szu y Hartley [12], para conseguir una convergencia más rápida. Se puede expresar de la siguiente forma $T_i = \frac{T_0}{1+i}$ siendo i el número de iteraciones de nuestro algoritmo. Esta convergencia puede acelerarse utilizando la función propuesta por Ingber [13] y que se define de la siguiente forma $T_k = \frac{T_0}{e^i}$, sin embargo, con la disminución lineal se consigue una convergencia adecuada que a la vez permite explorar la mayoría del área de la región *R*.

Para la *temperatura final*, la condición utilizada ha sido la siguiente $T_f \leq 0{,}005$, sin embargo, al realizar diferentes pruebas se ha observado como cuando el número de puntos que hay en el diagrama es mayor a 20, tener esta condición de parada tan baja no beneficiaba y el rendimiento empeoraba notablemente frente a la mejora en el porcentaje de área obtenido, que era mínimo. Por ello, para los diagramas en los que el número inicial de puntos es mayor o igual a 20 se ha elegido como *criterio de parada* $T_f \leq 0{,}025$.

Por último, el divisor que se ha explicado previamente y que se usa en la función de vecindad es de *div* = 3.5 para la mayoría de los diagramas, ya que, tras probar esta técnica, en los diagramas en los que el número de puntos iniciales es menor de 7 la región evaluada no era óptima y es por ello por lo que se utilizó un factor de *div* = 2. Además, como se quiere que este factor vaya aumentando poco a poco con cada iteración se va aumentando de la siguiente forma *div = div/0,999* para que al dividirlo en el sumando el número vaya siendo cada vez menor y los vecinos se generen más cerca del punto alrededor del cual se quiere buscar.

A continuación, se muestra el código utilizado para la realización de este algoritmo:

```
#Función que implementa el algorimto de simulated annealing
def simulated_annealing(lista,punto,voronoi):
    global div
    #Establecemo los parametros iniciales y finales de nuestro algoritmo
    temperatura_final = 0.005
    temperatura_inicial = 2*len(lista)

    if(len(lista) <7):
        div=2
    elif(len(lista) > 20):
```

```
        temperatura_final = 0.025
    n = 0
    k = 0
    puntos_evaluados = []
    if(lista ==[]):#Si el valor de voronoi que pasamos por pantalla es nulo
creamos la lista
        voronoi = []

    #bucle para el algoritmo
    while True:
        while True:
            if(k == 0):
                temperatura_iterable = temperatura_inicial/1
            vecino = generar_vecindad(punto)
            if punto not in puntos_evaluados:
                puntos_evaluados.append(punto)
            puntos_evaluados.append(vecino)
            lista.append(punto)
            voronoi.append(voronoiRegion(lista,len(lista)-1))
            #calculamos el area del punto almacenado
            area_punto = Area2(voronoi[len(lista)-1])
            lista.pop()
            voronoi.pop()
            lista.append(vecino)
            voronoi.append(voronoiRegion(lista,len(lista)-1))
            #Calculamos el area del vecino nuevo generado
            area_vecino = Area2(voronoi[len(lista)-1])
            lista.pop()
            voronoi.pop()
            #Calculamos la resta de las areas calculadas
            delta = area_punto - area_vecino
            if(delta < 0):
                #Si la resta es negativa es que el area del nuevo vecino es mejor
                punto = vecino
            else:
                if((delta > 0) and (numpy.random.uniform(0,1) < numpy.exp(-
delta/temperatura_iterable))):
                    #si no es mejor el area del vecino con cierta probabilidad
queremos cambiar ha dicho punto
                    punto = vecino
            n = n + 1
            if(n > temperatura_inicial/(1+k)):
                #Condicion de salida del bucle interior
                break
        print(n)
        k = k + 1
        temperatura_iterable = temperatura_inicial/(1+k) #Actualizamos la
temperatura
        div = div/0.999 #Modificamos el factor de división
        if(temperatura_iterable <= temperatura_final):
            #Evaluamos la condición de parada y si se cumple salimos del bucle
            break
    print(len(puntos_evaluados))
    div = 3.5 #Volvemos a dejar el parametro div como estaba inicialmente
    return punto,puntos_evaluados
```

En la Figura 38 se puede ver un ejemplo de solución para la heurística *Simulated Annealing* para un *diagrama de Voronoi* de 20 puntos.

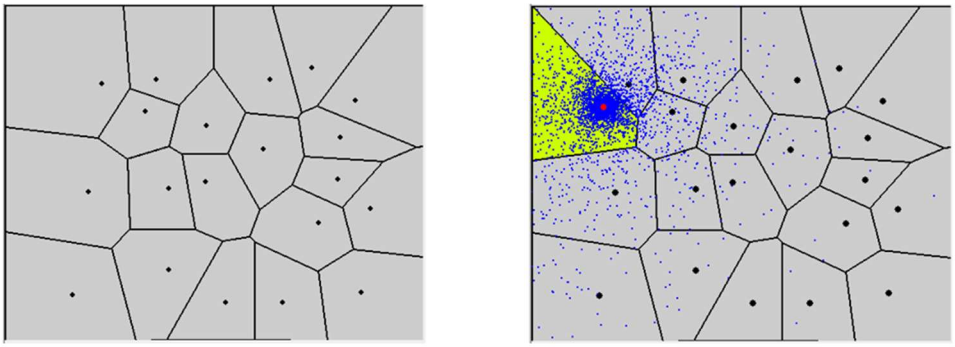


Figura 38. Ejemplo de salida para Simulated annealing

Los resultados obtenidos con este algoritmo y para este ejemplo son los siguientes:

Puntos evaluados: 8040

Porcentaje de área obtenida: 6,88%

Tiempo tardado: 24s

4.4 HEURÍSTICA ANT SYSTEMS

En esta sección se va a explicar la técnica heurística *Ant Systems* y la adaptación que se ha hecho de este algoritmo para el problema MAXA-P-VOR(N).

A continuación, se muestra el pseudocódigo utilizado para este algoritmo:

```
[1] for hormiga = 0 to max //Número máximo de hormigas que generamos
[2]   {for puntos_visitados = 0 to maxi// n° ptos que visita cada hormiga
[3]         {Resultado = C(x)
[4]               for rastro
[5]                     {if (resultado > rastro(i))
[6]                           rastro(i) = resultado
[7]                           break
[8]                     }
[9]   }Reducimos el área en el que se generan los puntos visitados
[10] }devolvemos el punto que genera el mayor área
[11] Fin
```

Esta técnica se inspira en la optimización biológica de los caminos que recorren las hormigas y por ello es necesario analizar los diferentes caminos y puntos que analiza nuestro agente. Esto quiere decir que la hormiga o agente irá recorriendo la región de forma aleatoria e irá analizando en función de una *función de costes* cuáles son los mejores lugares por los que ha pasado y los irá almacenando. Por ello, para este algoritmo debemos definir el *criterio de parada* y la *función de coste*.

La *función de coste* que se va a utilizar es la misma que la explicada con anterioridad para la técnica heurística *Simulated Annealing* y queda definida de la siguiente manera:

$$C(p_i) = Área\left(Rv(p_i, A \cup \{p_i\}, \mathrm{R})\right)$$

El *criterio de parada* utilizado depende del número de puntos inicial que contenga nuestro *diagrama de Voronoi*, si hay más de 20 puntos en el diagrama la *parada = N/2* siendo *N* el número de puntos del diagrama y para las situaciones en las que ha menos de 20 puntos se ha definido que la *parada = 2*N*. Esto quiere decir que en función del número de puntos que tenemos inicialmente en nuestro diagrama, generaremos *N/2* o *2*N* agentes. Por ejemplo, si tenemos un diagrama de 10 puntos, generaremos en total 20 agentes que recorrerán la región de búsqueda.

Para esta técnica es importante mencionar que hay que dejar un rastro, esto quiere decir que cada hormiga al pasar debe almacenar los mejores puntos con los que se encuentre a lo largo de todo el trayecto realizado. En este caso el rastro son los 10 mejores puntos que se van encontrando y que cada hormiga debe ir actualizando cuando pasa. Este rastro se usa para ir restringiendo la región en la que se permite a las hormigas moverse, es decir, la zona donde se pueden ir generando puntos aleatorios se va modificando en base al rastro que se tiene en cada una de las iteraciones. Esta región se obtiene entre el punto más grande, es decir, el punto que más dista del (0,0) en píxeles, hasta el punto más pequeño, o el más cercano al (0,0). Una vez se ha obtenido el máximo de la lista y el mínimo, se almacenan las coordenadas *(x,y)* de cada uno de los puntos y en la generación de números aleatorios se definen los límites de generación en base a estos valores. De esta forma se consigue que el

algoritmo se centre en la zona donde se considera que esta el punto que *maximiza la región de Voronoi.*

El código utilizado para este algoritmo se muestra a continuación:

```
#Función que implementa el algoritmo de ant systems
def ant_systems_funcion(lista,voronoi):
    #Definimos las condiciones de parada en función del número de puntos
introducidos
    if(len(lista)>20):
        parada = len(lista)/2
    else:
        parada = 2*len(lista)

    #Damos los valores iniciales a nuestras variables
    area = 0
    posicion = 0
    contador = 0
    contador_2 = 0
    puntos_buenos = []
    areas_buenas = []
    puntos_evaluados = []
    i_x_inf = 0
    i_x_sup = 500
    i_y_inf = 0
    i_y_sup = 400

    #Iniciamos el bucle que realiza el algoritmo
    while True:
        while True:
            x = numpy.random.uniform(i_x_inf,i_x_sup)
            y = numpy.random.uniform(i_y_inf,i_y_sup)
            punto = [x,y] #Generamos un número aleatorioa
            if (punto in puntos_evaluados):
                #Guardamos el puntos para luego poder representar los puntos
aleatorios generados
                punto = [punto[0]+1,punto[1]+1] #Si ya existe el punto evaluado
lo desplazamos para que no se pinten uno encima de otro
                puntos_evaluados.append(punto)
            else:
                puntos_evaluados.append(punto)
            #Añadimos el punto a la lista y creamos su región de voronoi
            lista.append(punto)
            voronoi.append(voronoiRegion(lista,len(lista)-1))
            area_punto = Area2(voronoi[len(lista)-1]) #Calculamos el área del
punto aleatorio
            lista.pop()
            voronoi.pop()
            if(len(puntos_buenos) <= 10):
                #Rellenamos el numero de puntos buenos que deseamos
                puntos_buenos.append(punto)
                areas_buenas.append(area_punto)
            else:
                #Cuando nuestra lista de puntos buenos esta rellena vamos
comparando el area de cada uno de los puntos aleatorios con los del vector de
puntos buenos
                for i in range(len(areas_buenas)):
```

```
                    if (area_punto > areas_buenas[i]):
                        #si el area es mejor eliminamos la que es peor y lo
añadimos a la lista
                        puntos_buenos.pop(i)
                        areas_buenas.pop(i)
                        puntos_buenos.append(punto)
                        areas_buenas.append(area_punto)
                        #Cuando encuentra uno que es mejor salimos del bucle
                        break
            contador +=1
            if(contador == 100):
                #Repetimos este proceso hasta llegar a 100 iteraciones
                contador = 0
                break
        contador_2 +=1
        minimo = min(puntos_buenos) #Buscamos el punto mínimo de la lista de
puntos buenos
        maximo = max(puntos_buenos) #Buscamos el punto máximo de la lista de
puntos buenos
        if(contador_2 >= 3 and len(lista) >= 6 ):
            #Vamos reduciendo los valores donde se pueden generar puntos mínimos
            i_x_inf = minimo[0]
            i_x_sup = maximo[0]
            i_y_inf = minimo[1]
            i_y_sup = maximo[1]
        elif(contador_2 >= 2 and len(lista)<6):
            #Reducimos los valores donde se pueden generar puntos buenos
            i_x_inf = minimo[0]
            i_x_sup = maximo[0]
            i_y_inf = minimo[1]
            i_y_sup = maximo[1]
        print(contador_2)
        if(contador_2 >= parada):
            #Cuando hemos terminado seleccionamos el que mejor area tiene y
devolvemos la posicion
            #los puntos buenos y los puntos evaluados
            for i in range(len(puntos_buenos)):
                lista.append(puntos_buenos[i])
                voronoi.append(voronoiRegion(lista,len(lista)-1))
                area_tmp = Area2(voronoi[len(lista)-1])
                if(area_tmp > area):
                    area = area_tmp
                    posicion = i
                voronoi.pop()
                lista.pop()
            break
    print(len(puntos_evaluados))
    return puntos_buenos,puntos_evaluados,posicion
```

En la Figura 39 se puede ver un ejemplo de este algoritmo para un *diagrama de Voronoi* de 20 puntos.

