Título: *Diagramas de Voronoi de alcance limitado.*

Alumno 1: *José Luis Moreno Durán.*
Alumno 2: *Sergio Ordóñez Pérez.*

Directora: *Vera Sacristán Adinolfi.*
Departamento: *Matemática Aplicada II.*
Fecha: *19 de Junio de 2009.*

DATOS DEL PROYECTO:

Título del proyecto: Diagramas de Voronoi de alcance limitado.

Nombre del estudiante 1: José Luis Moreno Durán.
Nombre del estudiante 2: Sergio Ordóñez Pérez.
Titulación: Ingeniería Informática.
Créditos: $37,5$.
Directora: Vera Sacristán Adinolfi.
Departamento: Matemática Aplicada II.

MIEMBROS DEL TRIBUNAL: *(Nombre y firma)*

Presidente: Mercè Mora Giné.

Vocal: Elvira Patricia Pino Blanco.

Secretario: Vera Sacristán Adinolfi.

CALIFICACIÓN:

Calificación numérica:
Calificación descriptiva:

Fecha: 19 de Junio de 2009.

Índice general

Capítulo 1

Introducción

En este proyecto presentamos un estudio de los diagramas de Voronoi de alcance limitado de un conjunto finito de puntos del plano y *DVALon*, una herramienta para la manipulación de dichos diagramas, los grafos de proximidad de alcance limitado y su aplicación a los caminos de desviación mínima y de separación máxima. El resultado de este trabajo ha sido aceptado para su presentación en los XIII Encuentros de Geometría Computacional [3].

1.1. Contexto

Recientemente, los diagramas de Voronoi y, más en general, los grafos de proximidad están siendo objeto del interés de comunidades científicas tan dispares como las surgidas alrededor del análisis y el desarrollo de redes de sensores inalámbricos [15] o del estudio de la evolución de los bosques [1].

En ingeniería forestal se emplea el concepto de región potencialmente disponible de un árbol (*APA: Area Potentially Available*) para estudiar la influencia entre los árboles vecinos en el desarrollo de un bosque. La primera referencia de la que hay constancia es del ingeniero alemán Koenig y se remonta a 1864. Existen varias descripciones de dicha región. Smith en 1987 [19] hace un repaso de las primeras aproximaciones y llega a la conclusión de que los diagramas de Voronoi y los algoritmos que proporciona la Geometría Computacional son los instrumentos adecuados para su modelización. En este caso, las regiones de influencia son necesariamente limitadas por la capacidad de crecimiento de cada especie vegetal. En [18] se emplean las propiedades de los diagramas de Voronoi obtenidos como resultado de la expansión de círculos para estudiar las islas verdes que se crean en los incendios forestales (zonas rodeadas de fuego).

Sean estas *ad hoc* o fijas, las redes de sensores tienen aplicaciones en la monitorización y el control de procesos industriales, el seguimiento de parámetros medioambientales, el control del tráfico, la automatización y la vigilancia domésticas, de edificios públicos y de oficinas, los cuidados sanitarios y sociales, la detección precoz de incendios, el seguimiento de movimientos sísmicos, las redes de radares y sónares, la detección de partículas químicas o nucleares en el aire, el control de robots móviles y, por descontado, las antenas de telefonía móvil.

En ninguno de estos casos sería realista asumir que los sensores sean capaces

de detectar calor, presión, sonido, luz, campos electromagnéticos, vibración o lo que quiera que sea su especialización, a distancias arbitrariamente grandes, como tampoco que sean capaces de comunicarse independientemente de la distancia a la que se encuentren los unos de los otros.

Es en estos contextos donde cobra interés el diagrama de Voronoi de alcance limitado: una descomposición del plano en regiones, cada una de ellas más cercana a un sensor -a un árbol- que a cualquier otro, pero limitadas por cierta distancia a este. Por ejemplo, en [6] se propone un método de eliminación de redundancia sin pérdida de cobertura en redes de sensores basado en el reconocimiento de los sensores cuya región de Voronoi de alcance limitado coincide con su región de Voronoi ordinaria, y se caracterizan los sensores que intervienen en la frontera del diagrama de Voronoi de alcance limitado. En el mencionado trabajo no se propone una terminología específica, pero en [22] se bautiza el diagrama de Voronoi de alcance limitado con el nombre de *Aberrant Voronoi Graph*.

Del mismo modo, tienen interés y aplicación los grafos de proximidad de alcance limitado: triangulación de Delaunay, grafo de Gabriel, grafo de vecinos relativos, etc. Estos grafos han sido introducidos independientemente en [9] y [13], donde reciben los nombres de *Unit Delaunay Triangulation*, *Constrained Gabriel Graph* y *Constrained Relative Neighborhood Graph*.

En particular, los diagramas de Voronoi de alcance limitado y sus grafos de proximidad asociados tienen aplicación directa al estudio de caminos de desviación mínima y de separación máxima, tal como se expone en [2, 5]. La importancia de estos caminos deriva de su uso como herramientas de medida de la calidad de una red de sensores para cubrir un cierto territorio o *área de interés* [14].

Consideremos un conjunto P de n puntos del plano. La desviación respecto de P de un camino en el plano que conecta dos puntos s y t es la máxima distancia entre los puntos del camino y el conjunto P (entendiendo que la distancia de un punto a un conjunto de puntos es la menor entre el punto y los puntos del conjunto). Nos interesan los caminos que conectan puntos del plano alejándose lo menos posible del conjunto P. Esta situación ocurre, por ejemplo, cuando se quiere desplazar un receptor de un punto a otro dentro de una región en la que hay una red de antenas emitiendo una señal. Cuanto más próximos estén los puntos del camino a las antenas, mejor será la señal que reciba el receptor en su desplazamiento. En [2] y [16] se describe cómo obtener caminos de desviación mínima entre dos puntos. En general, existen muchos caminos de desviación mínima entre dos puntos, por lo que tiene interés seleccionar entre todos ellos los que satisfagan otras propiedades, como por ejemplo: longitud mínima, giro total mínimo, exposición mínima o máxima a la señal de las antenas, etc.

En este trabajo nos interesamos por los caminos que, además de ser de desviación mínima, son tales que cualquier subcamino suyo hereda dicha propiedad. Es decir, todo subcamino es a su vez un camino de desviación mínima entre sus extremos. Estos caminos han sido introducidos y estudiados en [5].

Este tipo de caminos aparecen en el diseño y análisis de redes de sensores. En [15] se plantean dos problemas: Entre todos los caminos de desviación mínima, hallar el que se mantiene más alejado de los sitios (*Maximal Breach Path*), que se aplica como test a una red de sensores para medir hasta qué punto es fácil escapar a su vigilancia, y hallar el que se mantiene más próximo a los sitios (*Maximal Support Path*), que se aplica para medir la mejor cobertura. En [13]

se demuestra que existe un camino de desviación mínima en el grafo de Gabriel, por lo que se puede calcular localmente. También abordan los autores la minimización de la longitud del camino, pero se restringen al grafo de discos unidad (*Unit Disk Graph*), obteniendo una 5/2-aproximación del camino óptimo en dicho grafo. En [16] se subraya que la solución sobre el grafo de discos unidad no se corresponde con la solución al problema en el plano, se mejora el algoritmo de [15] para el problema Maximal Breach Path y se calcula el de mínima longitud. Además se indica, sin detallar, cómo hallar el camino de desviación mínima de mínima longitud.

1.2. Objetivos del proyecto

El objetivo principal del proyecto es el desarrollo de una aplicación de visualización y cálculo para el estudio de los diagramas de Voronoi de alcance limitado y los caminos de desviación mínima.

La aplicación a realizar debe ser capaz de construir y visualizar tanto los grafos de proximidad como los diagramas de Voronoi de alcance limitado de un conjunto de puntos P del plano.

Asimismo, se desea que la aplicación permita al usuario introducir caminos compuestos por segmentos y arcos de circunferencia con centro en alguno de los puntos de P, y sea capaz de decidir si los caminos introducidos son de desviación mínima y de desviación mínima local.

Para la consecución de este objetivo, es necesario profundizar en la comprensión de los conceptos teóricos asociados, algunos de los cuales están actualmente en desarrollo, y así adquirir los conocimientos necesarios que permitan desarrollar la aplicación.

El objetivo del proyecto, pues, es realizar una aplicación que responde a la necesidad de una investigación en curso, que necesita disponer de una serie de herramientas no presentes en otras aplicaciones hasta ahora, para que pueda ser usada por miembros de la comunidad científica para facilitar sus investigaciones relacionadas con los diagramas de Voronoi de alcance limitado y su aplicación a los caminos de desviación mínima y caminos de desviación mínima local.

1.3. Organización de la memoria

La memoria del proyecto se organiza del modo siguiente:

En el Capítulo 2 se presentan una serie de conceptos conocidos, relacionados con la geometría computacional, y que han sido necesarios para la consecución del trabajo. Se refiere a conceptos importantes, como diagramas de Voronoi o grafos de proximidad. También incluye conceptos más relacionados con la informática, como son las estructuras de datos y los algoritmos utilizados para resolver algunos problemas geométricos.

El Capítulo 3 se dedica a definir y estudiar los conceptos que están siendo actualmente objeto de investigación y desarrollo, y cuya implementación constituye el núcleo central de la aplicación desarrollada en este proyecto, los diagramas de Voronoi de alcance limitado, los caminos de desviación mínima y caminos de desviación mínima local y sus posibles optimizaciones utilizando diversos criterios.

El Capítulo 4 describe las funcionalidades de la aplicación *DVALon*, y para ello expone el trabajo interno que realiza la aplicación para la consecución de cada funcionalidad. Esto incluye el análisis de los algoritmos utilizados, la mención de las estructuras de datos empleadas para guardar los resultados y las consideraciones más importantes para entender el funcionamiento de la aplicación.

En el Capítulo 5 se presenta el manual de usuario de la aplicación, en el que se explica el modo de empleo de *DVALon* con las nociones necesarias para que el usuario pueda sacar el máximo rendimiento de la aplicación.

En el Capítulo 6 se plantean aspectos importantes referentes al desarrollo de este trabajo, como son la planificación con la que se ha llevado a cabo, o las ventajas e inconvenientes de haber realizado el proyecto entre dos personas.

Finalmente, en el Capítulo 7 se enumeran las conclusiones obtenidas después de haber realizado un proyecto de esta magnitud.

En la parte final de la memoria se recogen todas las referencias bibliográficas que se citan en el documento.

Capítulo 2

$Vor(P)$ y grafos de proximidad

2.1. Diagrama de Voronoi

El diagrama de Voronoi es una estructura que captura la información de proximidad de un conjunto de puntos P descomponiendo el plano en regiones poligonales convexas (ver [17]). Esta estructura tiene un gran interés para este proyecto por las aplicaciones geométricas que posee. Entre sus aplicaciones geométricas, se encuentra la construcción de la triangulación de Delaunay dualizando el diagrama de Voronoi, o la búsqueda del punto $p_i \in P$ más cercano a un punto q del plano. Entre sus aplicaciones no geométricas, se encuentran usos ecológicos, como la de determinar la supervivencia de organismos en competencia por alimentos o luz (por ejemplo, árboles en un bosque), y usos en redes sociales para representar las relaciones entre personas.

Proposición 2.1.1 *Dado un conjunto finito de puntos del plano, P, el diagrama de Voronoi de P, $Vor(P)$, es la descomposición del plano en las regiones asociadas, por proximidad, a cada uno de los puntos de P. Dichas regiones se denominan regiones de Voronoi.*

La *región de Voronoi* de un punto $p_i \in P$ es:

$$Vor(p_i) = \{x \in \mathbb{R}^2 \mid d(x, p_i) \leq d(x, p_j)\ \forall j \neq i\}$$

La Figura 2.1 ilustra el diagrama de Voronoi de un conjunto de puntos P.

Por lo tanto, la region de Voronoi asociada al punto $p_i \in P$, $Vor(p_i)$, contiene los puntos del plano que están más cerca del punto p_i que del resto de puntos de P. La región $Vor(p_i)$ es poligonal y convexa, y es acotada si, y sólo si, p_i es interior a la envolvente convexa de P. Las aristas de $Vor(p_i)$ son porciones de mediatrices de pares de puntos, y pueden ser de tres tipos:

1. Rectas, si los puntos de P están alineados. Ver Figura 2.2.

2. Semirrectas, si los dos puntos que determinan la arista son consecutivos en la envolvente convexa de P. Ver Figura 2.1.

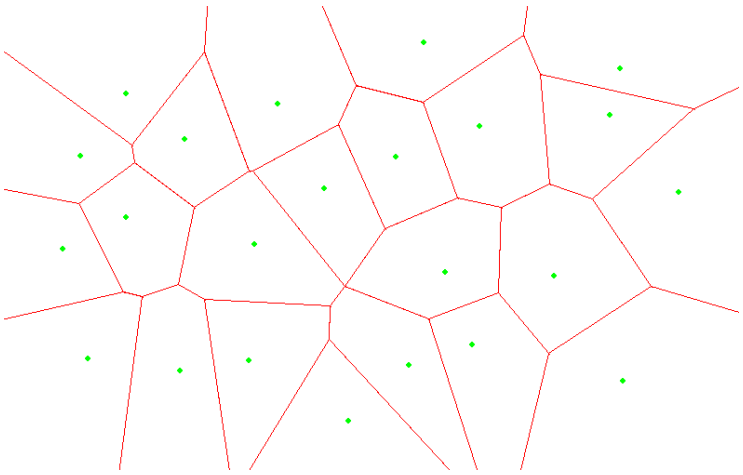

Figura 2.1: Diagrama de Voronoi de P.

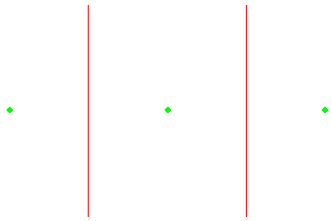

Figura 2.2: Regiones de Voronoi delimitadas por rectas.

3. Segmentos, si al menos uno de los puntos que determina la arista es interior a la envolvente convexa de P. Ver Figura 2.1.

Añadiendo un punto en el infinito, como extremo de todas las semirrectas y rectas, $Vor(P)$ es un grafo plano y, en consecuencia, se le puede aplicar la fórmula de *Euler* para obtener una relación entre el número de vértices (v), el número de aristas (a), y el de caras (n): $(v+1)+n = a+2$ de donde $v+n = a+1$. Además, como cada vértice de Voronoi es incidente al menos a 3 aristas y cada arista tiene exactamente 2 extremos, tenemos que $2a \geq 3(n+1)$ por lo que $v \leq 2n-5$ y $a \leq 3n-6$. Por lo tanto, la complejidad de $Vor(P)$ es $O(n)$. Aunque para algún $p_i \in P$, $Vor(p_i)$ puede tener $n-1$ aristas, el promedio de aristas de una región de Voronoi en cualquier conjunto P es:

$$\frac{2a}{n} \leq \frac{6n-12}{n} = 6 - \frac{12}{n} \leq 6$$

Es conveniente mencionar ahora que la triangulación de Delaunay, $Del(P)$, es el grafo dual rectílineo de $Vor(P)$. En concreto, las aristas de la triangulación de Delaunay unen aquellos puntos de P que son vecinos en $Vor(P)$, es decir, cuyas regiones de Voronoi son adyacentes.

2.2. Grafos de proximidad

En esta sección se definen y caracterizan los conceptos relativos a grafos de proximidad que ha sido necesario adquirir para la realización de este proyecto. En concreto, se dan las definiciones de grafo y grafo geométrico. Se definen también los grafos de proximidad calculados por la aplicación, que son: la triangulación de Delaunay, el grafo de Gabriel, el grafo de vecindad relativa y el árbol generador mínimo euclídeo. Estos últimos han sido introducidos en [20] y [21].

2.2.1. Grafo. Grafo geométrico

En matemáticas y ciencias de la computación, un grafo (del griego grafos: dibujo, imagen) es un conjunto V de vértices o nodos unidos por enlaces llamados aristas o arcos, que forman el conjunto E, que permiten representar relaciones binarias entre elementos de un conjunto.

Típicamente, un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas rectas o curvas (aristas), aunque existe todo un campo de investigación relacionado con los diversos modos de representación (véase [8, 10]).

Los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan terminales y las aristas representan conexiones (las cuales, a su vez, pueden ser cables o conexiones inalámbricas).

Prácticamente cualquier problema puede representarse mediante un grafo, y su estudio trasciende a las diversas áreas de las ciencias exactas y las ciencias sociales.

Llamamos grafo dirigido al grafo $G = (V, E)$ cuyas aristas son un conjunto de pares ordenados de elementos de V. Dada una arista (a, b), a es su nodo inicial y b su nodo final.

Terminología de los grafos:

1. Adyacencia: dos aristas son adyacentes si tienen un vértice en común, y dos vértices son adyacentes si una arista los une.

2. Incidencia: una arista es incidente a un vértice si este es uno de sus extremos.

3. Ponderación: corresponde a una función que a cada arista le asocia un valor (coste, peso, longitud, etc.), para aumentar la expresividad del modelo. Esto se usa mucho para problemas de optimización, como el del vendedor viajero o del camino más corto.

4. Etiquetado: distinción que se hace a los vértices y/o aristas mediante una marca que los hace unívocamente distinguibles del resto.

En matemáticas, un grafo geométrico es un grafo cuyos sus vértices o aristas están asociadas a objetos o configuraciones geométricas. Algunos de los grafos geométricos más notables y más interesantes para nuestro proyecto son los siguientes:

1. Un grafo rectilíneo plano es un grafo cuyos vértices son puntos del plano euclídeo y cuyas aristas son segmentos que no se cortan. Ejemplos de este tipo de grafos son las triangulaciones, definidas en la sección 2.2.2, o los grafos de Gabriel y de vecindad relativa definidos en 2.2.3 y 2.2.4.

2. Un 1-esqueleto de un poliedro es el conjunto de vértices y aristas del politopo. El esqueleto de cualquier poliedro convexo es un grafo.

3. Un grafo euclídeo es un grafo cuyos vértices representan puntos del plano, y a sus aristas se les asignan pesos iguales a la distancia euclídea entre sus dos vértices. El árbol generador mínimo euclídeo (ver sección 2.2.5) es el árbol de expansión mínimo de un grafo euclídeo completo.

En este proyecto hemos trabajado con grafos geométricos euclídeos. Concretamente, con diversos grafos geométricos euclídeos que capturan información de proximidad entre un conjunto dado de puntos del plano.

2.2.2. Triangulación de Delaunay

Una triangulación de un conjunto P de n puntos del plano es un grafo rectilíneo, plano, con vértices en P y con un conjunto maximal de aristas.

Definimos triangulación de Delaunay de un conjunto de puntos P, $Del(P)$, como el grafo dual rectilíneo del diagrama de Voronoi de P, $Vor(P)$. Para más información acerca de $Vor(P)$ y de su dualidad con $Del(P)$ consúltese la Sección 2.1 o la referencia [17] de este mismo capítulo.

1. Características

   a) Dos puntos $p_i, p_j \in P$ forman una arista de Delaunay si, y sólo si, existe un círculo vacío de puntos de P cuya frontera pasa por p_i y p_j. Esta característica está ilustrada en la Figura 2.3.a).

   b) Tres puntos $p_i, p_j, p_k \in P$ forman un triángulo de Delaunay si, y sólo si, el círculo que definen está vacío de puntos de P. Esta característica se ilustra en la Figura 2.3.b).

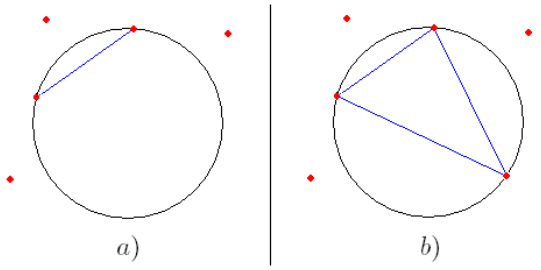

Figura 2.3: Ejemplo de arista de Delaunay (izquierda) y de triángulo de Delaunay (derecha).

2. Propiedades

   a) $Del(P)$ es un grafo plano, ya que si un segmento $\overline{rs}$ corta a una arista de Delaunay $\overline{pq}$, cuyo círculo es vacío, todo círculo por r y s contiene al menos p o q.

b) $Del(P)$ es una triangulación de P, es decir, todas sus caras son triángulos, excepto si P tiene cuatro o más puntos concíclicos. En este caso $Del(P)$ es una pre-triangulación y es trivial completarla.

En la Figura 2.4 podemos apreciar una nube de puntos y su triangulación de Delaunay.

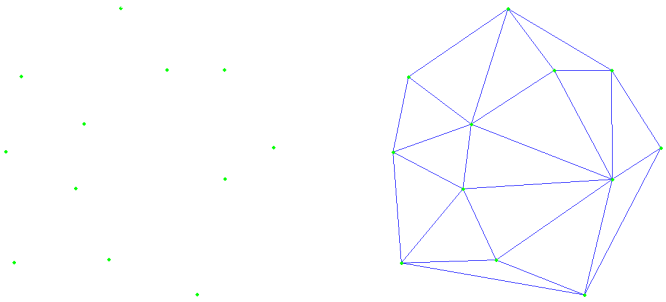

Figura 2.4: Nube de puntos y Triangulación de Delaunay.

Las aristas de la triangulación de Delaunay unen aquellos puntos de P que son vecinos en $Vor(P)$, es decir, cuyas regiones de Voronoi son adyacentes.

Por otro lado, cada triángulo de $Del(P)$ es incidente a tres puntos de P (sus tres vértices) tales que el circulo que pasa por ellos no contiene ninguno otro punto de la nube. Además, los vértices de Voronoi son aquellos lugares del plano que equidistan de tres o más sitios (puntos de P). El circuncentro de cada triángulo de $Del(P)$ es el lugar que equidista de los tres vértices del triángulo y está más alejado del resto de puntos de la nube que de estos tres ya que no puede haber otros puntos de P contenidos en el circulo. Por lo tanto, el circuncentro de cada triángulo $p_ip_jp_k$ coincide con el vértice de Voronoi común a las regiones de Voronoi de p_i, p_j y p_k.

Es sencillo, pues, encontrar los vértices y las aristas de $Del(P)$ a partir de $Vor(P)$, y viceversa. En la Figura 2.5 se pueden observar y comparar $Del(P)$ y $Vor(P)$.

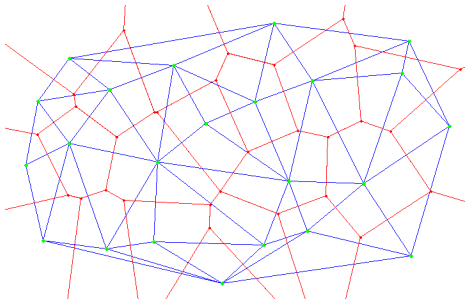

Figura 2.5: $DEL(P)$ y $Vor(P)$.

Es importante observar que la propiedad de Delaunay puede reducirse a una condición local.

Una triangulación es localmente de Delaunay si, para cada par de triángulos $p_ip_jp_k$ y $p_ip_jp_l$ que comparten una arista $\overline{p_ip_j}$, se cumple que p_l no está contenido en el círculo por $p_ip_jp_k$ y que p_k no está contenido en el círculo por $p_ip_jp_l$.

Proposición 2.2.1 *Basta que una triangulación sea localmente de Delaunay para que sea de Delaunay.*

Demostración: Para demostrarlo, supongamos que tenemos una triangulación $T(P)$ que es localmente de Delaunay pero no es de Delaunay. Existiría un triángulo T_{ijk} cuyo circuncírculo contendría un cierto p_l (véase la Figura 2.6). Supongamos que $\overline{p_ip_j}$ es el lado de T_{ijk} que separa p_l de T_{ijk}. De todas las tuplas $ijkl$ que están en esta situación, consideremos la que maximiza el ángulo $p_ip_lp_j$. Sea T_{ijm} el triángulo contiguo a T_{ijk} por el lado $\overline{p_ip_j}$. Como $T(P)$ es localmente de Delaunay, $m \neq l$. Entonces, p_l está en el círculo C_{ijm}. Por tanto, uno de los ángulos $p_ip_lp_m$ y $p_jp_lp_m$ es más grande que el ángulo $p_ip_lp_j$, con lo que llegamos a una contradicción. □

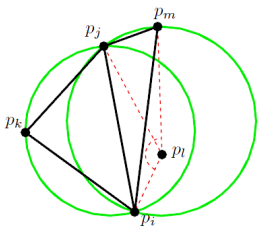

Figura 2.6: Ilustración de la demostración de la localidad de la condición de Delaunay.

El cálculo de la triangulación de Delaunay de un conjunto P de n puntos se puede hacer, en el peor de los casos, en tiempo $O(n \log n)$.

El cálculo de la triangulación de Delaunay de P se puede hacer, en el peor de los casos, en tiempo $O(n^2)$. Sin embargo, existen algoritmos capaces de realizarla en un tiempo promedio $O(n \log n)$.

Algunos de los algoritmos de construcción de triangulaciones de Delaunay se basan en flips de Delaunay. Un flip de Delaunay consiste en eliminar una diagonal de un cuadrilátero convexo, si este no es localmente de Delaunay, y sustituirla por la otra diagonal del cuadrilátero.

Lema 2.2.2 *En un cuadrilátero convexo con diagonales $\overline{ab}$ y $\overline{pq}$, si $\overline{ab}$ no es localmente de Delaunay, $\overline{pq}$ sí lo es, y viceversa.*

Demostración: Supongamos que $\overline{ab}$ no es localmente de Delaunay. Esto implica que q está en el interior del círculo C_{abp}, lo que implica que el ángulo aqp es mayor que el ángulo abp. Por tanto, también tenemos que b es exterior al círculo C_{aqp}, lo que implica que la diagonal $\overline{pq}$ es localmente de Delaunay. La situación se ilustra en la Figura 2.7. □

Este lema es especialmente interesante para nosotros, puesto que permite actualizar de manera muy sencilla una triangulación de Delaunay de forma incremental, como es nuestro caso, tal como se expone en la Sección 4.2.1.

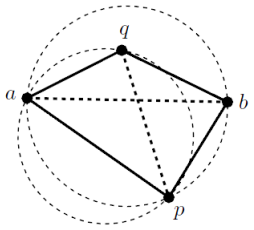

Figura 2.7: Ejemplo de flip de Delaunay.

Para acabar, definimos triangulación de Delaunay de alcance limitado por r como la triangulación que contiene únicamente las aristas de la triangulación de Delaunay ordinaria de longitud menor o igual a r. La triangulación de Delaunay de alcance limitado es el grafo dual rectilíneo del diagrama de Voronoi de alcance limitado por el radio r de P.

2.2.3. Grafo de Gabriel

El grafo de Gabriel de un conjunto P de n puntos, $GG(P)$, es un subgrafo de la triangulación de Delaunay de P. Una arista pertenece al grafo de Gabriel cuando se cumple que no queda ningún otro punto dentro de la circunferencia definida por el diámetro que forman sus dos vértices. El grafo de Gabriel es un grafo conexo.

Definimos grafo de Gabriel de alcance limitado por r como el grafo de Gabriel del que únicamente seleccionamos las aristas cuya longitud es inferior o igual al valor r. El grafo de Gabriel de alcance limitado por r es un subgrafo de la triangulación de Delaunay de alcance limitado por r.

2.2.4. Grafo de vecindad relativa

El grafo de vecindad relativa de un conjunto P de n puntos, $RNG(P)$, es un subgrafo del grafo de Gabriel de P. Una arista pertenece al grafo de vecinos relativos si sus extremos son vecinos relativos, es decir, si la lente intersección de las circunferencias con centro en los extremos de la arista y radio la longitud de la arista no contiene ningún otro punto. El grafo de vecinos relativos es un grafo conexo. En la Figura 2.8 observamos un ejemplo de grafo de Gabriel y de grafo vecinos relativos.

Definimos grafo de vecinos relativos de alcance limitado por r como el grafo de vecinos relativos del que únicamente seleccionamos las aristas cuya longitud es inferior o igual al valor r. El grafo de vecinos relativos de alcance limitado por r es un subgrafo del grafo de Gabriel de alcance limitado por r.

2.2.5. Árbol generador mínimo euclídeo

El árbol generador mínimo euclídeo de un conjunto P de n puntos, $EMST(P)$, es un subgrafo del grafo de vecinos relativos de P. Se trata del árbol generador mínimo del grafo completo de P, ponderado de acuerdo con la distancia euclídea entre sus vértices. De hecho, un conjunto de puntos P puede tener varios árboles generadores mínimos euclídeos, pero todos tienen el mismo coste (suma

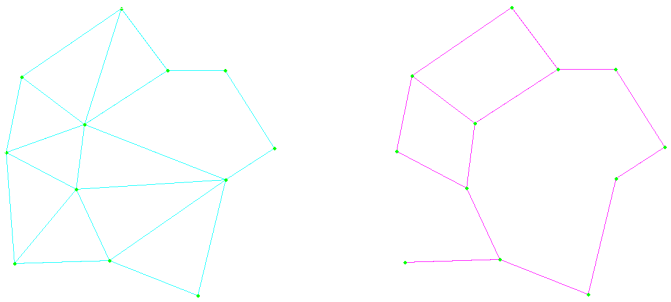

Figura 2.8: Grafo de Gabriel y grafo de vecindad relativa.

de las longitudes de sus aristas). En la Figura 2.9 tenemos un ejemplo de árbol generador mínimo euclídeo.

Definimos árbol generador mínimo de alcance limitado por r como el árbol generador mínimo del que únicamente seleccionamos las aristas cuya longitud es inferior o igual al valor r. El resultado es, en general, un bosque, puesto que la simple eliminación de una arista de un árbol produce la desconexión del grafo.

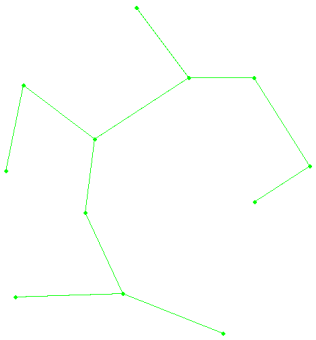

Figura 2.9: Árbol generador mínimo euclídeo.

2.3. Estructuras de datos y algoritmos básicos

En esta sección se describen las estructuras de datos más importantes que usa nuestra aplicación para almacenar la información, sin tener en cuenta las estructuras simples como pueden ser una tabla o un vector. En concreto, en esta sección se presentan y analizan la lista de aristas doblemente enlazada ($DCEL$), y las listas de adyacencias.

También se describen algunos de los algoritmos ya existentes que se han usado en nuestra aplicación. En concreto, veremos el algoritmo de Prim, utilizado para encontrar el árbol generador mínimo euclídeo a partir de un grafo conexo, y el algoritmo de ordenación por selección, utilizado principalmente para ordenar vectores o listas de elementos.

2.3.1. Lista de aristas doblemente enlazada (DCEL)

La DCEL (lista de aristas doblemente enlazada) es una estructura de datos que se usa para representar grafos planos y politopos en 3D. Esta estructura de datos proporciona una manipulación eficiente de la información topológica asociada a estos objetos (incidencia entre vértices, aristas y caras, orden de las aristas incidentes en un vértice o frontera de una cara, etc.). Se utiliza en muchos algoritmos geométricos para tratar subdivisiones poligonales del plano, comúnmente llamadas grafos rectilíneos planos.

Esta estructura fue originalmente propuesta por Muller y Preparata para representaciones de poliedros convexos 3D. Por simplicidad, esta estructura sólo considera grafos conexos, pero es extensible fácilmente para poder tratar grafos no conexos.

Normalmente está compuesta por una lista de vértices, una lista de semiaristas y una lista de caras. Para cada vértice almacena sus coordenadas y una semiarista incidente. Por cada arista del grafo, guarda dos semiaristas orientadas en la misma dirección y sentidos contrarios. Para cada una de ellas, guarda su vértice origen, su semiarista gemela (semiarista de sentido contrario), la cara a la que es incidente y la semiarista que la sigue (o precede) en la cara. Para cada cara, guarda una semiarista incidente. Sin embargo, no todas las DCELs guardan siempre exactamente la misma información (aunque sí información equivalente), y ésta puede estar sujeta cambios según para lo que se esté utilizando. En la Figura 2.10 se ilustra un ejemplo del tipo de estructuras que se pueden almacenar en una DCEL.

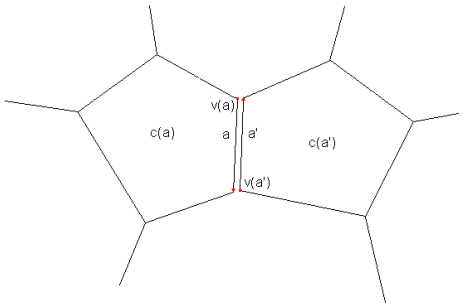

Figura 2.10: Ejemplo de la información de un grafo que se guarda en una DCEL.

2.3.2. Lista de adyacencias

La lista de adyacencias es una estructura de datos que es asocia a cada vértice i de un grafo una lista que contiene todos los vértices j que son adyacentes (ver Figura 2.11). De esta forma, a diferencia de la representación por matriz de adyacencias, la lista de adyacencias sólo reserva memoria para los arcos adyacentes a i y no para todos los posibles arcos que pudieran tener como origen i. El grafo $G = (V, A)$, por tanto, se representa por medio de un vector de n componentes (si $|V| = n$) donde la componente i es una lista de los vértices adyacentes al vértice i del grafo. Cada elemento de la lista consta de un campo que indica el vértice adyacente. En caso de que el grafo sea etiquetado, se añade un segundo campo para mostrar el valor de la etiqueta.

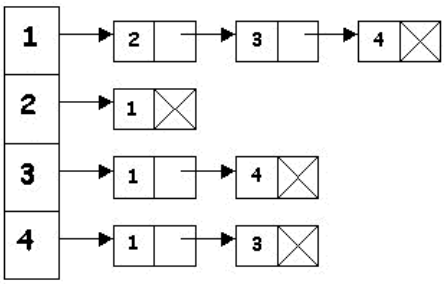

Figura 2.11: Representación de un grafo por su lista de adyacencias.

Esta representación requiere un espacio proporcional a la suma del número de vértices, más el número de arcos, y se suele usar cuando el número de arcos es mucho sustancialmente inferior al del grafo completo. Una desventaja respecto a la representación por matriz de adyacencias es que con la lista de adyacencias puede llevar tiempo O(n) determinar si existe un arco del vértice i al vértice j, ya que puede haber n vértices en la lista de adyacencia asociada al vértice i.

Por otro lado, esta estructura no permite añadir o suprimir vértices del grafo. Para solucionar esto se puede usar una lista de listas de adyacencia. Sólo los vértices del grafo que sean origen de algún arco aparecerán en la lista. De esta forma se pueden añadir y suprimir arcos sin desperdicio de memoria ya que simplemente habrá que modificar la lista de listas para reflejar los cambios.

2.3.3. Algoritmo de Prim

El algoritmo de Prim permite calcular un árbol recubridor mínimo en un grafo ponderado conexo y no dirigido.

En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, y tal que el peso total de las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encuentra el árbol recubridor mínimo para una de las componentes conexas del grafo.

El algoritmo fue diseñado en 1930 por el matemático Vojtech Jarnik y luego de manera independiente por el científico computacional Robert C. Prim en 1957 y redescubierto por Dijkstra en 1959. Por esta razón, el algoritmo es también conocido como algoritmo DJP o algoritmo de Jarnik.

