UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Triangulaciones de número cromático mínimo

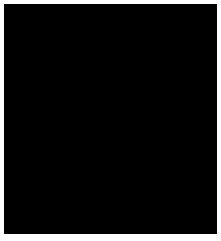
T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

JUAN ALFREDO CRUZ CARLÓN



DIRECTOR DE TESIS:
DR. JORGE URRUTIA GALICIA

2011

Hoja de datos del Jurado

1.- Datos del alumno
Cruz
Carlón
Juan Alfredo
54 46 94 01
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
096001228

2.- Datos del tutor
Dr
Jorge
Urrutia
Galicia

3.- Datos del sinodal 1
Dr
Ruy
Fabila
Monroy

4.- Datos del sinodal 2
M en C
Marco Antonio
Heredia
Velasco

5.- Datos del sinodal 3
Dr
David
Flores
Peñaloza

6.- Datos del sinodal 4
M en C
Canek
Peláez
Valdés

7.- Datos del trabajo escrito
Triangulaciones de número cromático mínimo
65 p
2011

Agradecimientos

A todos mis sinodales y en especial al Dr. Jorge Urrutia Galicia y al M. en C. Marco Antonio Heredia Velasco.

A TOD@S l@s que
hicieron, hacen y harán a la
UNAM una realidad.

Índice

Introducción

La Geometría Computacional es una de las áreas de Las Ciencias de la Computación donde los resultados puramente matemáticos también dan solución a problemas de la vida cotidiana. Dentro de esta joven disciplina hay una vasta cantidad de problemas; uno donde la investigación es muy activa, debido en parte a sus aplicaciones, es el que estudia las triangulaciones de conjuntos de puntos en posición general.

Una triangulación T de un conjunto P de puntos en posición general en el plano es una gráfica geométrica plana maximal en cuanto al número de aristas. Es decir, para cualquier par de aristas, éstas no se intersectan salvo en sus extremos y si hay dos puntos a, b de P que no sean adyacentes (no hay arista con extremos a, b) entonces el segmento de recta ab intersecta al menos una arista de T en un punto distinto de a y b. Ver Figura 1.

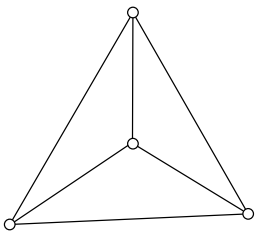


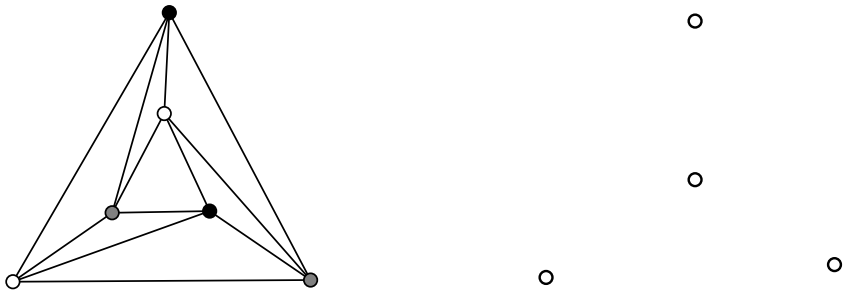
Figura 1: Triangulación de un conjunto de puntos en posición general.

Las triangulaciones podemos encontrarlas en diversos lugares algunos de los cuales son los sistemas de información geográfica, la graficación 3D, diseño asistido por computadora y análisis de datos finitos para la simulación de fenómenos, etc.

Todo conjunto de puntos en posición general con al menos tres elementos admite una triangulación. Pero hay veces en las que requerimos cierto *tipo* particular de triangulación.

En este trabajo estudiaremos un tipo de triangulaciones que llamaremos *triangulaciones 3 coloreables*. Una triangulación es 3 coloreable si sus vértices se pueden

colorear usando solamente tres colores y de forma que dos vértices adyacentes no compartan el mismo color (ver Figura 2a). El lector se podrá dar cuenta que no todo conjunto de puntos acepta una triangulación de este estilo, un ejemplo es el de la Figura 2b. Éste conjunto de puntos solamente acepta una triangulación y ésta no puede colorearse con tres colores.



(a) (b)

Figura 2: Conjunto de puntos que no acepta triangulación par.

Si no todo conjunto de puntos acepta una triangulación 3 coloreable, entonces es natural preguntarnos, ¿cuántos puntos necesitamos añadir a nuestro conjunto original para que aseguremos que existe una triangulación 3 coloreable? Este es el problema que abordamos en este trabajo.

En el primer capítulo definiremos algunos conceptos necesarios y haremos una breve revisión bibliográfica de los artículos relacionados. En el segundo capítulo demostraremos que insertando (en el peor caso) $O(n)$ puntos Steiner a cualquier conjunto de puntos en posición general el conjunto resultante acepta una triangulación tres coloreable. La demostración es constructiva y nos lleva a una heurística de complejidad $O(n \log(n))$. Finalmente expondremos algunas concusiones y trabajo futuro.

Intentamos ser autocontenidos, sin embargo referimos al lector a las obras [CLRS01] y [Har94] como referencia si se quiere profundizar en algunos conceptos aquí usados.

Capítulo 1

Definición del problema y resultados previos

1.1 Definiciones

Sea P un conjunto de n puntos en el plano en *posición general*, esto es, no hay tres de ellos sobre una recta. De ahora en adelante usaremos P para denotar a un conjunto de tal naturaleza.