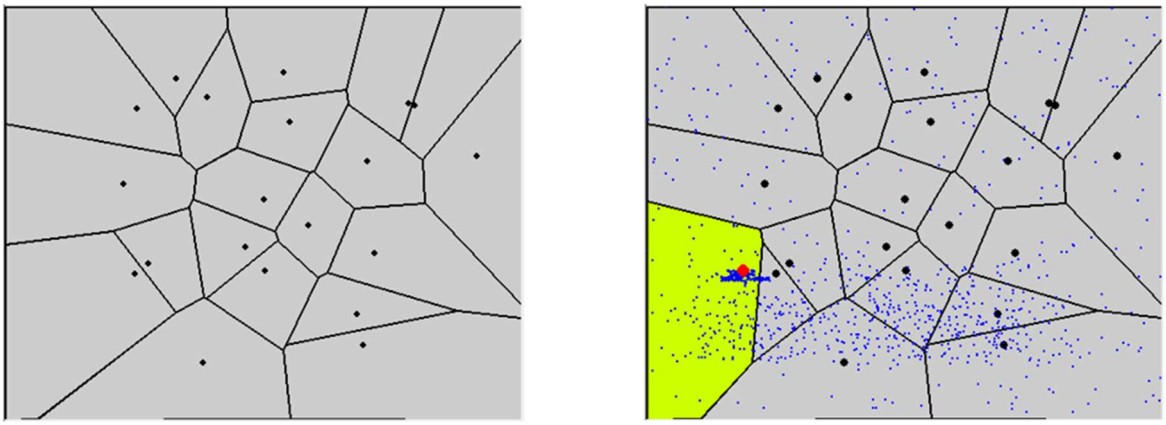


Figura 39. Ejemplo de solución para el algoritmo Ant Systems

Los resultados obtenidos con este algoritmo son los siguientes:

Puntos evaluados: 4000

Porcentaje de área obtenida: 9,16%

Tiempo tardado: 4s

Capítulo 5. DISEÑO DE LA APLICACIÓN

En esta sección se va a explicar el diseño de la aplicación, los elementos que contiene y una guía sencilla sobre como poder moverse por la misma.

En primer lugar, en la Figura 40 se muestra una imagen de la pantalla principal de la aplicación.

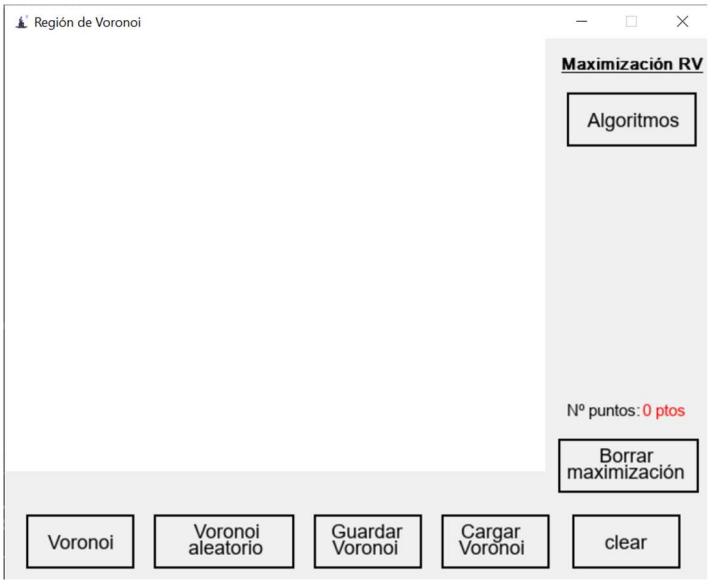


Figura 40. Ventana principal de la aplicación.

Se van a ir explicando todas las funcionalidades que tiene la aplicación y como se puede acceder a cada una de ellas.

En primer lugar, al iniciar el programa, el usuario es capaz de dibujar puntos en el lienzo con el ratón de su ordenador. Se puede observar un ejemplo en la Figura 41.

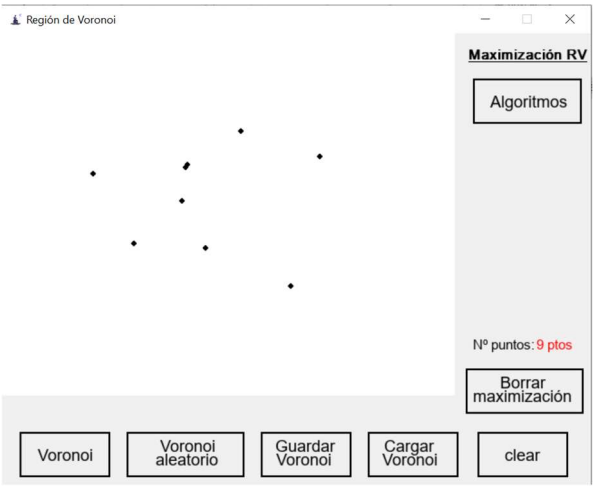


Figura 41. Puntos insertados por el usuario.

Las funcionalidades principales de la aplicación, en lo que se basa el proyecto, es en la construcción de *diagramas de Voronoi* y en la posterior maximización de la *región de Voronoi.*

Hay 3 formas en las que un usuario puede generar un diagrama:

- Si el usuario ha insertado los puntos en el mapa, debe pulsar el botón "*Voronoi*" y se calculará el diagrama de los puntos pintados por el usuario.
- El programa es capaz de generar un *diagrama de Voronoi* de forma aleatoria mediante el botón "Voronoi aleatorio". Aparecerá una ventana que pedirá al usuario el número de puntos que desea para crear el diagrama, se muestra un ejemplo en la Figura 42.

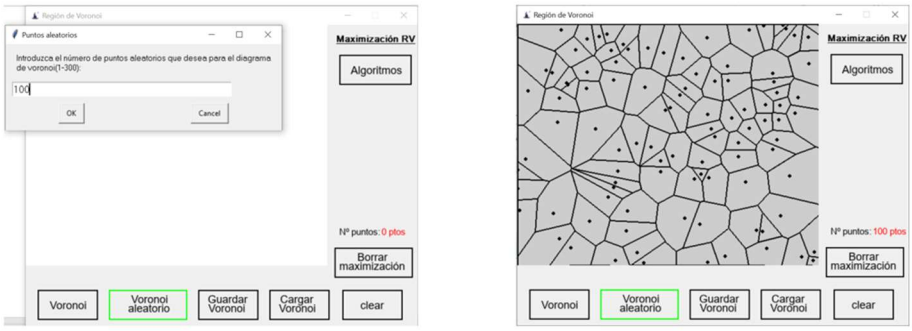


Figura 42. Generar diagrama de Voronoi aleatorio.

- La última opción que existe para generar un diagrama es cargar un archivo con un diagrama guardado con anterioridad mediante el botón "Guardar Voronoi". Este botón nos da la posibilidad de guardar cualquier diagrama que exista en el lienzo y pulsado el botón "Cargar Voronoi" podremos reutilizar este archivo cuando queramos. Esta funcionalidad ha sido de gran utilidad a la hora de realizar la comparación de las diferentes técnicas heurísticas, se muestra un ejemplo en la Figura 43.

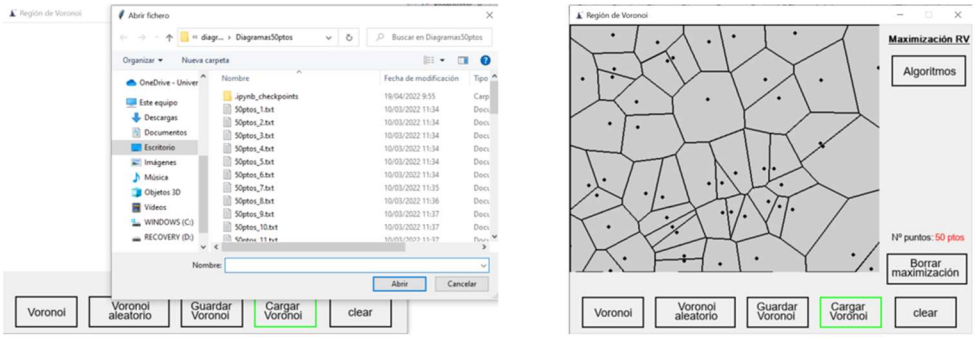


Figura 43. Cargar un diagrama de Voronoi almacenado.

La funcionalidad más importante es la de poder encontrar un nuevo punto *q* que *maximice la región de Voronoi*. Para acceder a cualquiera de los 3 algoritmos desarrollados se tiene que pulsar el botón "Maximización RV", nos aparecerá una ventana como la que se muestra en la Figura 44, y eligiendo una de las tres técnicas heurísticas se tendrá la posibilidad de calcular la *maximización de la región de Voronoi.*

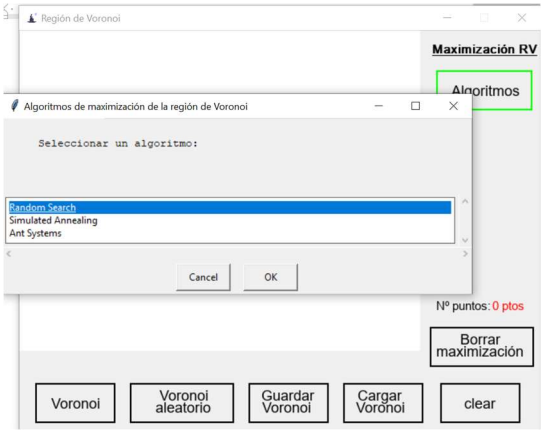


Figura 44. Selección del algoritmo para realizar la maximización de la región de Voronoi.

Al seleccionar un algoritmo para realizar la maximización, obtendremos gráficamente la solución encontrada por el algoritmo y los resultados obtenidos, tanto en tiempo de ejecución hasta obtener la respuesta como en porcentaje de área obtenido por la solución. El botón de "Borrar maximización" permite comparar diferentes algoritmos para el mismo diagrama ya que únicamente borra la solución obtenida por la heurística utilizada y vuelve a calcular el diagrama inicial, el de partida para poder utilizar otro algoritmo. Se muestra en las Figuras 45, 46 y 47 el resultado obtenido por cada uno de los 3 algoritmos para que sirva como antesala para el capítulo de análisis de las técnicas heurísticas.

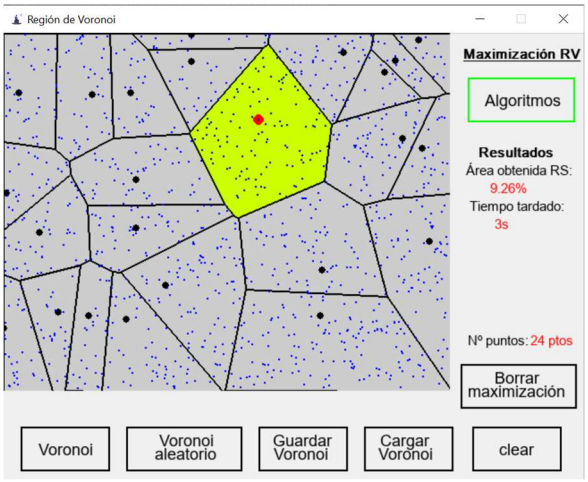


Figura 45. Resultado obtenido con Random Search para un diagrama de 23 ptos y 1150 ptos aleatorios.

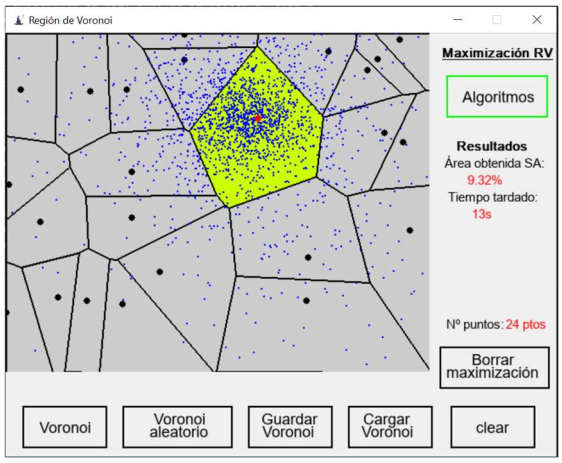


Figura 46. Resultado obtenido con el algoritmo Simulated Annealing para 23 ptos y 1886 ptos evaluados.

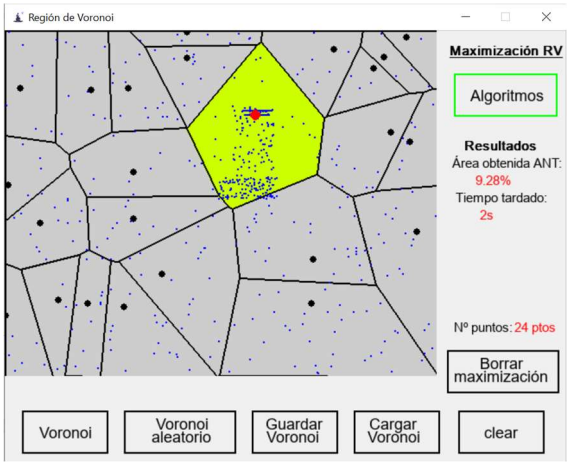


Figura 47. Resultados obtenidos con Ant Systems para un diagrama de 23 ptos y 1200 ptos evaluados.