El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol y que no forman ciclos con ellos. El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar. El algoritmo siguiente presenta el pseudocódigo del algoritmo de Prim.

```
PRIM (Grafo G, nodo_fuente s)
    // inicializamos todos los nodos del grafo.
    // La distancia la ponemos a infinito
    // y el padre de cada nodo a NULL
```

```
    for each u \in V[G] do
        distancia[u] = INFINITO
        padre[u] = NULL
    distancia[s]=0
    //encolamos todos los nodos del grafo
    Encolar(cola, V[G])
    while cola != 0 do
        // OJO: Se extrae el nodo que tiene
        // distancia mínima y se conserva la
        // condición de Cola de prioridad
        u = extraer_minimo(cola)
        for v \in adyacencia[u] do
            if ((v \in cola) && (distancia[v] > peso(u, v)) do
                padre[v] = u
                distancia[v] = peso(u, v)
                actualizar(cola,v);
```

2.3.4. Algoritmo de ordenación por selección

La ordenación por selección es un algoritmo de ordenación que requiere $O(n^2)$ operaciones para ordenar una lista de n elementos.

Su funcionamiento es el siguiente:

1. Buscar el mínimo elemento de la lista
2. Intercambiarlo con el primero
3. Buscar el mínimo en el resto de la lista
4. Intercambiarlo con el segundo

El pseudocódigo del algoritmo de ordenación por selección es el siguiente:

```
ORD_SEL (lista)
    para i=1 hasta n-1
        minimo = i;
        para j=i+1 hasta n
            si lista[j] < lista[minimo] entonces
                minimo = j;
            fin si
        fin para
        intercambiar(lista[i], lista[minimo])
    fin para
```

Capítulo 3

DVAL y caminos de desviación mínima

3.1. Diagrama de Voronoi de alcance limitado

Recientemente, los diagramas de Voronoi están siendo objeto de interés en campos científicos como los dedicados al análisis y el desarrollo de redes de sensores inalámbricos [15] o al estudio de la evolución de bosques [1].

No sería realista asumir por ejemplo en el caso de los sensores, que estos fueran capaces de detectar calor, presión, sonido, luz, campos electromagnéticos, vibración o lo que quiera que sea su especialización, a distancias arbitrariamente grandes, como tampoco que sean capaces de comunicarse independientemente de la distancia a que se encuentren los unos de los otros.

Es en estos contextos donde cobra interés el diagrama de Voronoi de alcance limitado: una descomposición del plano en regiones, cada una de ellas más cercana a un punto p_i perteneciente a un conjunto de puntos P, que a cualquier otro punto $p_j \in P$, pero limitadas por cierta distancia a este.

Los diagramas de Voronoi de alcance limitado tienen aplicación directa al estudio de caminos de desviación mínima y de separación máxima, tal como se expone en [2, 5].

3.1.1. Definiciones

Dado un conjunto $P = \{p_1, ..., p_n\}$ de n puntos del plano y un número real $r \geq 0$, la *región de Voronoi de alcance* r de un punto $p_i \in P$ es:

$$Vor(p_i, r) = \{x \in \mathbb{R}^2 \mid d(x, p_i) \leq d(x, p_j)\ \forall j \neq i\ \wedge\ d(x, p_i) \leq r\}.$$

Proposición 3.1.1 *La región $Vor(p_i, r)$ es la intersección de la región de Voronoi ordinaria $Vor(p_i)$ y el círculo con centro en p_i y radio r.*

El *diagrama de Voronoi de alcance* r del conjunto P, denotado $Vor(P, r)$, es la descomposición del plano en las regiones $Vor(p_i, r)$. Si $r = 0$, el diagrama coincide con P y consta de n componentes conexas. Si $r = +\infty$, $Vor(P, r)$ coincide con el diagrama de Voronoi ordinario $Vor(P)$, y consta de una única

componente conexa. Si $r \neq +\infty$, $Vor(P, r)$ puede constar de una o más componentes conexas, cada una de las cuales con posibles agujeros. La Figura 3.1 ilustra un ejemplo.

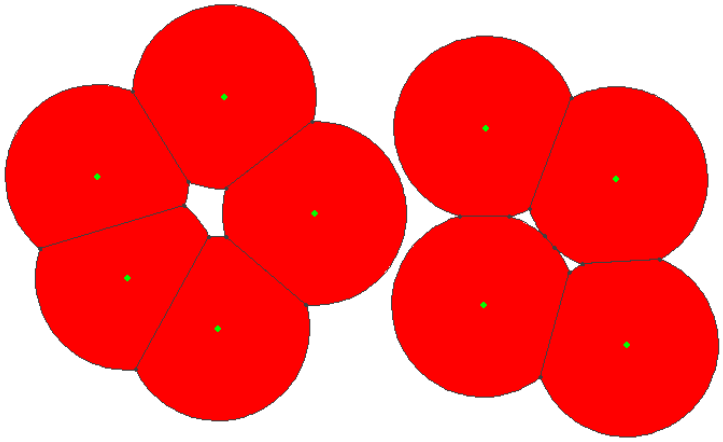

Figura 3.1: $Vor(P, r)$ con 2 componentes conexas y varios agujeros.

3.1.2. Estructura combinatoria de los diagramas

Para describir $Vor(P, r)$ para un r concreto y dado que este es una descomposición del plano, conviene considerar la estructura combinatoria de este conjunto (número de componentes conexas, número de agujeros, etc.).

Aunque la variación de los valores de r desde 0 hasta $+\infty$ es continua, existe un número finito de valores de r que generan cambios combinatorios entre los diversos $Vor(P, r)$. Estos cambios combinatorios, también llamados eventos, son de cuatro tipos, y están enumerados a continuación:

1. Conexión de dos componentes conexas en $Vor(P, r)$

   Proposición 3.1.2 *Cuando r alcanza la mitad de la longitud de una arista p_ip_j del árbol generador mínimo euclídeo de P, $EMST(P)$, se conectan dos de las componentes conexas de $Vor(P, r)$.*

   Demostración: Es inmediato ver que cuando r alcanza la mitad de la distancia entre dos puntos p_i y p_j sus regiones $Vor(p_i, r)$ y $Vor(p_j, r)$ quedan conectadas entre sí, por lo que lo único que nos queda demostrar es que p_i y p_j estaban en distintas componentes conexas antes de esto. Para ello, si suponemos que p_i y p_j estaban en la misma componente conexa, llegaremos a una contradicción. Para empezar, tendría que haber un camino entre p_i y p_j que pasara únicamente por aristas de tamaño menor a $2r$ y, si esto fuera cierto, la arista p_ip_j no estaría en el $EMST(P)$ por formar ciclo. Por lo tanto si $p_ip_j \in EMST(P)$, significa que p_i y p_j estaban en distintas componentes conexas en $Vor(P, r')$ $\forall r' < r$. □

   En la Figura 3.2 se puede ver un ejemplo de un evento de este tipo.

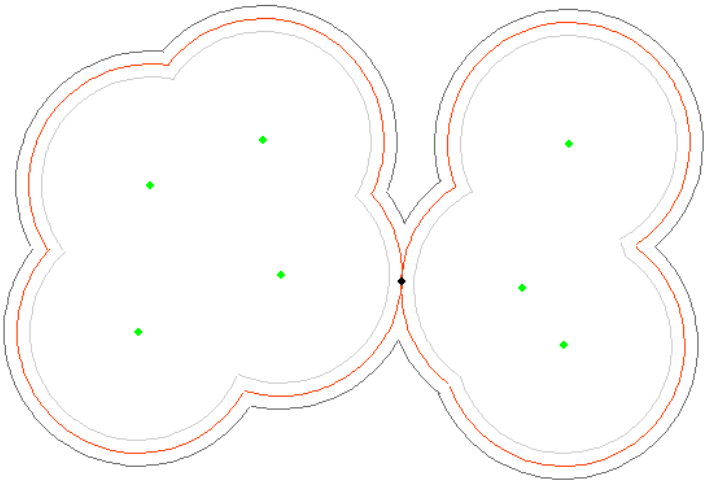

Figura 3.2: Unión de dos componentes conexas en $Vor(P, r)$.

2. Generación de un nuevo agujero en $Vor(P, r)$

   Proposición 3.1.3 *Cuando r alcanza la mitad de la longitud de una arista p_ip_j de Gabriel que no pertenece al $EMST(P)$ (una arista de Delaunay que corta a su dual de Voronoi pero no pertenece al $EMST(P)$) se produce un agujero en el diagrama.*

   Demostración: Para algún r menor que la mitad de la longitud de la arista p_ip_j que no pertenece a $EMST(P)$, p_i y p_j ya están en la misma componente conexa en $Vor(P, r)$, porque existe un camino en el $EMST(P)$ que los une con aristas de longitud estrictamente inferior a $2r$. Por otra parte, si la arista p_ip_j de Delaunay corta a su arista dual de Voronoi, significa que los circuncentros de los dos triángulos adyacentes a la arista de Delaunay p_ip_j están uno a cada lado de esta (son los dos extremos de la arista de Voronoi). Cuando r alcanza la mitad de la longitud de la arista p_ip_j, $Vor(P, r)$ contiene tanto la arista p_ip_j como el camino entre p_i y p_j que pasa por aristas del $EMST(P)$ formando un ciclo, pero no contiene un punto interior como es el circuncentro del triángulo que queda encerrado por el ciclo, y que está en el interior de este, pero a una distancia mayor que r de p_i, p_j y cualquier otro punto de P (por ser la triangulación de Delaunay). Tenemos así que, al cerrarse la arista p_ip_j en $Vor(P, r)$, queda un conjunto de puntos interiores que no pertenecen a $Vor(P, r)$, es decir, un agujero. □

   En la Figura 3.3 se puede apreciar un ejemplo del momento en el que aparece un agujero en $Vor(P, r)$.

3. Desaparición de un agujero en $Vor(P, r)$

   Proposición 3.1.4 *Cuando r alcanza el radio de la circunferencia circunscrita a un triángulo de Delaunay cuyo centro se encuentra dentro del*

Figura 3.3: Aparición de un agujero en $Vor(P, r)$.

triángulo, es decir, de un triángulo de Delaunay acutángulo, desaparece uno de los agujeros de $Vor(P, r)$ *y aparece un nuevo vértice de Voronoi.*

Demostración: El radio de la circunferencia circunscrita del triángulo es mayor que la mitad del lado mayor de este (a no ser que el circuncentro estuviera en la frontera del triángulo, en cuyo caso, serían iguales). Cuando r alcanza la mitad de la longitud del mayor de los lados, los tres lados del triángulo quedan contenidos en $Vor(P, r)$, pero se crea un agujero que consiste exclusivamente en cierta porción del interior del triángulo. El circuncentro del triángulo es uno de los puntos incluidos en el agujero, por ser interior al triángulo pero estar a una distancia mayor que r de sus tres vértices. Cuando r alcanza la longitud del radio de la circunferencia circunscrita, todo el interior del triángulo se encuentra a distancia menor o igual que r de alguno de sus tres vértices y, por tanto, todo el triángulo esta contenido en $Vor(P, r)$ y desaparece el agujero. Allí donde estaba el circuncentro aparece un vértice de Voronoi por ser este punto equidistante de los tres vértices del triángulo. □

En la Figura 3.4 se puede apreciar el momento en el que desaparece un agujero en $Vor(P, r)$.

4. Desaparición de un arco de circunferencia en la frontera de $Vor(P, r)$

Proposición 3.1.5 *Cuando r alcanza el radio de la circunferencia circunscrita a un triángulo de Delaunay cuyo centro se encuentra fuera del triángulo, es decir, de un triángulo de Delaunay obtusángulo, desaparece un arco de circunferencia en la frontera de* $Vor(P, r)$ *y aparece un nuevo vértice de Voronoi.*

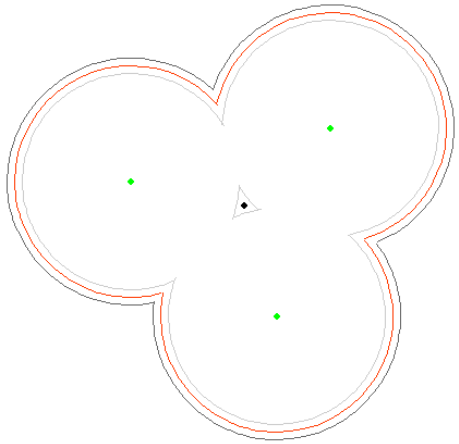

Figura 3.4: Desparición de un agujero en $Vor(P, r)$.

Demostración: Cuando r alcanza la longitud de la mitad del mayor de los lados del triángulo de Delaunay, los tres vértices delimitan tres arcos de circunferencia en la frontera de $Vor(P, r)$. Al incrementar el alcance r en $Vor(P, r)$, el arco correspondiente al vértice que forma un ángulo obtusángulo se va haciendo más pequeño por la parte que se acerca al circuncentro del triángulo. Cuando r finalmente alcanza el radio de la circunferencia circunscrita al triángulo, el arco desaparece justamente en el punto equidistante a los tres vértices del triángulo (el circuncentro) y, a partir de ese momento, cualquier incremento de r supone la inclusión de nuevos puntos en $Vor(P, r)$ que están más cerca de alguno de los otros dos vértices del triángulo que del vértice obtusángulo. Allí donde estaba el circuncentro aparece un vértice de Voronoi por ser este punto equidistante a los tres vértices del triángulo. □

En la Figura 3.5 se observa un ejemplo del momento en el que desaparece un arco en la frontera de $Vor(P, r)$.

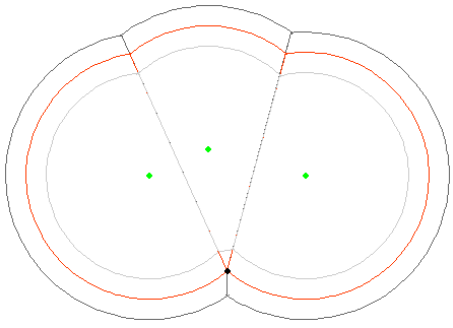

Figura 3.5: Desaparición de un arco de circunferencia en la frontera de $Vor(P, r)$.

En conclusión, hemos obtenido el siguiente resultado:

Proposición 3.1.6 *Dado un conjunto $P = \{p_1, \ldots, p_n\}$ de n puntos del plano, el número de diagramas de Voronoi de alcance limitado, $Vor(P, r)$, combinatoriamente distintos es $m = O(n)$ y los valores del alcance $r_1, \ldots, r_m$ en que se producen los cambios combinatorios en la estructura de $Vor(P, r)$ son los descritos en los lemas anteriores.*

3.1.3. Construcción de $Vor(P, r)$

En este apartado se estudia la forma de construir $Vor(P, r)$ para un radio r dado y de obtener $Vor(P, r_i)$ para todos los radios r_i que determinan algún cambio combinatorio en $Vor(P, r)$.

Proposición 3.1.7 *Para cualquier valor $r \geq 0$ y cualquier conjunto P de n puntos del plano, es posible calcular el diagrama de Voronoi de alcance r, $Vor(P, r)$, a partir del diagrama de Voronoi ordinario, $Vor(P)$, en tiempo $O(n)$.*

Demostración: Teniendo ya calculado $Vor(P)$, el cálculo de cada región $Vor(p_i, r)$ se puede realizar cortando la propia región $Vor(p_i)$ con el círculo $B(p_i, r)$, cosa que puede hacerse en tiempo proporcional a la complejidad de la región $Vor(p_i)$, dando lugar a un algoritmo de coste acumulado $O(n)$. □

Proposición 3.1.8 *El problema de obtener todos los diagramas $Vor(P, r_i)$ combinatoriamente distintos de un conjunto P de n puntos dados puede resolverse en tiempo $O(n \log n)$.*

Demostración: Se construyen la triangulación de Delaunay $Del(P)$ y el diagrama de Voronoi $Vor(P)$ en tiempo $O(n \log n)$ y se extraen los grafos $EMST(P)$ y $GG(P)$ en tiempo $O(n)$. A partir de aquí se pueden obtener los valores de todos los alcances $r_1, ..., r_m$ que producen cambios combinatorios en $Vor(P, r)$. Obsérvese que $m = O(n)$ y que los valores $r_1, ..., r_m$ se pueden obtener en tiempo $O(n)$ recorriendo las estructuras $Del(P)$, $Vor(P)$, $EMST(P)$ y $GG(P)$. Se ordenan estos valores en tiempo $O(n \log n)$ y a cada uno de ellos se le asigna el cambio combinatorio que le corresponde. Cada uno de estos cambios se puede insertar sucesivamente a partir del anterior en tiempo $O(\log n)$, de modo que el conjunto de todos ellos se construye en tiempo $O(n \log n)$. □

3.2. Caminos de desviación mínima

Una de las utilidades de los diagramas de Voronoi de alcance limitado es su aplicación a los caminos de desviación mínima. Los caminos de desviación mínima sirven para resolver ciertos problemas de optimización geométrica relacionados con el diseño de rutas que se desea que estén próximas a un conjunto de puntos del plano. Además de este criterio, es interesante encontrar la manera para escoger el camino cuya longitud sea menor de entre todos los caminos de desviación mínima.

3.2.1. Definiciones y propiedades

Dado un conjunto P de puntos del plano, y un par de puntos s y t de P, se desea obtener un camino entre s y t que no se aleje demasiado de los puntos de P. A continuación se precisa esta noción de cercanía.

Definición 3.2.1 *Llamamos desviación de un punto q del plano con relación a P a la mínima distancia de q a los puntos de P.*

$$dev(q, P) = \text{mín}\{dist(q, p_i) \mid p_i \in P\}$$

Definición 3.2.2 *Si Γ es un camino que conecta los puntos s y t, llamamos desviación de Γ respecto a P a la máxima desviación de los puntos del camino, es decir,*

$$dev(\Gamma, P) = \text{máx}\{dev(q, P) \mid q \in \Gamma\}$$

Definición 3.2.3 *Se llama desviación del par (s,t) con respecto a P a la mínima de las desviaciones de todos los caminos de s hasta t.*

$$dev(s, t, P) = \text{mín}\{dev(\Gamma_i, P) \mid \Gamma_i \textit{ camino de } s \textit{ hasta } t\}$$

Definición 3.2.4 *Una camino Γ entre s y t se dice que es de desviación mínima con respecto a P si $dev(\Gamma, P) = dev(s, t, P)$.*

Dicho de otra forma, si la desviación del par (s,t) respecto de P es r, un camino Γ entre s y t es de desviación mínima si para cualquier punto $q \in \Gamma$ se cumple que $dist(q, P) \leq r$.

3.2.2. Caracterización

Dado un conjunto de puntos P, la desviación mínima de todos los caminos entre s y t, puntos de P, depende de la longitud de una arista del árbol generador mínimo euclídeo del conjunto P, $EMST(P)$.

Definición 3.2.5 *Se llama arista crítica del par (s,t) con respecto a P a la arista de mayor longitud en el único camino que conecta s y t en $EMST(P)$. El punto medio de esta arista se llama punto crítico de (s,t) con respecto a P.*

Proposición 3.2.6 *Si e es la arista crítica del par (s,t) con respecto al conjunto P de puntos del plano entonces la desviación de (s,t) con respecto a P es:*

$$dev(s, t, P) = \frac{1}{2} long(e)$$

Demostración: Probaremos que todo camino de s a t con desviación mínima debe pasar por M, el punto medio de la arista crítica e.

Se construyen los discos con centro en cada punto de P y radio $r = \frac{1}{2} long(e)$. La unión de estos discos, C, tiene al punto M en su frontera. El interior de C tiene dos componentes conexas, una conteniendo a s y otra conteniendo a t, teniendo ambas el punto M en su borde. Así cualquier camino de s hasta t contenido en C tiene desviación r y pasa por M.

Además todo camino no contenido en C tiene algún punto con desviación mayor que r, por lo que r es $dev(s, t, P)$ como queríamos demostrar. □

En la Figura 3.6 podemos ver un ejemplo de camino de desviación mínima.

Proposición 3.2.7 *La desviación del par (s,t) con respecto al conjunto P de puntos del plano se puede calcular en $O(n \log n)$.*

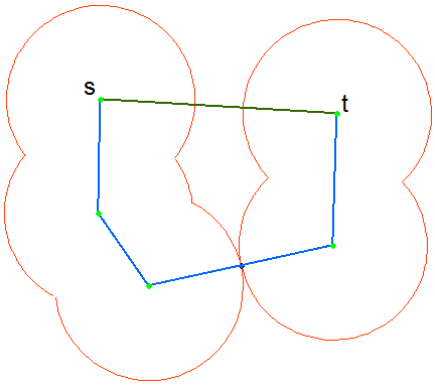

Figura 3.6: El camino azul es de desviación mínima, pero el camino verde no.

Demostración: El árbol generador mínimo, $EMST(P)$, se construye en tiempo lineal a partir de la triangulación de Delaunay de P. La arista de mayor longitud de $EMST(P)$ entre s y t se halla en tiempo lineal. Por tanto el tiempo total es $O(n \log n)$ marcado por el tiempo que se tarda en construir la triangulación de Delaunay. □

3.2.3. Minimización de la longitud

Todos los caminos de desviación mínima están contenidos en la región C unión de los discos centrados en los puntos de P y de radio r, la desviación mínima. Por tanto el camino de desviación y longitud mínimas entre s y t está también dentro de la región C.

Esta región C se puede poligonizar eliminando los segmentos circulares determinados por puntos de intersección de discos que sean consecutivos en el borde de C, pues el camino de longitud mínima nunca penetra en estos segmentos circulares, a excepción del caso en que el centro del disco correspondiente esté contenido en uno de dichos segmentos circulares. En ese caso el triángulo cuyos vértices son los dos puntos de intersección consecutivos y el centro del disco se mantiene formando parte del polígono y eliminando el resto del segmento circular. En la Figura 3.7 se puede ver la poligonización de C.

Llamemos D a la región poligonal así construida, unión de dos polígonos con un vértice común, el punto crítico M. Como C tiene como mucho $6n - 12$ vértices (ver [12]), la región poligonal D tiene $O(n)$ vértices y, además, puede tener agujeros (una cantidad lineal de ellos) según se muestra en la Figura 3.7.

Teorema 3.2.8 *Dado un conjunto de puntos P, el camino de desviación y longitud mínimas entre los puntos s y t de P puede construirse en tiempo $O(n \log n)$.*

Demostración: En primer lugar se construye la región C. Como es unión de n discos del mismo radio se puede realizar en $O(n \log n)$. La poligonización de C para obtener D se efectúa en tiempo lineal y la construcción del camino de longitud mínima en D se puede efectuar en $O(n \log n)$ (ver [11]). □

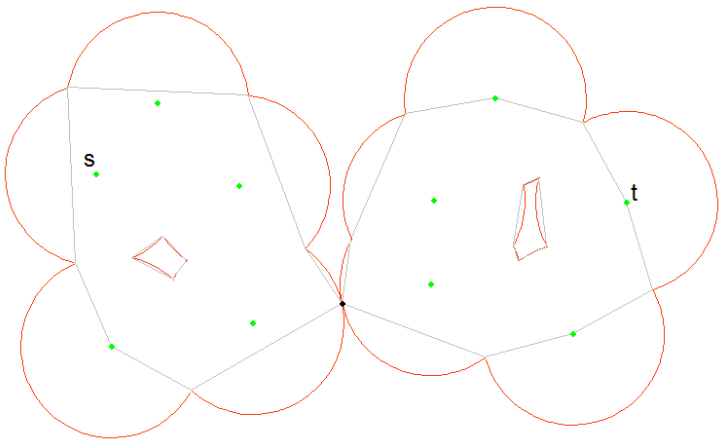

Figura 3.7: Poligonización de la región C.

3.3. Caminos de desviación mínima local

Nos interesan los caminos que conectan puntos del plano alejándose lo menos posible del conjunto P. Estos caminos son de desviación mínima. Esta situación ocurre, por ejemplo, cuando se quiere desplazar un receptor de un punto a otro dentro de una región en la que hay una red de antenas emitiendo una señal. Cuanto más próximos estén los puntos del camino a las antenas, mejor será la señal que reciba el receptor en su desplazamiento. En general, existen muchos caminos de desviación mínima entre dos puntos, por lo que tiene interés seleccionar entre todos ellos, los que satisfagan otras propiedades. En concreto nos interesamos por aquellos que, además de ser de desviación mínima, cualquier subcamino suyo hereda dicha propiedad. Es decir, todo subcamino es a su vez un camino de desviación mínima entre sus extremos. A estos caminos los llamamos de desviación mínima local.

3.3.1. Definiciones y propiedades

Los caminos de desviación mínima entres dos puntos pueden, en algunos casos, mantenerse a gran distancia de los sitios. Es por esto que surge, de forma natural, la definición de los caminos de desviación mínima local.

Definición 3.3.1 *El camino Γ es de desviación mínima local si, para cada par de puntos x e y de Γ, el camino $\Gamma_{|xy}$ es de desviación mínima, esto es si $dev(\Gamma_{|xy}, P) = dev(x, y, P)$.*

Definimos también a continuación los máximos locales estrictos de un camino, así como los puntos críticos y valores críticos de un conjunto de puntos P que servirán para futuras caracterizaciones de los caminos de desviación mínima local:

Definición 3.3.2 *Un punto $p \in \Gamma$ es un máximo local estricto de Γ si existe un entorno (a ambos lados) de p en Γ en el que todos los puntos tienen desviación estrictamente inferior a la desviación de p.*

Esta proposición se extiende de forma natural a intervalos de puntos del camino:

Definición 3.3.3 *Un intervalo (esto es, una porción conexa)* Γ' *de* Γ *es un máximo local estricto de* Γ*, si todos los puntos de* Γ' *tienen la misma desviación, y existe un entorno (a ambos lados) de* Γ' *en el que todos los puntos tienen desviación estrictamente inferior.*

De este modo, en ambos casos se tienen puntos a lo largo de Γ, a un lado y al otro de un máximo estricto, con desviación estrictamente inferior. En particular, los extremos del camino no son ni pertenecen a un máximo estricto.

En cuanto a los puntos y valores críticos de un conjunto de puntos P, se definen a continuación:

Definición 3.3.4 *Llamamos puntos críticos a los puntos medios de las aristas del árbol recubridor mínimo de* P*. Asimismo, llamamos valores críticos a la mitad de las longitudes de las aristas de dicho árbol, que consideramos ordenadas de forma creciente,* $r_i, ..., r_{n-1}$*.*

3.3.2. Caracterizaciones

Estamos ya en condiciones de ofrecer una caracterización de los caminos de desviación mínima local en términos de sus máximos locales.

Teorema 3.3.5 *Un camino simple* Γ *es de desviación mínima local si, y sólo si, todos sus máximos locales estrictos son o contienen un punto crítico.*

Demostración: Para demostrar este teorema, habría que definir toda una serie de conceptos no relacionados con este trabajo que harían que nos desviásemos del tema principal de esta sección. Si quiere ver la demostración al teorema, consúltese [5]. □

Una vez caracterizados los caminos de desviación mínima local en términos de sus máximos locales, podemos expresar que la desviación mínima local, es una propiedad local (valga la redundancia).

Teorema 3.3.6 *Un camino simple es de desviación mínima local si, y sólo si, es de desviación mínima.*

Demostración: Obviamente, todo camino de desviación mínima local es localmente de desviación mínima. En cuanto al recíproco, sea Γ un camino localmente de desviación mínima, y sean x e y dos puntos de Γ, supongamos que el camino $\Gamma_{|xy}$ no fuera de desviación mínima. Entonces tendría un máximo estricto que no coincidiría ni contendría un punto crítico. En ese caso, es trivial comprobar que Γ no sería localmente de desviación mínima en p. □

A continuación se dan algunos ejemplos de caminos de desviación mínima local.

Lema 3.3.7 *Dados dos sitios* $s = p_i$ *y* $t = p_j$*, el camino* Γ_1 *entre* s *y* t *contenido en el árbol generador mínimo de* P*, es de desviación mínima local.*

La Figura 3.8 ilustra un ejemplo del camino Γ_1 descrito en el Lema 3.3.7.

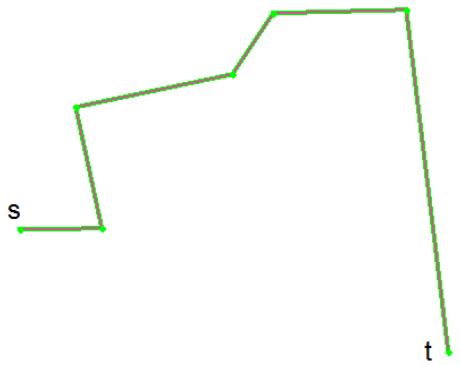

Figura 3.8: Camino de desviación mínima local descrito en el Lema 3.3.7(rojo) y $EMST(P)$(verde).

Lema 3.3.8 *Dados dos sitios $s = p_i$ y $t = p_j$, el camino Γ_2 entre s y t formado por los segmentos que unen los puntos medios de los segmentos de Γ_1, junto con los segmentos que unen sus extremos a s y t, es de desviación mínima local.*

La Figura 3.9 ilustra un ejemplo del camino Γ_2 descrito en el Lema 3.3.8.

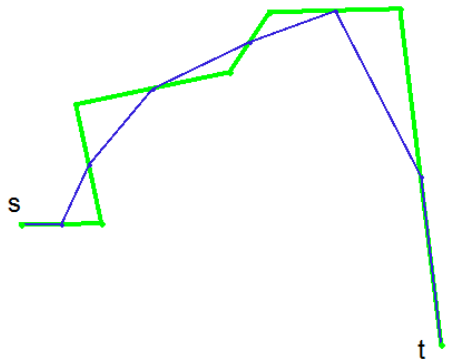

Figura 3.9: Camino de desviación mínima local descrito en el Lema 3.3.8(azul) y $EMST(P)$(verde).

3.3.3. El problema de decisión

Para determinar si un camino Γ compuesto por segmentos y arcos de circunferencia centrados en los puntos de un conjunto de puntos P, es o no de desviación mínima local, se presenta el siguiente teorema:

Teorema 3.3.9 *Sea Γ un camino simple formado por segmentos de recta y arcos de circunferencia de radios arbitrarios, centrados en los sitios de P y contenidos en la región de Voronoi correspondiente a su centro, y sea k la complejidad de Γ. Conocidos $Vor(P)$ y sus aristas críticas, es posible decidir en tiempo $O(nk)$ si Γ es o no de desviación mínima local.*

Demostración: El algoritmo de decisión comienza localizando el origen del camino en $Vor(P)$, es decir, hallando la región de Voronoi en la que se encuentra. A partir de aquí, se va recorriendo la zona de Γ, es decir, el conjunto de las regiones de Voronoi atravesadas por Γ, y se va comprobando, región por región, si Γ tiene máximos locales estrictos y si estos coinciden o incluyen puntos críticos.

El coste de este algoritmo es trivialmente $O(nk)$. □

Proposición 3.3.10 *La intersección de un camino poligonal de complejidad k de desviación mínima local con el 1-esqueleto del diagrama de Voronoi de un conjunto de n puntos tiene complejidad $O(nk)$ y existen ejemplos de complejidad $\Omega(nk)$.*

Demostración: La cota superior es obvia. La inferior se ilustra en la Figura 3.10. □

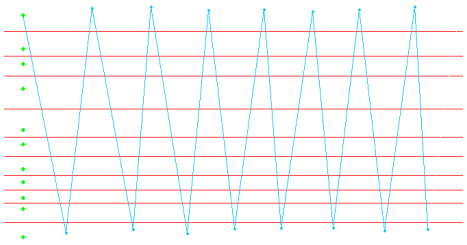

Figura 3.10: Camino de desviación mínima local que corta nk veces el 1-esqueleto del diagrama de Voronoi de P.

Así, pues, aunque no podemos asegurar que el problema de decisión tenga complejidad $\Omega(nk)$, sí podemos garantizar que cualquier algoritmo que pretenda resolverlo en tiempo $O(nk)$ deberá ser capaz de responder la cuestión sin explorar todos los puntos de corte del camino con las aristas de Voronoi de P.

Capítulo 4

Funcionalidades de la aplicación

En este capítulo se describen y analizan los algoritmos utilizados por la aplicación para calcular las distintas funcionalidades que ofrece , incluyendo el cálculo de los diagramas y grafos definidos en los capítulos anteriores y los algoritmos de decisión sobre los caminos introducidos por el usuario.

Cada sección profundiza en el funcionamiento de alguna de las funcionalidades de la aplicación, en los algoritmos implementados, y en los costes de estos algoritmos para que el lector conozca exactamente la forma en que la aplicación realiza los cálculos y muestra los resultados a partir de los datos suministrados (secciones 4.1 y 4.2).

Se presentan primero las funcionalidades básicas para comenzar a trabajar con nuestra aplicación. En primer las distintas maneras de introducir una nube de puntos en nuestra aplicación y, a continuación, se explica como trabajar y visualizar los distintos grafos de proximidad.

Una vez presentadas las funcionalidades básicas, se presentan las funcionalidades más específicas de nuestra aplicación, como son todas las relacionadas con los diagramas de Voronoi de alcance limitado y la introducción y tratamiento de caminos (secciones 4.3 y 4.4).

Por último, se presentan las funcionalidades orientadas a la personalización y adaptación de la aplicación por parte del usuario (secciones 4.5 y 4.6) y las de interacción mediante ficheros entre la aplicación y el usuario (sección 4.7).

4.1. Insertar puntos

En esta sección se presentan las tres principales maneras de introducir un conjunto de puntos en nuestra aplicación o de modificar uno ya existente. Se puede introducir o modificar una nube de puntos en nuestra aplicación a través de clics de ratón, mediante ficheros de puntos o generando un conjunto de puntos distribuidos aleatoriamente.

4.1.1. Por ratón

La manera más intuitiva y rápida de introducir puntos al conjunto P de puntos con el que trabaja la aplicación es mediante clics de ratón. En cualquier momento el usuario puede introducir un nuevo punto en la nube pulsando el botón izquierdo del ratón en la posición del plano donde quiera introducir el nuevo punto. Cuando esto sucede, la aplicación calcula las coordenadas del ratón en el momento del clic e incluye un nuevo punto en el vector de puntos cuyas coordenadas se extraen de la información leída del ratón. A continuación, se recalculan todos los grafos de proximidad y diagramas de manera local, como se ve más adelante en la sección 4.2.

4.1.2. Por fichero

Puede ser interesante para un usuario trabajar en distintas sesiones con el mismo conjunto P de puntos. También se puede dar el caso de que, por comodidad y, sobretodo, por necesidades de precisión, el usuario prefiera no introducir la nube de puntos mediante clics de ratón en la pantalla sino a través de las coordenadas de los mismos.

Por estos motivos, *DVALon* incluye la posibilidad de guardar y cargar ficheros de puntos. Los ficheros de puntos no tienen extensión concreta, de manera que queda a elección del usuario la extensión a asignar a este tipo de ficheros o incluso puede no asignarles extensión. Lo que sí tienen los archivos de puntos es un sencillo formato, que es común a todos los ficheros de puntos creados y leídos por la aplicación. Este formato es el siguiente:

1. Una línea al principio que contiene el número de puntos de la nube que se van a guardar en el fichero.

2. Para cada punto, una línea que contiene las coordenadas x e y de éste, separadas por un espacio.

Una vez se accede a esta funcionalidad, la aplicación abre una ventana de diálogo que permite al usuario navegar por su sistema de ficheros y escoger el fichero que guarda los puntos. Recordemos que este tipo de ficheros pueden tener cualquier extensión, con lo que es responsabilidad del usuario recordar la extensión utilizada, si es que decide utilizar alguna, al guardar los ficheros de puntos. Esto, que a priori podría parecer una carencia de la aplicación, en realidad tiene una gran ventaja.

Si comparamos el formato de los ficheros ".DATA" que guardan un proyecto (sección 4.7.2) y el descrito en esta sección para los ficheros de puntos, observamos que ambos empiezan de la misma manera: una línea para el número de puntos y tantas líneas como puntos haya con las coordenadas de estos separadas con espacios. Esta coincidencia de ambos formatos, unida a que los ficheros de puntos, como hemos dicho, no tienen extensión concreta, permite que la aplicación sea capaz de cargar una nube de puntos tanto de un fichero de puntos como de un fichero de un proyecto completo. Esto permite que, si en un momento determinado el usuario únicamente necesita la nube de puntos que tiene guardada en un proyecto, puede cargar también un fichero ".DATA"como si fuera un fichero de puntos, en cuyo caso la aplicación solo leerá la nube de puntos del fichero, obviando el resto de datos. Podría parecer absurda, entonces,

la existencia de ficheros de puntos pero, al contener mucha menos información, este tipo de ficheros ocupa mucho menos espacio que un fichero que guarda un proyecto completo. Además, nos ha parecido cómodo para el usuario facilitarle dos tipos de ficheros.