Capítulo 6. ANÁLISIS DE RESULTADOS

6.1 ÁREA OBTENIDA Y TIEMPO DE EJECUCIÓN

En esta sección se procede a realizar un análisis comparativo de las diferentes técnicas utilizadas para calcular la *maximización de la región de Voronoi*. Para este apartado se ha realizado el análisis de 50 *diagramas de Voronoi* aleatorios de 10, 50 y 100 puntos, se han aplicado a dichos diagramas las técnicas heurísticas explicadas, *Random Search, Simulated Annealing y Ant Systems* para comparar el porcentaje de área en media obtenido y el tiempo empleado por cada uno de los algoritmos. Para hacer un análisis lo más preciso posible, el algoritmo *random search*, obtiene los resultados utilizando un número de punto aleatorios predefinido y de valor 50*n siendo n el número de puntos del *diagrama de Voronoi* de partida. Los resultados medios obtenidos se muestran a continuación:

- Resultados para *Random Search – RS.*

Puntos	*% Medio Área Región Voronoi*	*Tiempo [segundos]*
10	15,78	0,06
50	4,51	9,88
100	2,42	23,64

*Tabla 10. Resultados Random Search con m = 50*Puntos*

- Resultados para *Simulated Annealing – SA.*

Puntos	*% Medio Área Región Voronoi*	*Tiempo [segundos]*
10	16,27	4,36
50	4,52	13,42
100	2,43	43,06

Tabla 11. Resultados Simulated Annealing con: T_0*= 2*Puntos,* T_f *= 0,005 y* $T_i = \frac{T_0}{1+i}$

- Resultados para *Ant Systems – ANT.*

Puntos	*% Medio Área Región Voronoi*	*Tiempo [segundos]*
10	15,95	1,86
50	4,48	7,7
100	2,38	25,24

Tabla 12. Resultados Ant Systems con iteraciones $= \frac{Puntos}{2}$ $\forall\ Puntos > 20$ *o iteraciones = 2*Puntos* $\forall\ Puntos < 20$

Como se puede observar en las tablas los valores de porcentaje medio de área son muy parecidos entre ellos, esto puede dar a entender que la solución que están encontrando, es decir, el nuevo punto *q* que pertenece a la región *R* y que *maximiza la región de Voronoi* es lo más cercano al óptimo posible.

Como se ha explicado con anterioridad, el problema planteado es NP-hard y por tanto no podemos encontrar un algoritmo exacto para solucionar el problema. Por ello las soluciones aproximadas obtenidas son cercanas al óptimo y los algoritmos dan resultados diferentes.

Aunque el porcentaje de área sea similar sí que es reseñable destacar que la técnica que proporciona los mejores resultados es *Simulated Annealing.* Como los resultados son la media para 50 diagramas diferentes, que un algoritmo obtenga mejores resultados es notable, eso indica que en la mayoría de los diagramas estudiados el algoritmo mencionado previamente obtiene resultados notablemente mejores. En la Figura 48 se muestra una gráfica en la que se compara el número de puntos del diagrama y los resultados del porcentaje medio de área obtenido por cada uno de ellos.

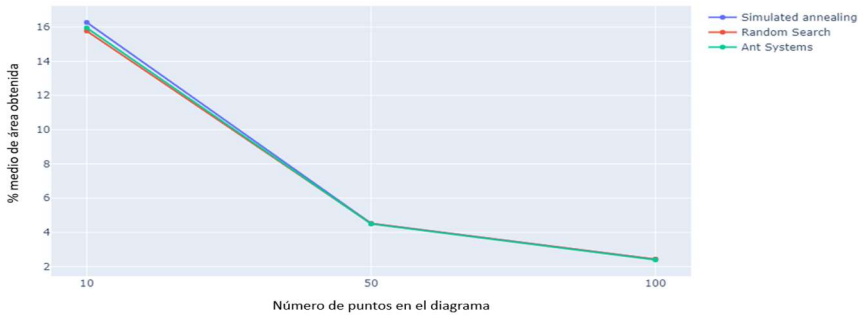


Figura 48. Comparativa de porcentaje medio de área obtenida por cada técnica

En la Figura 48 se puede observar lo explicado con anterioridad. Además, se puede ver como el porcentaje de área obtenido por cada uno de los algoritmos se va igualando y disminuyendo. Esto es el resultado esperado ya que a medida que nuestro *diagrama de Voronoi* inicial contiene más puntos, el área de cada una de las *regiones de Voronoi* se va haciendo más pequeña debido al mayor número de particiones del espacio.

Por último, es reseñable mencionar como a la hora de elegir que técnica heurística utilizar hay que analizar el tiempo que tarda cada una de las técnicas. *SA* es la heurística que peores resultados obtiene en cuanto al tiempo necesario para obtener la solución, sin embargo, es la técnica que mejores resultados obtiene para el porcentaje medio de área. Es por ello, que a la hora de elegir hay que encontrar un equilibrio o tomar una decisión en base a si se buscan los mejores resultados posibles u obtener un resultado en el menor tiempo.

6.1 CONVERGENCIA DE LAS TÉCNICAS HEURÍSTICAS

En esta sección se van a analizar los tiempos de convergencia de las diferentes técnicas heurísticas utilizadas, es decir, las curvas de crecimiento de los algoritmos.

En la Figura 49 se muestra la curva de crecimiento para el algoritmo de *Random Search* de un diagrama de 10 puntos.

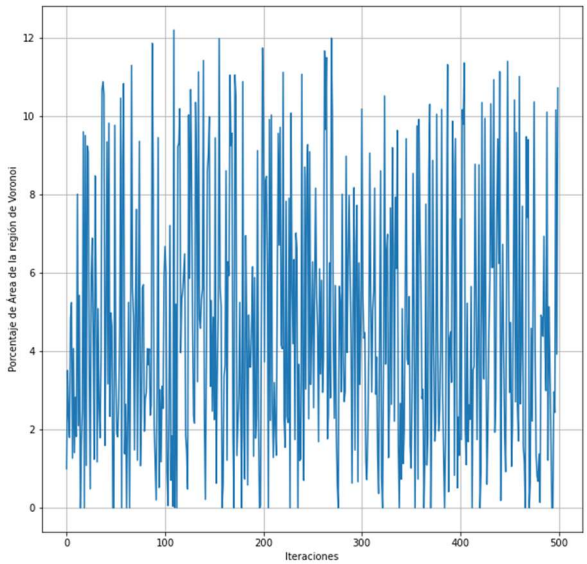


Figura 49. Curva de crecimiento para el problema de maximización de la región de Voronoi con RS

Como se puede observar este algoritmo no converge hacia una solución óptima ya que únicamente prueba los puntos aleatorios generados previamente. Es por ello por lo que con el objeto de poder analizar la convergencia de cada uno de los algoritmos se hará una aproximación de la curva de crecimiento del algoritmo eliminando los valores de los puntos aleatorios que hagan retroceder al algoritmo, es decir, que proporcionen un resultado peor al anterior. La gráfica modificada la podemos ver en la Figura 50.

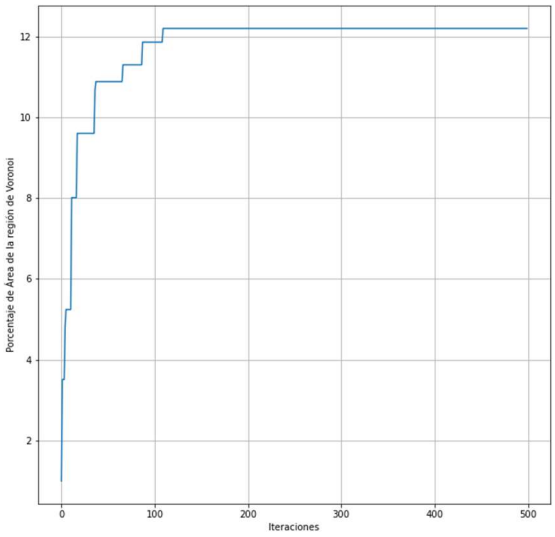


Figura 50. Crecimiento del algoritmo RS para el problema de maximización de la región de Voronoi

La curva de crecimiento para el algoritmo *Simulated Annealing* para un diagrama de 10 puntos se muestra en la Figura 51.

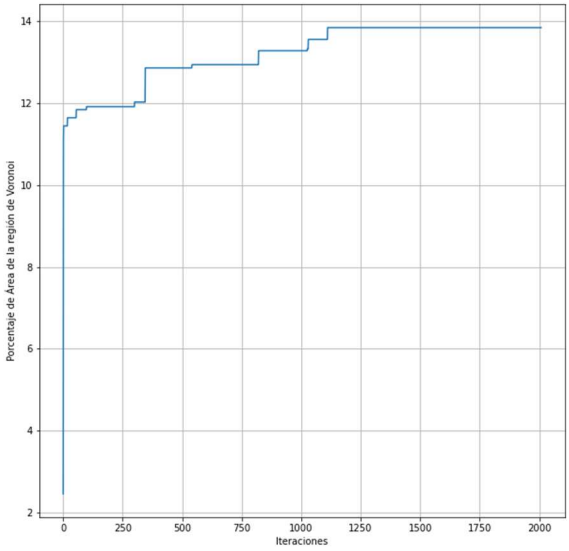


Figura 51. Crecimiento del algoritmo SA para el problema de maximización de la región de Voronoi

La curva de crecimiento para el algoritmo *Ant Systems* se muestra en la Figura 52 para un diagrama de 10 puntos.

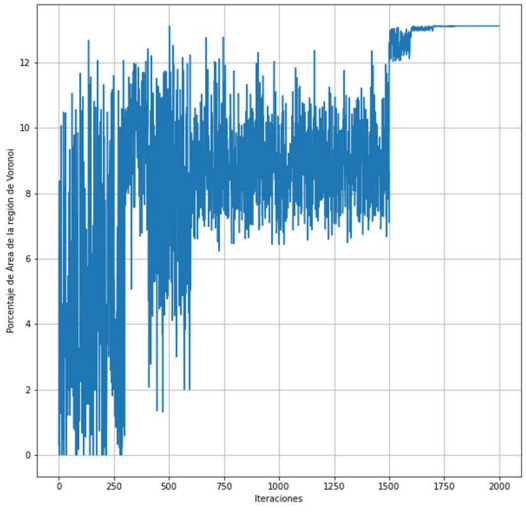


Figura 52. Crecimiento del algoritmo SA para el problema de maximización de la región de Voronoi

Como se puede observar, al igual que sucedía en la técnica *Random Search,* hay puntos que hacen retroceder al algoritmo por el simple hecho de realizar búsquedas en puntos aleatorios.

Por ello, se presenta en la Figura 53 una adaptación de esta gráfica eliminando los puntos que empeoran el crecimiento natural del mismo.

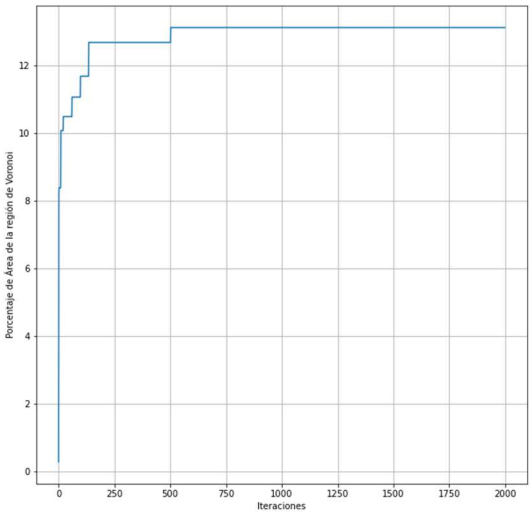


Figura 53.Crecimiento del algoritmo ANT para el problema de maximización de la región de Voronoi

Para poder ver más claramente y poder hacer una comparación de la convergencia de los algoritmos se muestra en la Figura 54 un gráfico en el que aparecen las curvas de crecimiento para un ejemplo de un diagrama de 10 puntos representadas sobre la misma gráfica.

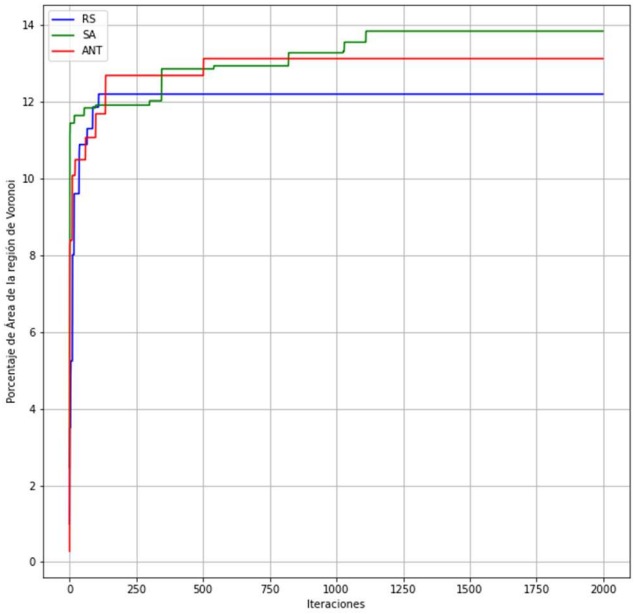


Figura 54. Crecimiento para los diagramas de Voronoi de 10 puntos

Como se observa en la Figura 54 la convergencia del algoritmo de *Random Search* es la más rápida, seguida del algoritmo de *Ant Systems* y por último la técnica *Simulated Annealing.*

A la hora de decidir que algoritmo utilizar, se puede observar que los resultados de porcentaje de área media que obtenemos con todos son bastante parecidos, por lo que habría que mirar otras condiciones.

Si se quisiera obtener un buen resultado para el problema de *maximización de la región de Voronoi* en el menor número de iteraciones posible, la heurística *RS* sería la vencedora. Si por el contrario lo que queremos es obtener la mejor solución imposible sin depender de otros factores se debería elegir la técnica de *SA.* Si lo que se busca es obtener el resultado en el menor tiempo posible, se podría dudar entre elegir la técnica de *RS* o *ANT,* sin embargo, como *ANT* es capaz de proporcionar resultados mejores en un tiempo muy parecido a la heurística *RS* sería la técnica elegida.

Capítulo 7. CONCLUSIONES Y TRABAJOS FUTUROS

Como se explicó en el Capítulo 3, el objetivo principal sobre el que se desarrolla este trabajo es el de realizar un software gráfico interactivo capaz de construir *diagramas de Voronoi* y de dar una solución aproximada al problema de encontrar un nuevo punto *q* dentro de la región *R* que maximice el área de su *región de Voronoi*.

Para afrontar este proyecto se dividió el trabajo en diferentes etapas, el diseño de la interfaz, la programación del algoritmo incremental para la construcción de los diagramas de Voronoi y por último el diseño de los algoritmos heurísticos para la maximización. En este capítulo se desea hacer un breve resumen de todas las etapas principales en las que se ha basado este proyecto.

7.1 DISEÑO DE INTERFAZ

La primera parte consistió en el diseño de la herramienta gráfica sobre la que se mostrarían los resultados obtenidos.

Esta herramienta cumplió con todos los objetivos propuestos, es interactiva, es decir, el usuario puede trabajar sobre la aplicación y explorar las diferentes funcionalidades. Es una herramienta gráfica ya que los resultados obtenidos los muestra en un diagrama representado por la propia aplicación, además, es capaz de dar resultados, valores, para el área y tiempo tardado hasta calcular la maximización.

Se ha incluido funcionalidad adicional ya que al ir observando las necesidades que podría tener el usuario se decidió ampliar las capacidades, es por ello por lo que se ha añadido la capacidad para guardar *diagramas de Voronoi,* algo de gran utilidad cuando se quiere comparar los resultados obtenidos por los diferentes algoritmos, así como la funcionalidad para borrar la maximización calculada y poder aplicar sobre el mismo diagrama otra técnica heurística.

7.2 DIAGRAMAS DE VORONOI

El programa es capaz de calcular el *diagrama de Voronoi* de los puntos que el usuario introduzca por pantalla. Este cálculo lo realiza rápidamente y muestra el resultado por la aplicación de forma que se puede observar gráficamente la solución.

Como ya se explicó en el Capítulo 4 la decisión en este apartado fue la de decantarnos por el algoritmo incremental para la construcción de los *diagramas de Voronoi* porque obtuvimos resultados muy buenos y la dificultad para entender el funcionamiento de este método era menor.

7.3 TÉCNICAS HEURÍSTICAS

La sección sobre la que giraba el proyecto era la programación de las técnicas heurísticas para la *maximización de la región de Voronoi.* Como idea de partida se pensó en realizar únicamente las heurísticas *Random Search* y *Simulated Annealing.* Sin embargo, y viendo la velocidad y el buen camino que llevaba el proyecto se decidió ampliar y añadir la heurística *Ant Systems*.

Esta sección fue la más complicada, ya que si bien la programación de las dos primeras técnicas, que pueden estar más orientadas a *diagramas de Voronoi*, el algoritmo *Ant Systems* ha sido una interpretación del original y tardo en pensarse cuál era la mejor opción para poder implementar la estrategia del movimiento de las hormigas y el rastro a la maximización de la *región de Voronoi*.

La complejidad en la programación de la heurística *Simulated Annealing* residió en las estrategias de templado explicadas con anterioridad, es decir, en encontrar los valores más óptimos para que el algoritmo pudiera converger y encontrar la solución más cercana al óptimo.

7.4 CONCLUSIONES

En este proyecto se han presentado diferentes técnicas heurísticas que dan solución al problema de encontrar un nuevo punto *q* dentro de la región *R* tal que el área que genere la nueva *región de Voronoi* se máxima. Las heurísticas utilizadas son *Simulated Annealing, Random Search y Ant Systems.* Como se ha visto en el capítulo anterior los resultados obtenidos con las diferentes técnicas son muy parecidos, es decir, el porcentaje obtenido con los algoritmos se diferencia muy poco por lo que se puede concluir que las tres técnicas son capaces de dar una solución cercana al óptimo. Sin embargo, la diferencia principal radica en el tiempo de cálculo que necesita cada algoritmo para alcanzar una solución, siendo en este apartado la heurística *Simulated Annealing* la técnica más perjudicada ya que el tiempo empleado es notablemente mayor. Esta técnica es capaz de obtener porcentajes de área superiores por lo que en ciertos casos puede ser interesante tener que utilizarla si se quiere encontrar la solución más cercana a la óptima.

Se ha conseguido la realización de todos los objetivos propuestos inicialmente. Se ha creado un software gráfico interactivo que puede ser utilizado por cualquier usuario, ya que el uso del programa es sencillo y aparecen explicaciones a lo largo del programa que explican el correcto funcionamiento de la aplicación. Además, se ha conseguido realizar la parte principal, la razón de realizar este proyecto, como ha sido adaptar diferentes técnicas heurísticas para la *maximización de la región de Voronoi.*

Tras realizar la investigación en la primera fase del proyecto, no existen softwares gráfico-interactivos, que sean actuales, capaces de calcular *diagramas de Voronoi* y posteriormente facilitar al usuario la capacidad de encontrar un nuevo punto *q* en el plano que *maximice la región de Voronoi.*

Según la propia opinión del autor de este proyecto, los *diagramas de Voronoi* son una rama de la geometría computacional con mucho potencial en diferentes áreas de investigación y con numerosas aplicaciones en robótica, inteligencia artificial y un amplio abanico de problemas relacionados con la matemática computacional y con numerosas aplicaciones en

diferentes ámbitos de actualidad. Como se explicó en el Capítulo 3 las ramas de aplicación de esta tecnología van desde la colocación de antenas de telefonía hasta campos como la robótica. Lo que se puede ver es que esta tecnología puede ser usada en diversos y diferentes campos de actuación, por lo que la investigación realizada y el software creado son un avance en la investigación y tienen como objetivo último el facilitar a las personas una forma para ser capaces de dividir el mundo, de hacer particiones basadas en un razonamiento matemático y con una lógica asociada.

7.5 TRABAJOS FUTUROS

Considerando este proyecto una versión inicial que se espera poder seguir desarrollando en un futuro, se proponen una serie de trabajos posibles como continuación de este proyecto realizado.

- Implementación de otra técnica heurística para la *maximización de la región de Voronoi.* En este caso se propone los *algoritmos genéticos.* Estos algoritmos fueron introducidos por Holland [10] y son técnicas metaheurísticas en las que se considera individuos que son soluciones factibles y esta población, conjunto de soluciones, es capaz de reproducirse o incluso mutar. La estructura básica de estos algoritmos es la siguiente:

```
[1]INICIO
[2]Generar población inicial
[3]do{
[4]    Calcula la capacidad de cada individuo
[5]    Selecciona de forma probabilística en base a la aptitud
[6]    Aplicar el cruce entre individuos seleccionados
[7]    Aplicar mutación a un porcentaje de la población inicial
[8]    Generar una nueva población
[9]}while(parada == false)
[10]FIN
```

- Se propone la adaptación de la región al cuadrado unidad. Inicialmente está en coordenadas de píxeles porque los algoritmos se comenzaron a realizar en base a dicho formato, sin embargo, de cara a poder ser usado en diferentes campos que

requieren más precisión en cuanto a posición y distancia parece razonable esta modificación. Además, sería interesante poder adjuntar imágenes como fondo del lienzo, por ejemplo, un mapa de una ciudad en la que se quiere colocar una nueva antena de telefonía y de esta forma poder esbozar una solución inicial sobre cuál sería el sitio óptimo para ubicar dicho dispositivo. Para facilitar la conversión entre unidades de pantalla (píxeles) y coordenadas de usuario (mm) se muestra una regla para realizar el proceso:

Llamando a las coordenadas de la pantalla xmm(milímetros) e ymm(milímetros) las dimensiones de un píxel son las siguientes:

$$\frac{xmm}{xmaxpíxel}; \frac{ymm}{ymaxpíxel}$$

El aspecto es lo que se conoce como la relación entre anchura y altura de un píxel y se puede calcular de la siguiente manera:

$$aspecto = \frac{ymm}{ymaxpíxel} : \frac{xmm}{xmaxpíxel}$$

- De cara a poder adaptar este programa a la rama de las comunicaciones, se propone que el programa sea capaz de obtener la *triangulación de Delaunay* explicada en el Capítulo 3 y que sea capaz de encontrar el camino más corto para enrutar la comunicación utilizando para ello, por ejemplo, el algoritmo de Dijkstra.
- Uno de los objetivos últimos que tenía el desarrollo de esta aplicación era el de poder ayudar a impartir la asignatura de Geometría Computacional en el nuevo grado en Ingeniería Matemática e Inteligencia Artificial (iMAT) de la Escuela Técnica Superior de Ingeniería (ICAI). Es por ello por lo que se sugiere la posibilidad de crear una función que vaya detallando paso a paso la creación de los diagramas e incluso explicando los pasos a seguir por las diferentes heurísticas para calcular la maximización de la *región de Voronoi* y de esta manera obtener una herramienta más pedagógica.

BIBLIOGRAFÍA

[1] Canales Cano, Santiago. "Métodos heurísticos en problemas geométricos. Visibilidad, iluminación y vigilancia", Tesis doctoral, 2004.

[2] Berg de, M.; Kreveld van, M.; Overmars, M.; Schwarzkopf, O.; "Computational Geometry. Algorithms and Applications", Springer, 1997.

[3] Okabe, A; Boots, B; Sugihara, K; Chiu, S.N. "Spacial Tesellations: Concepts and Applications of Voronoi Diagrams", John Wiley & Sons, Chichester, UK, 2000.

[4] Cheong, O; Har-Peled, S; Linial, N; Matousek, J. "The one-round Voronoi game", Proc. 18th Annu. ACMP Symp. on Computational Geometry, 2002.

[5] Ann, H-K; Cheng, S-W; Cheong, O; Golin, M; Oostrum, R. "Competitive facilitylocation along a highway", Proc. 7th Annu. Int. Conf. (COCOON 2001), Lectures Notes Comput. Sci. (2108), 2001.

[6] Dehne, F; Klein, R; Seidel, R. "Maximizing a Voronoi Region: The Convex Case", 2005.

[7] Cheong, O; Har-Peled, S; Linial, N; Matousek, J. "Finding a guard that sees most and a shop that sells most, Discrete Comput Geom, 2007.

[8] Mayoclinic.org Cólera- Síntomas y causas- Mayo Clinic. 26 de febrero 2022.

[9] Koch T. John Snow, hero of cholera: RIP. Can Medí Assoc J. 2008

[10] Holland, J.H.: "Adaptation in Natural and Artificial Systems", University of Michigan Press, 1975.

[11] Kirkpatrick, S; Gelatt, C.D; Vecchi, M.P. Optimization by simulated annealing, Science, 1983.

[12] Szu, H.H; Hartley, R.L. Fast simulated annealing, Physics Letters A, 1987.

[13] Ingber, L. Very fast simulated re-annealing, Mathl. Comput Modelling, 1989.

[14] García, B. Uso del sistema de Colonia de hormigas para optimizar circuitos lógicos combinatorios. Universidad de Veracruz, 2001.

[15] Dorigo, M. Behaviour of Real Ants. IRIDIA, Université Libre de Bruxelles, Belgium.

[16] Maldonado, C.E. Un problema fundamental en la investigación: Los problemas P vs NP. Revista Logos Ciencia, 2013.

ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS

Los ODS (Objetivos de desarrollo Sostenible) fueron establecidos en 2015 por la Asamblea General de las Naciones Unidas y tienen como objetivo cumplir con los 17 objetivos para el año 2030. Los objetivos que se propusieron son los siguientes:

- Fin de la pobreza
- Hambre cero
- Salud y bienestar
- Educación de calidad
- Igualdad de género
- Agua limpia y saneamiento
- Energía asequible y no contaminante
- Trabajo decente y crecimiento económico
- Industria, innovación e infraestructura
- Reducción de las desigualdades
- Ciudades y comunidades sostenibles
- Producción y consumo responsables
- Acción por el clima
- Vida submarina
- Vida de ecosistemas terrestres
- Paz, justicia e instituciones sólidas
- Alianzas para los objetivos

Debido al gran número de aplicaciones que tienen los diagramas de Voronoi se puede ver que cumple diferentes objetivos para los ODS.

Como se ha explicado previamente, los *diagramas de Voronoi* permiten hacer particiones de una determinada región de una forma razonada y siguiendo un criterio matemático concreto. Una de las capacidades que tienen los *diagramas de Voronoi* es el de ser capaz de colocar centros de salud u hospitales y organizar el área que debe ser capaz de manejar cada uno de dichos puntos de atención sanitaria. De esta forma se podría organizar la atención sanitaria poniendo como centros para los diagramas los núcleos de población que no tienen acceso a la sanidad.