Una vez se ha seleccionado el fichero que se va a cargar, la aplicación primero comprueba que el formato del fichero sea el correcto. En caso de que lo sea, lee los puntos y rellena la estructura de la aplicación que guarda la nube de puntos con los datos leídos, devolviendo el resto de estructuras a su estado inicial. De esta manera, conseguimos empezar un nuevo proyecto a partir de una nube de puntos leída desde un fichero. A continuación, aparece un mensaje que confirma que los puntos han sido cargados correctamente. En caso de que el formato de los puntos sea incorrecto, aparece un mensaje de error que avisa al usuario de este hecho, y se cargan aquellos puntos que tengan formato correcto. Un fichero de puntos se considera que tiene formato incorrecto cuando alguno de los puntos presenta menos o más coordenadas de las esperadas (sólo deben tener coordenadas x e y separadas por espacios), si alguna coordenada tiene algún carácter no numérico, a excepción del punto para expresar valores decimales, o si falta la primera línea que indica el número de puntos que hay que leer a continuación.

En caso de que haya menos puntos de los especificados en la primera línea del fichero, se cargan todos los que haya igualmente. En cambio, si hay más de los declarados, los puntos en exceso son obviados por la aplicación a la hora de leer. En estos dos últimos casos, el programa avisa al usuario mediante una ventana emergente. Si el usuario decide modificar por su cuenta un fichero de puntos es responsabilidad suya asegurarse de que el número de puntos introducidos es el correcto. *DVALon* siempre guarda los archivos de puntos con formato correcto, así que solo pueden existir errores de formato en caso de que se modifiquen los ficheros manualmente.

4.1.3. Aleatoriamente

Esta funcionalidad permite al usuario generar una nube de puntos distribuidos aleatoriamente. Una vez el usuario ha accedido a ella, se abre una nueva ventana en la aplicación donde se puede elegir el número de puntos a generar y los vértices superior izquierdo e inferior derecho del rectángulo que los va a contener. Por defecto, el número de puntos a generar es cero (sería como crear un nuevo proyecto, ver sección 5.7.1) y los puntos superior izquierdo e inferior derecho son el $(0, 0)$ y el $(800, 600)$ respectivamente.

Una vez se han rellenado estos parámetros, si pulsamos "Aceptar" el proyecto actual desaparece para dar paso a un nuevo proyecto que contiene únicamente la nueva nube de puntos generada aleatoriamente en el rectángulo especificado. La aplicación vuelve a su estado inicial, es decir, todas las estructuras vacías excepto el vector de *DVALs*, que contiene un único *DVAL* con radio igual a 0 para que el usuario pueda trabajar con *DVALs* desde el principio sin necesidad de crear uno. En cambio, el vector de puntos, claro está, no queda vacío, sino que contiene la información de los puntos que se acaban de generar. Dicha información, se genera usando el método "Random" la clase "Math" de Java. Dicho método genera números pseudoaleatorios dentro del rango especificado. Por tanto, al crear una nube de puntos aleatoriamente distribuidos, la aplicación genera números con este método de la clase "Math" de Java, siempre dentro del rango especificado por el usuario en el rectángulo contenedor.

4.2. Grafos de proximidad

En esta sección se detalla el modo en que nuestra aplicación calcula la triangulación de Delaunay, grafo de Gabriel, grafo de vecindad relativa y árbol generador mínimo euclídeo. Para cada uno de ellos, se describe y justifica el algoritmo implementado y se hace un análisis de su coste en tiempo y en espacio.

4.2.1. Triangulación de Delaunay

Tal como se explica en la Sección 2.2.2, una triangulación es de Delaunay si todos sus triángulos son localmente de Delaunay, y esto sucede si para cada par de triángulos $p_ip_jp_k$ y $p_ip_jp_l$ que comparten una arista $\overline{p_ip_j}$, se cumple que p_l es exterior al círculo por $p_ip_jp_k$ y que p_k no está en el círculo de $p_ip_jp_l$.

Para calcular la triangulación de Delaunay de una nube de puntos P, en adelante $Del(P)$, se ha implementado para la aplicación un algoritmo incremental. A cada iteración dicho algoritmo ejecuta esencialmente dos funciones. La primera consiste en localizar el nuevo punto en la triangulación construida hasta el momento, y la segunda es la actualización propiamente dicha de la triangulación. Al final de cada iteración, la triangulación resultante es la triangulación de Delaunay del conjunto de puntos considerados hasta el momento. Para la etapa de localización, el algoritmo utiliza un triángulo contenedor de todos los puntos de la nube y un punto interior a este triángulo que denominamos auxiliar y denotamos por q y que facilita las tareas de localización dentro de la triangulación. El triángulo contenedor es lo suficientemente grande como para que no se produzcan errores en la interpretación de si un triángulo interior es o no localmente de Delaunay, dada la precisión con la que se pueden introducir nuevos puntos en la nube utilizando el ratón.

La triangulación $Del(P)$ se guarda en una DCEL (véase la Sección 2.3.1) que consta de una lista de triángulos (con un apuntador a una semiarista de cada triángulo), una lista de vértices (con un apuntador a una semiarista incidente en cada vértice), y una lista de semiaristas (con apuntadores a su vértice origen, a sus semiaristas siguiente, anterior y gemela, y a su cara incidente). Dado que se guarda una cantidad de información constante para cada elemento, la estructura utilizada para guardar $Del(P)$ ocupa un espacio de tamaño $O(n)$.

Tal como se ha dicho, el algoritmo construye $Del(P)$ de forma incremental, empezando desde un estado inicial $Del_0(P)$ que se corresponde con el triángulo contenedor e introduciendo un nuevo punto p_i cada vez sobre el estado $Del_{i-1}(P)$.

Para introducir un punto p_i en la triangulación, el algoritmo realiza los siguientes pasos a partir del estado anterior $Del_{i-1}(P)$:

1. Localización

   Identifica el triángulo de $Del_{i-1}(P)$ que contiene el nuevo punto p_i introducido. Para conocer cuál es este triángulo, se considera el segmento entre p_i y el punto auxiliar q (que está en un triángulo conocido de $Del_{i-1}(P)$). Se utiliza la DCEL que guarda la triangulación para recorrer la triangulación a partir de q encontrando los puntos de intersección del segmento con los lados de los triángulos por los que pasa en su recorrido hasta p_i. En el peor de los casos, el segmento corta todos los triángulos de $Del_{i-1}(P)$

y, por tanto, el coste de la localización de p_i es $O(n)$. Es sabido, sin embargo, que para conjuntos de puntos distribuidos de forma uniforme en un rectángulo, el coste esperado de esta etapa es $O(\sqrt{n})$ (véase [7]).

Debemos aclarar que el punto auxiliar utilizado en la localización no es estático. Va desplazándose a los circuncentros de los triángulos por los que pasa para evitar detectar intersección con varios triángulos simultáneamente (puntos de la nube que son vértices de 3 triángulos o más). De esta manera, el punto auxiliar se mueve por la DCEL mientras va buscando el triángulo sobre el que está el nuevo punto introducido para acabar situándose en este al final de la búsqueda. En la Figura 4.1 se ilustra el proceso de localización.

Es relevante destacar que al realizar cualquier modificación en la nube de puntos, hay que asegurarse de que la posición del punto auxiliar se actualiza, pues este punto sólo es útil mientras sepamos con seguridad en qué triángulo de la DCEL está.

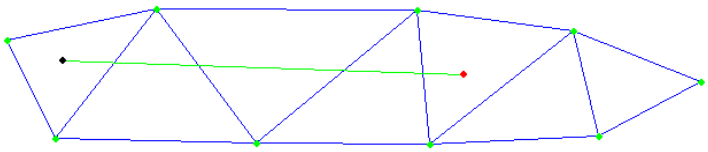

Figura 4.1: Ejemplo de localización.

2. Creación de nuevos triángulos

   Modifica la DCEL de $Del_{i-1}(P)$: desaparece el triángulo que contiene el punto p_i y se crean tres nuevos triángulos. Cada uno de los nuevos triángulos está formado por uno de los lados del antiguo triángulo de $Del_{i-1}(P)$ y el punto p_i introducido. Este paso se realiza en tiempo constante. La conversión se ilustra en la Figura 4.2.

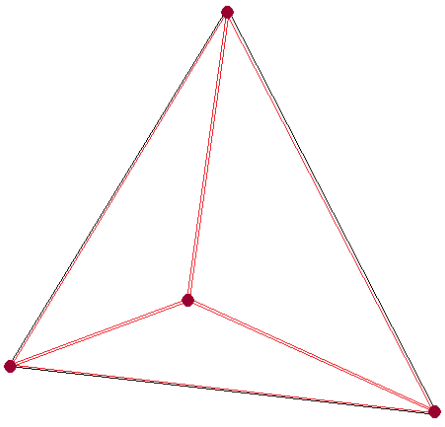

Figura 4.2: Creación de nuevos triángulos tras introducir un punto p_i.

3. Flips

   Tras la introducción del nuevo punto y la creación de los nuevos tres triángulos, una serie de triángulos de $Del_{i-1}(P)$ pueden no cumplir la condición local de Delaunay. Por ello se comprueba si los nuevos triángulos son localmente de Delaunay, y en caso de que no lo sean, se realizan *flips de Delaunay* recursivamente para todos los triángulos incidentes en p_i. El número de comprobaciones y *Flips* estimado es del orden del grado que tendrá p_i en el estado $Del_i(P)$, que en el peor de los casos, puede ser n, pero en promedio de todos los puntos, es < 6. Por lo tanto, el coste de este paso del algoritmo es en el peor de los casos $O(n)$, pero en promedio es $O(1)$.

Partiendo de $Del_{i-1}(S)$, la triangulación resultante después de aplicar estos tres pasos es $Del_i(S)$. Para una nube de puntos P, partiendo del estado $Del_0(P)$, el algoritmo realiza n iteraciones introduciendo un punto en la triangulación cada vez para acabar en el estado $Del_n(S)$.

Proposición 4.2.1 *Dada una nube de n puntos P, la aplicación calcula la triangulación de Delaunay de P, Del(P), de forma incremental en tiempo peor* $O(n^2)$ *y tiempo esperado* $O(n^{3/2})$*, y espacio* $O(n)$*.*

Una vez la triangulación de Delaunay está calculada, el usuario puede producir alteraciones indirectamente en esta mediante modificaciones en la nube de puntos. Así pues, si el usuario introduce nuevos puntos en la nube, o mueve puntos ya existentes, o elimina puntos de la nube, la aplicación utiliza algoritmos para reconstruir la triangulación de Delaunay. Estos algoritmos se describen a continuación.

Cuando el usuario introduce un nuevo punto p_i en la nube, la aplicación calcula la nueva triangulación de Delaunay, $Del(P \cup \{p_i\})$, a partir de la anterior, $Del(P)$, de la forma descrita anteriormente, esto es, realizando los mismos tres pasos que en una iteración del algoritmo incremental.

Proposición 4.2.2 *Dada una triangulación de Delaunay* $Del(P)$ *cualquiera, la nueva triangulación de Delaunay* $Del(P \cup \{p_i\})$ *resultante tras la adición de un nuevo punto* p_i *se calcula en tiempo peor* $O(n)$ *y tiempo esperado* $O(\sqrt{n})$ *a partir de la anterior.*

Cuando el usuario mueve un punto de la nube, la aplicación comprueba si los triángulos que han sido modificados (todos los incidentes en el punto que ha sido movido) siguen cumpliendo la condición local de Delaunay o si, por el contrario, alguno ha perdido esta propiedad. Esta comprobación tiene coste constante para cada triangulo y, por lo tanto, del orden del número de triángulos incidentes al punto, esto es, del orden de su grado como vértice de la triangulación. En el peor de los casos todos los triángulos de $Del(P)$ son incidentes al punto y la comprobación tiene un coste $O(n)$. Pero el número de triángulos incidentes a un punto es en promedio < 6 y, suponiendo que todos los puntos de la nube tienen las mismas posibilidades de ser movidos, esta comprobación tendrá de media coste constante.

Debemos tener en cuenta que cuando el usuario mueve un punto de la nube, las comprobaciones y reacciones se realizan en tiempo real por cada pequeña variación de la posición para que $Del(P)$ se vaya recalculando y redibujando

según el usuario realiza el movimiento. Por esta razón, las variaciones son casi continuas y las actualizaciones de la $DCEL$ se realizan en el tiempo en que los cambios de $Del(P)$ se producen.

Cuando la aplicación detecta que la triangulación ha dejado de ser localmente de Delaunay, realiza el *Flip de Delaunay* correspondiente al cuadrilátero que lo ha provocado. El número de $flips$ después de una variación es de orden $O(1)$, pero como el algoritmo debe mirar todos aquellos triángulos incidentes al punto desplazado, su coste en el peor de los casos en $O(n)$, mientras que su coste esperado es $O(1)$.

Proposición 4.2.3 *Dada una triangulación de Delaunay de una nube de puntos $Del(P)$, la aplicación recalcula $Del(P)$ después de una variación casi continua de uno de los puntos de la nube en tiempo peor $O(n)$ y tiempo esperado $O(1)$.*

Cuando el usuario elimina un punto p_i de la nube, la aplicación calcula el nuevo $Del(P - \{p_i\})$ recalculando el grafo de la forma descrita al principio de este apartado, es decir, partiendo desde el triángulo contenedor y recalculando incrementalmente el diagrama ex-novo.

Proposición 4.2.4 *Dada una triangulación de Delaunay de una nube de puntos $Del(P)$, si se elimina uno de los puntos p_i de la nube, la aplicación recalcula $Del(P - \{p_i\})$ en tiempo peor $O(n^2)$ y tiempo esperado $O(n^{3/2})$.*

Somos conscientes de la posibilidad de implementar un algoritmo que realice este cálculo localmente en tiempo menor, y este podría ser un aspecto de mejor potencial del proyecto. Teniendo presente la propiedad de localidad, el hecho de eliminar un punto p_i de la nube sólo afecta a los triángulos incidentes a dicho punto, y el resto de triángulos de $Del(P)$ se mantienen en la nueva triangulación $Del(P-\{p_i\})$. Por lo tanto, sabiendo que sólo es necesario retriangular una parte de $Del(P)$, vemos que el tiempo que tardaría el nuevo algoritmo en hacerlo sería en cualquier caso menor o igual al tiempo que tarda el actual algoritmo en recalcularlo todo. El coste del algoritmo sería proporcional al grado del punto p_i eliminado, puesto que se trata de triangular un polígono estrellado del que se conoce un punto del núcleo (el punto p_i eliminado). En el peor de los casos, si el grado de p_i fuera n, el coste del algoritmo sería de orden $O(n)$, pero el coste esperado con un grado promedio por punto de la nube < 6 sería de orden $O(1)$. Sin embargo, tal como se puede constatar con el uso de la aplicación, el coste actual del borrado de un punto no produce el más mínimo efecto visual de retraso, probablemente a causa de que esta aplicación se utiliza, en general, para visualizar la triangulación de Delaunay de decenas o, a lo sumo, centenares de puntos.

La visualización de una triangulación de Delaunay limitada por un radio r no necesita mayores cálculos que los de la triangulación de Delaunay ordinaria, ya que teniendo la triangulación de Delaunay sin limitaciones antes calculada, basta con visualizar sólo aquellas aristas cuya longitud sea $\leq r$.

Debemos mencionar también en este apartado, que la triangulación de Delaunay es el grafo que se encuentra en la primera capa de la aplicación. Con esto queremos decir que todos los demás grafos se calculan o bien a partir de este, o bien a partir de otro grafo calculado a partir de este. Es, por tanto, manifiesta la importancia que tiene que este grafo esté bien calculado y que se calcule en el

menor tiempo posible para no propagar retrasos en el resto de grafos calculados posteriormente.

4.2.2. Grafo de Gabriel

Como hemos visto en la sección 2.2.3, el grafo de Gabriel del conjunto de puntos P, $GG(P)$, es un subgrafo de la triangulación de Delaunay, $Del(P)$. Esto permite implementar un algoritmo que calcule eficientemente el grafo de Gabriel, ya que únicamente se debe recorrer la triangulación de Delaunay previamente calculada y, para cada una de sus aristas, comprobar si cumple la condición de Gabriel descrita en la sección 2.2.3 que, recordemos, caracteriza una arista de Gabriel como aquella tal que el circulo con centro en el punto medio de la arista y diámetro la longitud de la arista no contiene ningún otro punto de P.

En concreto, el algoritmo implementado en esta aplicación para calcular el grafo de Gabriel va recorriendo las aristas de $Del(P)$ una a una. Cada vez que visita una arista, la marca como vista para no volver a visitarla en el futuro y marca también como vista su arista gemela en la triangulación, puesto que el grafo de Gabriel no es dirigido y no es necesario mirar las aristas gemelas de aristas ya visitadas. Si la arista a tratar está marcada como visitada o es del triángulo contenedor, el algoritmo no hace nada. En cambio, si la arista aún no ha sido visitada basta con mirar si se cumple la condición de Gabriel. Para ello, es suficiente mirar sólo los dos puntos que forman cara con los vértices de la arista, como se demuestra a continuación.

Demostración: El círculo el cual debemos comprobar si está vacío o no es el círculo con centro en el punto medio de la arista y diámetro la longitud de la arista, llamado habitualmente círculo de Gabriel de la arista. Esta arista tiene un triángulo de Delaunay incidente a cada lado. Tomando uno de ellos, por ejemplo el superior en la Figura 4.3, si el tercer vértice de este es exterior al círculo de Gabriel entonces el ángulo que forma con la arista es menor de 90 grados y, por tanto, el circuncentro del triángulo está por encima de la arista y contiene el círculo que usamos para comprobar la condición de Gabriel. Por tanto, podemos concluir que el circuncírculo contenedor del triángulo contiene el semicírculo superior de Gabriel, por lo que necesariamente éste semicírculo no puede contener puntos, ya que en caso contrario la triangulación no sería de Delaunay. Esta situación está ilustrada en la Figura 4.3. La demostración para el otro punto adyacente a la arista es equivalente. □

Así, pues, el algoritmo comprueba, para la arista que estamos tratando, si los dos vértices de los triángulos incidentes en la arista pertenecen o no a su círculo de Gabriel. Para ello calcula el punto medio de la arista, al que llamaremos m, y la mitad de su longitud, valor al que llamamos d. A continuación, comprueba que ninguno de los dos puntos a comprobar esté a distancia menor o igual a d del punto m. Si esto se cumple, selecciona la arista. En caso contrario, queda descartada.

Si la arista ha sido seleccionada, se incluye en la lista de adyacencia (sección 2.3.2 de esta documentación) que guarda el grafo de Gabriel. Para cada uno de los dos vértices de la arista, se incluye el otro en su lista de adyacencias. Concretamente, en la lista de adyacencias de cada vértice se guarda, en cada posición, el identificador de uno de los vértices adyacente y la longitud de la arista a la que pertenecen.

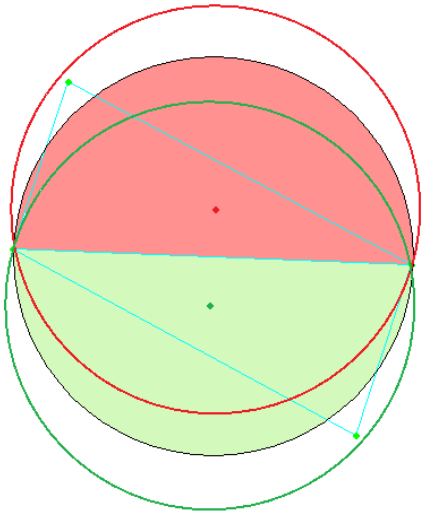

Figura 4.3: Círculo de Gabriel de una arista perteneciente a $GG(P)$.

Proposición 4.2.5 *Dada una triangulación de Delaunay, $Del(P)$, la aplicación construye el grafo de Gabriel de P a partir de $Del(P)$ en tiempo $O(n)$.*

Es inmediato comprobar la validez de esta proposición ya que, como hemos visto, el algoritmo que calcula el grafo de Gabriel recorre una única vez cada una de las aristas de $Del(P)$, marcando como vistas tanto la arista en cuestión como a su gemela para que no vuelvan a ser visitadas. Por tanto, recorre las m aristas de $Del(P)$. Para cada una de ellas solo realiza dos comprobaciones, y en cada comprobación realiza las operaciones descritas anteriormente. Por tanto, el coste de las comprobaciones para cada arista es constante. El algoritmo tiene, pues, coste proporcional al número de aristas de $Del(P)$. Tal como se ha visto en la Sección 4.2.1, $Del(P)$ es un grafo de complejidad $O(n)$, en el que el número de aristas es $a \leq 3n - 6$.

La aplicación calcula siempre el grafo de Gabriel siguiendo este algoritmo, independientemente de que el usuario decida o no visualizarlo o decida visualizarlo de manera completa o limitada por algún alcance. Si el usuario decide no visualizarlo, la aplicación simplemente no muestra las aristas del grafo. En cambio, si el usuario quiere visualizar el grafo, la aplicación mostrará todas las aristas, en caso de se quiera visualizar el grafo de Gabriel ordinario. Si el usuario prefiere ver el grafo de Gabriel de alcance limitado, la aplicación solo muestra aquellas aristas que tengan una longitud menor o igual alcance.

4.2.3. Grafo de vecindad relativa

El grafo de vecinos relativos del conjunto de puntos P, $RNG(P)$, se calcula de manera similar al grafo de Gabriel. Como se ha visto en la sección 2.2.4, el RNG es un subgrafo del grafo de Gabriel, con lo que es posible calcularlo a partir de éste. Para ello la aplicación recorre la lista de adyacencias que contiene el GG y para cada adyacencia comprueba si se cumple la condición para que

una arista forme parte del grafo de vecinos relativos descrita en la sección 2.2.4. Una vez el algoritmo ha comprobado una adyacencia, no vuelve a comprobarla en el futuro, como tampoco comprobará la misma adyacencia vista desde la lista del otro vértice.

En el caso del RNG, a diferencia de lo que sucede con el GG, para saber si una arista cumple la condición para ser una arista del grafo de vecinos relativos es necesario comprobar si alguno de los puntos de la nube está en la lente intersección de las circunferencias con centro en cada uno de los vértices de la arista y radio la longitud de la arista. Es necesario comprobar todos los puntos, ya que existen ejemplos que demuestran la no localidad de la condición. Por tanto, para cada adyacencia de la lista visitada, es necesario visitar el resto de puntos de la nube y, para cada uno de ellos, la aplicación comprueba si el punto se encuentra a distancia menor o igual a la longitud de la arista de sus dos vértices. Si esto sucede, la arista no pertenece al grafo de vecinos relativos y es descartada. En caso contrario se incluye la arista en la lista de adyacencias (sección 2.3.2 de esta documentación) en la que la aplicación guarda el grafo de vecinos relativos. En la lista de adyacencias de cada vértice se guarda, en cada posición, el identificador de uno de los vértices adyacentes y la longitud de la arista a la que pertenecen.

Proposición 4.2.6 *Dado un grafo de Gabriel, $GG(P)$, la aplicación construye el grafo de vecinos relativos de P a partir de $GG(P)$ con un coste de $O(n^2)$.*

Esta proposición se cumple ya que, para cada una de las m aristas del grafo de Gabriel de P, se recorren los n puntos de P, y para cada uno de ellos se ejecuta un cálculo de coste constante.

DVALon calcula siempre el grafo de vecinos relativos siguiendo este algoritmo, independientemente de que el usuario decida o no visualizarlo o decida visualizarlo de manera completa o limitada por algún alcance. Si el usuario decide no visualizarlo, la aplicación simplemente no muestra las aristas del grafo. En cambio, si el usuario quiere visualizar el grafo, la aplicación mostrará todas las aristas, en caso de se quiera visualizar el RNG ordinario. Si el usuario prefiere ver el grafo de vecinos relativos de alcance limitado, la aplicación solo muestra aquellas aristas que tengan una longitud menor o igual al alcance seleccionado.

4.2.4. Árbol generador mínimo euclídeo

Para la construcción del árbol generador mínimo del conjunto de puntos P, $EMST(P)$, la aplicación primero selecciona las aristas de la triangulación de Delaunay que va a tratar, puesto que no todas son válidas. Por tanto, descartamos las aristas que tienen algún vértice que pertenece al triángulo contenedor de la triangulación y las gemelas del resto de aristas. Una vez tenemos las aristas que vamos a tratar, el algoritmo utilizado para encontrar el $EMST(P)$ es el algoritmo de Prim, descrito en la sección (2.3.3). Se ha decidido construir el $EMST(P)$ a partir de la triangulación de Delaunay y no del grafo de vecindad relativa, a pesar de que es subgrafo de éste último, porque para otras funcionalidades de la aplicación relacionadas con los diagramas de Voronoi de alcance limitado, nos es útil también saber qué aristas de Delaunay no forman parte del $EMST(P)$ a la hora de detectar los eventos de tipo 3 y 4, (sección 4.3.4), y aprovechamos las construcción del propio $EMST(P)$ para obtener esta información simultáneamente.

En concreto, nuestro algoritmo primero ordena las aristas a tratar (todas las de $Del(P)$ salvo las tres que forman el triángulo contenedor y las que inciden en sus vértices) por orden creciente de peso (longitud). Una vez hecho esto, añade la arista de menos peso a la lista de adyacencias donde la aplicación guarda el $EMST(P)$ y aplica el algoritmo de Prim. Esto es, mientras queden aristas por visitar, si la arista actual no forma ciclo con las ya incluidas en el $EMST(P)$ y es adyacente a ellas, la incluimos y la marcamos para no volver a visitarla, además de volver a situar el puntero de nuevo al principio de las aristas a tratar, para volver a comenzar por la arista de menos peso no seleccionada en la siguiente iteración. Si no, pasamos a la siguiente.

Para saber si quedan aún aristas por tratar es suficiente con comparar en cada iteración la posición del puntero que usamos para movernos por el vector de aristas con el tamaño de dicho vector, operación que se realiza en tiempo constante. Si el puntero es mayor o igual al tamaño del vector de aristas significa que ya no quedan aristas por tratar. Esto es correcto ya que, como se ha descrito, el puntero siempre está dentro de los límites del vector mientras queden aristas por incluir en el $EMST(P)$, ya que cada vez que se incluye una nueva arista en el mismo, se devuelve el puntero al principio del vector para que vuelva a recorrerlo entero en la siguiente iteración. Esto significa que el puntero únicamente puede superar el tamaño del vector cuando ya no encuentre ninguna arista que pueda ser incluida en el $EMST(P)$, y esto solo es posible si ya no quede ninguna, ya que al recorrerlas todas siempre encuentra una que cumpla las condiciones para ser incluida a no ser que ya no exista ninguna.

Para realizar las comprobaciones sobre si una arista es adyacente a las ya incluidas y si forma ciclo con las ya incluidas, se usa un vector de booleanos auxiliar, cuyo tamaño es igual al número de vértices de P, y que nos dice si un vértice ha sido ya incluido o no en el $EMST(P)$. Cada vez que incluimos una arista en el $EMST(P)$, marcamos como incluidos sus dos vértices en este vector. De esta manera, cada vez que vayamos a tratar una nueva arista podemos saber en tiempo constante si sus vértices ya han sido incluidos o no en el $EMST(P)$. Por lo tanto, para mirar si una arista forma ciclo con el resto de las aristas incluidas en el $EMST(P)$ basta con mirar si sus dos vértices están marcados en este vector auxiliar. Si ambos lo están, significa que si añadimos la arista al $EMST(P)$ formaremos ciclo. Para saber, en cambio, si la arista que estamos tratando es adyacente a las ya incluidas en el $EMST(P)$ basta con mirar si uno de sus dos vértices ya está incluido, accediendo también a nuestro vector auxiliar. Por tanto, el algoritmo trata una arista siempre que no se cumpla la primera condición pero sí la segunda. Como vemos, ambas operaciones se realizan en tiempo constante.

Por lo tanto, para construir el $EMST(P)$, primero ordenamos las aristas por orden creciente de longitud. Hemos implementado para este propósito el algoritmo de selección descrito en la sección 2.3.4, cuyo coste es $O(n^2)$. Después, recorremos todas las aristas de $Del(P)$ y, para cada una de ellas, las comprobaciones a hacer se realizan en tiempo constante, con lo que esta etapa se lleva a cabo en tiempo $O(n)$. Por tanto, nuestro algoritmo calcula el $EMST(P)$ a partir de $Del(P)$ en tiempo $O(n^2 + n) = O(n^2)$.

Proposición 4.2.7 *La aplicación calcula el árbol generador mínimo de P a partir de la triangulación de Delaunay, $Del(P)$, en tiempo $O(n^2)$.*

Como en los casos de la triangulación de Delaunay y los grafos de Gabriel y

de vecinos relativos, nuestra aplicación calcula siempre el árbol generador mínimo siguiendo este algoritmo, independientemente de que el usuario decida o no visualizarlo, o decida visualizarlo de manera completa o limitada por algún alcance. Si el usuario decide no visualizarlo, la aplicación simplemente no muestra las aristas del árbol. En cambio, si el usuario quiere visualizar el árbol, la aplicación mostrará todas las aristas, en caso de se quiera visualizar el $EMST(P)$ ordinario, o aquellas que tengan una longitud menor o igual al alcance seleccionado, en caso de querer visualizar el $EMST(P)$ de alcance limitado.

4.3. Diagramas de Voronoi

En esta sección, analizamos detenidamente los algoritmos que la aplicación emplea para realizar los cálculos relacionados con el diagrama de Voronoi y los diagramas de Voronoi de alcance limitado así como para detectar los cambios combinatorios que se producen entre los distintos diagramas de Voronoi de alcance limitado cuando el valor del alcance, r, varía en la semirrecta real positiva.

4.3.1. Diagrama de Voronoi ordinario ($Vor(P)$)

Tal como se explica en la Sección 2.1, el diagrama de Voronoi de un conjunto finito P de puntos del plano, denotado $Vor(P)$, es la descomposición del plano en las regiones asociadas, por proximidad, a cada uno de los puntos de P. Dichas regiones se denominan regiones de Voronoi.

La aplicación calcula $Vor(P)$ a partir de la triangulación de Delaunay, $Del(P)$. Cada vez que se produce algún cambio en la nube de puntos y, por lo tanto, en $Del(P)$, el diagrama de Voronoi es recalculado completamente. No obstante, como $Vor(P)$ se calcula a partir de $Del(P)$, el hecho de que el cálculo de $Del(P)$ a cada cambio de la nube de puntos sea local, hace que $Vor(P)$ así como el resto de grafos que se calculan a partir de $Del(P)$ se calculen más rápidamente.

El diagrama de Voronoi $Vor(P)$, del mismo modo que la triangulación de Delaunay $Del(P)$, se guarda en una $DCEL$ (véase la Sección 2.3.1) que consta de una lista de caras poligonales (con un apuntador a una semiarista incidente que llamamos "semiarista inicial"), una lista de vértices de Voronoi (con un apuntador a una semiarista incidente), y una lista de semiaristas (con apuntadores a su vértice origen, a sus semiaristas siguiente y gemela, y a su cara incidente). Dado que se guarda una cantidad de información constante para cada elemento, la estructura utilizada para guardar $Vor(P)$ ocupa un espacio de tamaño $O(n)$.

El algoritmo que calcula $Vor(P)$ realiza una serie de transformaciones en la $DCEL$ de $Del(P)$ para obtener la $DCEL$ de $Vor(P)$. Los pasos que realiza para llevar a cabo la transformación son los siguientes:

1. Vértices de Voronoi

   Por la dualidad explicada en la Sección 2.1, los circuncentros de los triángulos de $Del(P)$ coinciden con los vértices de $Vor(P)$, de modo que la aplicación calcula el circuncentro de cada uno de los triángulos de $Del(P)$ e introduce un vértice de Voronoi en la $DCEL$ de $Vor(P)$ por cada uno de

ellos. Este paso tiene un coste proporcional al número de triángulos de $Del(P)$, que es $\leq 3n-6$ donde n es el número de puntos de P, es decir, el coste de este paso es de orden $O(n)$.

En la Figura 4.4 se puede observar cómo los circuncentros de los triángulos de $Del(P)$ son vértices en $Vor(P)$.

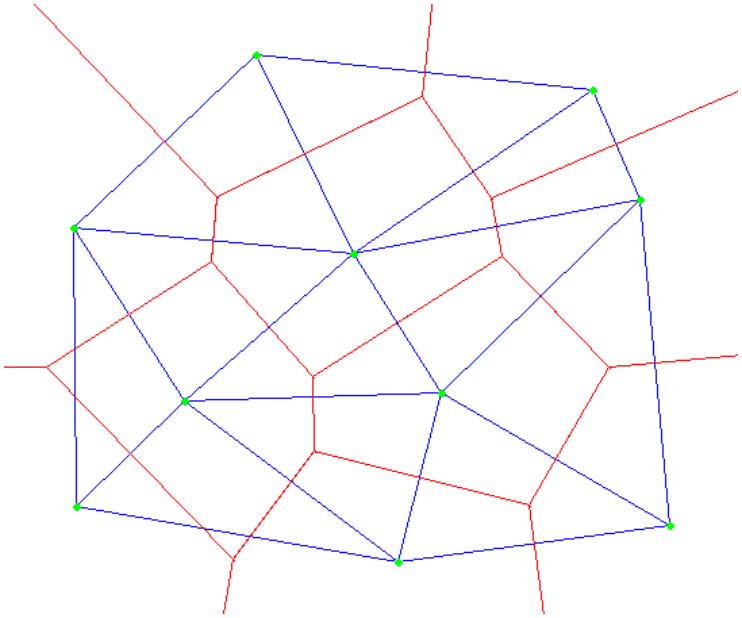

Figura 4.4: Triangulación de Delaunay (azul) y Diagrama de Voronoi (rojo).

2. Aristas de Voronoi

   La dualidad antes mencionada permite asociar cada arista de Voronoi con una arista de Delaunay y viceversa. Más concretamente, se corresponden la arista de Voronoi que une los circuncentros de dos triángulos adyacentes en $Del(P)$, t_1 y t_2, y la arista común en $Del(P)$ a t_1 y t_2. Gracias a esta dualidad podemos identificar las distintas aristas de Voronoi y su correspondencia con las aristas de Delaunay. Por cada semiarista de la $DCEL$ de $Del(P)$, conociendo el triángulo incidente de esta, t_1, y el de su gemela ,t_2, el algoritmo dualiza la semiarista para unir los vértices de Voronoi correspondientes a los circuncentros de t_1 y t_2 en la $DCEL$ de $Vor(P)$.

   En la Figura 4.5 puede verse un ejemplo de cómo se dualiza la información de las semiaristas de la $DCEL$ de $Del(P)$ a semiaristas de la $DCEL$ de $Vor(P)$.

   A diferencia de las aristas de $Del(P)$ que solo pueden ser segmentos, las aristas de $Vor(P)$ pueden ser segmentos, rectas o semirrectas. El algoritmo representa, sin embargo, las aristas siempre mediante dos puntos. En el caso de segmentos, estos dos puntos son el punto inicial y el punto final. En el caso de semirrectas, un punto es el punto inicial y el otro es un punto auxiliar en la dirección y sentido de la semirrecta. Por último, en el caso de

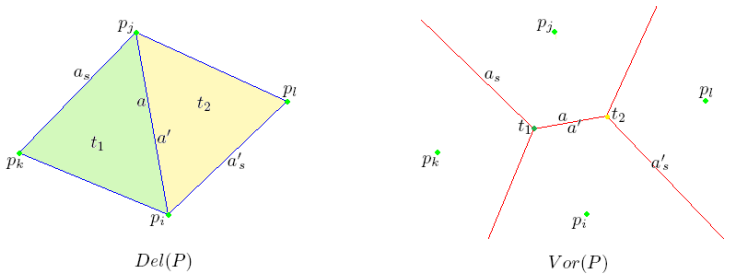

Figura 4.5: Identificadores de los elementos en la $DCEL$ de $Del(P)$ (izquierda) y su correspondencia con los elementos en la $DCEL$ de $Vor(P)$ (derecha).

rectas, los dos puntos son auxiliares y determinan la recta. Estos puntos se corresponden con vértices en la $DCEL$ de $Vor(P)$ tanto si se trata de vértices de Voronoi como de puntos auxiliares usados para determinar la dirección y el sentido de rectas y semirrectas. Como hemos visto, los puntos que son vértices de Voronoi son los circuncentros de los triángulos de la $DCEL$ que pertenecen a $Del(P)$. Los puntos auxiliares son también circuncentros de triángulos de la $DCEL$, pero que no pertenecen a $Del(P)$. Estos triángulos son aquellos en la $DCEL$ incidentes a alguno de los tres vértices del triángulo contenedor y por tanto que no existen realmente en la triangulación.

Este paso del algoritmo que adapta la información de las semiaristas de la $DCEL$ de $Del(P)$ a semiaristas en la $DCEL$ de $Vor(P)$ realiza un trabajo de coste constante por cada semiarista. Por lo tanto, el coste de la transformación de todas las semiaristas es proporcional al número de semiaristas que, a su vez, es proporcional al número de puntos de la nube, es decir, el coste de este paso es de orden $O(n)$.

3. Regiones de Voronoi

   Las caras poligonales resultantes, delimitadas por las aristas de Voronoi, son las regiones de Voronoi correspondientes a cada sitio. Por lo tanto, hay una cara en la $DCEL$ por cada punto de la nube, y su arista de Voronoi asociada corresponde a la que ya tenía el vértice de Delaunay correspondiente.

   Para realizar la transformación de este paso en la $DCEL$, el algoritmo crea por cada vértice v de la $DCEL$ de $Del(P)$ una cara en la $DCEL$ de $Vor(P)$ y le asigna como arista inicial a esta la que era arista incidente de v. La frontera de la región ya ha sido generada en el paso anterior al transformar las semiaristas de $Del(P)$ en semiaristas de $Vor(P)$. En este paso sólo hemos tenido que asignar una arista inicial a cada cara de Voronoi para poder recorrer la frontera de las regiones. Los cálculos realizados en la generación de cada cara tienen un coste constante, y el número de caras es proporcional al número de vértices, n, por lo que el coste de este paso del algoritmo es de orden $O(n)$.

Proposición 4.3.1 *La aplicación calcula $Vor(P)$ a partir de $Del(P)$ en tiempo $O(n)$ y espacio $O(n)$.*

El diagrama de Voronoi, además de ser una funcionalidad indispensable en la aplicación para sacar conclusiones sobre la forma en la que están distribuidos los puntos y conocer con rapidez cuál es el punto de la nube más cercano a un punto del plano dado, se utiliza también para cálculos posteriores relacionados con los diagramas de Voronoi de alcance limitado ($DVAL$), que requieren que $Vor(P)$ esté calculado. Veremos los algoritmos que utiliza la aplicación para calcular los $DVAL$ a partir de $Vor(P)$ en la Sección 4.3.2.

4.3.2. Diagramas de Voronoi de alcance limitado (*DVALs*)

La construcción de diagramas de Voronoi de alcance limitado, es un paso necesario para la decisión de si un camino introducido es o no de desviación mínima o de desviación mínima local. Vea la Sección 4.4 para saber cómo la aplicación realiza estas decisiones.

Tal como se ha visto en la Proposición 3.1.1 de la Sección 3.1 la *región de Voronoi de alcance* r de un punto p_i, $Vor(p_i, r)$, es la intersección de la región de Voronoi ordinaria $Vor(p_i)$ y un círculo con centro en p_i y radio r.

El *diagrama de Voronoi de alcance* r del conjunto P, $Vor(P, r)$, es la descomposición del plano en las regiones $Vor(p_i, r)$ tales que $p_i \in P$.

La Figura 3.1 de la Sección 3.1 ilustra un ejemplo.

La aplicación calcula las distintas regiones de Voronoi de los puntos $p_i \in P$ de alcance limitado por la distancia r, $Vor(p_i, r)$, a partir de sus correspondientes regiones de Voronoi ordinarias, $Vor(p_i)$, aplicando un algoritmo que obtiene la intersección entre estas y los círculos de radio r centrados en p_i.

Tal como se explica en la Sección 2.1, las regiones de Voronoi son en cualquier caso regiones poligonales convexas. Para calcular la *región de Voronoi de alcance* r de un punto $p_i \in P$, se ha implementado un algoritmo que retorna la región resultante después de hacer la intersección entre una región poligonal convexa (acotada o no) y un círculo de radio r. Este algoritmo va recorriendo las aristas de la región poligonal convexa, $Vor(p_i)$, y transformándolas, según si el alcance r es mayor o menor que la distancia de la arista al punto p_i, en una combinación de segmentos y arcos de circunferencia.

Por cada arista de $Vor(p_i)$ el algoritmo primero comprueba si la arista está a una distancia de p_i menor o mayor que r. Si la distancia es mayor o igual que r, el algoritmo substituye la arista por un arco de circunferencia. Si la distancia es menor, comprueba cuál es la distancia de p_i a los extremos de la arista para conocer los elementos que deben sustituir a la arista. Estos elementos pueden ser un solo segmento, o un segmento unido con uno o dos arcos, a uno o a ambos lados de este. En cualquier caso, el coste de realizar estas comprobaciones y decisiones es constante para cada arista.

En la Figura 4.6 se puede ver el resultado de aplicar el mencionado algoritmo en una región de Voronoi con un radio r.

El tiempo que tarda el algoritmo en calcular $Vor(p_i, r)$ es proporcional al número de aristas de $Vor(p_i)$. En el peor de los casos, $Vor(p_i)$ puede tener $n-1$ aristas y el algoritmo tardar tiempo $O(n)$, pero el número de aristas que hay en promedio en una región $Vor(p_i)$ es ≤ 6, por lo que el coste promedio del algoritmo es $O(1)$ por región.

Este proceso se realiza con cada región $Vor(p_i)$, es decir, para calcularlas todas es necesario repetir el algoritmo tantas veces como puntos hay en P.

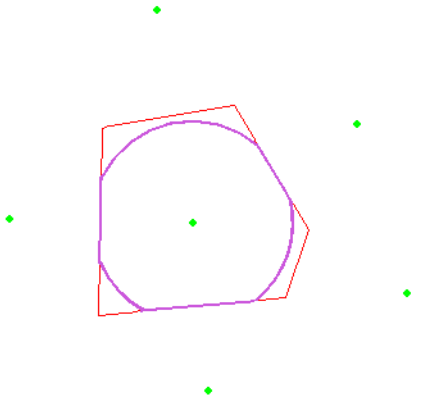

Figura 4.6: $Vor(p_i)$ (rojo) y $Vor(p_i, r)$ (lila).

Proposición 4.3.2 *La aplición calcula el conjunto de regiones de Voronoi de alcance limitado en tiempo amortizado $O(n)$ y en espacio $O(n)$.*

En la frontera de las regiones $Vor(p_i, r)$ sólo puede haber 2 tipos de elementos; segmentos y arcos. Como el alcance está limitado, el diagrama no puede constar de rectas o semirrectas.

La estructura elegida para guardar la información de cada región es una lista de elementos. Empezando por un elemento cualquiera de la región que llamamos *elemento inicial* y recorriendo la región en sentido antihorario, vamos guardando los distintos segmentos y arcos que forman su frontera (especificando de qué tipo es cada elemento) hasta llegar al elemento inicial de nuevo, ya que las regiones son necesariamente acotadas gracias a la limitación del alcance. El tamaño que ocupa cada región en memoria es proporcional al número de aristas de $Vor(p_i)$. En el peor de los casos, puede tener $n-1$ aristas y la lista ocupar espacio $O(n)$, pero en promedio cada región $Vor(p_i)$ ocupa un espacio de tamaño constante.

Pero para definir $Vor(P, r)$ no basta con definir todas sus regiones, ya que la simple unión de estas no nos da información combinatoria como el número de componentes conexas, el número de agujeros en cada componente conexa o qué elementos de las regiones pertenecen a la frontera exterior de una componente conexa, al interior de la componente o a la frontera interior de esta (los agujeros).

Para calcular $Vor(P, r)$, una vez tenemos toda la información de sus regiones, utilizamos otro algoritmo que calcula la información combinatoria que nos interesa conocer.

En la lista de regiones de Voronoi de alcance limitado que tenemos calculada, seguimos teniendo parte de la información del diagrama de Voronoi ordinario. En concreto, para aquellos elementos que son segmentos, conocemos de qué arista del diagrama de Voronoi proceden. Por lo tanto, podemos saber para una región de Voronoi de alcance limitado determinada, $Vor(p_i, r)$, cuál es su región vecina, $Vor(p_j, r)$, al otro lado de un segmento, porque sabemos cuál es la arista gemela en $Vor(P)$ y, por tanto, la región de Voronoi $Vor(p_j)$ a la que pertenece dicha arista gemela. De esta manera, seguimos teniendo la información de vecindad que puede ser directamente consultada en la $DCEL$ de $Vor(P)$.

Así pues, en primer lugar, nuestro algoritmo detecta el número de componentes conexas de $Vor(P)$ y a qué componente conexa pertenece cada punto de P. Para ello busca los segmentos de las regiones $Vor(p_i, r)$, porque las regiones a ambos lados de un segmento están en una misma componente conexa. En esta etapa se realizan los pasos siguientes:

1. Inicialización

   Empieza con tantas componentes conexas como puntos hay en P y asigna a cada $p_i \in P$ una componente conexa distinta.

2. Detección de solapamiento

   Recorre las regiones en busca de segmentos, hasta que encuentra un segmento en la frontera de dos regiones $Vor(p_i, r)$ y $Vor(p_j, r)$. Cuando detecta esta situación, realiza el paso siguiente. Cuando ha comprobado todos los segmentos de todas las regiones, el algoritmo acaba.

3. Unión de componentes

   Asigna a todas las regiones que están en la misma componentes conexa que $Vor(p_i, r)$, la componente conexa de $Vor(p_j, r)$. Vuelve al paso de detección de solapamiento.

Este algoritmo recorre todos los elementos de las fronteras de las regiones $Vor(p_i, r)$, y hay un número lineal de ellos. Para aquellos elementos que sean segmentos (y puede haber un número lineal de ellos) la asignación del tercer paso puede llegar a ser lineal en el peor de los casos. Por lo tanto, en el caso peor, el algoritmo de decisión tiene un coste $O(n^2)$.

El siguiente paso es detectar qué elementos forman la frontera exterior de cada componente conexa o la frontera de sus agujeros (la frontera interior de las componentes). Para ello, sabemos que tanto en la frontera de $Vor(P, r)$ como en la de sus agujeros sólo puede haber arcos de circunferencia. También sabemos, llegados a este punto, qué regiones pertenecen a qué componentes conexas. Con toda esta información, el algoritmo que se encarga de construir la frontera de la componente conexa k realiza los pasos siguientes:

1. Arco inicial

   Como todos los elementos de la frontera deben ser arcos, el algoritmo escoge un arco inicial a partir del cual construye la frontera. Para ello, el algoritmo debe escoger un arco correspondiente a alguna de las regiones $Vor(p_i, r)$ que pertenecen a la componente conexa k. Pero no le vale cualquier arco, porque no todos los arcos pertenecen a la frontera exterior de k, algunos arcos pertenecen a la frontera de los agujeros de esta. Para escoger un arco que pertenezca con certeza a la frontera exterior, el algoritmo busca el punto $p_i \in k$ con ordenada mayor y, en la región $Vor(p_i, r)$ de este punto, busca el arco que contenga el ángulo 90^o ya que, al ser el arco que se encuentra más arriba en la componente conexa, pertenece seguro a la frontera. En la Figura 4.7 se ilustra el arco que escoge el algoritmo para cada componente conexa.

   El coste de este paso es $O(n)$ porque encontrar un punto con una coordenada y máxima tiene coste lineal, y el coste encontrar el arco mencionado en una región que puede tener un número lineal de elementos es también lineal.

Figura 4.7: El algoritmo escoge los arcos en lila como iniciales.

2. Recorrido de la región

   El algoritmo va recorriendo en sentido antihorario la región $Vor(p_i, r)$ en la que se encuentra. Va introduciendo en la frontera los arcos de circunferencia que encuentra en el recorrido de la frontera de la región. Si encuentra un segmento salta al paso siguiente. Si encuentra el arco de circunferencia inicial, termina el algoritmo porque la frontera esta completada.

   El coste de este paso es $O(n)$ en el peor de los casos, pero $O(1)$ en promedio ya que el número de elementos en la frontera de una región es constante en promedio.

3. Cambio de región

   Cuando en el paso anterior encuentra un segmento, el algoritmo detecta que para seguir recorriendo la frontera de la componente conexa tiene que cambiar de región. Para eso, consulta la $DCEL$ de $Vor(P)$ para saber cuál es la región $Vor(p_j)$ vecina a la región actual por dicho segmento. Una vez sabe por qué región debe seguir la construcción de la frontera, vuelve al paso anterior.

   El coste de este paso es $O(1)$ en cualquier caso.

En la Figura 4.8 puede observarse el proceso de generación de la frontera de una componente conexa que realiza el algoritmo y su resultado.

El coste del algoritmo que detecta los arcos que pertenecen a la frontera y los guarda tiene un coste acumulado de orden $O(n)$.

Como el algoritmo recorre las regiones en sentido antihorario, los arcos que se van guardando en la frontera están también en sentido antihorario. Por lo tanto, el algoritmo define la frontera mediante arcos de circunferencia en sentido antihorario, dejando el interior de la componente conexa a la izquierda de la frontera.

Por último es necesario detectar y descubrir los agujeros de cada componente conexa e introducirlos en la estructura que guarda la información de la

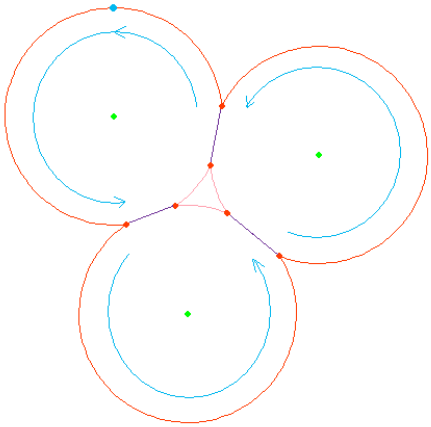

Figura 4.8: Generación de la frontera de una componente conexa.

componente. Para establecer la frontera de los agujeros se utiliza un algoritmo análogo al anterior. Sin embargo, no es tan sencillo detectar un arco inicial que sepamos con seguridad que pertenece a la frontera de un agujero. Por eso es necesario realizar este paso después de haber detectado la frontera exterior de la componente conexa.

Los arcos que pertenecen a la frontera exterior no pertenecen a ningún agujero y, del mismo modo, los arcos que no pertenecen a la frontera exterior pertenecen por fuerza a algún agujero. Por lo tanto, si al construir la frontera exterior de la componente conexa vamos marcando los arcos que utilizamos, cuando hayamos terminado de construirla sabremos exactamente qué arcos pertenecen a la frontera de agujeros y qué arcos no.

Utilizando esta ventaja, el algoritmo que detecta los agujeros y construye sus fronteras realiza los pasos siguientes:

1. Nuevo agujero

   Si queda algún arco no utilizado, crea un nuevo agujero, marca dicho arco como visitado y lo asigna como arco inicial, para, a continuación, ejecutar el paso siguiente. Si no queda ningún arco sin utilizar es que ya no hay más agujeros en la componente conexa y termina el algoritmo.

2. Recorrido de la región

   Va recorriendo la región $Vor(p_i, r)$ en la que se encuentra el arco, del mismo modo que para el algoritmo anterior. El sentido de recorrido es, también, el que deja la región a su izquierda, esto es, el que recorre los arcos de circunferencia en sentido antihorario y, por consiguiente, recorre los elementos de la frontera del agujero en sentido horario. El algoritmo introduce en la frontera del agujero los arcos de circunferencia que se encuentra en el recorrido y va marcando estos arcos como visitados, para que no sean utilizados en agujeros posteriores. Si encuentra un segmento salta al paso siguiente. Si encuentra el arco de circunferencia inicial, termina el agujero porque la frontera del agujero se ha completado, y vuelve al primer paso por si quedan más agujeros por descubrir.

3. Cambio de región

   Cuando en el paso anterior encuentra un segmento, el algoritmo detecta que para seguir recorriendo la frontera del agujero tiene que cambiar de región. Para eso, consulta la $DCEL$ de $Vor(P)$ para saber cual es la región $Vor(p_j)$ vecina a la región actual por dicho segmento. Una vez sabe por qué región debe seguir la construcción de la frontera, vuelve al paso anterior.

En la Figura 4.9 se puede ver el proceso de generación de la frontera de un agujero en una componentes conexa y su resultado.

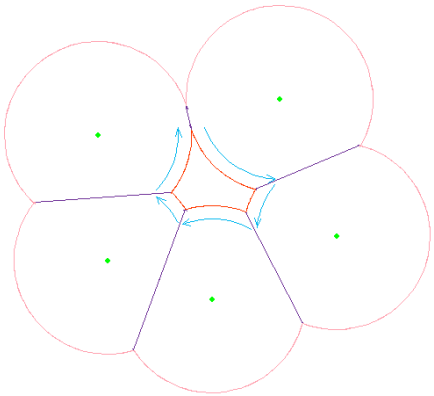

Figura 4.9: Generación de la frontera de un agujero en una componentes conexa.

El coste de la generación de la frontera de un agujero es en el peor caso $O(n)$, porque este puede tener un número lineal de elementos (arcos), y la adición de un arco se hace en tiempo $O(1)$. En el caso promedio el coste de la generación de la frontera de un agujero es $O(1)$, porque cada agujero tiene en promedio un número de arcos constante.

El tiempo que tarda el algoritmo en generar las fronteras de todos los agujeros de una componentes conexa es de orden de $O(n)$ ya que aunque puede haber agujeros con una complejidad lineal, la complejidad promedio de los agujeros es constante, con lo que el coste acumulado de generar las fronteras de todos los agujeros de una componente conexa (puede haber un número lineal de ellos) es lineal.

Después de identificar los arcos de la frontera exterior de la componente conexa y de las fronteras de los agujeros de esta, no quedan mas arcos, debido a que todo arco está en contacto con el exterior de la componente conexa y por tanto debe estar en la frontera interior o exterior de esta. Por otra parte, en la frontera de la componente sólo puede haber arcos, porque los segmentos implican contacto entre regiones. Por ello, cuando todos los arcos han sido utilizados, sabemos que no quedan agujeros por definir, y el algoritmo acaba.

La frontera de los agujeros, y también la frontera exterior de las componentes conexas, se guardan cada una en su propia lista de arcos, en la que primero nos encontramos con el arco inicial que hemos escogido, y al que le siguen el resto de arcos que hemos ido aceptando en el recorrido. En el caso de la frontera exterior, cada arco está tomado en sentido antihorario, y ello resulta en una

descripción de la frontera que sigue un recorrido antihorario. En el caso de la frontera de los agujeros, cada arco está tomado también en sentido antihorario, pero ello resulta en una descripción de la frontera del agujero que sigue un recorrido horario. En ambos casos, el recorrido de las fronteras deja la región a su izquierda. Esta descripción de la frontera es, pues, consistente con la forma utilizada habitualmente para describir regiones con agujeros (polígonos, por ejemplo). La descripción de la frontera de un agujero ocupa en el peor de los casos un espacio de orden $O(n)$, porque puede haber un cantidad lineal de arcos en su frontera, pero en el caso promedio ocupa un espacio de orden $O(1)$.

Nuestra aplicación almacena cada componente conexa por su frontera exterior más una lista de sus agujeros definidos mediante sus fronteras. Por lo tanto, como la frontera exterior puede ocupar un espacio de orden $O(n)$, y puede haber un número lineal de agujeros ocupando cada agujero en promedio un espacio constante, la aplicación almacena la componente conexa en una estructura que ocupa un espacio de orden $O(n)$.

Para guardar $Vor(P, r)$, la aplicación guarda tanto las regiones $Vor(p_i, r)$ definidas al principio, como la información combinatoria de las componentes conexas y sus fronteras interiores y exteriores. La lista de regiones ocupa un tamaño $O(n)$ ya que cada región ocupa en promedio $O(1)$. La información sobre qué regiones están en qué componentes conexas se guarda en una matriz cuyas filas son las regiones, y columnas las componentes conexas. Esta matriz ocupa en el peor de los casos (que haya tantas componentes conexas como regiones), $O(n^2)$. Por último, la definición de las componentes conexas ocupa un espacio de orden $O(n)$ ya que el número total de arcos que puede haber en las fronteras de todas las componentes es proporcional al número de aristas de Voronoi, que es $O(n)$.

Proposición 4.3.3 *La aplicación calcula $Vor(P, r)$ a partir de $Del(P)$ en tiempo $O(n^2)$ y espacio $O(n^2)$.*

Demostración: La estructura que guarda $Vor(P, r)$, ocupa un espacio de orden $O(n^2)$ determinada por la matriz de componentes conexas que ocupa $O(n^2)$ si hay tantas componentes conexas como puntos en P.

El coste total del algoritmo que genera la estructura que define $Vor(P, r)$ viene dado por la parte del algoritmo más costosa, que es la que detecta el número de componentes conexas, y qué puntos pertenecen a cada componente conexa, que en el caso peor (si hay tantas componentes conexas como puntos en P) tiene un coste en tiempo de $O(n^2)$. □

4.3.3. Poligonización

Tras calcular $Vor(P, r)$, la aplicación calcula también la poligonización de las componentes conexas. Esto incluye la construcción de la frontera del polígono contenido en cada componente conexa, y la frontera de sus agujeros, de forma que el interior del polígono esté completamente contenido en el interior de la componente conexa. En la Sección 3.2.2 se cuenta que el uso principal de esta funcionalidad es el de facilitar la minimización de la longitud en caminos de desviación mínima.

En la Figura 4.10 podemos ver un ejemplo de poligonización de componentes conexas.

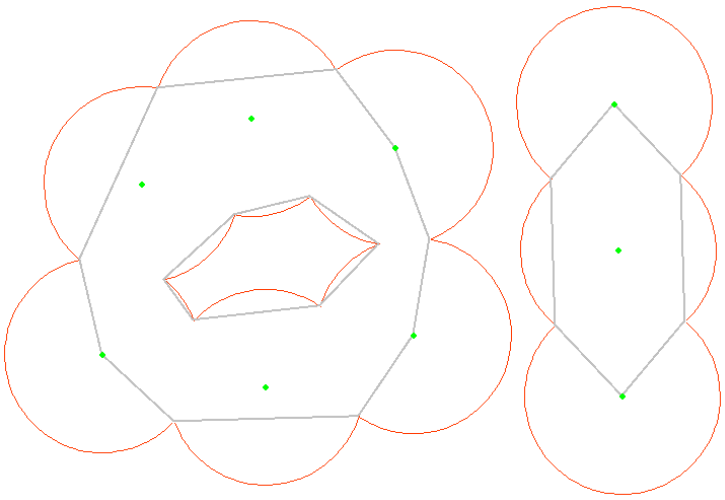

Figura 4.10: Poligonización de 2 componentes conexas.

El algoritmo que poligoniza cada componente conexa y también sus agujeros, realiza los pasos siguientes:

1. Vértice inicial

   Tanto la frontera de las componentes conexas como la de los agujeros se definen utilizando un arco inicial al que le siguen el resto de elementos. El algoritmo de poligonización parte del punto inicial de dicho arco y empieza a poligonizar a partir de él pasando al siguiente paso.

2. Añadir vértice

   El algoritmo recorre la estructura que guarda la frontera exterior de la componente conexa, o del agujero. En caso de estar poligonizando la frontera exterior, el recorrido es en sentido antihorario, y si está poligonizando un agujero el recorrido de la frontera es en sentido horario. Desde el punto actual, traza un segmento hasta el punto inicial q del arco siguiente. Si el punto $p_i \in P$, centro del arco actual, se encuentra a la derecha del mencionado segmento, el algoritmo guarda en la estructura de la poligonización, primero el punto p_i y después el punto q. Si por el contrario, p_i se encuentra a la izquierda del segmento, el algoritmo sólo añade a la estructura el punto q del arco siguiente. Después de haber hecho una cosa u otra, vuelve a repetir este paso para el arco siguiente, tomando como punto actual el último introducido, que en ambos casos es q. Si el punto del arco q que nos encontramos coincide con el punto punto inicial escogido en el paso anterior, el algoritmo acaba porque ha terminado la poligonización.

La poligonización se guarda en una lista ordenada de vértices, en la que el primer elemento es el punto inicial escogido en el primer paso.

La información combinatoria sobre la poligonización de una componente conexa se guarda en una estructura similar a la utilizada para guardar la frontera

de las componentes conexas. Esta estructura contiene la lista de vértices de la frontera exterior del polígono en sentido antihorario y una lista de agujeros en la que por cada agujero se guarda también la lista de los vértices que definen el agujero del polígono en sentido horario. Esta estructura ocupa espacio $O(n)$ porque la estructura que guarda la frontera de una componente conexa ocupa también $O(n)$ y el número de vértices del polígono es proporcional al número de arcos de la frontera.

Proposición 4.3.4 *La aplicación calcula la poligonización de $Vor(P, r)$ en tiempo $O(n)$ y espacio $O(n)$.*

Demostración: El número de vértices en la poligonización es proporcional al número de arcos que hay en la frontera de $Vor(P, r)$, y este es de orden $O(n)$. Por lo tanto el espacio necesario para guardar la poligonización de $Vor(P, r)$ es $O(n)$.

Por la misma razón, el tiempo necesario para recorrer los elementos de la frontera de $Vor(P, r)$ y extraer los vértices es $O(n)$, ya que cada elemento se trata en tiempo $O(1)$. □

4.3.4. Cambios combinatorios

Cuando el valor del alcance, r, varía en la semirrecta real positiva, se producen cambios combinatorios en $Vor(P, r)$ en un número finito de valores de r, asociados intrínsecamente al conjunto de puntos P. Así pues, la aplicación los calcula y los mantiene actualizados en todo momento, con independencia de los diagramas de Voronoi de alcance limitado que se estén visualizando en cada momento.

Los distintos cambios combinatorios posibles están explicados en la Sección 3.1.2. En esta sección se explican los algoritmos utilizados para detectar cada uno de ellos, así como su coste. Para ello, se analizan los cuatro tipos de eventos para explicar por separado la forma en que se detectan y se guardan.

Para guardar los eventos, se utiliza una lista de eventos en la que están en orden creciente de alcance para poder consultar de manera rápida qué evento hay antes o después de un valor r dado.

A cada evento guardado en la lista se le asocia la información relativa al tipo de evento de que se trata, de los cuatro posibles, el valor del alcance en el momento en que se produce el evento, y las coordenadas del punto del plano en el que se produce el cambio combinatorio asociado al evento. Por lo tanto, la información guardada para cada evento ocupa un espacio de tamaño $O(1)$.

Los cuatro tipos de eventos son los siguientes:

1. Conexión de dos componentes conexas en $Vor(P, r)$

   De acuerdo con la Proposición 3.1.2 en la Sección 3.1.2, cuando r alcanza la mitad de la longitud de una arista $p_i p_j$ del árbol generador mínimo euclídeo de P ($EMST(P)$) se conectan dos de las componentes conexas de $Vor(P, r)$.

   Los eventos de este tipo se detectan mientras se está construyendo el $EMST(P)$. En concreto, se produce un evento de este tipo en el punto medio de cada arista del EMST. Por tanto, cada vez que en el algoritmo

de construcción del $EMST(P)$ detecta una arista que pertenece al grafo, está detectando también un evento, y este se guarda en la lista de eventos.

El hecho de detectar estos eventos no produce coste adicional, porque el hecho de guardar cada evento se hace en tiempo constante, y aunque el número de eventos sea lineal, el tiempo utilizado en detectarlos se usa también para la construcción del $EMST(P)$, que es $O(n)$.

2. Generación de un nuevo agujero en $Vor(P, r)$

   De acuerdo con la Proposición 3.1.3 en la Sección 3.1.2, cuando r alcanza la mitad de la longitud de una arista $p_i p_j$ de Gabriel que no pertenece al $EMST(P)$ (una arista de Delaunay que corte con su dual de Voronoi pero no pertenezca al $EMST(P)$) se produce un agujero en el diagrama.

   Los eventos de este tipo también se detectan durante la construcción del $EMST(P)$. Sin embargo, este tipo de eventos sólo se producen en aristas de Gabriel que no pertenecen al $EMST(P)$. Así pues, para comprobar si se produce algún evento de este tipo, se miran aquellas aristas descartadas por el algoritmo de construcción del $EMST(P)$.

   Como en el caso anterior, detectar estos eventos no supone ningún coste adicional, ya que se realiza un trabajo constante por cada arista descartada del $EMST(P)$, y por lo tanto, todas las aristas sobre las que se realiza la comprobación están siendo consultadas durante la construcción del $EMST(P)$.

3. Desaparición de un agujero en $Vor(P, r)$

   De acuerdo con la Proposición 3.1.4 en la Sección 3.1.2, cuando r alcanza el radio de la circunferencia circunscrita a un triángulo de Delaunay cuyo centro se encuentra dentro del triángulo, es decir, un triángulo de Delaunay acutángulo, desaparece uno de los agujeros de $Vor(P, r)$ y aparece un nuevo vértice de Voronoi.

   Estos eventos se detectan durante el proceso de construcción de $Vor(P)$. Aunque para detectar estos eventos, necesitamos explorar los triángulos de $Del(P)$, no podemos detectarlos en la construcción de $Del(P)$ porque esta se hace de forma incremental, y hasta que no acaba el algoritmo no se dispone de todos los triángulos de $Del(P)$. Por eso la búsqueda de estos eventos se realiza simultáneamente con la construcción de $Vor(P)$, porque este se calcula después de haber terminado la construcción de $Del(P)$, y se recalcula completamente después de cada cambio en $Del(P)$.

   Durante la construcción de $Vor(P)$, se visitan todos los triángulos de $Del(P)$ para dualizarlos en vértices de $Vor(P)$. Es en estas visitas cuando se comprueba si el triángulo visitado es acutángulo o no. Esta comprobación se realiza en tiempo constante. Si el triángulo resulta ser acutángulo se introduce un evento de este tipo en la lista de eventos.

   Detectar estos eventos no supone un coste adicional al algoritmo de construcción de $Vor(P)$ ya que el coste de la comprobación es constante, y el algoritmo visita igualmente todos los triángulos de $Del(P)$ para dualizarlos.

4. Desaparición de un arco de circunferencia en la frontera de $Vor(P, r)$

De acuerdo con la Proposición 3.1.5 en la Sección 3.1.2, cuando r alcanza el radio de la circunferencia circunscrita a un triángulo de Delaunay cuyo centro se encuentra fuera del triángulo, es decir, un triángulo de Delaunay obtusángulo, desaparece un arco de circunferencia en la frontera de $Vor(P, r)$ y aparece un nuevo vértice de Voronoi.

Como en el caso anterior y por las mismas razones, la detección de estos eventos se realiza durante el proceso de construcción de $Vor(P)$. En este caso, se comprueba si los triángulos visitados son obtusángulos y, si lo son, se introduce un evento de este tipo en la lista de eventos. Esta comprobación tiene también un coste constante, por lo que la detección de estos eventos tampoco supone un coste adicional para el algoritmo que calcula $Vor(P)$.

Por lo tanto, el coste de detectar los 4 tipos de eventos está incluido en el de los algoritmos de generación de $EMST(P)$ y $Vor(P)$. No es necesario un algoritmo aparte, posterior a estos cálculos y con un coste adicional para detectarlos. Tras detectar todos los eventos (primero los de tipo 3 y 4, y después los de tipo 1 y 2) el algoritmo ordena estos radios de forma creciente para después poder consultar fácilmente qué evento viene antes o después de otro. Para ello, utilizamos un algoritmo de ordenación, en concreto, el algoritmo de ordenación por selección (ver Sección 2.3.4). Este algoritmo tiene un coste $O(n^2)$, y por lo tanto, podríamos decir que el tiempo extra empleado en la construcción y preparación de la lista de eventos es finalmente $O(n^2)$.

Proposición 4.3.5 *La aplicación obtiene y ordena la lista de eventos en un tiempo $O(n^2)$ marcado por el proceso de ordenación de los eventos, y la estructura utilizada para guardar la lista ocupa un espacio de orden $O(n)$.*

La lista de eventos contiene, en cualquier caso, una cantidad lineal de eventos. Se producen exactamente $n-1$ eventos del primer tipo (unión de componentes) y en el peor de los casos una cantidad lineal del resto de eventos. Por ello, y dado que un evento ocupa un espacio de memoria constante, esta lista de eventos ocupa un espacio lineal de memoria, $O(n)$.

Como al producirse cualquier cambio en la nube de puntos, tanto $Vor(P)$ como $EMST(P)$ se recalculan completamente, el proceso de detección y ordenación de los eventos también se repite. Es decir, cuando se detecta una modificación en la nube de puntos, la lista de eventos se vacía, y se vuelven a detectar y ordenar los nuevos eventos a partir de $Vor(P)$ y $EMST(P)$ con un coste $O(n^2)$ como ya hemos visto.

4.4. Caminos

La aplicación permite al usuario introducir caminos de varios tipos de manera sencilla, y expone una serie de información útil sobre estos caminos que facilita al usuario el estudio, todavía en desarrollo, sobre los caminos de desviación mínima y de separación máxima.

4.4.1. Tipos de caminos

La aplicación permite al usuario introducir caminos de dos tipos. En primer lugar permite introducir caminos poligonales, formados únicamente por segmen-

tos rectilíneos. La Figura 4.11 muestra un ejemplo de camino poligonal.

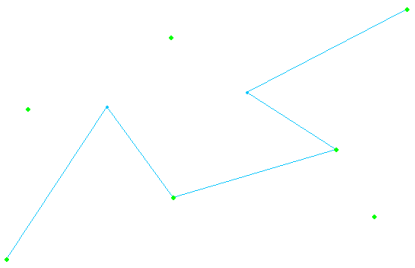

Figura 4.11: Ejemplo de camino poligonal.

Por otro lado también permite introducir caminos formados por segmentos rectilíneos y arcos de circunferencia centrados en los puntos de P. La Figura 4.12 muestra un ejemplo de este tipo de caminos. El usuario debe tener en cuenta a la hora de introducir este tipo de caminos, que para que la aplicación realice las decisiones acerca de estos caminos correctamente, es necesario que los arcos de circunferencia centrados en los puntos de P estén totalmente contenidos en las regiones de Voronoi correspondientes a sus centros. Llamamos a esta propiedad de los arcos de un camino, propiedad de contención. Si los arcos de un camino no cumplen la propiedad de contención, la aplicación podría no dar una respuesta correcta a las decisiones tomadas sobre estos caminos.

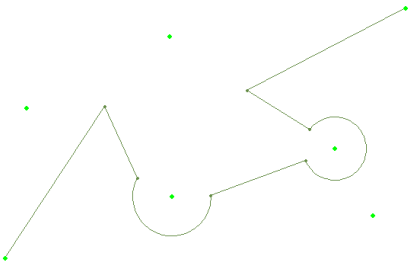

Figura 4.12: Ejemplo de camino formado por segmentos rectilíneos y arcos de circunferencia.