Como se ha mencionado, la división del espacio es una de las características principales de estos diagramas, es por ello, que, en zonas de conflicto por territorios, se podría realizar una partición de este utilizando *diagramas de Voronoi* y así poder facilitar la paz y la justicia en todos los territorios.

Uno de los objetivos que tiene este proyecto es el de ser didáctico y poder llevar a todas las personas una rama de la geometría computacional que no es muy conocida hoy en día. Es por ello, que se ha creado con la intención de proveer una herramienta que sea capaz de proveer una mejor calidad de la educación.

Este proyecto también propone nuevas ideas para realizar la comunicación, nuevas formas para la colocación de infraestructuras para las comunicaciones y de esa forma mejorar la eficiencia de éstas. Es por ello por lo que también promueve el principio de industria, innovación e infraestructura.

Los ODS deberían ser tenidos en cuenta a la hora de realizar cualquier proyecto, ya que uno de los objetivos que debería tener cualquier persona que realice un trabajo debería ser el del crecimiento personal y global, así como la mejora en sociedad, respetando el planeta y a las personas que forman parte del mismo.

ANEXO II: CÓDIGO

Programa principal, (Voronoi.py)

```
import pygame
import sys
import numpy
from funciones_gc import *
from datetime import datetime
from popup import cuadro
from popup import popNadaGuardar
from popup import popNadaIntroducido
from popup import seleccionarAlgoritmo
from popup import puntosvoraleatorio
from popup import posicionNoDisponible
from popup import noHayptosEnElLienzo
from popup import maximizacionCalculada
import time
from manejo_archivos import save_file
from manejo_archivos import open_file

#Inicializamos pygame
pygame.init()

#Lista de puntos original
p=[]
#lista puntos comparación
m=[]
#Para mover el punto cuando lo situamos encima de otro
l=1
#Inicializamos región para random search
RS=[]
#Iniciamos región para simulated annealing
SA=[]
#Inicializamos región para ant systems
ANT=[]
#Inicializamos región para diagrama de voronoi
V=[]

#Inicializamos la variable area_total
area_total=400*500

#Numero aleatorio para sa
def aleatorio_sa():
    a = numpy.random.uniform(0,500)
    b = numpy.random.uniform(0,400)

    punto_inicial = [a,b]
    return punto_inicial

#Función para generar puntos aleatorios
def aleatorio(numero):
    m =[]
    #bucle para generar números aleatorios y crear puntos
    for i in range (numero):
```

```
        a = numpy.random.uniform(2,498)
        b = numpy.random.uniform(2,398)
        nuevo = (a,b)
        m.append(nuevo)

    #bucle para mostrar los puntos aleatorios por pantalla
    for k in range(len(m)):
        pygame.draw.circle(ventana,(0,0,255),m[k],1)
        pygame.display.flip()
    return m

#Función para almacenar los puntos que vamos escribiendo por pantalla
def almacenar():
    global l, p
    raton = pygame.mouse.get_pos() #guardamos la posición del ratón
    if(raton[1]>400):
        posicionNoDisponible()
        print('Posición del punto no disponible')
    elif(raton[0]>500):
        posicionNoDisponible()
        print('Posición del punto no disponible')
    else:
        #si el punto ya estaba en la lista de puntos, hacemos un ligero cambio
        # para desplazarlo y que no moleste el solapamiento
        if raton in p:
            p.append((raton[0]+l,raton[1]+l)) #añadimos el punto desplazado a la
lista
            l=l*(-1)
            raton=(raton[0]+1,raton[1]+1)
            pygame.draw.circle(ventana,(0,0,0),raton,3) #pintamos el punto
            pygame.display.flip() #mostramos el lienzo con el punto
        else:
            pygame.draw.circle(ventana,(0,0,0),raton,3) #pintamos el punto
            pygame.display.flip() #mostramos el lienzo con el punto
            #Si no esta en la lista añadimos el punto directamente
            p.append(raton)

#Función encargada de mostrar botones en la parte inferior
def mostrar_Botones(a,b,c,e,g,h,z):
    #Rectángulo que sirve de contenedor horizontal
    pygame.draw.rect(ventana, (240,240,240),contenedor)

    #Rectángulo que sirve de contenedor vertical
    pygame.draw.rect(ventana,(240,240,240),contenedor_vertical)

    #Botón de limpiar nuestro lienzo y el vector de puntos
    ventana.blit(clear_cadena, (clear_rect.centerx-30, clear_rect.centery-10))
    pygame.draw.rect(ventana,a,clear_rect,2)

    #botón para calcular la región de Voronoi
    ventana.blit(calcular_cadena, (calcular_rect.centerx-30,
calcular_rect.centery-10))
    pygame.draw.rect(ventana,b,calcular_rect,2)

    #botón para guardar un archivo
    ventana.blit(guardar_Archivo_Cadena, (302,guardar_Archivo_rect.centery-20))
    ventana.blit(guardar2_Archivo_Cadena, (303,guardar_Archivo_rect.centery-5))
    pygame.draw.rect(ventana,g,guardar_Archivo_rect,2)

    #botón para cargar un archivo
    ventana.blit(cargar_Archivo_Cadena, (422,cargar_Archivo_rect.centery-20))
```

```
    ventana.blit(cargar2_Archivo_Cadena, (420,cargar_Archivo_rect.centery-5))
    pygame.draw.rect(ventana,h,cargar_Archivo_rect,2)

    #botón para calcular el punto de maximización de Voronoi con Random search
    ventana.blit(algoritmos_cadena, (algoritmos_rect.centerx-42,
algoritmos_rect.centery-10))
    pygame.draw.rect(ventana,c,algoritmos_rect,2)

    #título del contenedor de los algoritmos de maximización
    ventana.blit(Titulo_cadena, (515,15))

    #botón para borrar la región de maximización
    ventana.blit(vor_aleatorio_cadena, (175,vor_aleatorio_rect.centery-20))
    ventana.blit(vor_aleatorio2_cadena, (170,vor_aleatorio_rect.centery-5))
    pygame.draw.rect(ventana,e,vor_aleatorio_rect,2)

    #botón para borrar la región de maximización
    ventana.blit(borrar_maximizacion_cadena,
(550,borrar_maximizacion_rect.centery-20))
    ventana.blit(maximizacion_cadena, (520,borrar_maximizacion_rect.centery-5))
    pygame.draw.rect(ventana,z,borrar_maximizacion_rect,2)

    #Se actualiza lo que se muestra por pantalla
    pygame.display.update()

def mostrar_numero_puntos(numero):
    ventana.blit(numero_puntos,(520,335))
    cadena_para_mostrar = font.render(str(numero)+" ptos",True,(255,0,0))
    ventana.blit(cadena_para_mostrar,(590,335))

    #Se actualiza lo que se muestra por pantalla
    pygame.display.update()

#tamaño de la ventana.
ventana = pygame.display.set_mode((650,500))

#titulo de la ventana
pygame.display.set_caption("Región de Voronoi")

#Icono de la ventana
pygame_icon = pygame.image.load("math_icon.png")
pygame.display.set_icon(pygame_icon)

#contendor para almacenar los botones
contenedor=pygame.Rect(0,400,500,100)
pygame.draw.rect(ventana, (240,240,240),contenedor)

#contenedor vertical para almacenar botones
contenedor_vertical=pygame.Rect(500,0,150,500)
pygame.draw.rect(ventana,(240,240,240),contenedor_vertical)

#fuente utilizada en las letras
font = pygame.font.Font('C:\Windows\Fonts\Arial.ttf', 18)

#Botón para limpiar el lienzo
clear_cadena=font.render('  clear', True, (0, 0, 0))
clear_rect=pygame.Rect(525,440,100,50)

#Botón para calcular la región de Voronoi
calcular_cadena = font.render('Voronoi',True, (0,0,0))
calcular_rect = pygame.Rect(20,440,100,50)
```

```
#Botón para borrar la maximización
vor_aleatorio_cadena = font.render('Voronoi',True, (0,0,0))
vor_aleatorio2_cadena = font.render('aleatorio',True, (0,0,0))
vor_aleatorio_rect = pygame.Rect(138,440,130,50)

#Botón para borrar la maximización
borrar_maximizacion_cadena = font.render('Borrar',True, (0,0,0))
maximizacion_cadena = font.render('maximización',True, (0,0,0))
borrar_maximizacion_rect = pygame.Rect(512,370,130,50)

#Boton para guardar archivos
guardar_Archivo_Cadena = font.render('Guardar',True, (0,0,0))
guardar2_Archivo_Cadena = font.render('Voronoi',True, (0,0,0))
guardar_Archivo_rect = pygame.Rect(286,440,100,50)

#Boton para cargar archivos
cargar_Archivo_Cadena = font.render('Cargar',True, (0,0,0))
cargar2_Archivo_Cadena = font.render('Voronoi',True, (0,0,0))
cargar_Archivo_rect = pygame.Rect(404,440,100,50)

#Botón para elegir algoritmo
algoritmos_cadena = font.render('Algoritmos',True, (0,0,0))
algoritmos_rect = pygame.Rect(520,50,120,50)

#fuente utilizada en las letras del titulo
font = pygame.font.Font('C:\Windows\Fonts\Arial.ttf', 15)
font.set_bold(1)
font.set_underline(1)

#Letras de "Algoritmo maximización"
Titulo_cadena = font.render('Maximización RV',True, (0,0,0))

#Letras sin subrayar
font.set_underline(0)

#Letras resultados
resultados_cadena = font.render('Resultados',True, (0,0,0))

#Letra sin negrita
font.set_bold(0)

#Cadena para mostrar por pantalla el área con RS
cadena_area_RS = font.render('Área obtenida RS: ',True,(0,0,0))

#Cadena para mostrar por pantalla el área con SA
cadena_area_SA = font.render('Área obtenida SA: ',True,(0,0,0))

#Cadena para mostrar por pantalla el área con ANT
cadena_area_ANT = font.render('Área obtenida ANT: ',True,(0,0,0))

#Cadena de tiempo
tiempo_cadena = font.render('Tiempo tardado:',True,(0,0,0))

#Cadena de número de puntos
numero_puntos = font.render('N° puntos:',True,(0,0,0))

ventana.fill((255,255,255)) #lienzo en blanco
pygame.display.flip()#Se muestra la ventana

#Codigo de color para los botones
```

```
a=(0,0,0)#clear
b=(0,0,0)#calcular Voronoi
c=(0,0,0)#algoritmos de maximización
e=(0,0,0)#voronoi aleatorio
g=(0,0,0)#guardar archivo
h=(0,0,0)#cargar archivo
z=(0,0,0)#Borrar maximización
mostrar_Botones(a,b,c,e,g,h,z)
mostrar_numero_puntos(len(p))

#Función para mostrar el algoritmo de random search
def mostrar_random_search(posicion,m):
    global p
    global RS
    ventana.fill((255,255,255)) #limpiamos nuestra ventana
    p.append(m[posicion]) #añadimos a nuestro vector de puntos el nuevo punto que
                          # maximiza la región de Voronoi
    for k in range(len(p)):
        RS = calculVoronoi_sinPintar()
    area = Area2(RS[len(p)-1])
    for k in range(len(p)-1): #recorremos nuestro vector de puntos original,
                              # sin tener en cuenta el punto aleatorio añadido al
                              # final ya que se representará
                              # posteriormente
        pygame.draw.polygon(ventana,(205,205,205),RS[k]) #representación de la
                                                         # región de voronoi, el
                                                         # fondo del
                                                         # polígono
        pygame.draw.circle(ventana,(0,0,0),p[k],4) #representación del punto
        pygame.draw.polygon(ventana,(0,0,0),RS[k],2) #representación de la
                                                     # región de voronoi,
                                                     # las aristas del
                                                     # polígono

    pygame.draw.polygon(ventana,(204,255,0),RS[len(p)-1]) #representación de la
                                                          #región de voronoi,
                                                          #el fondo del polígono
                                                          # asociado a la región
                                                          # maximizada
    pygame.draw.circle(ventana,(255,0,0),m[posicion],6) #representación del punto
                                                        #aleatorio que maximiza
                                                        # la región de voronoi
    pygame.draw.polygon(ventana,(0,0,0),RS[len(p)-1],2) #representación de la
                                                        #región de voronoi,
                                                        # las aristas del
                                                        # polígono asociado a la
                                                        # región maximizada

    for i in range(len(m)): #recorremos el vector de puntos aleatorios para
                            #representarlos en nuestro lienzo
        pygame.draw.circle(ventana,(0,0,255),m[i],1)

    porcentaje_area = (area/area_total)*100
    return porcentaje_area

#Función que nos permite mostrar el algorimto de simulated annealing, tanto la
región y el punto de maximización como los puntos evaluados
def mostrar_sa(punto,puntos_evaluados):
    global SA
    SA=[]
    global p
```

```
    ventana.fill((255,255,255)) #limpiamos nuestra ventana
    p.append(punto) #añadimos a nuestro vector de puntos el nuevo punto que
                    #maximiza la región de Voronoi
    for k in range(len(p)):
        SA = calculVoronoi_sinPintar()#Calculamos la región de Voronoi con el
                                      #punto de maximización
    area = Area2(SA[len(p)-1])#Calculamos el área de la región "ganadora"
    for k in range(len(p)-1): #recorremos nuestro vector de puntos original, sin
                              #tener en cuenta el punto aleatorio añadido al
                              # final ya que se representará posteriormente
        pygame.draw.polygon(ventana,(205,205,205),SA[k]) #representación de la
                                                         #región de voronoi, el
                                                         # fondo del polígono
        pygame.draw.circle(ventana,(0,0,0),p[k],4) #representación del punto
        pygame.draw.polygon(ventana,(0,0,0),SA[k],2) #representación de la región
                                                     #de voronoi, las aristas del
                                                     # polígono
    pygame.draw.polygon(ventana,(204,255,0),SA[len(p)-1]) #representación de la
                                                          #región de voronoi, el
                                                          # fondo del polígono
                                                          #  asociado a la región
                                                          # maximizada
    pygame.draw.polygon(ventana,(0,0,0),SA[len(p)-1],2) #representación de la
                                                        #región de voronoi, las
                                                        # aristas del polígono
                                                        # asociado a la región
                                                        # maximizada

    for i in range(len(puntos_evaluados)): #recorremos el vector de puntos
                                           # evaluados por el algoritmo
                                           # para representarlos en
                                           # nuestro lienzo
        pygame.draw.circle(ventana,(0,0,255),puntos_evaluados[i],1)

    pygame.draw.circle(ventana,(255,0,0),punto,4) #representación del punto
                                                  #aleatorio que maximiza
                                                  # la región de voronoi

    porcentaje_area = (area/area_total)*100 calculamos el porcentaje que
                                            #representa el área
                                            # maximizada
    return porcentaje_area

def mostrar_ant(puntos_mirados,punto):
    global ANT
    ANT = []
    global p
    ventana.fill((255,255,255))
    p.append(punto)
    for k in range(len(p)):
        ANT = calculVoronoi_sinPintar()#Calculamos la región de Voronoi con el
                                       #punto de maximización
    area = Area2(ANT[len(p)-1])#Calculamos el área de la región "ganadora"

    for k in range(len(p)-1): #recorremos nuestro vector de puntos original, sin
                              #tener en cuenta el punto aleatorio añadido al
                              # final ya que se representará posteriormente
        pygame.draw.polygon(ventana,(205,205,205),ANT[k]) #representación de la
                                                          #región de voronoi, el
                                                          # fondo del polígono
        pygame.draw.circle(ventana,(0,0,0),p[k],4) #representación del punto
```

```
        pygame.draw.polygon(ventana,(0,0,0),ANT[k],2) #representación de la
                                                      #región de voronoi,
                                                      #las aristas del
                                                      # polígono

    pygame.draw.polygon(ventana,(204,255,0),ANT[len(p)-1]) #representación de la
                                                           #región de voronoi, el
                                                           # fondo del polígono
                                                           # asociado a la región
                                                           # maximizada
    pygame.draw.polygon(ventana,(0,0,0),ANT[len(p)-1],2) #representación de la
                                                         #región de voronoi, las
                                                         # aristas del polígono
                                                         # asociado a la región
                                                         # maximizada

    for i in range(len(puntos_mirados)): #recorremos el vector de puntos
                                         #evaluados por el algoritmo para
                                         # representarlos en nuestro lienzo
        pygame.draw.circle(ventana,(0,0,255),puntos_mirados[i],1)

    pygame.draw.circle(ventana,(255,0,0),punto,6) #representación del punto
                                                  #aleatorio que maximiza la
                                                  # región de voronoi

    porcentaje_area = (area/area_total)*100
    return porcentaje_area

#Función que nos permite representar los puntos y el diagrama de voronoi desde un
archivo que almacena puntos
def representar_archivo(puntos):
    global p
    p=[]
    p=puntos
    V=[] #Inicialización del vector que contendrá los polígonos

    if(len(p) == 0):#Si no hay puntos, no representa nada
        return
    else:
        #Bucle para ir calculando la región de Voronoi con el
        #algoritmo incremental
        for k in range(len(p)):
            #llamada a la función de voronoiRegion
            V.append(voronoiRegion(p,k))
            pygame.draw.polygon(ventana,(205,205,205),V[k]) #representación de la
                                                            #región de voronoi,
                                                            # fondo del polígono
            pygame.draw.circle(ventana,(0,0,0),p[k],3) #representación del punto
            pygame.draw.polygon(ventana,(0,0,0),V[k],2) #representación de la
                                                        #región de voronoi,
                                                        # las aristas del
                                                        # polígono
#Función que borra todo lo asociado a la maximización, tanto los puntos evaluados
como la región maximizada
def borrar_maximizacion():
    global RS
    global p
    global V
    global SA
    global ANT
    V=[]
```

```
    if(RS!=[]):#Borra la maximizacion con random search
        if(len(p) == 0):
            ventana.fill((255,255,255)) #lienzo en blanco
            a=(0,0,0)
            b=(0,0,0)
            c=(0,0,0)
            e=(0,0,0)
            g=(0,0,0)
            h=(0,0,0)
            z=(0,0,0)
            mostrar_Botones(a,b,c,e,g,h,z)
            pygame.display.flip()#Se muestra la ventana
            RS=[]
        else:
            RS=[]
            p.pop()
            #Bucle para ir calculando la region de Voronoi con el
            #algoritmo incremental
            for k in range(len(p)):
                #llamada a la funcion de voronoiRegion
                V.append(voronoiRegion(p,k))
                pygame.draw.polygon(ventana,(205,205,205),V[k]) #representación
                                                                #de la región
                                                                # de voronoi,
                                                                #el fondo del
                                                                # polígono
                pygame.draw.circle(ventana,(0,0,0),p[k],3) #representación del
                                                           #punto
                pygame.draw.polygon(ventana,(0,0,0),V[k],2) #representación de la
                                                            #región de voronoi,
                                                            # las aristas del
                                                            # polígono
            return