La plicación sólo permite introducir caminos cuyos vértice inicial y vértice final esten sobre puntos de P. El resto de vértices del camino pueden estar también sobre puntos de P, o en cualquier otro lugar del plano. Si se intenta introducir un camino cuyo vértice inicial o vértice final no se encuentra sobre un punto de P, la aplicación añade un punto p_i al conjunto P en las ubicaciones mencionadas.

El hecho de que la aplicación solo trabaje con caminos cuyos vértices inicial y final están sobre puntos de P, facilita tareas de localización y de decisión. No obstante, la adapatación de la aplicación para que esta sea capaz de decidir sobre caminos que no cumplan esta propiedad, podría ser una ampliación interesante a realizar en el futuro.

Para saber de qué formas se pueden introducir los distintos tipos de camino en la aplicación, consulte la Sección 5.4 del Manual de usuario.

4.4.2. Cálculo de la longitud

Para calcular la longitud de un camino, la aplicación calcula y suma las longitudes de todos los elementos del camino (segmentos y arcos) por separado. La longitud de un segmento rectilíneo se calcula midiendo la distancia entre los dos extremos del segmento. La longitud de un arco de circunferencia se calcula obteniendo una parte proporcional al ángulo del arco, sobre el perímetro de la circunferecnia con el mismo radio que el arco.

Proposición 4.4.1 *La aplicación calcula la longitud de un camino en tiempo $O(k)$ siendo k la complejidad del camino.*

Demostración: El cálculo de la longitud de cada elemento del camino se realiza en tiempo constante, por lo que el tiempo que tarda la aplicación en calcular la longitud de un camino con k elementos es $O(k)$ □

4.4.3. Caminos de desviación mínima

Esta funcionalidad permite, dados dos puntos s y t de P y un camino Γ que les conecta, conocer si Γ es de desviación mínima, esto es, si la distancia máxima de los puntos de Γ al conjunto P es mínima.

Para una mayor profundización en la definición de caminos de desviación mínima, consúltese la Sección 3.2.

El primer paso para saber si un camino Γ que va desde un punto s a un punto t es de desviación mínima o no, es conocer la desviación entre s y t, que como se ha visto en la Sección 3.2 es igual a la mitad de la longitud de la arista más larga (arista crítica) del camino entre s y t contenido en $EMST(P)$. Para encontrar esta distancia crítica, el algoritmo primero calcula el camino entre s y t en el $EMST(P)$, para que, una vez conocidas las aristas del $EMST(P)$ por las que pasa, averiguar cuál es la arista crítica y calcular la desviación.

Para encontrar el camino entre s y t en el $EMST(P)$, el algoritmo parte de s y lleva a cabo una búsqueda en profundidad por el árbol, hasta llegar a t. En el peor de los casos, pasará 1 vez por cada arista del arbol, es decir, el coste del algoritmo de búsqueda del camino es $O(n)$.

Una vez conocido el camino entre s y t en el $EMST(P)$ y habiendo guardado este camino es una lista de aristas, encontrar cuál de todas ellas es la arista crítica se puede realizar en tiempo $O(n)$, recorriendo la lista de aristas y quedándonos con la que tenga una longitud mayor. La distancia crítica es la mitad de la longitud de la arista crítica encontrada.

En la Figura 4.13 se observa un camino Γ, el recorrido que une los vértices inicial y final de Γ pasando por aristas del $EMST(P)$, y la arista crítica de dicho recorrido.

Una vez la aplicación conoce la distancia crítica, esto es, la desviación entre s y t, se trata de comprobar si la desviación del camino Γ es o no superior. Para ello, el algoritmo encuentra todas las intersecciones del camino con el 1-esqueleto del diagrama de Voronoi, $Vor(P)$. Son los puntos por los que el camino sale de una región de Voronoi para entrar en otra, y por lo tanto, junto con los

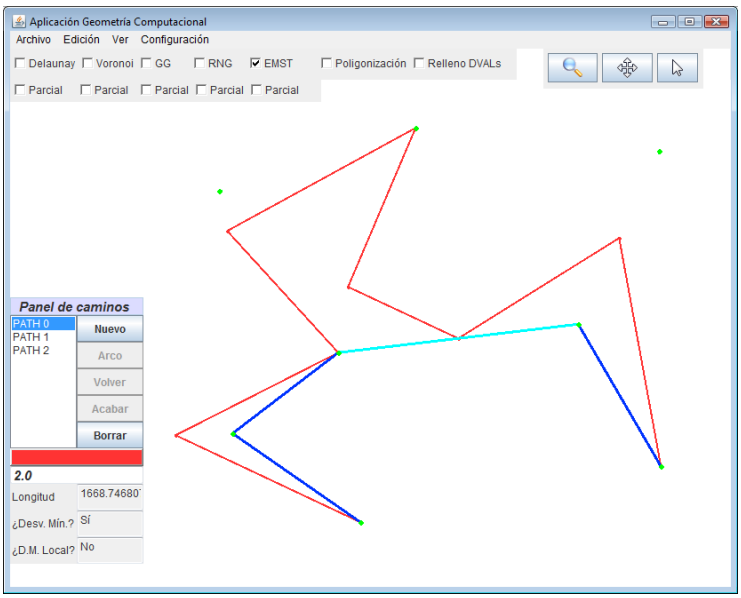

Figura 4.13: Camino (rojo) y su recorrido pasando por $EMST(P)$ (azul).

vértices del camino, son los únicos puntos de este en los que se puede alcanzar la distancia máxima a los puntos de P.

Para encontrar todos esos puntos de intersección entre el camino Γ y las aristas de $Vor(P)$, recorremos la $DCEL$ de $Vor(P)$ y vamos guardando en cada cambio de región, el punto exacto por donde Γ sale de la región. Gracias al hecho de que el punto inicial s está sobre un punto de P, conocemos en qué región se encuentra s para iniciar el recorrido. Desde la región del punto de P sobre el que está s, miramos si hay alguna intersección entre el segmento actual del camino y las aristas que delimitan la región en la $DCEL$ de $Vor(P)$. Si la hay, guardamos el punto de intersección en la lista, pasamos a la región vecina y volvemos a repetir el proceso. Si no hay intersección, guardamos el vértice del camino que se encuentra en esta misma región y pasamos al siguiente elemento del camino. Si el elemento actual del camino es un arco, no comprobamos las intersecciones, ya que el arco debe estar contenido en la región de Voronoi del punto que hace de centro. Por ello, si es un arco, simplemente comprobamos que tanto el punto inicial como el punto final estén dentro de la región, y pasamos al elemento siguiente. Si el punto inicial o el punto final de un arco no están en la misma región de Voronoi que el centro de este arco, significa que el arco no cumple la propiedad de contención, y el algoritmo retorna que el camino no es de desviación mínima.

Los puntos de intersección se guardan en una lista, junto con los vértices del camino en el orden en el que aparecen a lo largo del camino y, junto a cada uno de ellos, se guarda el identificador de la región a la que pertenecen. Generar esta lista de puntos tiene un coste proporcional a la complejidad del camino y sus intersecciones con el diagrama de Voronoi. En el peor de los casos, si el camino tiene k elementos y cada elemento del camino atraviesa completamente $Vor(P)$, el algoritmo tendría una complejidad $O(nk)$.

En la Figura 4.14 se pueden observar los puntos que introduciría este paso del algoritmo en la lista, para el camino que se muestra.

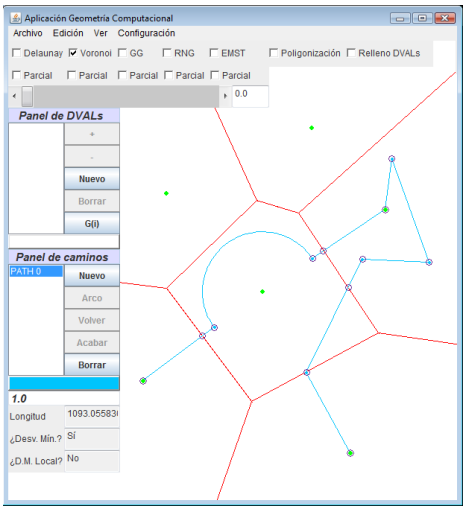

Figura 4.14: Posibles puntos de desviación máxima del camino.

Una vez tenemos esta lista de todos los puntos que pueden maximizar la distancia del camino a P, debemos comprobar que ninguno de ellos esté a una distancia mayor que la distancia crítica que hemos calculado previamente. Si todos los puntos de la lista están a una distancia menor o igual que la distancia crítica, entonces el camino es de desviación mínima, pero sólo con que uno de ellos tenga una desviación mayor que la crítica, el camino no es de desviación mínima.

Para averiguar la distancia de un punto q de la lista a la nube P no hace falta calcular la distancia de q a todos los puntos de P, porque en la lista hemos guardado además de los puntos, las regiones sobre las que están en $Vor(P)$, es decir, sabemos exactamente cuál es el punto $p_i \in P$ que está más cerca de q. Así pues, calcular la distancia de q a p_i se hace en tiempo $O(1)$, y comprobar si estas distancias son o no superiores a la desviación mínima se hace en tiempo proporcional a la complejidad de la lista. En el peor de los casos, si el camino tiene k elementos y cada elemento del camino atraviesa completamente $Vor(P)$, el algoritmo tendría una complejidad $O(nk)$.

Proposición 4.4.2 *La aplicación determina si un camino* Γ *que cumple la propiedad de contención es o no de desviación mínima en tiempo* $O(nk)$*, donde* n *es el número de puntos de* P*, y* k *el número de elementos del camino* Γ*.*

Debemos aclarar que si los arcos no cumplen la propiedad de contención (por la que todos los arcos del camino deben estar completamente contenidos en la región de Voronoi correspondiente al centro del arco), la aplicación podría no dar una respuesta correcta a la pregunta de si el camino es o no de desviación mínima. Ahora bien, siempre que la aplicación resuelva positivamente, el camino es efectivamente de desviación mínima. Sólo para ciertos caminos de desviación mínima cuyos arcos no cumplen la propiedad de contención, el algoritmo puede (no en todos los casos) devolver falso erróneamente, pero nunca a la inversa

(nunca devuelve cierto si, en realidad, el camino no es de desviación mínima). Si los arcos del camino cumplen la propiedad de contención, la respuesta del algoritmo es correcta en todos los casos.

4.4.4. Caminos de desviación mínima local

Tal y como se ha definido en la Sección 3.3, caracterizamos los caminos de desviación mínima local de dos maneras:

Proposición 4.4.3 *Diremos que* Γ *es de desviación mínima local si, para cada par de puntos* x *e* y *de* Γ*, el camino* $\Gamma_{|xy}$ *es de desviación mínima, esto es si* $d(\Gamma_{|xy}, P) = d(x, y, P)$*.*

La segunda caracterización de los caminos de desviación mínima local es en términos de sus máximos locales:

Para determinar si un camino es de desviación mínima local o no, la aplicación utiliza la caracterización por máximos locales: Un camino simple Γ es de desviación mínima local si, y sólo si, todos sus máximos locales estrictos son o contienen un punto crítico.

Hay que tener en cuenta que si un camino es de desviación mínima local, necesariamente es de desviación mínima. Por lo tanto, la aplicación realiza en primer lugar la comprobación de si el camino es de desviación mínima y, si el resultado es negativo, no hace falta realizar ningún otro cálculo para saber que el camino tampoco es de desviación mínima local.

Cuando el camino es de desviación mínima, y para determinar si es también de desviación mínima local o no, se utiliza la misma lista de puntos de Γ generada en el algoritmo de decisión de desviación mínima. Esta lista de puntos contiene de forma ordenada, los vértices del camino y los puntos de intersección entre Γ y las fronteras de las regiones de $Vor(P)$ porque sólo estos puntos pueden ser máximos locales. Como la lista es exactamente igual a la utilizada en la decisión anterior, no hace falta volver a recalcularla, simplemente reutilizarla, por lo que el coste añadido de conseguir esta lista para este algoritmo es constante. Consúltese la Sección 4.4.3 para más información acerca de la generación de esta lista.

Teniendo la lista de posibles máximos locales, el algoritmo los comprueba uno por uno y separa aquellos que efectivamente son máximos locales de aquellos que no lo son. Después comprueba si los máximos locales del camino son o contienen algún punto crítico (llamamos puntos críticos a los puntos medios de las aristas del $EMST(P)$).

Para comprobar si un punto q (o un arco del camino) es un máximo local o no, se compara con el anterior y el siguiente en la lista. Se trazan dos segmentos, el primero desde el punto anterior a q en la lista hasta q y el segundo desde q hasta el punto siguiente a q en la lista. Una comparación elemental sobre estos dos segmentos permite comprobar si q es o no un máximo local en tiempo $O(1)$.

En la Figura 4.15 se destacan los máximos locales del camino mostrado.

Los puntos de la lista que no son máximos locales se descartan y, para aquellos que sí lo son, se comprueba si son o contienen un punto crítico. Si se encuentra algún máximo local que no es ni contiene un punto crítico, el algoritmo acaba y devuelve que el camino no es de desviación mínima local. Si todos los máximos locales son o contienen un punto crítico, el algoritmo devuelve que el camino si es de desviación mínima local.

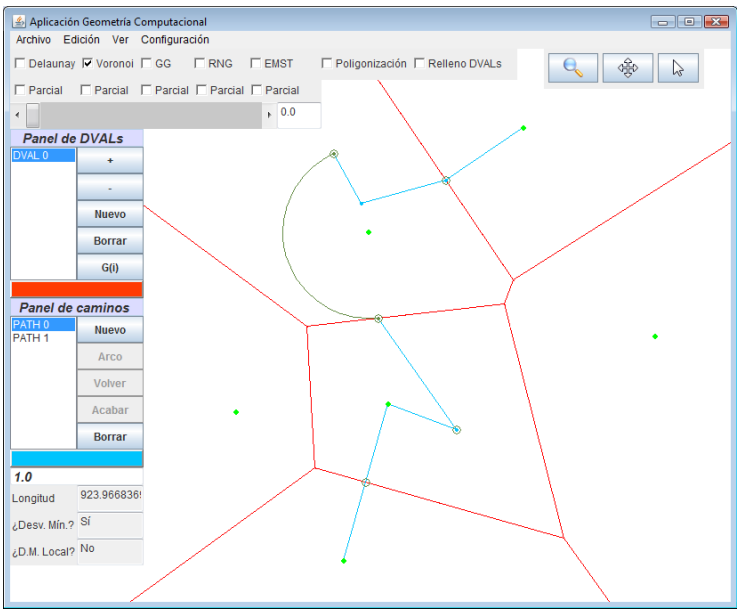

Figura 4.15: Máximos locales de un camino.

Ni el primero ni el último punto de Γ son máximos locales. Además, como estos puntos están sobre algún punto de P, la distancia de estos puntos a P es siempre 0.

Para comprobar si un máximo local q es o contiene un punto crítico, se compara con los puntos críticos de las aristas del $EMST(P)$ incidentes al punto cuya región de Voronoi contiene a q (como máximo son 6). Por lo tanto, la comprobación que dictamina si un punto de la lista es o contiene un punto crítico tiene un coste en tiempo $O(1)$.

Como esta comprobación se realiza en tiempo $O(1)$ para aquellos elementos de la lista antes mencionada que sean máximos locales (y pueden serlo todos ellos), el coste total del algoritmo es $O(nk)$, ya que puede haber en la lista nk elementos en el peor de los casos.

Proposición 4.4.4 *La aplicación determina si un camino Γ que cumple la propiedad de contención es o no de desviación mínima local en tiempo $O(nk)$, donde n es el número de puntos de P, y k el número de elementos del camino Γ.*

Al igual que ocurre con el algoritmo de decisión sobre caminos de desviación mínima, si los arcos del camino no cumplen la propiedad de contención (todos los arcos de un camino deben estar contenidos completamente en la región de Voronoi correspondiente al centro de cada arco), la aplicación puede no dar siempre una respuesta correcta a la pregunta de si el camino es o no de desviación mínima local. Ahora bien, siempre que la aplicación resuelva positivamente, el camino es efectivamente de desviación mínima local. Sólo para ciertos caminos de desviación mínima local cuyos arcos no cumplen la propiedad de contención, el algoritmo puede (no en todos los casos) devolver falso erróneamente, pero nunca a la inversa (nunca devuelve cierto si, en realidad, el camino no es de

desviación mínima local). Si los arcos del camino cumplen la propiedad de contención, la respuesta del algoritmo es correcta en todos los casos.

4.5. Visualización

En esta sección se describen todas las funcionalidades que el usuario tiene a sus disposición para adaptar el modo en que la aplicación muestra sus elementos y datos a sus necesidades. En concreto, se exponen las funcionalidades que permiten al usuario centrarse en una zona del plano y desplazarse por el mismo (Zoom y Pan), y las funcionalidades que permiten personalizar los colores y grosores de todos los elementos que aparecen en pantalla. También, se explican los panales con los que cuenta nuestra aplicación y como mostrarlos / ocultarlos.

4.5.1. Modos de ejecución: normal, Zoom y Pan

DVALon, además del modo normal de ejecución, cuenta con otros dos modos de ejecución: Zoom y Pan. Para acceder a cualquiera de estos dos modos o volver al modo normal la aplicación cuenta con un panel que cuenta con tres botones, cada uno para entrar en un modo distinto de ejecución. También es posible acceder a cada uno de los modos de ejecución de la aplicación mediante teclas de acceso rápido o mediante el botón central del ratón (ver sección 5.5.1).

El modo normal de ejecución es el modo en que la aplicación se ejecuta por defecto y a través del cual se accede al resto de funcionalidades descritas en este capítulo.

El modo Zoom de la aplicación permite acercar o alejar la cámara respecto de una zona del plano a elección del usuario. Cuando la aplicación se está ejecutando en modo Zoom no es posible introducir cambios en la nube de puntos, pero sí es posible crear/borrar *DVALs* o caminos (en este último caso la aplicación devuelve al usuario al modo normal automáticamente). Una vez dentro del modo Zoom, el cursor y el funcionamiento del ratón cambian. En concreto, al hacer un clic izquierdo se acerca la cámara a la zona clicada y al hacer un clic derecho la cámara se aleja de ella. Para comodidad del usuario, la aplicación también permite aumentar o reducir el Zoom desde cualquier modo de ejecución, sin necesidad de acceder al modo Zoom, a través de la rueda del ratón. Se podría cuestionar pues, la necesidad de tener un modo de ejecución específico para esta funcionalidad, pero este modo en realidad es necesario ya que no todos los potenciales usuarios tienen un ratón con rueda.

Cuando acerca la cámara, la aplicación multiplica por 0,7 el valor de escala utilizado, para reducirla, y recalcula los nuevos x_0 y y_0, variables que usa la aplicación para determinar la posición de la cámara, a partir de las coordenadas del punto sobre el que hacemos el Zoom y sus valores anteriores. También recalcula el radio máximo visible y las nuevas coordenadas (de pantalla, puesto que las coordenadas reales no cambian) de cada uno de los puntos de la nube. En caso de reducir el Zoom, los cálculos que realiza la aplicación son los mismos pero el valor escala se multiplica por 1,4 para aumentarla. La aplicación, como máximo, permite acercar la cámara 15 veces desde la posición inicial y también reducirla un máximo de 15 veces desde la posición inicial, lo que da un total de 31 estados de Zoom posibles.

En modo Pan, la aplicación permite trasladar la posición de la cámara sobre el plano, pero no acercarse ni alejarse de él. Este modo, permite al usuario desplazarse por el plano y centrar la cámara en los puntos que le interesen. Al acceder a este modo, el cursor y el funcionamiento del ratón también cambian y, como sucede con el modo Zoom, tampoco es posible introducir modificaciones en la nube de puntos. Todos los botones del ratón cambian su funcionalidad. Para usar la funcionalidad, una vez dentro del modo Pan, el usuario debe hacer clic, ya sea izquierdo, derecho o incluso central, sobre una zona del plano y, sin soltar el botón pulsado, desplazar el ratón en la dirección de desplazamiento deseada. Esta acción modifica las coordenadas de las variables que determinan la zona visible del plano a medida que se desplaza el ratón y, por tanto, la zona del plano visible para el usuario.

4.5.2. Colores

DVALon permite al usuario modificar los colores de los grafos y diagramas que se muestran, y de los *DVALs* y caminos creados para que estos se ajusten a sus preferencias. Para ello, existen una serie de ventanas accesibles desde el menú principal o de una combinación de teclas de acceso rápido que permiten modificar el color de todos estos elementos. Para el caso de *DVALs* y caminos, además, es posible hacer modificaciones de color a través de sus paneles correspondientes (ver sección 5.5.2).

La aplicación cuenta con una ventana que permite modificar los colores de la nube de puntos, diagrama de Voronoi y grafos de proximidad. Esta ventana contiene el nombre de los seis elementos básicos calculados por la aplicación (Puntos, triangulación de Delaunay, diagrama de Voronoi, grafo de Gabriel, grafo de vecinos relativos, árbol generador mínimo) y, al lado de cada uno de ellos, un botón cuyo color actual es el color del elemento en cuestión. Si clicamos en alguno de estos botones, la aplicación muestra una paleta de colores donde el usuario puede elegir el nuevo color del elemento en cuestión. El color seleccionado en esa paleta será el nuevo color con el que la aplicación mostrará el elemento que esta siendo modificando.

Nuestra aplicación permite, también, modificar el color de cada uno de los *DVALs* presentes en un proyecto. En este caso, en la ventana de selección de color aparecen el nombre de cada uno de los *DVALs* existentes en el proyecto y, junto a cada uno de ellos, un botón del color actual del *DVAL* con el cual es posible abrir una paleta de colores y asignarle un nuevo color al *DVAL* en cuestión.

Sin embargo, para el caso de los *DVALs* es posible modificar su color de una manera alternativa. Siempre que el usuario seleccione un *DVAL*, el botón del panel de *DVALs* que nos indica que *DVAL* tenemos seleccionado (ver sección 4.5.4) muestra el color del *DVAL* seleccionado. Si clicamos en este botón, la aplicación muestra una paleta de colores que permite modificar el color del *DVAL* seleccionado en ese momento, sin necesidad de tener que acceder a la ventana de selección de colores comentada anteriormente.

De igual manera que sucede con la selección de color de los *DVALs*, la aplicación permite modificar el color de cada uno de los caminos, ya sea mediante una ventana específica mediante la cual podemos modificar el color de cualquier camino (ver sección 5.5.2), ya mediante el botón del panel de caminos que indica en cada momento el camino que tenemos seleccionado (ver sección 4.5.4).

4.5.3. Grosores

DVALon permite también modificar los grosores de los distintos elementos que se muestran por pantalla. Para ello cuenta con una ventana específica para esta funcionalidad. Sin embargo, hay un caso especial para la modificación del grosor de los caminos, que se explica también a continuación.

La ventana de modificación de grosores sigue una estructura similar a las ventanas de selección de colores. Esta ventana permite modificar el grosor de once elementos que muestra la aplicación. Para modificar cada uno de ellos, la aplicación cuenta con una etiqueta y un campo de texto editable, donde el usuario debe introducir el valor del nuevo grosor que quiere asignar al elemento en cuestión.

Cada uno de estos elementos tiene restricciones y valores por defecto propios. Por eso, es conveniente explicar cada uno de ellos por separado.

1. Nube de puntos: Se utiliza para modificar el grosor de todos los puntos de la nube. El valor mínimo y el valor que tiene este grosor por defecto es 1, esto es, la aplicación no permite no visualizar la nube de puntos. Este grosor no tiene un valor máximo.

2. Eventos: Se utiliza para modificar el punto que marca la aparición de un nuevo evento. Su valor por defecto es 2, ya que es un punto importante y debe resaltar en la escena que está viendo el usuario. Sin embargo, es posible cambiar su grosor teniendo en cuenta que debe tener un valor mínimo de 1.

3. DVALs: Permite modificar el grosor de la frontera de las regiones de Voronoi de alcance limitado. Su valor por defecto y también valor mínimo es 1. También establece el grosor con el cual se muestran por pantalla los nuevos *DVALs* creados a partir de ese momento.

4. Vértices de Voronoi: Se utiliza para modificar el grosor de los vértices del diagrama de Voronoi ordinario. Su valor por defecto es 0, lo que provoca que estos vértices no sean visibles por defecto aunque estemos viendo el diagrama de Voronoi. Esta decisión se ha tomado ya que se considera que los vértices de Voronoi no tienen gran relevancia en este proyecto. Sin embargo, es posible modificar su grosor para que estos sean visibles con el grosor que queramos.

5. Delaunay: Permite modificar el grosor de la triangulación de Delaunay. Su valor por defecto y también mínimo es 1 y no tiene valor máximo.

6. GG: Permite modificar el grosor del grafo de Gabriel. Su valor por defecto y también mínimo es 1 y no tiene valor máximo.

7. RMG: Permite modificar el grosor del grafo de vecinos relativos. Su valor por defecto y también mínimo es 1 y no tiene valor máximo.

8. EMST: Permite modificar el grosor del árbol generador mínimo. Su valor por defecto y también mínimo es 1 y no tiene valor máximo.

9. Poligonización: Permite modificar el grosor de las poligonizaciones de los *DVALs*. Su valor por defecto y también mínimo es 1 y no tiene valor máximo.

10. Caminos: Permite modificar el grosor de los segmentos o arcos de todos los caminos existentes. También establece el valor del grosor que tienen los nuevos caminos que se crean a partir de ese momento. Su valor por defecto y también mínimo es 1 y tampoco tiene valor máximo.

11. Vértices caminos: Permite modificar el grosor de los vértices de los caminos. Su valor por defecto es 1 y no tiene valor máximo. Sin embargo, en este caso el valor mínimo es 0, ya que entendemos que la importancia de ver o no estos vértices es relativa y el usuario puede preferir no verlos.

Como consideraciones generales, cualquiera de los casos anteriores acepta cualquier valor entero o decimal, con lo que la aplicación nunca da error, cualquiera que sea el valor asignado. Sin embargo, ante un valor negativo o un valor por debajo del mínimo permitido en cada caso, la aplicación asigna el grosor mínimo automáticamente. En el caso del grosor de puntos, es decir, en el caso del grosor de la nube de puntos, del punto que marca un nuevo evento, de los vértices de Voronoi y de los vértices de los caminos, la aplicación siempre redondea los valores decimales al entero inferior. En el resto de casos, se aceptan todo tipo de valores decimales, siempre que la parte entera y la parte decimal estén separadas por un punto. En caso contrario, la aplicación no detecta el valor introducido.

Finalmente, hay que destacar que el grosor de cada camino se puede modificar también individualmente a través del panel de caminos (ver sección 5.5.3). Basta con introducir el valor del camino seleccionado en el campo de texto habilitado para ello en dicho panel. Esta acción permite modificar únicamente el grosor del camino seleccionado. Hay que tener en cuenta que si se modifica el grosor global de los caminos, las modificaciones individuales que se hayan hecho previamente quedan sin efecto.

4.5.4. Paneles

Nuestra aplicación cuenta con seis paneles además de la barra del menú principal. Estos panales están orientados principalmente a que el usuario sea capaz de acceder de manera rápida e intuitiva a todas las funcionalidades de la aplicación, sin necesidad de conocer las teclas de acceso rápido de las distintas funcionalidades. Los paneles principales con los que cuenta *DVALon* son los siguientes:

1. Panel de grafos: permite visualizar o ocultar los distintos grafos y diagramas. Cuenta con siete casillas que se pueden activar o desactivar en función de si se quieren observar o no los grafos de proximidad, el diagrama de Voronoi y la poligonización y relleno de *DVALs*. Cuenta también con cinco casillas para decidir si se quieren observar los grafos de proximidad y el diagrama de Voronoi limitados por algún alcance.

2. Panel de *DVALs*: permite la creación, modificación y destrucción de *DVALs*. Cuenta con una lista que nos permite seleccionar los distintos *DVALs* que están presentes en la aplicación. Además, cuanta con los botones "+" y "-", que sirven para agrandar o disminuir un *DVAL* hasta que se produce el siguiente o anterior evento respectivamente, los botones "Nuevo" y "Borrar", para crear o borrar un *DVAL*, y el botón "G(i)", que sirve

para generar automáticamente los *DVALs* críticos de la nube de puntos. Además, cuenta con un botón que nos indica qué *DVAL* tenemos seleccionado mediante su color y permite también modificar el color del mismo.

3. Panel de radios: permite la modificación del radio del *DVAL* seleccionado. Para ello, cuenta con una barra que podemos desplazar para modificar el radio del *DVAL* y con un campo de texto en el que podemos introducir directamente el nuevo alcance del *DVAL* seleccionado.

4. Panel de caminos: permite la creación, modificación y destrucción de caminos. Cuenta con una lista que nos permite seleccionar los distintos caminos que están presentes en la aplicación. Tiene los siguientes botones: "Nuevo", que permite crear un nuevo camino, "Arco", que permite decidir si el siguiente segmento a introducir es un arco o un segmento, "Volver", que elimina el último elemento introducido en el camino, "Acabar", para indicar que se ha terminado de construir el camino, y "Borrar", para eliminar el camino seleccionado. Además, cuenta con un botón que nos indica qué camino tenemos seleccionado mediante su color y permite también modificar el color del mismo. De forma similar, cuenta con un cuadro de texto editable que permite modificar el grosor del camino seleccionado.

5. Panel de modos: permite alternar entre los modos de ejecución de la aplicación: normal, Zoom o Pan. Cuenta con tres botones, cada uno con una imagen distinta. De izquierda a derecha, nos permiten entrar o salir del modo Zoom, Pan y normal respectivamente.

6. Panel de edición: permite deshacer o rehacer los cambios que se han ido introduciendo en el proyecto. Cuenta únicamente con dos botones, "Deshacer" y "Rehacer".

Para más comodidad del usuario, la aplicación permite ocultar o mostrar cada uno de los seis paneles. Su funcionamiento es básicamente la modificación de una variable de tipo booleano para cada panel. Si el panel es visible pasa a ser invisible al acceder a la funcionalidad de ocultarlo y, si es invisible, pasa a ser visible. También es posible ocultar o mostrar todos los paneles simultáneamente si lo que desea el usuario es ver solamente la escena o, por el contrario, asegurarse de que está viendo todos los paneles.

4.6. Interacción

En esta sección se describen algunas funcionalidades que permiten una interacción más cómoda entre usuario y aplicación, como pueden ser las teclas de acceso rápido a las distintas funcionalidades, los mensajes de aviso que muestra la aplicación y las funcionalidades de deshacer y rehacer cambios de manera rápida en la aplicación.

4.6.1. Teclas de acceso rápido y menús

Todas las funcionalidades que se describen en este capítulo son accesibles desde los distintos paneles descritos en la sección 4.5.4 o a través del menú principal del programa. Sin embargo, un usuario que utilice habitualmente la aplicación

tiene la opción de aprender las teclas de acceso rápido a cada una de las funcionalidades de la aplicación. De esta manera, la interacción entre usuario es mucho más rápida, ya que no es necesario desplazarse por los distintos paneles y menús para acceder a las funcionalidades de la aplicación puesto que todas ellas son accesibles también desde el teclado. Para conocer cuáles son las teclas de acceso rápido de todas y cada una de las funcionalidades de la aplicación, consúltese el manual de usuario (capítulo 5) de esta documentación.

4.6.2. Mensajes de confirmación y de error

La aplicación, en determinadas ocasiones, muestra mensajes de confirmación y error para que el usuario tenga controlado lo que está sucediendo en todo momento. En concreto, nuestra aplicación produce mensajes de confirmación cuando el usuario guarda o carga algún fichero, ya sea de puntos o todo un proyecto, correctamente. De modo similar, si el usuario intenta cargar un fichero cuya extensión o formato es incorrecto, la aplicación le avisa mediante un mensaje de error o un mensaje de advertencia, dependiendo de la gravedad del error. Para más información sobre este tipo de mensajes, consúltese las sección 4.7 de este capítulo o la sección 5.7 del manual de usuario.

4.6.3. Deshacer y rehacer

DVALon, como muchas de las aplicaciones existentes en el mercado, incorpora también la posibilidad de deshacer y rehacer cambios realizados en nuestro proyecto, a la que se puede acceder a través del menú principal o bien a través de un panel específico para esta funcionalidad (ver sección 4.5.4). Para ello, la aplicación sigue un proceso parecido al de cargar un proyecto guardado, aunque sin escribir ni leer de ningún fichero. La aplicación cuenta con un vector de estados, inicialmente vacío, en el que se van encolando los estados. Cada estado está representado por una estructura de tipo "structFile" (consúltese sección 4.7.2 para ver en detalle la estructura). Cada vez que el usuario hace una acción que potencialmente podría querer deshacer, la aplicación guarda el estado del proyecto en una instancia de "structFile" y la almacena en el vector de estados. En concreto, la aplicación salva automáticamente el estado del proyecto y lo almacena en el vector siempre que se produzca una modificación en la nube de puntos, ya sea inclusión, modificación o borrado de un punto, siempre que se introduce o borra un nuevo camino y siempre que se pulsa la tecla de generación de *DVALs* críticos (ver sección 4.5.4) o se borra algún *DVAL*. En el caso de crear un nuevo *DVAL* se ha decidido no salvar el estado, ya que siempre se puede usar la tecla de borrado de *DVALs* para deshacer este cambio.

La función que crea un "structFile" y lo rellena con los datos del proyecto actual es la misma para este caso que para la funcionalidad "Guardar proyecto" (sección 4.7.2), de ahí que el funcionamiento de ambas funcionalidades sea prácticamente el mismo, con la salvedad de que en la funcionalidad que nos ocupa, la aplicación no escribe ni lee de fichero, como ya se ha dicho.

A la hora de deshacer y rehacer, la aplicación simplemente cuenta con un puntero que se va desplazando por el vector de estados y va cargando un estado u otro según lo que haga el usuario. Este puntero, siempre apunta por defecto a la última posición del vector de estados, que es, o bien el estado actual, o bien el último estado al que se ha llegado antes de comenzar a deshacer cambios.

En caso de que el usuario decida deshacer, el puntero del vector de estados se desplaza hasta la posición anterior a la que esté, si la hay, y carga el estado, volviendo así al estado anterior. En cambio, en caso de que el usuario decida rehacer, el puntero del vector de estados se desplaza hasta la siguiente posición, si la hay, y carga el estado de esa posición. Cuando el usuario ya ha deshecho o rehecho todo lo que cree conveniente y realiza una nueva acción, esta acción se guarda en la siguiente posición de la que nos encontremos del vector de estados. El puntero del vector de estados pasa a apuntar a esta nueva posición y todas las posiciones que vienen a continuación de esta, si las hay, se eliminan, de manera que la acción recién hecha por el usuario pasa a ser el estado actual y el último estado en el vector de estados.

Para cargar un estado del vector de estados, al ser estos una instancia de la clase "structFile", la aplicación utiliza la misma función que para cargar un proyecto (sección 4.7.2).

4.7. Ficheros

Nuestra aplicación usa varios tipos de ficheros. Principalmente, existen dos tipos de ficheros. Los ficheros de datos son los que usa la aplicación para guardar y cargar información introducida por el usuario. En este sentido, existen dos tipos de ficheros de datos, uno para guardar la información necesaria para guardar o cargar un proyecto y otro para guardar y cargar conjuntos de puntos. A parte de los ficheros de datos, también existen los ficheros de información, ficheros generados por la aplicación para presentar todos los datos de un proyecto al usuario en un formato de texto entendible.

4.7.1. Proyecto

DVALon permite almacenar en el disco duro del usuario un proyecto, de manera que éste pueda ser recuperado en futuras sesiones. Para ello, se sirve de ficheros de datos que almacenan toda la información relativa un proyecto. Estos ficheros son los que se escriben al guardar un proyecto y se leen al cargarlo y, por tanto, los que hacen posible recuperar proyectos almacenados previamente.