    elif(SA!=[]): #borra la maximización con simulated annealing
        if(len(p) == 0):
            ventana.fill((255,255,255)) #lienzo en blanco
            a=(0,0,0)
            b=(0,0,0)
            c=(0,0,0)
            e=(0,0,0)
            g=(0,0,0)
            h=(0,0,0)
            z=(0,0,0)
            mostrar_Botones(a,b,c,e,g,h,z)
            pygame.display.flip()#Se muestra la ventana
            SA=[]
        else:
            SA=[]
            p.pop()
            #Bucle para ir calculando la región de Voronoi con el algoritmo
            # incremental
            for k in range(len(p)):
                #llamada a la función de voronoiRegion
                V.append(voronoiRegion(p,k))
                pygame.draw.polygon(ventana,(205,205,205),V[k]) #representación
                                                                #de la región de
                                                                # voronoi, el
                                                                # fondo del
                                                                # polígono
                pygame.draw.circle(ventana,(0,0,0),p[k],3) #representación del
                                                           #punto
```

```
                pygame.draw.polygon(ventana,(0,0,0),V[k],2) #representación de la
                                                            #región de voronoi,
                                                            # las aristas del
                                                            # polígono
            return
        return
    elif(ANT!=[]): #borra la maximizacion con simulated annealing
        if(len(p) == 0):
            ventana.fill((255,255,255)) #lienzo en blanco
            a=(0,0,0)
            b=(0,0,0)
            c=(0,0,0)
            e=(0,0,0)
            g=(0,0,0)
            h=(0,0,0)
            z=(0,0,0)
            mostrar_Botones(a,b,c,e,g,h,z)
            pygame.display.flip()#Se muestra la ventana
            ANT=[]
        else:
            ANT=[]
            p.pop()
            #Bucle para ir calculando la región de Voronoi con el
            # algoritmo incremental
            for k in range(len(p)):
                #llamada a la funcion de voronoiRegion
                V.append(voronoiRegion(p,k))
                pygame.draw.polygon(ventana,(205,205,205),V[k]) #representación
                                                                #de la región
                                                                #de voronoi,
                                                                # el fondo del
                                                                # polígono
                pygame.draw.circle(ventana,(0,0,0),p[k],3) #representación del
                                                           #punto
                pygame.draw.polygon(ventana,(0,0,0),V[k],2) #representación de la
                                                            #región de voronoi,
                                                            # las aristas del
                                                            # polígono
            return
        return

#Función que nos permite calcular el diagrama de Voronoi pero sin necesidad de
representarlo
def calculVoronoi_sinPintar():
    global l,p #variables globales inicializadas previamente
    Voronoi_inicial=[] #Inicialización del vector que contendrá los polígonos

    if(len(p) == 0):
        return
    else:
        #Bucle para ir calculando la región de Voronoi con el
        # algoritmo incremental
        for k in range(len(p)):
            #llamada a la función de voronoiRegion
            Voronoi_inicial.append(voronoiRegion(p,k))
    return Voronoi_inicial

#Función que calcula y representa el diagrama de Voronoi
def calculaVoronoi():
    global l,p,V #variables globales inicializadas previamente
    V=[] #Inicialización del vector que contendrá los polígonos
```

```
    global ANT,SA,RS
    ANT = []
    SA=[]
    RS=[]

    if(len(p) == 0):
        return
    else:
        #Bucle para ir calculando la región de Voronoi con el
        # algoritmo incremental
        for k in range(len(p)):
            #llamada a la función de voronoiRegion
            V.append(voronoiRegion(p,k))
            pygame.draw.polygon(ventana,(205,205,205),V[k]) #representación de la
                                                            #región de voronoi,
                                                            # el fondo del
                                                            # polígono
            pygame.draw.circle(ventana,(0,0,0),p[k],3) #representación del punto
            pygame.draw.polygon(ventana,(0,0,0),V[k],2) #representación de la
                                                        #región de voronoi, las
                                                        # aristas del polígono
        return

#Función para crear un diagrama de voronoi aleatorio de número de puntos elegido
por el usuario
def random_voronoi(numero):
    global p
    global V
    p=[]
    V=[]
    for i in range(numero):
        aleatoriox = numpy.random.uniform(0,500)
        aleatorioy = numpy.random.uniform(0,400)
        nuevo = [aleatoriox,aleatorioy]
        p.append(nuevo)

    #Bucle para ir calculando la región de Voronoi con el algoritmo incremental
    for k in range(len(p)):
        #llamada a la funcion de voronoiRegion
        V.append(voronoiRegion(p,k))
        pygame.draw.polygon(ventana,(205,205,205),V[k]) #representación de la
                                                        #región de voronoi,
                                                        # el fondo del
                                                        # polígono
        pygame.draw.circle(ventana,(0,0,0),p[k],3) #representación del punto
        pygame.draw.polygon(ventana,(0,0,0),V[k],2) #representación de la región
                                                    #de voronoi, las aristas del
                                                    # polígono

def inicio():
    global p
    #Bucle de control del juego
    while True:
        for event in pygame.event.get():
            #para cada evento que ocurre ejecuta una de las acciones de debajo
            if event.type == pygame.QUIT:
                #si se pulsa cerrar la ventana
                pygame.quit() #cierra pygame
                sys.exit() #cierra el programa
            elif event.type == pygame.MOUSEBUTTONDOWN:
                #si se pulsa con el ratón en la pantalla, menos donde cerrar
```

```
                raton=pygame.mouse.get_pos() #Obtener posición del ratón
                if calcular_rect.collidepoint(raton):
                    #Codigo de color para los botones
                    a=(0,0,0)
                    b=(0,255,0)
                    c=(0,0,0)
                    e=(0,0,0)
                    g=(0,0,0)
                    h=(0,0,0)
                    z=(0,0,0)
                    mostrar_Botones(a,b,c,e,g,h,z)
                    mostrar_numero_puntos(len(p))
                    #si la posición del ratón coincide con el botón de calcular
                    # llama a la función de calcular Voronoi
                    calculaVoronoi()
                    pygame.display.flip()
                elif vor_aleatorio_rect.collidepoint(raton):
                    a=(0,0,0)
                    b=(0,0,0)
                    c=(0,0,0)
                    e=(0,255,0)
                    g=(0,0,0)
                    h=(0,0,0)
                    z=(0,0,0)
                    mostrar_Botones(a,b,c,e,g,h,z)
                    mostrar_numero_puntos(len(p))
                    numero = puntosvoraleatorio()
                    if(type(numero) == int):
                        random_voronoi(numero)
                    else:
                        popNadaIntroducido()
                    mostrar_Botones(a,b,c,e,g,h,z)
                    mostrar_numero_puntos(len(p))
                    pygame.display.flip()
                elif clear_rect.collidepoint(raton):
                    #si la posición del ratón coincide con el botón de clear
                    p=[] #limpiar el vector de puntos cuando limpiamos el lienzo
                    m=[] #limpiar el vector de puntos aleatorios para rs
                    ventana.fill((255,255,255)) #mostrar el lienzo en blanco
                    #Código de color para los botones
                    a=(0,255,0)
                    b=(0,0,0)
                    c=(0,0,0)
                    e=(0,0,0)
                    g=(0,0,0)
                    h=(0,0,0)
                    z=(0,0,0)
                    mostrar_Botones(a,b,c,e,g,h,z)
                    mostrar_numero_puntos(len(p))
                    pygame.display.flip() #mostrar la pantalla refrescada
                elif borrar_maximizacion_rect.collidepoint(raton):
                    #Código de color para los botones
                    borrar_maximizacion()
                    a=(0,0,0)
                    b=(0,0,0)
                    c=(0,0,0)
                    e=(0,0,0)
                    g=(0,0,0)
                    h=(0,0,0)
                    z=(0,255,0)
                    mostrar_Botones(a,b,c,e,g,h,z)
```

```
                    mostrar_numero_puntos(len(p))
                elif algoritmos_rect.collidepoint(raton): #botón que implementa
                                                          #la funcionalidad del
                                                          # algoritmo random
                                                          # search
                    #Código de color para los botones
                    a=(0,0,0)
                    b=(0,0,0)
                    c=(0,255,0)
                    e=(0,0,0)
                    g=(0,0,0)
                    h=(0,0,0)
                    z=(0,0,0)
                    mostrar_Botones(a,b,c,e,g,h,z)
                    mostrar_numero_puntos(len(p))
                    algoritmo = seleccionarAlgoritmo()
                    if algoritmo == 'Random Search':
                        if(p==[]):
                            noHayptosEnElLienzo()
                        else:
                            numero = cuadro() #esta función recoge el número de
                                              #puntos aleatorios deseado por el
                                              # usuario
                            if(type(numero) == int):
                                inicio = datetime.now()
                                m = aleatorio(numero) #Generamos un vector de
                                                      #numeros aleatorios
                                voronoi = calculVoronoi_sinPintar()#Calculamos el
                                                                   #diagrama de
                                                                   # Voronoi
                                                                   # inicial
                                #LLamamos a la función que implementa el
                                # algoritmo de
                                # random search

                                posicion = random_search_funcion(m,voronoi,p)
                                final = datetime.now()
                                time.sleep(0.5)
                                #Mostramos la solución grafica del algoritmo de
                                # random search
                                porcentaje = mostrar_random_search(posicion,m)
                                if(len(p)==1):
                                    porcentaje = 2*porcentaje
                                #redondeamos el valor del porcentaje de área que
                                # genera la nueva región para que únicamente
                                # tenga dos decimales
                                porcentaje = round(porcentaje,2)

                                mostrar_Botones(a,b,c,e,g,h,z)
                                mostrar_numero_puntos(len(p))
                                maximizacionCalculada()
                                ventana.blit(resultados_cadena,(532,125))
                                #Lugar que queremos que ocupe la cadena
                                ventana.blit(cadena_area_RS,(517,145))
                                #Cadena para mostrar por pantalla el area
                                cadena_porcentaje =
font.render(str(porcentaje)+'%',True,(255,0,0))
                                #Lugar que queremos que ocupe la cadena
```

```
                                ventana.blit(cadena_porcentaje,(545,165))
                                #Cadena para el tiempo
                                ventana.blit(tiempo_cadena,(522,185))
                                tiempo_total = final - inicio
                                segundos = tiempo_total.seconds
                                #Cadena para mostrar por pantalla el tiempo
                                tiempo_segundos_cadena =
font.render(str(segundos)+'s',True,(255,0,0))
                                #Lugar que queremos que ocupe la cadena
                                ventana.blit(tiempo_segundos_cadena,(550,205))
                                pygame.display.flip()
                            else:
                                #popup para indicar que no se ha introducido nada
                                popNadaIntroducido()
                                print("No se ha introducido nada")
                    elif algoritmo == 'Simulated Annealing':
                        if(p==[]):
                            noHayptosEnElLienzo()
                        else:
                            inicio = datetime.now()
                            #Generamos un numero aleatorio
                            punto_inicial = aleatorio_sa()
                            #Calculamos el diagrama de voronoi de los
                            # puntos iniciales
                            voronoi = calculVoronoi_sinPintar()
                            #Llamada a la funcion que implementa el
                            # algoritmo sa
                            punto,puntos_evaluados =
simulated_annealing(p,punto_inicial,voronoi)
                            final = datetime.now()
                            print(len(puntos_evaluados))
                            time.sleep(0.5)
                            #Funcion que muestra el algoritmo
                            # simulated annealing
                            area = mostrar_sa(punto,puntos_evaluados)
                            if(len(p)==1):
                                area = 2*area
                            #redondeamos el valor del porcentaje del area que
                            # ocupa la region
                            area = round(area,2)
                            mostrar_Botones(a,b,c,e,g,h,z)
                            mostrar_numero_puntos(len(p))
                            #maximizacionCalculada()
                            #Mostramos en la ventana el valor de dicho area
                            maximizacionCalculada()
                            ventana.blit(resultados_cadena,(532,125))
                            ventana.blit(cadena_area_SA,(517,145))
                            cadena_porcentaje_sa =
font.render(str(area)+'%',True,(255,0,0))
                            ventana.blit(cadena_porcentaje_sa,(545,165))
                            #Cadena para el tiempo
                            ventana.blit(tiempo_cadena,(522,185))
                            tiempo_total = final - inicio
                            segundos = tiempo_total.seconds
```

```
                            #Cadena para mostrar por pantalla el tiempo
                            tiempo_segundos_cadena =
font.render(str(segundos)+'s',True,(255,0,0))
                            #Lugar que queremos que ocupe la cadena
                            ventana.blit(tiempo_segundos_cadena,(550,205))

                            pygame.display.flip()
                    elif algoritmo == 'Ant Systems':
                        if(p==[]):
                            noHayptosEnElLienzo()
                        else:
                            ventana.blit(resultados_cadena,(532,125))
                            inicio = datetime.now()
                            voronoi = calculVoronoi_sinPintar()
                            puntos_buenos,puntos_mirados,posicion_bueno =
ant_systems_funcion(p,voronoi)
                            punto = puntos_buenos[posicion_bueno]
                            final = datetime.now()
                            time.sleep(0.5)
                            area_ANT = mostrar_ant(puntos_mirados,punto)
                            if(len(p)==1):
                                area_ANT = 2*area_ANT
                            #redondeamos el valor del porcentaje del
                            # area que ocupa la region
                            area_ANT = round(area_ANT,2)
                            mostrar_Botones(a,b,c,e,g,h,z)
                            mostrar_numero_puntos(len(p))
                            #Mostramos en la ventana el valor de dicho area
                            maximizacionCalculada()
                            ventana.blit(resultados_cadena,(532,125))
                            ventana.blit(cadena_area_ANT,(517,145))
                            cadena_porcentaje_sa =
font.render(str(area_ANT)+'%',True,(255,0,0))
                            ventana.blit(cadena_porcentaje_sa,(545,165))
                            #Cadena para el tiempo
                            ventana.blit(tiempo_cadena,(522,185))
                            tiempo_total = final - inicio
                            segundos = tiempo_total.seconds
                            #Cadena para mostrar por pantalla el tiempo
                            tiempo_segundos_cadena =
font.render(str(segundos)+'s',True,(255,0,0))
                            #Lugar que queremos que ocupe la cadena
                            ventana.blit(tiempo_segundos_cadena,(550,205))
                            pygame.display.flip()

                            pygame.display.flip()
                    else:
                        print('Nada')
                elif guardar_Archivo_rect.collidepoint(raton):
                    #Codigo de color para los botones
                    a=(0,0,0)
                    b=(0,0,0)
                    c=(0,0,0)
                    e=(0,0,0)
```

```
                    g=(0,255,0)
                    h=(0,0,0)
                    z=(0,0,0)
                    mostrar_Botones(a,b,c,e,g,h,z)
                    mostrar_numero_puntos(len(p))
                    if(p!=[]):
                        #Llamada a la funcion para almacenar los puntos
                        # que hay sobre el diagrama
                        save_file(p)
                    else:
                        #pop up que indica que no hay puntos qu eguardar
                        popNadaGuardar()
                        print("No hay nada que guardar")
                elif cargar_Archivo_rect.collidepoint(raton):
                    #Codigo de color para los botones
                    a=(0,0,0)
                    b=(0,0,0)
                    c=(0,0,0)
                    e=(0,0,0)
                    g=(0,0,0)
                    h=(0,255,0)
                    z=(0,0,0)
                    mostrar_Botones(a,b,c,e,g,h,z)
                    mostrar_numero_puntos(len(p))
                    #LLamada a la función que devuelve los puntos
                    # desde un archivo
                    puntos=open_file()
                    #Llamada a la función que representa los
                    # puntos del archivo
                    representar_archivo(puntos)
                    mostrar_Botones(a,b,c,e,g,h,z)
                    mostrar_numero_puntos(len(p))
                else:
                    #Codigo de color para los botones
                    a=(0,0,0)#clear
                    b=(0,0,0)#calcular Voronoi
                    c=(0,0,0)#algoritmos de maximización
                    e=(0,0,0)#generar voronoi aleatorio
                    g=(0,0,0)#guardar archivo
                    h=(0,0,0)#cargar archivo
                    z=(0,0,0)#Borrar maximizacion
                    mostrar_Botones(a,b,c,e,g,h,z)
                    #En cualquier otro caso llama a la función
                    # para almacenar los puntos
                    almacenar()
                    mostrar_numero_puntos(len(p))

if __name__=='__main__':
    inicio()
```