Cuando el usuario accede a la opción de guardar un fichero, la aplicación abre una ventana de diálogo que permite al usuario navegar por su sistema de ficheros y decidir dónde va a guardar su proyecto. Una vez seleccionada la ruta donde se quiere guardar el proyecto, la aplicación procede primero a salvar el estado del proyecto en el cual está trabajando el usuario en el momento en que decide guardarlo y, después, crea un fichero de datos en el cual escribe toda la información necesaria para poder cargar este estado en futuras sesiones. Para salvar el estado, la aplicación utiliza una estructura de datos llamada "structFile" en la que es posible almacenar todos los datos necesarios para retornar un proyecto guardado en cualquier momento sin pérdida de información. En concreto la estructura "structFile" consta de los siguientes campos:

1. Un vector de puntos, que almacena el conjunto P de puntos. Para cada punto almacena su posición (coordenadas x e y).

2. Un vector de seis posiciones, que almacena los colores (valores RGB) de la nube de puntos, triangulación de Delaunay, diagrama de Voronoi, grafo

de Gabriel, grafo de vecinos relativos y árbol generador mínimo respectivamente.

3. Dos campos de tipo "double" (X_0, Y_0) que guardan la posición de la cámara.

4. Otros dos campos, uno de tipo "double" llamado escala, y otro de tipo entero llamado "estZoom" que guardan el escalado de los objetos y el estado del Zoom respectivamente.

5. Un vector que almacena los radios de los DVALs con los que está trabajando el usuario en el momento de guardar.

6. Un vector del mismo tamaño que el anterior que almacena los colores de los DVALs.

7. Un vector del mismo tamaño que los anteriores que guarda los nombres de la lista de DVALs.

8. Un entero que guarda el alcance del $DVAL$ que el usuario tiene seleccionado en el momento de salvar el proyecto.

9. Cuatro enteros que guardan el grosor de los puntos de la nube, de los vértices de Voronoi, el punto que señala la aparición de un evento y de los puntos de los caminos, respectivamente.

10. Un vector de siete posiciones que guarda los grosores de la triangulación de Delaunay, el diagrama de Voronoi, el grafo de Gabriel, el grafo de vecinos relativos, el árbol generador mínimo, de los segmentos de la poligonización de los DVALs y del trazado de los caminos, respectivamente.

11. Un vector de doce booleanos que guarda las casillas que hay activadas / desactivadas en el panel de grafos.

12. Un vector de seis booleanos que guarda los paneles que el usuario ha decidido visualizar o ocultar.

13. Un vector que guarda los caminos que el usuario ha creado hasta el momento de salvar el proyecto.

14. Dos enteros que guardan el número de caminos existentes y el índice del camino seleccionado por el usuario en el momento de guardar el proyecto.

La aplicación, cuando el usuario decide guardar el proyecto, crea una nueva instancia de la clase "structFile" y la rellena con toda la información que se detalla en la descripción anterior de la clase. Una vez creada la estructura y rellenada con todos los datos, la aplicación procede a escribir toda la información en un fichero. Para ello, la clase "file", encargada de guardar y cargar información en ficheros, se sirve de la estructura "structFile" para crear un fichero con extensión ".DATA" que contiene toda la información de la estructura y alguna información adicional necesaria para la posterior lectura del fichero. El usuario no debe preocuparse de añadirle la extensión al fichero a la hora de guardar, ya que la aplicación se encarga de añadírsela en caso de que no se indique extensión o incluso de corregirla en caso de que se intente guardar por error en un fichero

de información con extensión ".txt", ya que los ficheros de este tipo se reservan para otro fin, tal como se explica en la sección 4.7.3. Para facilitar la posterior carga de un proyecto, este fichero ".DATA" tiene un formato fijo, el cual se describe a continuación:

1. Una línea con el número de puntos de P.
2. Una línea para cada punto, que contiene sus coordenadas.
3. Seis líneas más, una para cada uno de los colores que se guardan en el segundo campo de la estructura (valores RGB).
4. Cuatro líneas para los campos X_0, Y_0, escala y "estZoom" respectivamente.
5. Una línea para el número de DVALs existentes.
6. Una línea para guardar el radio de cada uno de ellos.
7. Una línea para guardar el color de cada uno de ellos (valores RGB).
8. Una línea para guardar el nombre de cada uno de ellos, información necesaria para rellenar la lista de DVALs.
9. Una línea para guardar el DVAL seleccionado en el momento de guardar el proyecto.
10. Una línea para cada uno de los grosores referentes a puntos: puntos de la nube, vértices de Voronoi, punto que marca la aparición de eventos y puntos de los caminos.
11. Una línea que guarda los siete grosores de las aristas de grafos y caminos.
12. Una línea que guarda siete booleanos que indican cuáles de las siete casillas del panel de grafos están activadas.
13. Una línea que guarda cinco booleanos que indican cuáles de las cinco casillas para mostrar los cinco grafos de forma parcial están activadas.
14. Una línea de seis booleanos que guarda el estado de los seis panales (oculto o no).
15. Una línea para cada uno de los eventos. En cada una se guarda toda su información: radio, tipo, su arista, las coordenadas del punto que lo marca y sus valores r1 y r2, que guardan las regiones incidentes en este evento.
16. Una línea con el número de caminos presentes.
17. Para cada camino, cuatro líneas que guardan su número de elementos, su color, su nombre y su grosor individual, respectivamente. Además, para cada camino también se escribe una línea por cada uno de sus elementos en la que se guarda un booleano que indica si se trata de un arco o de un segmento rectilíneo, las coordenadas de su punto final y su índice, en caso de que estemos hablando de un segmento y, si estamos tratando un arco, guarda, además, el índice de su centro, sus dos ángulos inicial y final, su radio y si su sentido es antihorario o no.

18. Una línea con el número de caminos creados hasta el momento por el usuario.

19. Una línea que guarda el índice del camino seleccionado en el momento de guardar el proyecto.

Como se puede apreciar, algunas líneas tienen más de un valor. Siempre que una línea tiene varios valores almacenados, estos se separan con espacios.

Al guardar un proyecto, la aplicación muestra un mensaje de aviso con la ruta y el nombre del fichero donde el usuario ha salvado su proyecto para facilitar la comprobación por parte del usuario de que el proyecto se ha guardado en el lugar deseado.

La funcionalidad de cargar un proyecto guardado que ofrece nuestra aplicación es similar, en cuanto a estructuras de datos y ficheros externos utilizados, a la de guardar proyecto. El usuario puede cargar un proyecto guardado en la aplicación en cualquier momento. Una vez se ha accedido a esta opción, la aplicación muestra una ventana de diálogo con la que el usuario puede buscar de manera fácil el proyecto a cargar. Como hemos visto en la sección anterior, los ficheros que contienen los datos para cargar el proyecto y, por tanto, los que debe seleccionar el usuario a la hora de cargar su proyecto son los que tienen extensión ".DATA". En caso de que el usuario seleccione un fichero sin extensión o con otra extensión distinta, la aplicación avisa del error mediante una ventana emergente y no carga nada.

La aplicación, por otra parte, también es capaz de detectar errores de formato en el fichero de datos. Si el fichero ".DATA" que el usuario está intentando cargar no tiene el formato exacto descrito anteriormente, la aplicación le avisa mediante una ventana emergente, indicándole qué parte del fichero de datos es incorrecta. En caso de que haya más de un error de formato, la aplicación avisa de cada uno de ellos con una ventana aparte. Una vez la aplicación ha avisado de todos los errores de formato, muestra un mensaje avisando de que el fichero no se ha cargado correctamente. A continuación, la aplicación carga todos los datos que puede, teniendo en cuenta que es posible que algunas estructuras no estén completas o que haya alguna incoherencia entre el estado de los paneles y los elementos mostrados por pantalla. *DVALon* no puede generar ficheros de datos con formato o extensión incorrectos al guardar un proyecto. Sin embargo, la aplicación comprueba siempre el formato del fichero ya que el usuario puede decidir modificar el fichero de datos manualmente y cometer errores de formato al intentarlo.

Si todo va bien al cargar un fichero de datos, la aplicación avisa al usuario de que todo ha ido correctamente y se cargan los datos del proyecto seleccionado. Para cargar estos datos, la aplicación realiza un proceso similar al seguido al guardar un proyecto pero en orden inverso. Primero crea una instancia de la clase "structFile", descrita en la sección anterior, y a continuación abre el fichero ".DATA" seleccionado. Una vez abierto el fichero, va leyendo línea por línea, siguiendo el formato descrito también en la sección anterior, y rellenando con los datos leídos la estructura de tipo "structFile". Como es lógico, de un fichero de texto siempre se leen caracteres con lo que, según convenga, la aplicación convierte estos caracteres a los tipos que se correspondan con los de las estructuras de datos de los distintos elementos de la aplicación (enteros, doubles, etc.). Una vez rellena la estructura de datos "structFile", la aplicación pasa toda

esta información a la clase que gestiona la aplicación, y ésta va leyendo cada uno de sus campos y rellenando con ellos las estructuras de datos que guardan y muestran las distintas funcionalidades de la aplicación. De esta manera, el usuario consigue recuperar su proyecto en el punto exacto donde lo guardó.

4.7.2. Puntos

Como se ha visto en la sección anterior, *DVALon* permite salvar un proyecto al completo para que sea posible retomarlo en futuras sesiones. Sin embargo, puede ser interesante para un usuario trabajar en distintas sesiones con el mismo conjunto P de puntos sin cargar necesariamente todo el proyecto tal y como lo dejó en la última sesión. También se puede dar el caso de que, por comodidad y, sobretodo, por necesidades de precisión, el usuario prefiera no introducir la nube de puntos mediante clics de ratón en la pantalla, sino a través de las coordenadas de los mismos.

Por estos motivos, *DVALon* incluye la posibilidad de guardar y cargar ficheros de puntos. Una vez se accede a esta funcionalidad, la aplicación abre una ventana de diálogo que permite al usuario navegar por su sistema de ficheros y decidir dónde va a guardar su fichero de puntos. El fichero seleccionado es el fichero destino de las coordenadas del conjunto de puntos con la que el usuario esté trabajando en ese momento, sin importar el resto de elementos que haya en el proyecto.

A diferencia de la funcionalidad de guardar un proyecto, los ficheros de puntos no tienen extensión concreta, de manera que queda a elección del usuario la extensión a asignar a este tipo de ficheros o incluso puede no asignarles extensión. Lo que sí tienen los archivos de puntos es un sencillo formato, que es común a todos los ficheros de puntos creados y leídos por la aplicación. Este formato es el siguiente:

1. Una primera línea que contiene el número de puntos de la nube que se guardan en el fichero.

2. Para cada punto, una línea que contiene las coordenadas x e y de éste, separadas por un espacio.

Siempre que se guarda un fichero de puntos, aparece un mensaje para recordar al usuario la ubicación del fichero guardado y facilitar así la comprobación de que se ha guardado en la ubicación deseada.

Los ficheros de puntos se pueden cargar en la aplicación. Al hacerlo, la aplicación muestra una ventana de diálogo con la que el usuario puede buscar de manera fácil el fichero de puntos que va a cargar. Puesto que este tipo de ficheros pueden tener cualquier extensión, es responsabilidad del usuario recordar la extensión utilizada, si es que decide utilizar alguna, al guardar los ficheros de puntos. Esto, que a priori podría parecer una carencia de la aplicación, en realidad tiene una gran ventaja.

Si comparamos el formato de los ficheros ".DATA" que guardan un proyecto (sección anterior) y el descrito aquí para los ficheros de puntos, observamos que ambos empiezan de la misma manera: una línea para el número de puntos y tantas líneas como puntos haya con las coordenadas de estos separadas con espacios. Esta coincidencia de ambos formatos, unida a que los ficheros de puntos, como hemos dicho, no tienen extensión concreta, permite que la aplicación sea

capaz de cargar una nube de puntos tanto de un fichero de puntos como de un fichero de un proyecto completo. Esto permite que, si en un momento determinado el usuario únicamente necesita la nube de puntos que tiene guardada en un proyecto, puede cargar también un fichero ".DATA" como si fuera un fichero de puntos, en cuyo caso la aplicación solo lee la nube de puntos del fichero, obviando el resto de datos. Podría parecer redundante, entonces, la existencia de ficheros de puntos pero, al contener mucha menos información, este tipo de ficheros ocupa mucho menos que un fichero que guarda un proyecto completo. Además, nos ha parecido cómodo para el usuario facilitarle dos tipos de ficheros.

Una vez se ha seleccionado el fichero que se va a cargar, la aplicación primero comprueba que el formato del fichero sea el correcto. En caso de que lo sea, lee los puntos y rellena la estructura de la aplicación que guarda la nube de puntos con los datos leídos, devolviendo el resto de estructuras a su estado inicial. De esta manera, conseguimos empezar un nuevo proyecto a partir de una nube de puntos leída desde un fichero. A continuación, aparece un mensaje que confirma que los puntos han sido cargados correctamente. En caso de que el formato de los puntos sea incorrecto, aparece un mensaje de error que avisa al usuario de este hecho, y se cargan aquellos puntos que tengan formato correcto. Un fichero de puntos se considera que tiene formato incorrecto cuando alguno de los puntos presenta menos o más coordenadas de las esperadas (sólo deben tener coordenadas x e y separadas por espacios), si alguna coordenada tiene algún carácter no numérico, a excepción del punto para expresar valores decimales, o si falta la primera línea que indica el número de puntos que hay que leer a continuación.

En caso de que haya menos puntos de los especificados en la primera línea del fichero, se cargan todos los que haya igualmente. En cambio, si hay más de los declarados, los puntos en exceso son obviados por la aplicación a la hora de leer. En estos dos últimos casos, el programa avisa al usuario mediante una ventana emergente. Si el usuario decide modificar por su cuenta un fichero de puntos es responsabilidad suya asegurarse de que el número de puntos introducidos es el correcto. *DVALon* siempre guarda los archivos de puntos con formato correcto, así que solo pueden existir errores de formato en caso de que se modifiquen los ficheros manualmente.

4.7.3. Fichero de información

Además de este fichero ".DATA", la aplicación también crea un fichero de información con extensión ".txt" a la hora de guardar un proyecto. Este es un fichero de carácter informativo, destinado a que el usuario conozca las características del proyecto que acaba de guardar. Este fichero contiene información numérica sobre los grafos, diagramas y caminos, y da todo tipo de información detallada sobre cada uno de los elementos que el usuario tiene en su proyecto. La información se presenta siempre de la misma manera y en el mismo orden, con lo que se podría decir que este fichero de información también tiene un cierto formato. En concreto, este fichero de información contiene, en este orden:

1. Toda la información de la DCEL que guarda la triangulación de Delaunay. Para cada vértice, sus coordenadas, para cada arista, su vértice origen, sus aristas siguiente, anterior y gemela y su cara incidente y, para cada cara, su arista incidente.

2. Toda la información de la DCEL que guarda el diagrama de Voronoi, con la especificación de qué puntos son vértices de Voronoi y cuáles pertenecen a rectas o semirrectas y son, por tanto, puntos auxiliares, qué aristas son segmentos y qué aristas son rectas y semirrectas, etc.

3. Información sobre los diagramas de Voronoi de alcance limitado existentes en el momento de guardar el proyecto. Para cada uno de ellos, el fichero muestra el valor de su alcance, el número de componentes conexas que lo forman y la descripción de su frontera. La descripción de la frontera incluye vértices, arcos y segmentos, con detalle de coordenadas, ángulos, etc. Además, para cada una de las componentes conexas del DVAL describe los puntos de la nube, con sus coordenadas, que pertenecen a ella. También se describen los segmentos que forman su poligonización.

4. Toda la información sobre los eventos o cambios combinatorios que se producen en esta nube de puntos. Aparecen ordenados por orden creciente de radio y, para cada uno de ellos, aparece su tipo, el radio en el que se produce y el vértice de Voronoi o punto de tangencia en el que se produce el evento correspondiente, ambos con sus coordenadas.

5. Toda la información referente a los grafos de Gabriel y vecinos relativos y al árbol generador mínimo. En concreto, para cada vértice se dan todos sus vértices adyacentes y la longitud al cuadrado de la arista que los une.

6. Toda la información referente a los caminos existentes en el momento de guardar el proyecto. En concreto, para cada uno se muestra su longitud total, si es o no de desviación mínima y si es o no de desviación mínima local. Además, se da el número de elementos que lo forman y, para cada uno de ellos, una información distinta dependiendo de si son segmentos o arcos. En caso de los segmentos, aparecen las coordenadas de sus puntos origen y final, y el índice de su punto origen dentro de la DCEL. Si se trata de un arco, además de la información anterior, también se indican las coordenadas de su centro.

Capítulo 5

Manual de usuario

Este capítulo es un manual de usuario destinado a que cualquiera que se disponga a usar la aplicación conozca la forma de introducir los datos necesarios y de obtener los resultados deseados de forma sencilla y eficaz.

En cada uno de las secciones se explica detalladamente cómo visualizar cada una de las funcionalidades y la forma de consultar los resultados producidos por la aplicación.

5.1. Insertar puntos

En este sección se explica con detalle cómo crear un nuevo proyecto y cómo introducir un conjunto de puntos en la aplicación o modificar uno ya existente. En concreto, en el primer apartado se detalla qué ha de hacer el usuario para crear un nuevo proyecto y cómo introducir puntos en la aplicación mediante el ratón. En el segundo apartado se explica cómo introducir una nube de puntos mediante un fichero de puntos y, en el tercer apartado, se describe cómo genera un conjunto de puntos distribuidos aleatoriamente.

5.1.1. Nuevo proyecto y puntos por ratón

Es posible crear un nuevo proyecto con *DVALon* de dos maneras distintas pero equivalentes. Una de ellas es a través del menú principal del programa. Debe accederse al menú "Archivo" y, dentro de él, seleccionar la opción "Nuevo". La otra opción es pulsando simultáneamente las teclas Control, Shift(Mayúscula) y "N" (5.1). En ambos casos el resultado es el mismo: el estado en el que se encuentra la aplicación se pierde y ésta vuelve al estado inicial. Por ello es recomendable guardar el proyecto antes de acceder a esta funcionalidad.

Si el usuario está trabajando sobre un proyecto, nuevo o no, puede incluir puntos en el conjunto P en cualquier momento mediante el ratón. Para ello, solo debe hacer clic en la zona del plano donde quiere que aparezca un nuevo punto. Si hace clic sobre otro punto de P ya existente, en lugar de crear un nuevo punto, el usuario puede arrastrar este punto manteniendo pulsado el botón izquierdo del ratón, modificando así la posición del mismo.

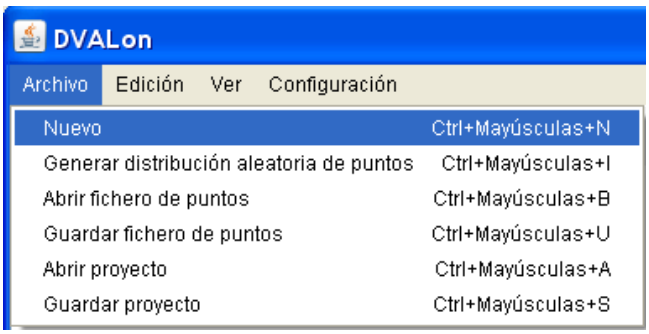

Figura 5.1: Crear un nuevo proyecto.

5.1.2. Por fichero

Si el usuario desea cargar un conjunto de puntos guardado, puede hacerlo accediendo al menú "Archivo" y seleccionando la opción "Abrir fichero de puntos" o bien mediante una combinación determinada de teclas (ver Figura 5.2).

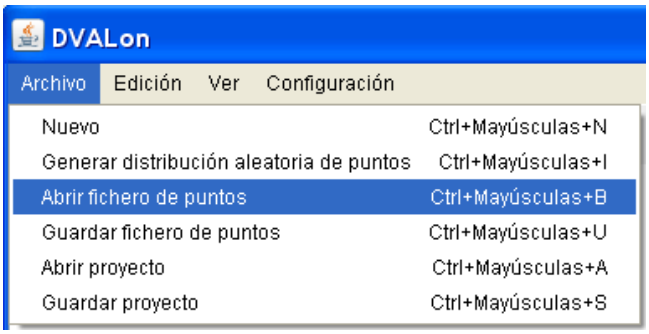

Figura 5.2: Cargar un fichero de puntos.

Una vez seleccionada esta opción, aparece en pantalla una ventana de diálogo similar a la de la Figura 5.37, en la cual se debe seleccionar el fichero de puntos a abrir o bien especificar manualmente su ruta completa en el campo "Nombre del Archivo". Una vez hecho esto, el usuario debe pulsar el botón "Aceptar". Si el fichero tiene el formato especificado en la sección 4.1.2, la aplicación carga los puntos y muestra un mensaje informando de que se ha cargado todo correctamente. En caso contrario, la aplicación carga únicamente los puntos con formato correcto, y muestra un mensaje avisando de que el formato del fichero no es correcto.

5.1.3. Aleatoriamente

La funcionalidad de generar una distribución de puntos aleatoria también es accesible desde el menú principal o mediante la combinación de teclas que aparece en la Figura 5.3. Esta opción se encuentra en el menú "Archivo", y su nombre es "Generar distribución de puntos aleatoria".

Una vez el usuario accede a esta opción, la aplicación muestra la ventana que se recoge en la Figura 5.4. Como vemos en la imagen, en la parte superior

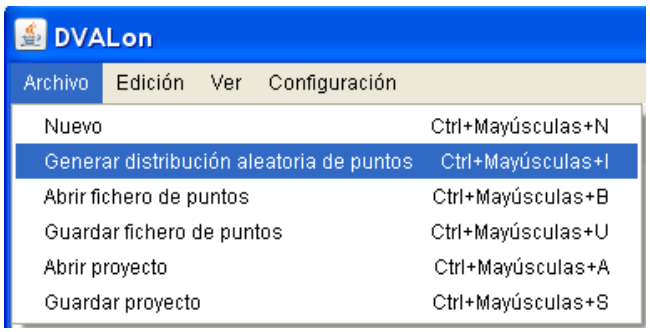

Figura 5.3: Generar distribución de puntos aleatoria.

de esta ventana se debe indicar el número de puntos a generar, valor que por defecto es 0. Un poco más abajo, se pueden ver cuatro cuadros de texto más, en los que el usuario debe indicar las coordenadas de las esquinas superior izquierda e inferior derecha del rectángulo contenedor de los puntos. Por defecto, son el $(0, 0)$ y el $(800, 600)$.

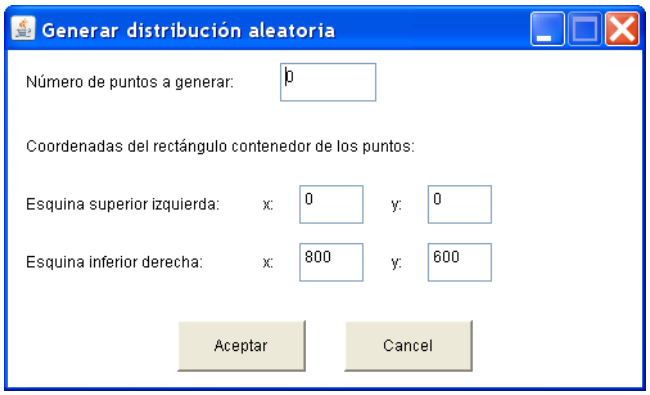

Figura 5.4: Ventana para generar una distribución de puntos aleatoria.

Una vez rellenos los cinco campos mencionados, se termina el proceso de generación de una distribución aleatoria de puntos pulsando el botón "Aceptar" que aparece en la parte inferior de la ventana. En ese momento el estado en el que se encuentra la aplicación se pierde y esta vuelve al estado inicial, para luego generar una nueva nube de puntos cuyas coordenadas tienen un valor aleatorio contenido en el rectángulo especificado. Por ello es recomendable guardar el proyecto antes de acceder a esta funcionalidad (para saber cómo guardar, véase a la sección 5.7.1). Si, en cambio, el usuario pulsa el botón "Cancelar", la aplicación cierra la ventana de generación de distribuciones aleatorias de puntos y no se produce ningún cambio.

5.2. Grafos de proximidad

Estas sección consta de cuatro apartados en los que se describe cómo visualizar, modificar y, en general, trabajar con los grafos de proximidad calculados

por nuestra aplicación: triangulación de Delaunay, grafo de Gabriel, grafo de vecindad relativa y árbol generador mínimo euclídeo respectivamente.

5.2.1. Triangulación de Delaunay

DVALon construye la triangulación de Delaunay automáticamente sin que el usuario deba indicárselo de manera explícita y la va actualizando de manera automática siempre que se produce algún cambio en la nube de puntos introducida. Sin embargo, aunque la triangulación de Delaunay esté siempre calculada, no es visible por defecto. Para poder visualizarla basta con activar la casilla "Delaunay", que se encuentra en primer lugar en el panel de grafos de la parte superior izquierda de la ventana (ver Figura 5.5), o pulsar la tecla de acceso rápido 'd'. Una vez marcada, aparece en la ventana la triangulación de Delaunay de los puntos de la nube.

Figura 5.5: Panel de grafos en la parte superior izquierda de la aplicación.

Si el usuario quiere únicamente visualizar la triangulación de Delaunay de alcance limitado por el radio seleccionado, debe marcar la casilla "Parcial" situada bajo la casilla "Delaunay" marcada previamente. De esta manera, sólo se visualizan aquellas aristas de la triangulación que están contenidas dentro del diagrama de Voronoi de alcance limitado, *DVAL*, seleccionado, esto es, las aristas de longitud menor o igual al alcance escogido. Sólo se puede marcar la casilla "Parcial" si está marcada la casilla "Delaunay".

En la Figura 5.6 se ilustran ambas funcionalidades.

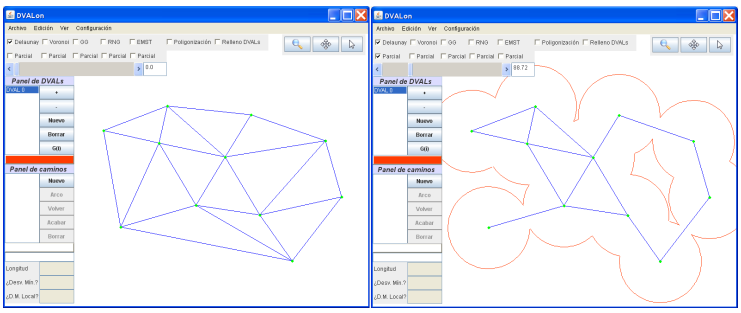

Figura 5.6: Triangulación de Delaunay y Triangulación de Delaunay de alcance limitado.

Para desactivar esta funcionalidad, basta con desmarcar la casilla "Delaunay" o volver a pulsar la tecla de acceso rápido 'd'.

5.2.2. Grafo de Gabriel

De la misma manera que lo hace con la triangulación de Delaunay, *DVALon* construye el grafo de Gabriel automáticamente sin que el usuario deba indicárselo y va actualizándolo automáticamente al modificar la nube de puntos. Para

poder visualizarlo basta con activar la casilla “GG”, que se encuentra también en el panel de grafos de la parte superior de la ventana, o pulsar la tecla de acceso rápido ‘g’. Una vez marcada, aparece en la ventana el grafo de Gabriel de los puntos de la nube.

Si el usuario quiere únicamente visualizar el grafo de Gabriel de alcance limitado por el radio seleccionado, debe marcar la casilla “Parcial” situada bajo la casilla “GG” marcada previamente. De esta manera, sólo se visualizan las aristas del grafo que están contenidas dentro del *DVAL* seleccionado, esto es, las aristas de longitud menor o igual al alcance escogido. Sólo se puede marcar la casilla “Parcial” si está marcada la casilla “GG”.

En la Figura 5.7 se ilustran ambas funcionalidades.

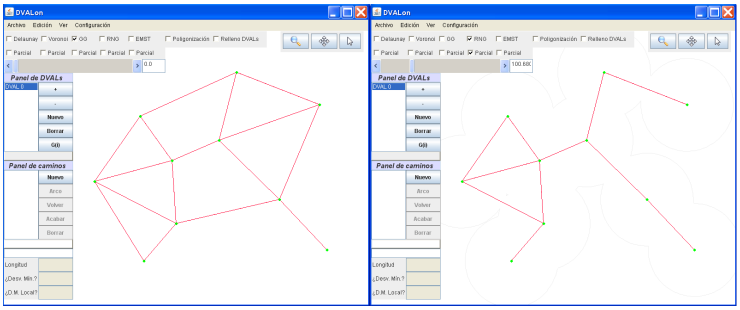

Figura 5.7: Grafo de Gabriel y grafo de Gabriel de alcance limitado.

Para desactivar esta funcionalidad, basta con desmarcar la casilla “GG” o volver a pulsar la tecla de acceso rápido ‘g’.

5.2.3. Grafo de vecindad relativa

Como en los casos anteriores, *DVALon* construye el grafo de vecinos relativos automáticamente sin que el usuario deba indicárselo y va actualizándolo automáticamente al modificar la nube de puntos. Para poder visualizarlo basta con activar la casilla “RNG”, que se encuentra también en el panel de grafos de la parte superior de la ventana, o pulsar la tecla de acceso rápido ‘r’. Una vez marcada, aparece en la ventana el grafo de vecinos relativos de los puntos de la nube.

Si el usuario quiere únicamente visualizar el grafo de vecinos relativos de alcance limitado por el radio seleccionado, debe marcar la casilla “Parcial” situada bajo de la casilla “RNG” marcada previamente. De esta manera, sólo se visualizan las aristas del grafo que están contenidas dentro del *DVAL* seleccionado, esto es, las de longitud menor o igual al alcance escogido. Sólo se puede marcar la casilla “Parcial” si está marcada la casilla “RNG”.

En la Figura 5.8 se ilustran ambas funcionalidades.

Para desactivar esta funcionalidad, basta con desmarcar la casilla “RNG” o volver a pulsar la tecla de acceso rápido ‘r’.

5.2.4. Árbol generador mínimo euclídeo

Como en el resto de grafos, *DVALon* construye árbol generador mínimo automáticamente sin que el usuario deba indicárselo y va actualizándolo au-

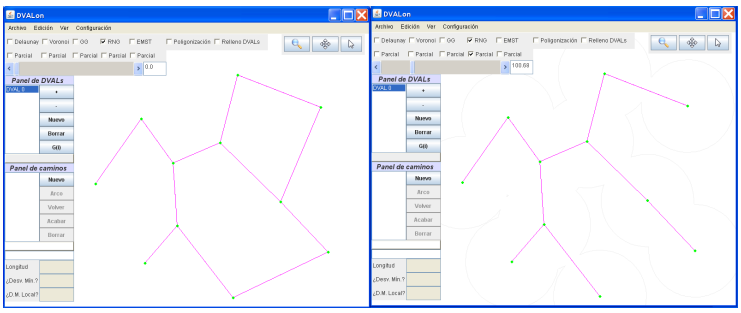

Figura 5.8: Grafo de vecinos relativos y grafo de vecinos relativos de alcance limitado.

tomáticamente al modificar la nube de puntos. Para poder visualizarlo, basta con activar la casilla "EMST", que se encuentra también en el panel de grafos de la parte superior de la ventana, o pulsar la tecla de acceso rápido 'm'. Una vez marcada, aparece en la ventana el árbol generador mínimo euclídeo de los puntos de la nube.

Si el usuario quiere únicamente visualizar el árbol generador mínimo de alcance limitado por el radio seleccionado, debe marcar la casilla "Parcial" situada bajo de la casilla "EMST" marcada previamente. De esta manera, sólo se visualizan las aristas del árbol que están contenidas dentro del *DVAL* seleccionado, esto es, las de longitud menor o igual al alcance escogido. Sólo se puede marcar la casilla "Parcial" si está marcada la casilla "EMST".

En la Figura 5.9 se ilustran ambas funcionalidades.

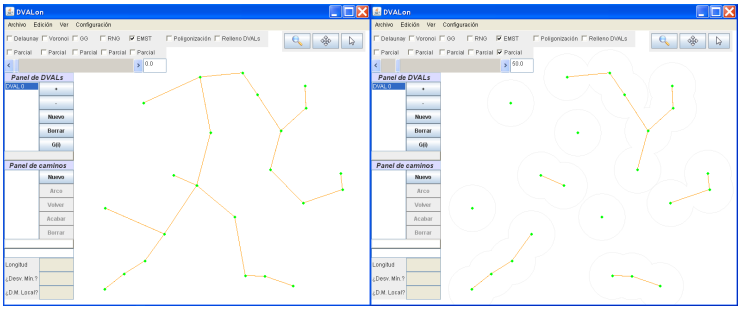

Figura 5.9: Árbol generador mínimo y árbol generador mínimo de alcance limitado.

Para desactivar esta funcionalidad, basta con desmarcar la casilla "EMST" o volver a pulsar la tecla de acceso rápido 'm'.

5.3. Diagrama de Voronoi

En esta sección se explican todas las funcionalidades disponibles para el usuario relacionadas con la visualización y tratamiento del diagrama de Voronoi y de los diagramas de Voronoi de alcance limitado.

5.3.1. Diagrama de Voronoi ordinario ($Vor(P)$)

DVALon calcula el diagrama de Voronoi, $Vor(P)$ sin que el usuario se lo indique, y lo va actualizando automáticamente después de producirse cualquier cambio en la nube de puntos. El diagrama $Vor(P)$ no es visible por defecto en la aplicación y, para visualizarlo, el usuario debe activar la casilla "Voronoi", situada en el panel de grafos (ver Figura 5.5), o pulsar la tecla de acceso rápido "v". Una vez marcada, aparecerá el diagrama de Voronoi ordinario de los puntos de la nube en la ventana de la aplicación.

Para visualizar las regiones de Voronoi correspondientes a cada punto de la nube, que han sido limitadas por un alcance r, el usuario debe haber creado un *DVAL*, haberle asignado el radio r, y tenerlo seleccionado (para saber como crear y manipular *DVALs* véase la Sección 5.3.2). El usuario puede elegir entre visualizar las regiones individuales de Voronoi de P de alcance limitado por r, o visualizar el diagrama de Voronoi ordinario, activando o desactivando la casilla "Parcial" situada justo debajo de la casilla "Voronoi" en el panel de grafos.

En la Figura 5.10 se pueden observar las diferencias entre ambas funcionalidades.

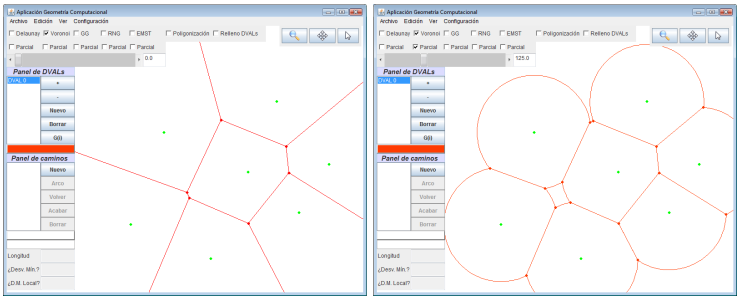

Figura 5.10: Diagrama de Voronoi ordinario y de alcance limitado.

Para dejar de visualizar $Vor(P)$ basta con desactivar la casilla "Voronoi" en el panel de grafos o volver a pulsar la tecla de acceso rápido "v".

5.3.2. Diagramas de Voronoi de alcance limitado (*DVALs*)

La aplicación puede visualizar tantos diagramas de Voronoi de alcance limitado, *DVALs*, como el usuario desee, pero para ello el usuario debe comunicarle a la aplicación qué alcance quiere que tenga cada uno. A diferencia del resto de grafos de los que hemos hablado en este capítulo, los *DVALs* no se calculan por defecto, sino cuando el usuario los crea o modifica, aunque se actualizan automáticamente al producirse cualquier cambio en la nube.

Al iniciarse la ejecución de la aplicación, o cuando se elige la opción "nuevo", se crea por defecto un único *DVAL* de color rojo y radio 0 para que el usuario pueda manipularlo si lo necesita.

Cuando la nube de puntos dispone de 2 o más puntos, el usuario dispone de una serie de opciones para crear, eliminar o manipular *DVALs*. A continuación se enumeran las funcionalidades relacionadas con los *DVALs*.

1. Crear *DVAL*

Para crear un nuevo *DVAL*, el usuario puede hacer clic en el botón "Nuevo" del panel de *DVALs* situado en la parte izquierda de la ventana de la aplicación (este panel puede verse en la Figura 5.11). También puede crear un nuevo *DVAL* pulsando la tecla de acceso rápido "n".

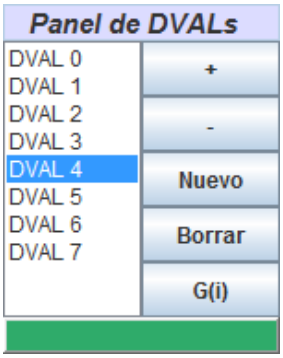

Figura 5.11: Panel de *DVALs*.

Al crear un nuevo *DVAL*, la aplicación lo crea por defecto con el radio del *DVAL* seleccionado en ese momento y, en caso de no haber ninguno, lo crea con radio 0. El color con el que crea el *DVAL* por defecto lo elige la aplicación en función del número de *DVALs* que se hayan creado hasta entonces, de forma que los colores sean lo suficientemente distintos como para poder distinguirlos con facilidad. Después de la creación de un *DVAL*, el usuario puede cambiar tanto el radio como el color con los que ha aparecido.

Cuando se crea un nuevo *DVAL*, este se selecciona en la lista del panel de *DVALs* por defecto.

2. Seleccionar *DVAL*

   Cuando se crea un nuevo *DVAL*, este se selecciona por defecto, pero existen otras maneras de seleccionar *DVALs* incluso después de que hayan sido deseleccionados. En el panel de *DVALs* se puede ver qué *DVAL* está seleccionado en cada momento. Es aquel que aparece marcado en un fondo azul (en la Figura 5.11 está seleccionado el *DVAL* 4).

   Para seleccionar un *DVAL* determinado basta con hacer clic sobre su nombre en el panel de *DVALs*. También es posible cambiar de *DVAL* seleccionado pulsando la barra espaciadora. Al pulsar esta tecla, la aplicación va cambiando la selección de *DVAL*, seleccionando el siguiente al que hay seleccionado en ese momento. De esa manera el usuario también puede ir cambiando de DVAL hasta llegar al que desea seleccionar.

   Cuando se elimina un *DVAL*, la aplicación selecciona el siguiente *DVAL* en la lista, si queda alguno.

3. Eliminar *DVAL*

   Para eliminar un *DVAL* el usuario debe tener seleccionado el *DVAL* que quiere eliminar. No se puede eliminar un *DVAL* si primero no se selecciona.

Una vez el *DVAL* escogido está seleccionado, el usuario puede hacer clic en el botón "Borrar" del panel de *DVALs* para eliminarlo. También puede eliminar un *DVAL* pulsando la tecla de acceso rápido "x".

Cuando se elimina un *DVAL*, la aplicación selecciona el siguiente *DVAL* en la lista, si queda alguno.

4. Modificar *DVAL*

   Para modificar un *DVAL*, el usuario debe tenerlo seleccionado previamente.

   Cuando hablamos de modificar un *DVAL* no nos referíamos a los cambios inducidos en todos los *DVALs* al añadir, eliminar o modificar un punto de la nube. Esos cambios los realiza la aplicación automáticamente. Las modificaciones a las que nos referimos aquí, son aquellas que afectan al color o al radio de un *DVAL* determinado.

   Para cambiar el color de un *DVAL*, el usuario puede hacer clic en el botón que muestra el color actual del *DVAL* en el panel de *DVALs*. Al hacer clic le aparece una sencilla interficie en la que puede elegir el color que desea que tenga el *DVAL*. Otro modo de cambiar el color de un *DVAL* que no requiere que este esté seleccionado es seleccionando en el menú "Configuración" la opción "Modificar colores de *DVALs*". En este caso, al usuario le aparece una interficie que muestra los colores de todos los *DVALs* creados, con botones del color de estos. Esta interficie se puede ver en la Figura 5.12. Haciendo clic en cualquiera de esos botones el usuario puede cambiar el color del *DVAL* al que corresponde el botón que ha clicado.

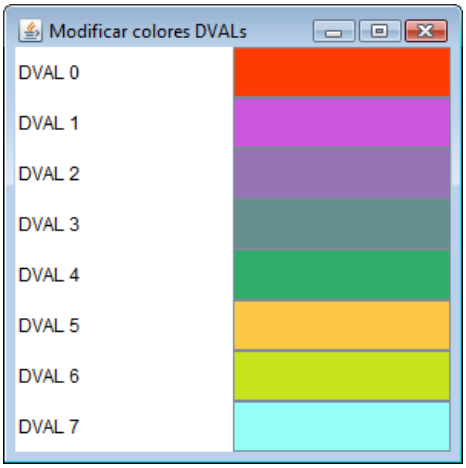

Figura 5.12: Interficie de selección de colores de *DVALs*.

Para cambiar el radio de un *DVAL*, el usuario tiene varias opciones que puede utilizar.

En primer lugar, el usuario puede ir variando el radio, saltando entre los valores de r que suponen un cambio combinatorio o evento para la nube de puntos. Estos cambios combinatorios pueden producirse en valores de r en los que se unen dos componentes conexas, se genera o desaparece

un agujero, o desaparece uno de los arcos de circunferencia de la frontera de una componentes conexa. Para ir saltando entre estos valores críticos de r, el usuario puede hacer clic en el botón "+" del panel de *DVALs* o pulsar la tecla "+" del teclado para saltar al siguiente evento, o hacer clic en el botón "−" del panel de *DVALs* o pulsar la tecla "−" del teclado para saltar al evento anterior. El evento siguiente o anterior se decide en función del valor actual que tiene el radio del *DVAL* seleccionado.

Otro método que puede utilizar el usuario para modificar el radio es manipular la barra del panel de radios situada justo encima del panel de *DVALs* (el panel de radios se puede ver en la Figura 5.13). El usuario puede variar el radio de una manera casi continua entre los límites establecidos por la aplicación. El movimiento no es totalmente continuo, pues la barra consta de 1000 "lugares" en los que puede detenerse. Por lo tanto, el conjunto de valores elegibles para r es finito. El límite inferior de los valores elegibles mediante la barra siempre es 0, y el superior depende de la ventana de visualización. No se permiten valores mayores a aquellos a partir de los cuales no se detectaría ningún cambio aunque incrementáramos el radio.

Figura 5.13: Panel de radios de la aplicación.

Por último, el usuario puede variar el radio de un *DVAL* estableciendo exactamente el valor del radio que desea que este tenga. Esto puede hacerse introduciendo el valor exacto en la casilla habilitada para ello a la derecha del panel de radios. El usuario puede introducir cualquier valor, puede incluso introducir valores superiores al límite superior establecido para la barra, pero si introduce valores inferiores a 0, la aplicación no reproduce los cambios.

Cuando el usuario ha introducido, de cualquiera de las tres maneras posibles, el valor de un radio en el que se produce un evento, en la ventana de visualización se ve el lugar en el que se produce el evento mediante un punto negro. Este lugar puede ser, por ejemplo, el punto por el que se unen dos componentes, o en que se cierra o desaparece un agujero. En la Figura 5.14 podemos ver un ejemplo de *DVAL* que tiene un radio crítico y sobre el que se muestra el lugar donde se produce el evento.

5. Conjunto de *DVALs* $G(i)$

   Esta funcionalidad permite al usuario generar con un simple clic todo el conjunto de *DVALs* con radios críticos que se produce en la nube de puntos actual. Los radios críticos son aquellos en los que se unen dos componentes conexas en la nube de puntos. Cuando el usuario decide ejecutar esta funcionalidad, se eliminan todos los *DVALs* preexistentes para que estos no interfieran, y se crean tantos *DVALs* como cambios combinatorios de este tipo se produzcan para la nube de puntos introducida, cada uno con un radio correspondiente a uno de estos eventos. Los *DVALs* quedan ordenados en el panel de *DVALs* en orden creciente de radio, para que el usuario pueda ver la progresión en el crecimiento y los momentos exactos en que se unen dos componentes de forma ordenada. Para activar esta

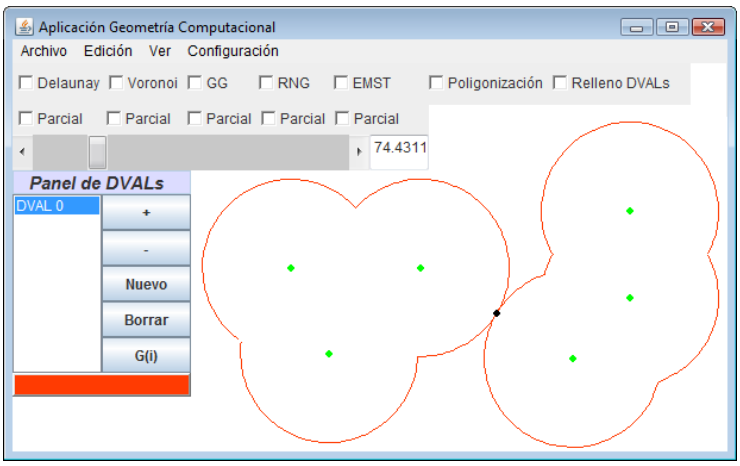

Figura 5.14: *DVAL* con radio crítico en el que se unen dos componentes conexas.

funcionalidad, basta con que el usuario haga clic en el botón "$G(i)$" del panel de *DVALs* o pulse la tecla de acceso rápido "q".

En la Figura 5.15 puede verse un ejemplo del resultado de esta funcionalidad para una nube de puntos muy simple.

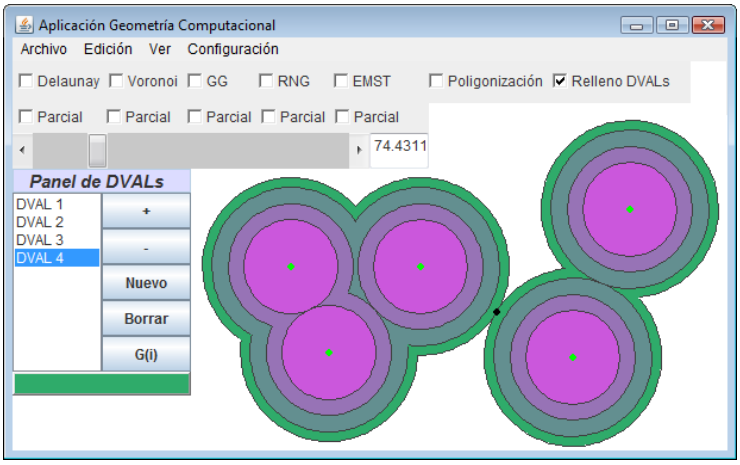

Figura 5.15: Conjunto $G(i)$ para una nube de puntos simple.

6. Rellenar componentes

   Para rellenar las componentes conexas del color correspondiente a cada una y visualizar así mejor la zona que cubren, el usuario debe activar la casilla "Relleno *DVALs*", del panel de grafos (ver Figura 5.5). No hay tecla de acceso rápido para esta funcionalidad.

   En la Figura 5.16 se observa un ejemplo con varios *DVALs* rellenos.

   Para dejar de visualizar los *DVALs* rellenos basta con desactivar la casilla "Relleno *DVALs*" del panel de grafos.

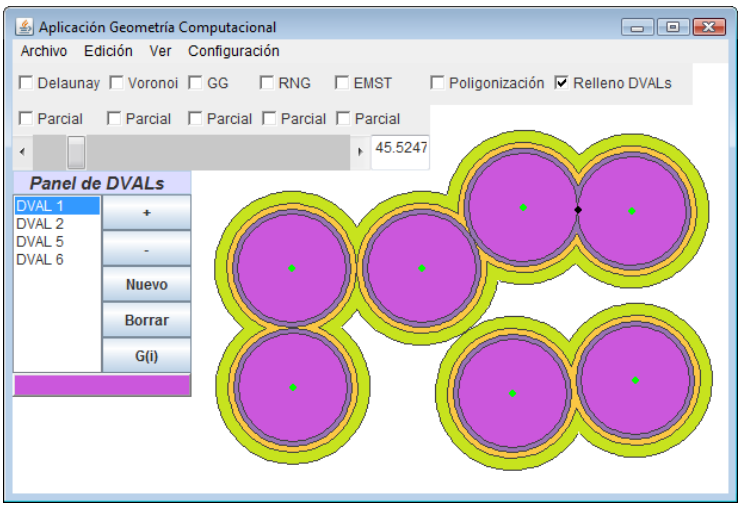

Figura 5.16: Ejemplo de *DVALs* rellenos.

5.3.3. Poligonización

La poligonización de las componentes conexas es calculada por la aplicación de forma automática, pero el usuario puede decidir si quiere visualizarla o no. Para visualizarla el usuario debe activar la casilla “”Poligonización”, situada en el panel de grafos (ver Figura 5.5), o pulsar la tecla de acceso rápido “p”.

En la Figura 5.17 se puede observar un ejemplo de poligonización de una componente conexa.

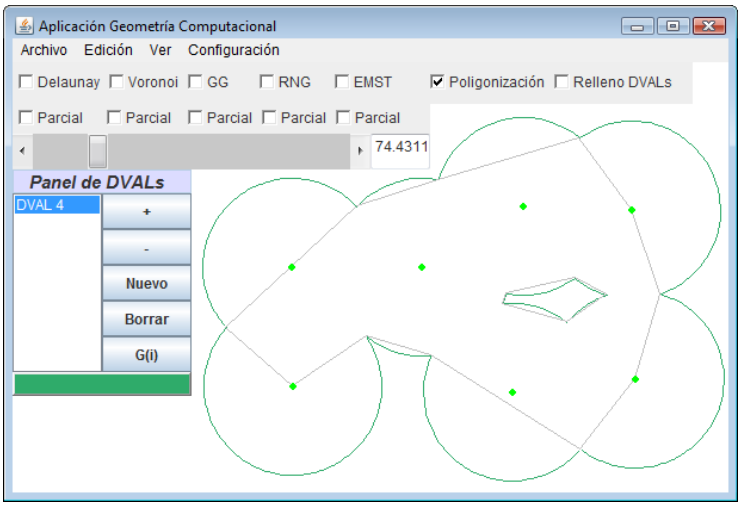

Figura 5.17: Poligonización de una componente conexa.

Para dejar de visualizar la poligonización basta con desactivar la casilla “Poligonización” del panel de grafos, o volver a pulsar la tecla de acceso raápido “p”.

5.4. Caminos

El usuario puede crear tantos caminos como desee, realizar toda clase de modificaciones sobre dichos caminos y consultar una serie de parámetros de estos.

Además de todas las modificaciones que el usuario puede realizar sobre los caminos, estos se actualizan automáticamente cada vez que hay algún cambio en la nube de puntos. Estas modificaciones automáticas incluyen cambios en el diseño del camino (puntos por los que pasa) o cambios en la información de interés para el usuario calculada sobre los caminos. Esta información de interés incluye la longitud del camino, y si el camino es o no de desviación mínima y si es o no desviación mínima local.

Cuando la nube de puntos consta de 3 o más puntos, el usuario dispone de una serie de opciones para crear, eliminar o manipular caminos, así como consultar los parámetros de los caminos existentes. A continuación se enumeran las funcionalidades relacionadas con los caminos.

1. Crear camino

   Para entrar en el modo de creación de caminos, el usuario puede hacer clic en el botón "Nuevo" del panel de Caminos situado en la parte izquierda de la ventana de la aplicación, justo debajo del panel de *DVALs* (el panel de Caminos se puede ver en la Figura 5.18). También puede entrar en el modo de creación de caminos pulsando la tecla de acceso rápido "c".

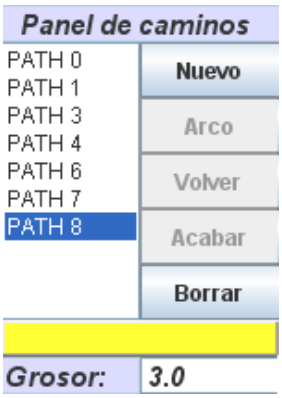

Figura 5.18: Panel de Caminos.

   Una vez la aplicación está en modo de creación de caminos, el usuario puede definir el camino. Para ello, el usuario puede introducir segmentos del camino mediante clics en la ventana de visualización o introducir arcos de circunferencia.

   Para introducir un arco de circunferencia, el usuario debe hacer clic en el botón "Arco" del panel de Caminos o pulsar la tecla de acceso rápido "a" estando en el modo de creación de caminos y habiendo introducido al menos el punto inicial del camino. Después de esto, el usuario debe determinar el centro del arco que va a generar haciendo clic sobre él en la ventana de visualización (este centro debe estar siempre sobre un punto de

la nube). Tras determinar el centro, el usuario debe establecer el ángulo intermedio y el ángulo final del arco en este orden. El punto inicial del arco es el último punto ya introducido en el camino, y el radio del arco es la distancia entre este último punto y el centro introducido. Así pues, establecidos el ángulo final y un ángulo intermedio (para discernir cuál de los dos arcos posibles desea el usuario), el arco queda totalmente definido, este es introducido en el camino, y la aplicación vuelve al modo de creación de caminos. En caso de haber seleccionado la opción arco, pero no querer completar la operación de introducción de un arco en el camino, el usuario tiene que volver a hacer clic en el botón "Arco" del panel de Caminos o pulsar de nuevo la tecla de acceso rápido "a".

Al introducir elementos en un camino, debemos tener en cuenta la presencia de los llamados "puntos de atracción". Estos son puntos tales que al acercarnos a ellos en la ventana de visualización, el cursor los detecta y se posa sobre ellos. Los puntos de atracción son de dos tipos. Los primeros son los puntos de la nube. Cuando pasamos el cursor cerca de un punto de la nube, la aplicación lo detecta y, si intentamos introducir ahí un elemento, la aplicación asocia el elemento del camino introducido con el punto de la nube. Esta asociación no sólo significa que ambos puntos tienen las mismas coordenadas, sino también que si movemos posteriormente el punto de la nube, también estaremos modificando todo camino que pasa por ese punto, moviendo el o los elementos del camino, vértices, segmentos y arcos, incidentes en él. El segundo tipo de puntos de atracción lo constituyen los puntos medios de las aristas del $EMST(P)$ porque son puntos de interés para la construcción de caminos de desviación mínima y de desviación mínima local. La asociación de los vértices de los caminos con estos últimos puntos no es total: comparten las coordenadas, pero no sufren modificaciones cuando lo hace el $EMST(P)$.

Si, estando en el modo de creación de caminos, el usuario define un segmento o un arco que no desea, puede volver atrás y eliminar el último elemento introducido haciendo clic en el botón "Volver" del panel de Caminos o pulsando la tecla de acceso rápido "retroceso". Si hace clic en el botón "Volver" o pulsa la tecla "retroceso" cuando el camino todavía no tiene ningún punto, el camino queda cancelado y la aplicación sale del modo de creación de caminos.

Para terminar la creación de un camino una vez introducidos los elementos deseados, el usuario debe hacer clic en el botón "Acabar" del panel de Caminos, pulsar la tecla de acceso rápido "Intro", o hacer clic en el botón derecho del ratón en cualquier parte de la ventana de visualización. Para que el camino sea aceptado, debe tener al menos dos vértices. Si no los tiene, es descartado y no aparece en la lista de caminos. En caso contrario, el camino es aceptado y se selecciona por defecto en la lista del panel de Caminos.

A la hora de crear un camino, hay que tener en cuenta que la aplicación exige que tanto el punto inicial del camino como el punto final tienen que estar sobre puntos de la nube para facilitar las decisiones sobre el camino. Si el usuario no pone el punto inicial del camino sobre un punto de la nube, la aplicación introduce automáticamente un punto en la nube en la misma localización que el vértice inicial del camino y genera los cambios

necesarios en los grafos. Del mismo modo, si cuando el usuario decide terminar un camino, el último punto introducido no se encuentra sobre un punto de la nube, la aplicación introduce un punto en esa localización y realiza las modificaciones necesarias en los grafos. De esta manera nos aseguramos de que todos los caminos tienen siempre su punto inicial y su punto final sobre puntos de la nube.

2. Seleccionar un camino

   Cuando se crea un camino, este se selecciona por defecto, pero existen otras maneras de seleccionar caminos incluso después de que hayan sido deseleccionados. En el panel de Caminos se puede ver qué camino está seleccionado en cada momento. Es aquel que aparece marcado en un fondo azul (en la Figura 5.18 está seleccionado el camino 0).

   Para seleccionar un camino basta con hacer clic sobre su nombre en el panel de Caminos.

   Cuando se elimina un camino, la aplicación selecciona el siguiente camino en la lista, si queda alguno.

3. Eliminar un camino

   Para eliminar un camino el usuario debe tener seleccionado el camino que quiere eliminar. Una vez el camino escogido está seleccionado, el usuario tiene que hacer cick en el botón "Borrar" del panel de Caminos para eliminarlo.

   Otra manera de borrar un camino es borrar el punto de la nube que es punto inicial o final del camino. Si el punto de la nube asociado a un punto inicial o final de uno o más caminos se borra, como todos los caminos deben tener estos dos puntos sobre puntos de la nube, todos los caminos que tuvieran su punto inicial o final sobre este son eliminados. De esta manera nos aseguramos de que todos los caminos activos tienen siempre su punto inicial y su punto final sobre puntos de la nube.

   Cuando se elimina un camino, la aplicación selecciona el siguiente camino en la lista, si queda alguno.

4. Modificar un camino

   Las modificaciones que se pueden realizar a un camino son de dos tipos. El primer tipo lo constituyen los cambios que sólo afectan a la visualización del camino en la ventana, y que no influyen en los cálculos realizados sobre este. Este tipo incluye la modificación del color del camino y la del grosor de este para su visualización por pantalla. El segundo tipo de modificaciones se realizan sobre los elementos del camino, vértices, segmentos y arcos, y producen cambios en los resultados de los cálculos realizados sobre estos. Este tipo de modificaciones incluye eliminación o desplazamiento de puntos del camino, modificación de arcos o incluso adición de puntos en ciertos lugares del camino.

   Los caminos aparecen por defecto con un color que los diferencia del resto. Para cambiar el color de un camino seleccionado, el usuario puede hacer clic en el botón que muestra el color actual del camino en el panel de Caminos. Al hacer clic, aparece una sencilla interficie en la que puede elegir

el color que desea que tenga el camino. Otro modo de cambiar el color de un camino que no requiere que este esté seleccionado es seleccionando en el menú "Configuración" la opción "Modificar colores de caminos". En este caso, aparece una interficie que muestra los colores de todos los caminos creados, con botones del color de estos (esta interficie puede verse en la Figura 5.19). Haciendo clic en cualquier de esos botones, el usuario puede cambiar el color del camino asociado al botón que ha clicado.

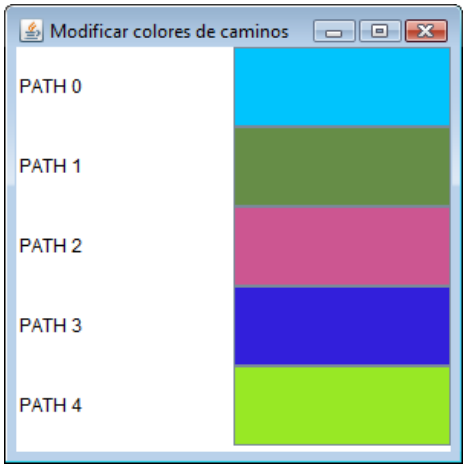

Figura 5.19: Interficie de selección de colores de caminos.

Los caminos aparecen por defecto con un grosor de 1,0, pero este grosor puede ser modificado. Para cambiar el grosor de un camino seleccionado, el usuario debe introducir el valor del grosor que desea en el campo grosor del panel de Caminos y pulsar intro para aceptar dicho grosor. El valor del grosor tiene que ser un número entero o de coma flotante con un valor ≥ 0.

Como hemos dicho antes, también se pueden modificar los elementos (vértices, segmentos y arcos) de un camino En primer lugar, es posible borrar un punto del camino. Para ello hay que hacer clic con el botón derecho del ratón en la ventana de visualización sobre el punto a borrar. Si este punto es el vértice inicial o final de un camino, este desaparece, si no es así, el nuevo camino conecta los vértices anterior y posterior al eliminado mediante un segmento. Si el punto eliminado era un extremo de un arco, desaparece y se conectan los vértices anterior y posterior al eliminado con un segmento.

También se pueden mover puntos del camino. Para ello hay que hacer clic sobre ellos en la ventana de visualización, y arrastrarlos. Al mover un punto del camino, si este no es punto inicial o final de un arco, se puede mover libremente, cambiando únicamente el vértice desplazado y su conexión con sus vértices anterior y siguiente. Si el vértice movido es extremo de un arco de circunferencia, este solo se puede mover sobre la circunferencia salvo cuando el vértice a mover está sobre un punto de la nube, en cuyo caso este se puede mover libremente, produciendo, en su caso, un cambio en el radio del arco.

Al arrastrar puntos del camino, también actúan los "puntos de atracción" de forma que cuando pasamos cerca de uno de ellos, la aplicación tiende a asociarlos. Esto permite también modificar los radios de los arcos al intentar mover un punto de estos, porque si pasamos cerca de un punto de atracción el elemento del camino tiende a posarse sobre él, y esto implica si es un arco, que la distancia al centro del arco varía.

Por último, es posible añadir segmentos al camino. No es posible, en cambio, añadir un arco al camino después de haber finalizado la construcción de este, sin embargo, es posible modificar un arco al añadir un nuevo segmento del camino. Para añadir un segmento al camino, hay que arrastrar un punto del camino como si quisiéramos desplazarlo, pero manteniendo pulsada la tecla "Ctrl". De esta manera el vértice sobre el que hemos clicado se mantiene pero aparece otro entre este y el vértice siguiente que puede moverse libremente si el vértice clicado no era punto inicial de un arco, y radialmente en caso contrario.

Estas modificaciones provocan que se rehagan los cálculos relativos al camino.

5. Consultar los parámetros de un camino

   Los parámetros de un camino son su longitud, y la decisión sobre si dicho camino es o no de desviación mínima y si es o no de desviación mínima local. Estos parámetros pueden consultarse en la parte inferior del panel de Caminos. En esta zona, aparecen los 3 parámetros correspondientes del camino actualmente seleccionado. Para consultar los parámetros de otro camino, hay que seleccionarlo, en cuyo caso los campos de consulta se actualizan. En estos campos se visualizan adecuadamente de arriba a abajo, primero la longitud del camino, a continuación si el camino es o no de desviación mínima, y por último si el camino es o no de desviación mínima local.

   Debemos aclarar que es necesario que los arcos de circunferencia de los caminos estén contenidos en las regiones de Voronoi de los puntos sobre los que está situado su centro. Si el arco con centro en el punto p_i se sale de la región de Voronoi de p_i, $Vor(p_i)$, la información de si el camino es o no de desviación mínima o de si es o no de desviación mínima local puede ser errónea. En cualquier caso, si la aplicación determina que un camino es de desviación mínima o de desviación mínima local, esta información es siempre correcta, pero puede ocurrir que para ciertos caminos que no cumplen la propiedad de contención, la aplicación indique que el camino no es de desviación mínima o de desviación mínima local, pero en realidad, sí lo sea.

En la Figura 5.20 se puede ver un ejemplo en el que la aplicación trabaja con varios caminos y muestra los resultados de los cálculos sobre uno de ellos.

5.5. Visualización

En esta sección se explica con detalle cómo acceder a los distintos modos de ejecución con los que cuenta la aplicación y cómo hacer uso de ellos. También

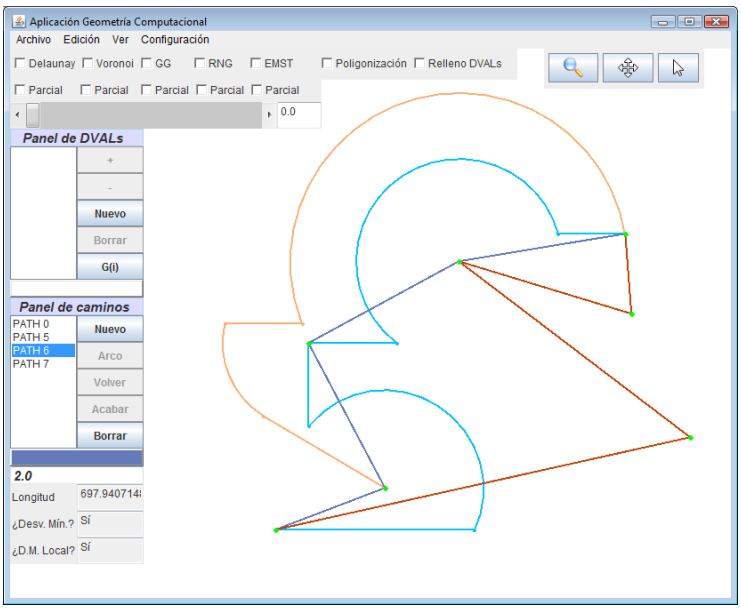

Figura 5.20: Ejemplo de caminos introducidos en la aplicación.

se detalla como modificar colores y grosores de cada uno de los elementos de la aplicación, y como ocultar cada uno de los paneles.

5.5.1. Modos de ejecución: normal, Zoom y Pan

La forma más inmediata de cambiar de un modo de ejecución a otro es a través del panel de modos de ejecución que se muestra en la Figura 5.21. Para entrar en el modo Zoom basta con pulsar el botón que tiene la imagen de una lupa. Para entrar en el modo Pan, hay que pulsar el botón que tiene dos flechas cruzadas. Para volver al modo normal, se puede elegir entre pulsar el icono con la imagen de una flecha o volver a pulsar el botón del modo de ejecución que previamente se haya pulsado.

Figura 5.21: Panel de modos de ejecución.

También es posible acceder a los modo Zoom y Pan mediante teclas de acceso rápido, siempre y cuando estemos en modo normal. En concreto, si se pulsa la letra “Z”, la aplicación entra en modo Zoom, y si se pulsa de nuevo la aplicación vuelve al modo normal. Una vez dentro del modo Zoom, si se hace clic izquierdo sobre un punto cualquiera del plano la cámara se acerca a éste. Si se hace clic derecho, la cámara se aleja. También es posible acercar o alejar la cámara estemos en el modo de ejecución que estemos moviendo la rueda del ratón hacia arriba o hacia abajo respectivamente.

Para entrar y salir del modo Pan a través del teclado, la tecla de acceso rápido que se debe pulsar es "W". También es posible acceder y salir del modo Pan directamente desde el ratón pulsando la rueda o botón central, según el ratón que se use. Una vez dentro del modo Pan, el usuario debe clicar sobre un punto del plano (da igual con qué botón del ratón lo haga) y, sin soltar el botón pulsado, mover el ratón en la dirección del plano en la que quiere desplazarse.

5.5.2. Colores

Para modificar los colores de la nube de puntos, la triangulación de Delaunay, el diagrama de Voronoi, los grafos de Gabriel y de vecinos relativos y el árbol generador mínimo, la aplicación utiliza la misma interficie. A ella se accede a través del menú "Configuración" ubicado en el menú principal del programa (ver Figura 5.22). Como también se ve en la figura, se puede acceder a esta opción mediante una combinación de teclas de acceso rápido: Control, Shift (Mayúsculas) y C pulsadas simultáneamente.

Figura 5.22: Opción para modificar colores de grafos y diagramas.

Una vez el usuario selecciona esta opción, se abre la ventana de la Figura 5.23. Como vemos, en ella aparecen seis etiquetas con los nombres abreviados de los seis elementos mencionados al principio de esta sección. Al lado de cada una de estas etiquetas se pueden apreciar botones con el color actual del elemento respectivo.

Figura 5.23: Ventana de modificación de colores.

Para modificar el color de alguno de los elementos que aparecen en la ven-

tana, se hace clic en el botón del elemento que se desea modificar. Entonces, la aplicación muestra una paleta de colores como la de la Figura 5.24, que permite seleccionar o incluso confeccionar un color del agrado del usuario. Una vez seleccionado el color adecuado, basta con pulsar el botón "Aceptar" y el elemento asociado al botón que se había pulsado quedará modificado. Si, en cambio, se decide pulsar "Cancelar" el color no se modifica.

Figura 5.24: Paleta de colores de la aplicación.

Para modificar los colores de los diagramas de Voronoi de alcance limitado, el proceso a seguir es muy similar al de la sección anterior. Primero, el usuario debe acceder a la ventana de modificación de colores de *DVALs*, que está ubicada en el menú "Configuración". También se puede acceder a través del teclado, como se puede apreciar en la Figura 5.25.

Figura 5.25: Opción para modificar colores de los *DVALs*.

En la ventana de modificación de color de *DVALs* (Figura 5.26), aparecen tantas etiquetas y botones como *DVALs* presentes en el momento de acceder a la ventana. Del mismo modo que en la sección anterior, para cambiar el color de un *DVAL* se debe hacer clic en su botón correspondiente y elegir el nuevo color en la paleta de colores que se abre al hacerlo.

También es posible modificar el color de un *DVAL* sin necesidad de acceder a la ventana descrita anteriormente, a través del panel de *DVALs* (Figura 5.27). Para modificar el color de un *DVAL* se selecciona en la lista situada a la izquierda del panel y se hace clic sobre el botón del color del *DVAL* seleccionado situado en la parte inferior del panel. Entonces, la aplicación abre una paleta de colores

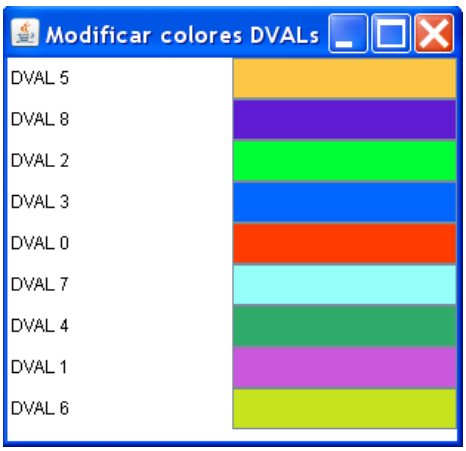

Figura 5.26: Ventana de modificación de colores de los *DVALs*.

en la que se puede seleccionar el nuevo color que se va a asignar al *DVAL* seleccionado.

Figura 5.27: Panel de edición de *DVALs*.

La funcionalidad para modificar los colores de los caminos es equivalente a la de modificar los colores de los *DVALs*. Esta funcionalidad es accesible a través del menú principal o teclado, como se puede ver en la Figura 5.28.

Figura 5.28: Opción para modificar colores de los caminos.

Una vez seleccionada esta opción, se abre una ventana como la de la Figura 5.29 en la que aparecen los caminos que en ese momento forman parte del

proyecto actual y sus respectivos botones de modificación de color. Haciendo clic sobre estos botones se abre una paleta de colores donde podemos escoger el nuevo color del camino en cuyo botón se ha clicado.

Figura 5.29: Ventana de modificación de colores de los caminos.

También es posible modificar el color de los caminos sin acceder a la ventana de selección de color de caminos, a través del panel de caminos (Figura 5.30). Para modificar el color de un camino se selecciona en la lista situada a la izquierda del panel y se hace clic sobre el botón del color del camino seleccionado situado en la parte inferior del panel. Entonces, la aplicación abre una paleta de colores en la que se puede seleccionar el nuevo color que se va a asignar al camino seleccionado.