Funciones Geometría Computacional, (funciones_gc.py)

```
from scipy import rand
import scipy.spatial as sp
import sys
from numpy.linalg import det
import numpy

div = 3.5

#Funcion que nos genera un punto vecino para el algoritmo de
# simulated annealing
def generar_vecindad(punto):
    global div

    while True:
        #generamos dos numeros aleatorios entre 0 y 1
        aleatorio1 = numpy.random.uniform(0,1)
        aleatorio2 = numpy.random.uniform(0,1)

        #Calculamos los sumandos para los nuevos puntos x e y
        sumando1 = (numpy.sqrt(-
2*numpy.log(aleatorio1))*numpy.sin(2*numpy.pi*aleatorio2))
        sumando2 = (numpy.sqrt(-
2*numpy.log(aleatorio1))*numpy.cos(2*numpy.pi*aleatorio2))

        #Calculamos las coordenadas de nuestro punto vecino
        x = punto[0]/500 + sumando1/div
        y = punto[1]/400 + sumando2/div

        #repetimos este bucle mientras que nos salgamos del cuadrado unidad
        if((x>=0)and(y>=0)and(x<=1)and(y<=1)):
            break

    #Volvemos a coordenadas de pantalla (pixels)
    x = x*500
    y = y*400
    vecino = [x,y] #Creamos el punto vecino y lo devolvemos
    return vecino

#Función que implementa el algorimto de simulated annealing
def simulated_annealing(lista,punto,voronoi):
    global div
    #Establecemo los parametros iniciales y finales de nuestro algoritmo
    temperatura_final = 0.005
    temperatura_inicial = 2*len(lista)

    if(len(lista) <7):
        div=2
    elif(len(lista) > 20):
        temperatura_final = 0.025
    n = 0
```

```
    k = 0
    puntos_evaluados = []
    #Si el valor de voronoi que pasamos por pantalla es nulo creamos la lista
    if(lista ==[]):
        voronoi = []

    #bucle para el algoritmo
    while True:
        while True:
            if(k == 0):
                temperatura_iterable = temperatura_inicial/1
            vecino = generar_vecindad(punto)
            if punto not in puntos_evaluados:
                puntos_evaluados.append(punto)
            puntos_evaluados.append(vecino)
            lista.append(punto)
            voronoi.append(voronoiRegion(lista,len(lista)-1))
            #calculamos el area del punto almacenado
            area_punto = Area2(voronoi[len(lista)-1])
            lista.pop()
            voronoi.pop()
            lista.append(vecino)
            voronoi.append(voronoiRegion(lista,len(lista)-1))
            #Calculamos el area del vecino nuevo generado
            area_vecino = Area2(voronoi[len(lista)-1])
            lista.pop()
            voronoi.pop()
            #Calculamos la resta de las areas calculadas
            delta = area_punto - area_vecino
            if(delta < 0):
                #Si la resta es negativa es que el area del nuevo vecino es mejor
                punto = vecino
            else:
                if((delta > 0) and (numpy.random.uniform(0,1) < numpy.exp(-
delta/temperatura_iterable))):
                    #si no es mejor el area del vecino con cierta probabilidad
                    # queremos cambiar ha dicho punto
                    punto = vecino
            n = n + 1
            if(n > temperatura_inicial/(1+k)):
                #Condicion de salida del bucle interior
                break
        print(n)
        k = k + 1
        #Actualizamos la temperatura
        temperatura_iterable = temperatura_inicial/(1+k)
        div = div/0.999 #Modificamos el factor de división
        if(temperatura_iterable <= temperatura_final):
            #Evaluamos la condición de parada y si se cumple salimos del bucle
            break
    print(len(puntos_evaluados))
    div = 3.5 #Volvemos a dejar el parametro div como estaba inicialmente
    return punto,puntos_evaluados
```

```
#Funcion que implementa el algoritmo de random search
def random_search_funcion(m,RS,lista):
    posicion = 0
    area = 0
    if(lista == []):
        RS=[]
    #bucle para ir añadiendo a nuestra lista de puntos dibujada cada
    # punto nuevo
    for k in range(len(m)):
        lista.append(m[k])
        RS.append(voronoiRegion(lista,len(lista)-1))
        area_nueva = Area2(RS[len(lista)-1]) #calculamos el area de
                                             #cada una de las
                                             # regiones que
                                             # formarian los nuevos
                                             # puntos aleatorios
        if(area_nueva > area):
            #si dicho area es nayor que el area que ya teniamos lo
            # guardamos y nos quedamos con la posicion de
            # dicho punto aleatorio
            area = area_nueva
            posicion = k
        RS.pop()
        #eliminamos el ultimo elemento de nuestra lista de puntos
        lista.pop()
        print(k)
    return posicion

#Función que implementa el algoritmo de ant systems
def ant_systems_funcion(lista,voronoi):
    #Definimos las condiciones de parada en función del número
    # de puntos introducidos
    if(len(lista)>20):
        parada = len(lista)/2
    else:
        parada = 2*len(lista)

    #Damos los valores iniciales a nuestras variables
    area = 0
    posicion = 0
    contador = 0
    contador_2 = 0
    puntos_buenos = []
    areas_buenas = []
    puntos_evaluados = []
    i_x_inf = 0
    i_x_sup = 500
    i_y_inf = 0
    i_y_sup = 400

    #Iniciamos el bucle que realiza el algoritmo
    while True:
```

```
        while True:
            x = numpy.random.uniform(i_x_inf,i_x_sup)
            y = numpy.random.uniform(i_y_inf,i_y_sup)
            punto = [x,y] #Generamos un número aleatorioa
            if (punto in puntos_evaluados):
                #Guardamos el puntos para luego poder representar
                # los puntos aleatorios generados
                punto = [punto[0]+1,punto[1]+1] #Si ya existe el
                                                #punto evaluado
                                                # lo desplazamos
                                                # para que no se
                                                # pinten uno
                                                # encima de otro
                puntos_evaluados.append(punto)
            else:
                puntos_evaluados.append(punto)
            #Añadimos el punto a la lista y creamos su
            # región de voronoi
            lista.append(punto)
            voronoi.append(voronoiRegion(lista,len(lista)-1))
            area_punto = Area2(voronoi[len(lista)-1]) #Calculamos
                                                      #el área
                                                      # del punto
                                                      # aleatorio
            lista.pop()
            voronoi.pop()
            if(len(puntos_buenos) <= 10):
                #Rellenamos el numero de puntos buenos que deseamos
                puntos_buenos.append(punto)
                areas_buenas.append(area_punto)
            else:
                #Cuando nuestra lista de puntos buenos esta rellena vamos
                # comparando el area de cada uno de los puntos aleatorios
                # con los del vector de puntos buenos
                for i in range(len(areas_buenas)):
                    if (area_punto > areas_buenas[i]):
                        #si el area es mejor eliminamos la que es peor y
                        # lo añadimos a la lista
                        puntos_buenos.pop(i)
                        areas_buenas.pop(i)
                        puntos_buenos.append(punto)
                        areas_buenas.append(area_punto)
                        #Cuando encuentra uno que es mejor salimos del bucle
                        break
            contador +=1
            if(contador == 100):
                #Repetimos este proceso hasta llegar a 100 iteraciones
                contador = 0
                break
        contador_2 +=1
        minimo = min(puntos_buenos) #Buscamos el punto mínimo de la lista de
                                    #puntos buenos
        maximo = max(puntos_buenos) #Buscamos el punto máximo de la lista de
```

```
                                    #puntos buenos
        if(contador_2 >= 3 and len(lista) >= 6 ):
            #Vamos reduciendo los valores donde se pueden generar puntos mínimos
            i_x_inf = minimo[0]
            i_x_sup = maximo[0]
            i_y_inf = minimo[1]
            i_y_sup = maximo[1]
        elif(contador_2 >= 2 and len(lista)<6):
            #Reducimos los valores donde se pueden generar puntos buenos
            i_x_inf = minimo[0]
            i_x_sup = maximo[0]
            i_y_inf = minimo[1]
            i_y_sup = maximo[1]
        print(contador_2)
        if(contador_2 >= parada):
            #Cuando hemos terminado seleccionamos el que mejor area tiene
            # y devolvemos la posicion
            #los puntos buenos y los puntos evaluados
            for i in range(len(puntos_buenos)):
                lista.append(puntos_buenos[i])
                voronoi.append(voronoiRegion(lista,len(lista)-1))
                area_tmp = Area2(voronoi[len(lista)-1])
                if(area_tmp > area):
                    area = area_tmp
                    posicion = i
                voronoi.pop()
                lista.pop()
            break
    print(len(puntos_evaluados))
    return puntos_buenos,puntos_evaluados,posicion

    #Punto medio del segemento dado por dos puntos
def puntoMedio(A,B):
    value = [(A[0]+B[0])/2,(A[1]+B[1])/2]
    return value

#Mediatriz de una recta definida por dos puntos
def mediatriz(r):
    #Vector director de la recta
    [A,B]=r
    #Vector perpendicular de la recta, obtenido a partir del
    # vector director
    v_perpendicular=[-(B[1]-A[1]),B[0]-A[0]]
    #Llamada a la función para calcular el
    # punto medio de la recta
    m=puntoMedio(A,B)
    #Obtenemos otro punto de la recta sumándole al punto
    # medio el vector perpendicular
    c=[m[0]+v_perpendicular[0],m[1]+v_perpendicular[1]]
    #devolvemos la recta como dos puntos
    return [m,c]

#Area signada de un triangulo dado por
```

```
# tres puntos en el plano
def areaSignada(A,B,C):
    value = (1/2*(((B[0]-A[0])*(C[1]-A[1]))-((C[0]-A[0])*(B[1]-A[1]))))
    return value

#Funcion para hacer clipping de un poligono
# con una recta dada
def recorte_poligono(P,r):
   #Se va a construir la region resultante en la lista C
   C=[]
   #Para cada punto del poligono
   for i in range(len(P)):
       if areaSignada(P[i],r[0],r[1])>=0:
           C.append(P[i])
           if areaSignada(P[(i+1)%len(P)],r[0],r[1])<0 :
                   C.append(Interseccion_Rectas(r,[P[i],P[(i+1)%len(P)]]))
       elif areaSignada(P[(i+1)%len(P)],r[0],r[1])>=0:
                   C.append(Interseccion_Rectas(r,[P[i],P[(i+1)%len(P)]]))
    #Se devuelve el poligono que se ha contruido
   return C

#Punto de interseccion de 2 rectas
def Interseccion_Rectas(r,s):
    #rectas no son paralelas
    if (r[1][0]-r[0][0])!=0 and (s[1][0]-s[0][0])!=0:
        #Pendientes de las rectas
        m1=(r[1][1]-r[0][1])/(r[1][0]-r[0][0])
        m2=(s[1][1]-s[0][1])/(s[1][0]-s[0][0])

        #Terminos independientes
        c1=r[0][1]-m1*r[0][0]
        c2=s[0][1]-m2*s[0][0]

        #Son paralelas pues tienen misma pendiente
        if m1==m2 :
            return []

        #Son la misma recta
        if m1==m2 and c1==c2:
            return r

        #Resolución por determinantes
        x= det(numpy.matrix([[c1,1],[c2,1]]))/det(numpy.matrix([[-m1,1],[-
m2,1]]))
        y= det(numpy.matrix([[-m1,c1],[-m2,c2]]))/det(numpy.matrix([[-m1,1],[-
m2,1]]))
        return [x,y]
     #Recta s paralela a eje y
    elif (s[1][0]-s[0][0])==0:
         m1=(r[1][1]-r[0][1])/(r[1][0]-r[0][0])
         c1=r[0][1]-m1*r[0][0]
         return [s[0][0],(m1*s[0][0])+c1]
    #2 rectas paralelas al eje y
```

```
    elif  (r[1][0]-r[0][0])==0 and (s[1][0]-s[0][0])==0:
        return []
    #r paralela al eje y
    else:
        m1=(s[1][1]-s[0][1])/(s[1][0]-s[0][0])
        c1=s[0][1]-m1*s[0][0]
        return [r[0][0],(m1*r[0][0])+c1]

#Region de Voronoi para un punto dentro un conjunto de puntos
def voronoiRegion(p,i):
    #Si solo hay un punto en el plano, su region es
    # toda la region del plano
    if len(p)==1 and i==0:
        #medidas que delimitan la pantalla
        return [[0,0],[500,0],[500,400],[0,400]]
    else:
        #Se comienza con la region de la pantalla
        Region=[[0,0],[500,0],[500,400],[0,400]]
        for k in range (len(p)):
            if k!=i :
                Region=recorte_poligono(Region,mediatriz([p[i],p[k]]))
        return Region

def Area2(corners):
    area = 0.0
    for i in range(1,len(corners)):#recorremos nuestra lista
                                   #de puntos que son los que
                                   # forman los vértices del
                                   # polígono
        #vamos llamando a la funcion del area
        # signada para calcular el area
        area += areaSignada(corners[1],corners[i-1],corners[i])
    return area #devolvemos el area calculada
```

Funciones para manejar archivos, (manejo_archivos.py)

```
from tkinter import filedialog,Tk
import pickle
from popup import noEsPosibleLeerDeEsteFichero

#Funcion que nos permite gurdar los puntos de un diagrama
def save_file(p):
    root = Tk()
    root.withdraw()

    #Método que nos permite crear el archivo que almacenará nuestros puntos y el
lugar de nuestro ordenador donde queremos que lo guarde
    root.file_save = filedialog.asksaveasfilename(title="Guardar
fichero",defaultextension=".txt",filetypes=[("txt files", '*.txt')])

    if root.file_save:
        #Escribimos en nuestro archivo la información que deseamos
```

```
        with open(root.file_save,"wb") as f:
            pickle.dump(p,f)
    root.destroy() #Cerramos nuestra ventana

#Funcion que nos permite abrir el archivo que deseemos
def open_file():
    root = Tk()
    root.withdraw()
    objeto =[]

    #Método que nos permite buscar el archivo de nuestro ordenador que deseamos
abrir
    root.file_open = filedialog.askopenfilename(title = "Abrir fichero")

    if (root.file_open):
        #Implementamos la lectura en un try ya que si no es un archivo almacenado
con pickle desde nuestro programa, no lo vamos a poder leer
        try:
            #Forma de leer de nuestro fichero
            with open(root.file_open,'rb') as f:
                objeto = pickle.load(f)
        except:
            noEsPosibleLeerDeEsteFichero()
            print('No es posible leer de ese fichero')
    root.destroy()
    return objeto
```

Funciones para informar al usuario de la aplicación, (popup.py)

```
import easygui as eg

#Funcion que implementa la ventana emergente para elegir el numero de puntos
aleatorios que se desean
def cuadro():
    numero = eg.integerbox(msg='Introduzca el número de puntos aleatorios que
desea (1-100000):',title='Random
search',default=1,lowerbound=1,upperbound=100000,image=None)
    #devolvemos el numero de puntos que deseamos
    return numero

#Esta función muestra un cuadro de mensaje para indicar que no se puedeguardar
ningun diagrama ya que no hay puntos en el lienzo
def popNadaGuardar():
    eg.msgbox (msg = "No hay puntos que se puedan guardar", title = "Save file",
ok_button = "OK")

#Esta función muestra un cuadro de mensaje para indicar que no se ha introducido
ningun numero de puntos
def popNadaIntroducido():
    eg.msgbox (msg = "No se ha introducido ningún número de puntos aleatorios",
title = "puntos aleatorios", ok_button = "OK")

#Esta función muestra un cuadro de dialogo cuando no es posible leer del fichero
que queremos cargar
def noEsPosibleLeerDeEsteFichero():
    eg.msgbox (msg = "No es posible leer este fichero con este programa", title =
"open file", ok_button = "OK")
```

```
#Función para seleccionar el algoritmo de maximización que se desee
def seleccionarAlgoritmo():
    seleccion = ''
    lista = ['Random Search', 'Simulated Annealing', 'Ant Systems',]
    algoritmo = eg.choicebox(msg='Seleccionar un algoritmo:',
                    title='Algoritmos de maximización de la región de Voronoi',
                    choices=(lista))

    if algoritmo != None:
        eg.msgbox('Algoritmo elegido: '+ algoritmo,
                'Lista de opciones',
                ok_button='Continuar')
    else :
        eg.msgbox('No se ha elegido ningún algoritmo',ok_button='Continuar')

    return algoritmo

#Funcion para introducir el numero de puntos que se desea que tenga el diagrama
de voronoi aleatorio
def puntosvoraleatorio():
    numero = eg.integerbox(msg='Introduzca el número de puntos aleatorios que
desea para el diagrama de voronoi(1-300):',title='Random
search',default=1,lowerbound=1,upperbound=300,image=None)
    #devolvemos el numero de puntos que deseamos
    return numero

#Función para indicar que la posición del punto no esta disponible
def posicionNoDisponible():
    eg.msgbox (msg = "Posición del punto no disponible", title = "pintar punto",
ok_button = "OK")

#Función para indicar que no hay puntos en el lienzo para realizar la
maximizacion
def noHayptosEnElLienzo():
    eg.msgbox (msg = "No hay puntos en el lienzo, no se puede realizar la
maximización de la región de Voronoi", title = "Maximización RV", ok_button =
"OK")

#Función para indicar que ha terminado de calcular la maximización
def maximizacionCalculada():
    eg.msgbox (msg = "Maximización calculada", title = "Maximización RV",
ok_button = "OK")
```