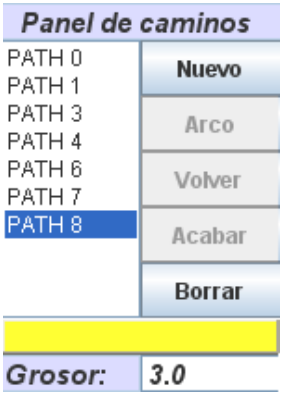

Figura 5.30: Panel de edición de caminos.

5.5.3. Grosores

La modificación de los grosores de los distintos elementos del proyecto sigue un proceso muy similar al de modificación de colores. Esta funcionalidad también se encuentra en el menú "Configuración" del menú principal del programa, aunque es posible acceder a ella también pulsando las teclas Control, Shift (Mayúsculas) y O simultáneamente, como se observa en la Figura 5.31.

Figura 5.31: Opción para modificar grosores.

Una vez se selecciona la opción de modificado de grosores, la aplicación abre una ventana como la de la Figura 5.32. En ella es posible ver los distintos elementos a los que se les puede modificar el grosor. Para ello, basta con modificar el valor del cuadro de texto que hay a la derecha de cada uno de ellos y pulsar la tecla "Enter" para confirmar el nuevo valor. Si éste está entre los valores permitidos del elemento en cuestión, el grosor se modifica. Si el valor introducido esta por debajo del mínimo permitido, el elemento en cuestión asume el mínimo grosor permitido para él. Los grosores mínimos de cada uno de los elementos son los valores que aparecen por defecto y que se pueden observar también en la Figura 5.32, a excepción del grosor de los vértices de los caminos, cuyo valor mínimo es cero. Los valores relativos a puntos (nube de puntos, vértices de Voronoi y caminos, y puntos que marcan la aparición de un nuevo evento) admiten únicamente valores enteros, por lo que si se introduce un valor decimal, éste es redondeado al entero inmediatamente inferior. El resto de elementos admiten valores decimales, que se deben expresar separando la parte entera de la parte decimal mediante un punto.

Figura 5.32: Ventana de modificación de grosores.

También es posible modificar el grosor de los caminos individualmente sin acceder a la ventana de modificación de grosores, a través del panel de caminos (Figura 5.30). Para modificar el grosor de un camino se selecciona en la lista situada a la izquierda del panel y se introduce el valor del grosor deseado en el campo de texto situado a la derecha de la etiqueta "Grosor: ". En ese momento,

el grosor de ese camino queda modificado. Hay que tener en cuenta de que si accedemos a la ventana de modificación de grosores y modificamos el apartado "Caminos", se modifica el grosor de todos los caminos simultáneamente, con lo que las modificaciones individuales de grosor de caminos realizadas previamente quedan sin efecto. Al modificar este apartado de la ventana de selección de grosores, estamos determinando también el grosor con el que se visualizarán los nuevos caminos que se creen a partir de ese momento.

5.5.4. Paneles

Acceder a la funcionalidad de mostrar u ocultar paneles es muy simple. A través menú "Ver" del menú principal del programa, se puede mostrar, en caso de que esté oculto, o ocultar, en caso de que sea visible, cualquiera de los paneles de la aplicación haciendo clic sobre el botón del panel a mostrar u ocultar. También es posible mostrar u ocultar paneles mediante las teclas de acceso rápido que aparecen en la Figura 5.33.

Figura 5.33: Menú para mostrar u ocultar panales.

Para conocer en detalle cómo usar y qué hace cada panel, consúltese los apartados anteriores de este capítulo o la Sección 4.5.4 del capítulo anterior.

5.6. Interacción

En esta sección se detalla, principalmente, cómo conocer las teclas de acceso rápido de las distintas funcionalidades del programa y los mensajes de confirmación o error que emite la aplicación. En el último apartado, se explica al detalle como hacer uso de las funcionalidades de deshacer y rehacer.

5.6.1. Teclas de acceso rápido y menús

Las teclas de acceso rápido para cada una de las funcionalidades de la aplicación se describen en el resto de apartados de este manual de usuario. Para saber qué teclas de acceso rápido tiene una funcionalidad concreta, consúltese el apartado correspondiente a esa funcionalidad de este manual.

5.6.2. Mensajes de confirmación y de error

Para conocer con detalle los mensajes de confirmación y error que muestra nuestra aplicación, consúltese la sección 5.7.

5.6.3. Deshacer y rehacer

Si se desea en algún momento deshacer o rehacer algún cambio, *DVALon* permite hacerlo de manera sencilla. La más rápida es a través del panel de edición. Como vemos en la Figura 5.34, este panel cuenta únicamente con los botones "Deshacer" y "Rehacer", y basta con pulsarlos para hacer una acción u otra según convenga.

Figura 5.34: Panel de edición.

Sin embargo, también es posible acceder a esta funcionalidad a través del menú principal. Para ello, es necesario acceder al menú "Edición", en el cual tenemos también los botones "Deshacer" y "Rehacer", como vemos en la Figura 5.35. Basta con hacer clic sobre alguno de ellos para deshacer o rehacer la última modificación realizada. Las modificaciones que se pueden deshacer o rehacer son cualquier modificación en la nube de puntos (inclusión, borrado o modificación de un punto), las de borrado de un *DVAL* (la creación de un *DVAL* se puede deshacer borrándolo desde el panel de *DVAL*), la generación de *DVALs* críticos y las de creado y borrado de caminos (la inclusión de un punto intermedio de un camino se puede deshacer mediante la tecla específica para esta función del panel de caminos).

Figura 5.35: Deshacer y rehacer a través del menú principal.

5.7. Ficheros

En esta sección se detalla cómo guardar y cargar un proyecto y un fichero de puntos. En cada caso se describen exactamente los pasos a seguir por el usuario, las distintas opciones que tiene a la hora de guardar y cargar, y los mensajes y problemas que pueden surgir durante este proceso.

5.7.1. Proyecto

Para guardar un proyecto con nuestra aplicación basta con acceder a la opción "Guardar proyecto", a través del menú "Archivo" situado en la barra de opciones principal del programa o pulsando simultáneamente las teclas "Control+Shift(o Mayúsculas)+S" en cualquier momento (véase la Figura 5.36).

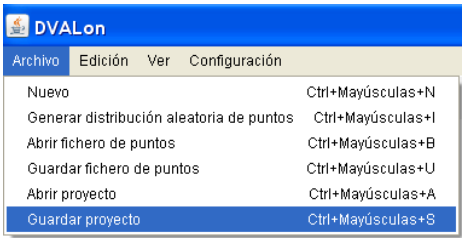

Figura 5.36: Opción "Guardar proyecto".

En ambos casos, aparece en la pantalla una ventana de diálogo como la de la Figura 5.37, donde podremos navegar por nuestro sistema de ficheros para seleccionar el archivo a sobrescribir o bien podemos crear un nuevo escribiendo su nombre en el campo "Nombre del Archivo".

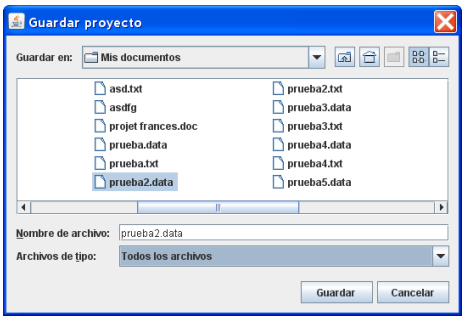

Figura 5.37: Ventana de diálogo para seleccionar la ubicación del proyecto.

Una vez tengamos la ruta donde queremos guardar el proyecto, pulsamos el botón "Aceptar". A continuación aparece un mensaje como el de la Figura 5.38 que indica dónde se ha guardado el proyecto.

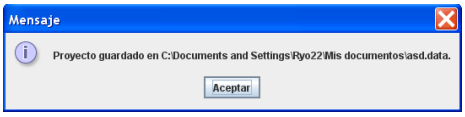

Figura 5.38: Ventana de confirmación al guardar.

Para cargar un proyecto ya guardado, el proceso es muy similar al de guardado de proyectos. Basta con acceder a la opción 'Abrir proyecto", a través del menú "Archivo" situado en la barra de opciones principal del programas o pulsando simultáneamente las teclas "Control+Shift(o Mayúsculas)+A" en cualquier momento (Figura 5.39).

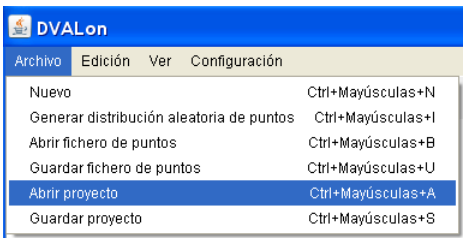

Figura 5.39: Opción "Abrir proyecto".

A continuación aparece en pantalla una ventana de diálogo muy similar a la de la Figura 5.37, donde podemos navegar por nuestro sistema de ficheros para seleccionar el archivo ".DATA" a cargar o bien podemos escribir directamente su ruta completa en el campo "Nombre del Archivo".

Hecho esto, pulsamos el botón "Aceptar" o bien hacemos doble clic sobre el fichero a cargar. A continuación aparece un mensaje diciendo que todo ha ido bien y se carga nuestro proyecto en caso de que el fichero tenga le extensión y formato correcto (ver Figura 5.40).

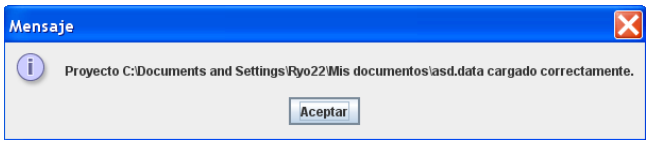

Figura 5.40: Ventana de confirmación al cargar.

Si el fichero seleccionado no tiene la extensión adecuada o bien su formato es incorrecto, el fichero no se carga o se carga parcialmente con errores y aparece un mensaje describiendo el error producido. En las Figuras 5.41, 5.42 y 5.43 podemos ver algunos ejemplos.

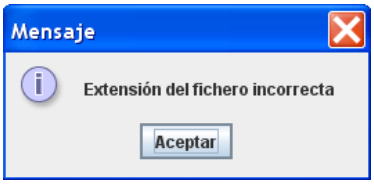

Figura 5.41: Aviso de extensión incorrecta.

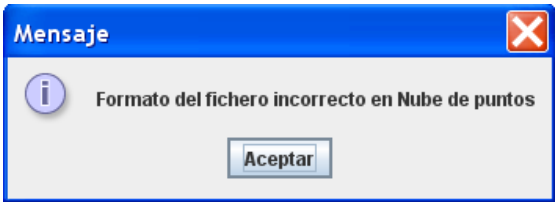

Figura 5.42: Aviso de formato del fichero incorrecto en la nube de puntos.

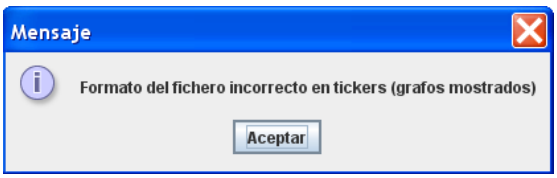

Figura 5.43: Aviso de formato del fichero incorrecto en el panel de grafos.

A continuación, después de mostrar todos los errores que contiene el fichero de datos, aparece el mensaje de advertencia de la Figura 5.44. En él se indica al usuario que el fichero no se ha cargado correctamente(aunque en ocasiones pueda parecer lo contrario).

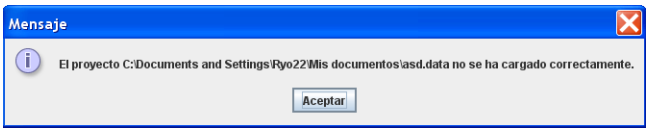

Figura 5.44: Aviso de fichero cargado con errores.

5.7.2. Puntos

Para guardar únicamente la nube de puntos del proyecto en el que estamos trabajando existen dos opciones, como podemos ver en la Figura 5.45. Podemos acceder a esta funcionalidad a través del menú "Archivo", seleccionando la opción "Guardar fichero de puntos" o bien pulsando simultáneamente las teclas Control, Shift (Mayúsculas) y U.

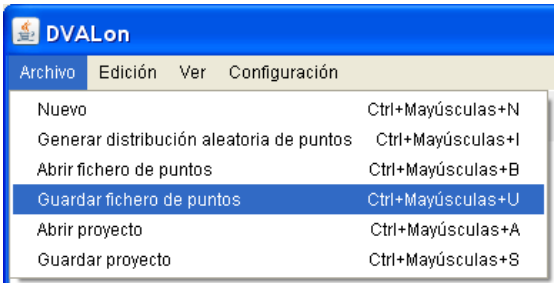

Figura 5.45: Guardar un fichero de puntos.

Una vez se ha accedido a esta opción aparece en pantalla una ventana de diálogo similar a la de la Figura 5.37, dónde podemos navegar por nuestro sistema de ficheros para seleccionar el archivo a sobrescribir o bien podemos crear un nuevo escribiendo su nombre en el campo "Nombre del Archivo". Una vez tengamos la ruta donde queremos guardar el fichero de puntos, pulsamos el botón "Aceptar". A continuación aparece un mensaje similar al de la Figura 5.38 indicando dónde se ha guardado el fichero.

Si el usuario desea cargar un conjunto de puntos guardado, puede hacerlo accediendo al menú "Archivo" y seleccionando la opción "Abrir fichero de puntos"

o bien mediante una combinación determinada de teclas (ver Figura 5.46).

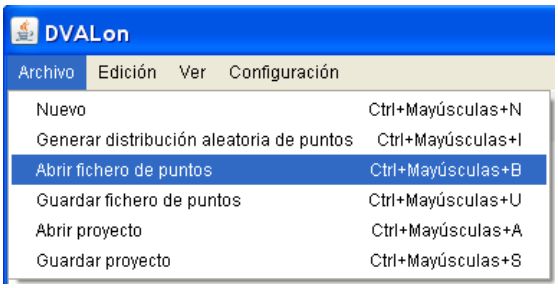

Figura 5.46: Cargar un fichero de puntos.

Una vez seleccionada esta opción, aparece en pantalla una ventana de diálogo similar a la de la Figura 5.37, en la cual se debe seleccionar el fichero de puntos a abrir o bien especificar manualmente su ruta completa en el campo "Nombre del Archivo". Una vez hecho esto, el usuario debe pulsar el botón "Aceptar". Si el fichero tiene el formato especificado en la sección 4.7.2, la aplicación carga los puntos y muestra un mensaje informando de que se ha cargado todo correctamente. En caso contrario, la aplicación carga únicamente los puntos con un formato correcto, y muestra un mensaje avisando al usuario de que el formato del fichero no es correcto.

5.7.3. Fichero de información

El fichero de información se genera automáticamente al guardar un proyecto (ver sección 5.7.1).

Capítulo 6

Metodología de trabajo

En este capítulo de describe la metodología de trabajo seguida durante la realización del proyecto. Uno de los aspectos más importantes que se abordan en este capítulo son la planificación del proyecto, donde se habla no sólo de la planificación en sí, sino también de las modificaciones que hemos tenido que hacer en la planificación inicial, los imprevistos que han ido surgiendo y cómo se han solucionado, y si se han cumplido todos los plazos marcados para sacar adelante cada uno de los apartados del proyecto propuestos al inicio del mismo. Otro aspecto importante a destacar es el trabajo en equipo, ya que este proyecto se ha llevado a cabo entre dos personas. En este capitulo se habla también de cómo se ha distribuido el trabajo y del funcionamiento que ha tenido el equipo durante el transcurso del proyecto.

6.1. Planifiación

Este proyecto comenzó oficialmente a principios de enero de 2009. Decimos oficialmente porque en realidad llevábamos meses pensando en hacer un proyecto de Geometría computacional. Sin embargo, el tema del proyecto no quedó totalmente definido hasta el 12 de enero, en la primera reunión con nuestra tutora Vera Sacristán. En esa primera reunión se concretó el problema a resolver y comenzamos a estudiar la información que había hasta el momento sobre diagramas de Voronoi de alcance limitado. Ese día se convino que dos semanas después, volveríamos a reunirnos ya con la especificación del problema a resolver por el programa que íbamos a implementar y con una planificación detallada de todo el proyecto a partir de ese día, planificación que se presenta a continuación.

1. Del 27 al 31 de Enero (4 días): Recordatorio. Buscar y recordar los conceptos necesarios para resolver el problema (diagramas de Voronoi, triangulaciones de Delaunay, etc.).

   Este paso se cumplió sin problemas, ya que el volumen de conceptos a adquirir no era demasiado grande y nuestra dedicación al proyecto es a tiempo completo.

2. Del 5 al 12 de Febrero (1 semana): Resolución. Utilizando estos conceptos, proponer una solución correcta y eficiente para la implementación del

algoritmo que resolverá el problema (estructuras necesarias, uso de estas, coste computacional, etc.).

Este apartado también se pudo cumplir sin problemas.

3. Del 14 al 17 de Febrero (4 días): Herramientas. Buscar y seleccionar las herramientas que utilizaremos para la implementación (lenguaje, librerías graficas, etc.) y aprender a utilizar estas herramientas correctamente (en caso de no saber utilizarlas ya).

   Este paso se cumplió también en el plazo establecido, ya que las herramientas a utilizar las tuvimos claras desde el principio. Uno de los objetivos de la aplicación es que pueda ser ejecutada directamente como un "Applet" insertado en una página web. Para esto, lo ideal es que esté implementada en el lenguaje de programación Java, ya que facilita mucho esta labor. Además, Java es un lenguaje muy completo y de sobras conocido por nosotros, con lo que no tuvimos demasiadas dudas en elegirlo como lenguaje a utilizar para la implementación de nuestro proyecto. En cuanto a las librerías gráficas usadas, nos decantamos por AWT, que habíamos visto por encima durante la carrera y que es famosa por ser de las más completas y por su facilidad de uso. En estos cuatro aprendimos a usar AWT , tarea que no nos ocasionó demasiados problemas al conocer ya gran parte de las librerías de Java.

4. Del 18 de Febrero al 18 de Marzo (1 mes): Implementación (Primera fase). Implementación de las funcionalidades iniciales de la aplicación. Primera parte de la fase de implementación.

   La primera fase de la implementación consistía en la implementación de todas las funcionalidades descritas en esta documentación, a excepción de la gestión de caminos. Esta primera fase en realidad no existía como tal en la planificación inicial del proyecto, ya que en un principio no se tenía pensado incluir las funcionalidades relativas a los caminos en el proyecto. Sin embargo, viendo que era una opción interesante y que íbamos bien de tiempo, decidimos incluirla y hacer la implementación del proyecto en dos fases.

   Inicialmente esta fase iba a durar aproximadamente un mes y quince días, tiempo que pensamos que sería suficiente. Sin embargo, unas dos semanas antes del final del plazo nos surgió la oportunidad de presentar nuestro proyecto en los XIII Encuentros de Geometría Computacional, que este año tendrá lugar los días 29,30 y 1 de junio y julio en Zaragoza, oportunidad que decidimos aprovechar. Esto significó que tuvimos que acortar dos semanas el plazo de esta fase de implementación para poder preparar el artículo que debíamos enviar a la organización del evento. A pesar de este inconveniente, aumentando el ritmo de trabajo conseguimos acabar esta fase de la implementación dentro del nuevo plazo establecido.

5. Del 19 al 22 de Marzo (4 días): Aprender a usar LaTeX.

   Esta fase del proyecto consistente en aprender a usar el lenguaje de edición de texto LaTeX inicialmente se encontraba a finales de abril, ya que no sería necesario aprender LaTeX hasta justo antes de empezar con la documentación del proyecto. Sin embargo, debido al artículo que

debíamos redactar para los XIII Encuentros de Geometría Computacional nos obligó a moverlo a mediados de marzo. No supuso ningún problema, ya que habíamos acortado la fase anterior en dos semanas.

6. Del 23 al 31 de Marzo (10 días): Preparación del articulo. Preparación en equipo del documento que se enviará a la convención de Zaragoza.

   Los diez días siguientes nos dedicamos a preparar el artículo. Diez días para preparar un artículo de unas 4 o 5 páginas puede parecer un tiempo excesivo, pero se decidió dar este margen ya que eran los diez días restantes para la implementación de la primera fase del proyecto después del cambio de planes.

7. Del 1 al 22 de Abril (3 semanas): Implementación (Segunda fase). Implementación de las funcionalidades restantes de la aplicación. Última parte de la fase de implementación.

   Esta segunda fase de la implementación inicialmente no existía, como se ha comentado anteriormente. En su lugar, había un mes entero dedicado a la resolución de imprevistos y a acabar posibles fases de la planificación anteriores que no se hubieran acabado cuando se había previsto. Viendo que todo estaba saliendo según lo previsto, decidimos incluir aquí la segunda fase de la implementación y dejar únicamente una semana para la resolución de imprevistos, que hasta el momento eran inexistentes. En estas tres semanas implementamos todas las funcionalidades relacionadas con caminos y completamos la aplicación con nuevas funcionalidades (como la posibilidad de deshacer y rehacer y de cambiar los grosores).

8. Del 23 al 30 de abril (1 semana): Resolución de Imprevistos. Resolver imprevistos si hubiera habido, o terminar la segunda fase de implementación en caso de que nos hubiera faltado tiempo.

   La implementación de la segunda fase del proyecto se llevó a cabo en las 3 semanas destinadas a ello, con lo que esta semana reservada para imprevistos se utilizó para corregir algunos “bugs” de la aplicación y para ir comenzando con la documentación.

9. Del 1 al 31 de Mayo (1 mes): Memoria. Redactar las diferentes partes de la memoria y revisarlas sucesivamente hasta darla por finalizada.

   El mes de mayo se ha reservado exclusivamente para redactar la memoria. Primero se decidió redactar una sección de cada capítulo y revisarlo con nuestra tutora para corregir fallos y poder hacer mejor el resto de la documentación. Esto se hizo en la primera semana de mayo. Una vez hecho esto, se redactaron primero los capítulos correspondientes al núcleo de la memoria (dos, tres, cuatro y cinco) durante las dos semanas siguientes para que también fuesen revisados con ayuda de nuestra tutora. Finalmente, en la última semana de mayo se ha corregido todo lo escrito hasta el momento y se han escrito los capítulos de introducción y conclusiones (uno, seis y siete) de esta documentación. La documentación se ha terminado definitivamente el día 2 de junio, únicamente con dos días de retraso respecto a lo planificado inicialmente. Salvo imprevistos de última hora, el resto de apartados de esta planificación se cumplirán todos dentro de su debido plazo.

10. Del 1 al 14 de Junio (2 semanas): Preparación del material. Preparar la defensa del proyecto. Consta de preparar las transparencias y el guión tanto de la defensa del PFC como de la presentación en Zaragoza.

11. Día 9 de Junio: Entrega de las memorias. Entregamos las memorias definitivas al menos 10 dias antes de la defensa del proyecto.

12. Día 19 de Junio: Defensa del proyecto.

13. Día 1 de Julio: Presentación del proyecto en los XIII Encuentros de Geometría Computacional.

En general, se puede decir que todo el proyecto ha salido según lo previsto. Es cierto que la planificación inicial se ha ido alterando debido a nuevas ideas y a la preparación del artículo para los XIII Encuentros de Geometría Computacional, pero esto no se ha traducido en ningún momento en necesidad de tiempo extra para cumplir todos los objetivos que nos marcamos antes de empezar. Es más, este proyecto incluye algunas características, como la inclusión de caminos en la aplicación, que en un principio no se tenía pensado implementar, con lo que estamos realmente satisfechos con el resultado final de nuestro proyecto.

6.2. Trabajo en equipo

Este proyecto se ha llevado a cabo entre dos personas y el desarrollo del mismo ha tenido una duración de seis meses. El hecho de que se haya tardado este tiempo la realización de este proyecto con dos personas trabajando en él a tiempo completo, es indicador de la magnitud e importancia del mismo. Es por este motivo que decidimos llevar a cabo este proyecto entre dos personas. Ambos miembros del grupo estábamos interesados en hacer un proyecto similar y este nos pareció un proyecto interesante, tanto desde el punto de vista geométrico como desde el punto de vista informático.

Debido a su magnitud, posiblemente no habría sido posible llevarlo a cabo por una sola persona, con lo que pensamos que fue un acierto hacerlo en equipo. Además, el hecho de hacer un proyecto en equipo durante seis meses nos ha enseñado, al margen de todos los conocimientos geométricos e informáticos adquiridos, a tomar decisiones en común y a dividir y distribuir el trabajo de manera equilibrada.

El trabajo se ha ido repartiendo siempre de manera que ambos pudiéramos trabajar simultáneamente para luego acabar poniendo ese trabajo en común. Ambos miembros hemos trabajado a la vez en todos los aspectos del proyecto (implementación, documentación, etc.). Sería muy difícil hacer una partición exacta del trabajo realizado por cada miembro, ya que ambos hemos trabajado en todo prácticamente por igual. En lo que se refiere a la implementación del programa, ambos miembros hemos trabajado en la gran mayoría de clases que lo componen en mayor o menor medida. Por tanto, es complicado separar lo que ha hecho uno del otro. En cuanto a la documentación, se ha repartido en partes iguales. Cada uno ha redactado su parte de manera individual para luego juntarlas en una única documentación.

A pesar de ir trabajando de manera individual, ambos miembros del grupo hemos estado en contacto constante y siempre nos hemos ido consultado las

dudas o decisiones que tomábamos. Para ello, herramientas como Gmail o MSN Messenger han sido imprescindibles. Gmail es una herramienta muy cómoda para enviar y almacenar todo el progreso del proyecto. Esto permite tener un acceso rápido, no solo a la última versión del proyecto, sino a todas las versiones que se vayan enviando a lo largo del desarrollo del mismo. MSN Messenger nos ha permitido permanecer en contacto constante mientras estábamos trabajando, lo que ha permitido que haya mucha fluidez en la toma de decisiones y la resolución de dudas por parte de ambos miembros del equipo.

Concluyendo, creemos que realizar este proyecto en equipo ha sido muy positivo ya que nos ha permitido realizar un proyecto del que estamos satisfechos que habría sido imposible realizar de manera individual. También, nos ha ayudado a mejorar nuestras capacidades de cooperación y trabajo en equipo, a parte de proporcionarnos los conocimientos geométricos y informáticos necesarios para realizar el proyecto.

Capítulo 7

Conclusiones

Estamos en condiciones de decir que los objetivos del proyecto han sido completados satisfactoriamente en el tiempo programado. El objetivo principal de este proyecto, que era la creación de una aplicación capaz de calcular y visualizar múltiples diagramas de Voronoi de alcance limitado y otros grafos de proximidad, así como calcular, dado un camino compuesto por segmentos y arcos de circunferencia, su longitud, y decidir si es o no de desviación mínima y si es o no de desviación mínima local, ha sido realizado con éxito.

La aplicación resultante, *DVALon*, promete ser una herramienta muy útil para la investigación en comunidades científicas tan dispares como las surgidas alrededor del análisis y el desarrollo de redes de sensores inalámbricos [15] o del estudio de la evolución de los bosques [1].

Cabe destacar que algunas de las funcionalidades que ofrece *DVALon* son exclusivas, es decir, no se pueden encontrar por el momento en ninguna otra aplicación, y que por lo tanto, es una buena candidata a ser usada por los investigadores que requieran del uso de estas funcionalidades.

La aplicación *DVALon* también resulta útil como herramienta para la ilustración docente en asignaturas con base informática y matemática como puede ser la asignatura Geometría Computacional. La herramienta puede facilitar a los alumnos la visión y caracterización de los grafos de proximidad, los diagramas de Voronoi de alcance limitado, y los caminos de desviación mínima o de desviación mínima local.

Durante el desarrollo de la aplicación, se nos han ocurrido ciertas optimizaciones a realizar para que *DVALon* hiciera los cálculos más rápidamente, o realizara un mayor número de funcionalidades. Estas optimizaciones no se pudieron realizar de forma simultánea a la programación de las funcionalidades básicas de la aplicación, debido a la estricta planificación con la que realizamos esta. Sin embargo, las optimizaciones iban siendo apuntadas a medida que se nos ocurrían, para intentar incluirlas cuando finalizáramos la implementación de las funcionalidades previstas inicialmente. Algunas de ellas pudieron ser introducidas en *DVALon*, como por ejemplo la función $G(i)$ que genera un *DVAL* por cada cambio combinatorio del tipo unión de componentes del conjunto de puntos P. Esta funcionalidad fue introducida al final de la fase de implementación a pesar de no haber estado prevista inicialmente. Otras funcionalidades no fueron implementadas por falta de tiempo, pues la planificación nos daba muy poco margen. Sin embargo, creemos que es conveniente citar algunas de estas opti-

mizaciones que no se han producido, para dar la posibilidad de introducirlas en el futuro a terceras personas y mejorar así la aplicación.

Como se explica en otras secciones de la documentación, la decisión de si un camino con arcos es o no de desviación mínima y de si es o no de desviación mínima local no siempre es fiable, pues puede pasar que los arcos del camino no cumplan la propedad de contención, y aun así el camino sea de desviación mínima o de desviación mínima local, pero la aplicación responda negativamente. De esto surge una de las posibles modificaciones más interesantes y quizás más problemáticas de realizar. Se basa en conseguir que la aplicación responda correctamente a estas preguntas incluso cuando los arcos del camino no cumplan la propiedad de contención.

Otra posible modificación de la aplicación, también relacionada con los caminos, tiene que ver con el hecho de que actualmente todo camino introducido en la aplicación deba tener su vértice inicial y su vértice final sobre puntos de P. Sería igualmente posible, aunque más complejo, conseguir que la aplicación aceptara caminos cuyos vértices inicial y final no estuvieran sobre puntos de P, y adaptar la aplicación de forma que pudiera hacer los cálculos que ya hace pero sobre estos nuevos caminos.

Otra modificación interesante tiene que ver con la forma de recalcular $Del(P)$ cuando un punto de P es eliminado. Hemos explicado cómo la aplicación no recalcula completamente $Del(P)$ cuando un nuevo punto se introduce en P o cuando se mueve uno de los puntos de P, pero sin embargo, cuando se elimina un punto de P, la aplicación sí recalcula completamente la triangulación de Delaunay. Es posible implementar un algoritmo que modifique parcialmente $Del(P)$ después de la eliminación de un punto de P, y que por tanto, pueda obtener la nueva $Del(P)$ en un tiempo menor al que necesita actualmente. El cálculo del resto de grafos cuando se produce alguna modificación en P, que actualmente se hace a partir de $Del(P)$ también puede hacerse más eficientemente, aplicando cambios parciales a cada grafo, como se hace con $Del(P)$.

La última modificación que proponemos, aunque podría haber más, tiene que ver con el espacio que ocupan en memoria las estructuras utilizadas para calcular las funcionalidades. Existe margen de optimización en espacio para algunas de las estructuras utilizadas por la aplicación, con lo que se puede reducir la cantidad de memoria necesaria para el óptimo funcionamiento de esta.

Después de la realización de un trabajo de esta envergadura, estamos satisfechos con los resultados obtenidos, sobretodo por el hecho de saber que hemos realizado una aplicación que va a ser usada y que va a ayudar a otras personas en sus investigaciones. Hemos adquirido una gran cantidad de conocimientos, tanto matemáticos como informáticos, necesarios para la implementación de algunas de las funcionalidades de la aplicación, y ahora que hemos acabado, podemos decir que estamos contentos de haber escogido un proyecto práctico como lo es este, con unos resultados visibles y un producto final usable.

Bibliografía

[1] B. Abellanas, M. Abellanas, C. Vilas. VOREST: Modelización de bosques mediante diagramas de Voronoi. *Actas de los XII Encuentros de Geometría Computacional (EGC'07)*, pp. 249–256, 2007.

[2] M. Abellanas, G. Hernández. Optimización de rutas de evacuación, *Actas de los XII Encuentros de Geometría Computacional (EGC'07)*, pp. 273–280, 2007.

[3] M. Abellanas, G. Hernández, J. L. Moreno, S. Ordóñez, V. Sacristán. DVALon: una herramienta para diagramas de Voronoi y grafos de proximidad de alcance limitado. aceptado en los *XIII Encuentros de Geometría Computacional (EGC'09)*, 2009.

[4] M. Abellanas, G. Hernández, J. L. Moreno, S. Ordóñez, V. Sacristán. DVALon. `http://www-ma2.upc.es/~geoc/DVALon/`.

[5] M. Abellanas, G. Hernández, V. Sacristán. Caminos de desviación mínima local, aceptado en los *XIII Encuentros de Geometría Computacional (EGC'09)*, 2009.

[6] B. Cărbunar, A. Grama, J. Vitek, O. Cărbunar. Redundancy and Coverage Detection in Sensor Networks, *ACM Transactions on Sensor Networks*, Vol. 2, N. 1, pp. 94–128, 2006.

[7] O. Devillers, S. Pion, M. Teillaud. Walking in a Triangulation, *Internat. J. Found. Comput. Sci.*, Vol. 13, pp. 181–199, 2002.

[8] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis. Graph Drawing: Algorithms for the Visualization of Graphs, *Prentice Hall*, 1999.

[9] J. Gao, L. J. Guibas, J. Hershberger, L. Zhang, A. Zhu. Geometric Spanner for Routing in Mobile Networks, *Proc. 2nd ACM International Symposium on Mobile and Ad Hoc Networking and Computing (MobiHoc'01)*, pp. 45–55, 2001.

[10] J. E. Goodman, J. O'Rourke. Handbook of Discrete and Computational Geometry, *Chapman & Hall / CRC Press, 2a edició*, pp. 1163–1185, 2004.

[11] J. Hershberger, S. Suri. On Computing Shortest Paths in the Plane, *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 508–517, 1993.

[12] K. Kedem, J. Pach, M. Sharir. On the Union of Jordan regions and collision-free translational motion amidst polygonal obstacles, *Discrete and Computational Geometry, 1*, pp. 59–71, 1986.

[13] X.-Y. Li, G. Calinescu, P.-J. Wan, Y. Wan. Localized Delaunay Triangulation with Application in Ad Hoc Wireless Networks, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, N. 10, pp. 1035–1047, 2003.

[14] S. Megerian, F. Koushanfar, M. Potkonjak, M. B. Srivastava. Worst and Best-Case Coverage in Sensor Networks, *IEEE Transactions on Mobile Computing*, Vol. 4, N. 1, pp. 84–92, 2005.

[15] S. Meguerdichian, F. Koushanfar, M. Poktonjak, M. B. Srivastava. Coverage Problems in Wireless Ad-hoc Sensor Networks, *Proc. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFORCOM 2001)*, Vol. 3, pp. 1380–1387, 2001.

[16] D. P. Mehta, M. A. Lopez, L. Lin. Optimal Coverage Paths in Ad-hoc Sensor Networks, *Proc. IEEE International Conference on Communications, ICC'03*, pp. 507–511, 2003.

[17] A. Okabe, B. Boots, K. Sugihara. Spatial Tessellations Concepts and Applications of Voronoi Diagrams, *Ed. JHON WILEY & SONS*, 1992.

[18] W. L. Roque, H. Choset. The Green Island Formation in Forest Fire Modeling with Voronoi Diagrams, *Proc. 3rd CGC Workshop on Computational Geometry*, 1998.

[19] W. R. Smith. Area Potentially Available to a Tree: A Research Tool, *Proc. 19th Southern Forest Tree Improvement Conference, SFTIC*, pp. 22–29, 1987.

[20] G. T. Toussaint. The Relative Neighbourhood Graph of a Finite Planar Set, *Pattern Recognition*, Vol 12, pp. 261–268, 1980.

[21] G. T. Toussaint, J. W. Jaromczyk. Relative Neighbourhood Graphs and Their Relatives, *Proc. IEEE, vol.80, nº 9*, pp. 1502–1517, 1992.

[22] H. Xu, L. Huang, Y. Wan, K. Lu. Localized Algorithm for Coverage in Wireless Sensor Networks, *Proc. IEEE Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'05)*, pp. 750–754, 2005.