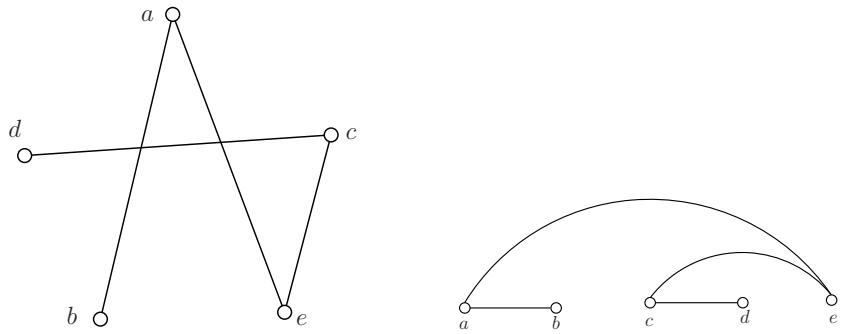
Una *gráfica* $G = (V, E)$ (o simplemente G) es un par ordenado de conjuntos disjuntos. A V se le denomina el conjunto de vértices y E el de aristas. E es un subconjunto del conjunto de pares desordenados de V. Si $e \in E$, $e = \{a, b\}$, decimos que a y b $(a, b \in V)$ son adyacentes y que e incide en a y en b. Otra notación utilizada para e es ab.

Al trabajar con gráficas es usual recurrir a representaciones visuales (dibujos) de las mismas. Sea $G = (V, E)$ una gráfica, representamos cada vértice u de G con un punto p_u del plano y cada arista uv con un arco a_{uv} que una a los puntos p_u y p_v de modo que a_{uv} no pase por ningún p_z con $z \neq u, v$. Si a^-_{uv} denota al arco a_{uv} sin sus extremos, exigimos también las siguientes condiciones: Si $a^-_{uv} \cap a_{u'v'}$ es no vacío, entonces consiste de un único punto siempre y cuando u, v, u', v' son distintos dos a dos. La figura que se obtiene de esta manera tiene también estructura de gráfica y es una *copia*, *representación geométrica* o *dibujo* de G. Por ejemplo, si se tiene la gráfica $H = (\{a, b, c, d, e\}, \{\{a, b\}, \{c, d\}, \{a, e\}, \{c, e\}\})$ dos posibles dibujos serían los de la Figura 1.1

Los dibujos de una gráfica son tan importantes que, por ejemplo, el concepto de *planaridad* es definido en términos de ellas. Decimos que una gráfica es *planar* si existe un dibujo tal que sus aristas no se cruzan. El lector podrá observar que en el

dibujo de H de la Figura 1.1a la arista cd cruza a las aristas ab y ae. Sin embargo H es planar porque en el dibujo de la Figura 1.1b las aristas no se cruzan.

Una gráfica *plana* es una gráfica tal que su conjunto de vértices son puntos del plano y su conjunto de aristas son curvas simples que no se intersectan salvo a lo más en los extremos.



(a) Dibujo no plano de H. (b) Dibujo plano de H.

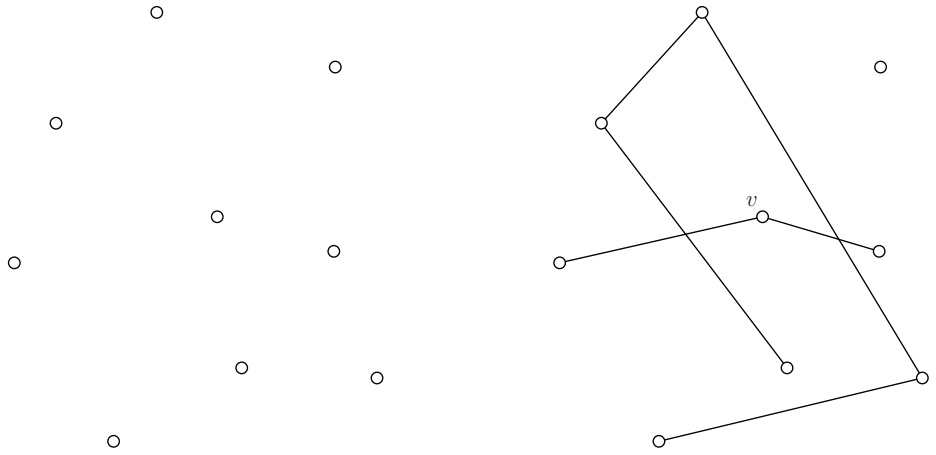
Figura 1.1: Dos dibujos distintos para la gráfica H.

Una gráfica geométrica $\mathcal{G} = (V, E)$ es una gráfica tal que $V \subseteq \mathbb{R}^2$ y E es un conjunto de segmentos de recta que unen elementos de V sujetos a las condiciones en la definición de dibujo. Ver Figura 1.2.

Si v es un vértice de G y denotamos por $\Gamma(v)$ al conjunto de sus vértices adyacentes, entonces el *grado* de v (denotado por $\delta(v)$) es $\delta(v) = |\Gamma(v)|$. Por ejemplo, el vértice v de la Figura 1.2 tiene grado dos.

Sea A un conjunto de puntos en el plano. El *cierre convexo* de A (denotamos por $CC(A)$) es el polígono más pequeño que contiene a todos los elementos de A. A los vértices de $CC(A)$ que también son elementos de A se les conoce como *vértices del cierre convexo*, al resto $(A \setminus V(CC(A))$, donde $V(CC(A))$ son los vértices de $CC(A))$ se les conoce como *puntos interiores* de A y a su conjunto lo denotamos por $Int(CC(A))$. Ver Figura 1.3.

Sea $G = (V, E)$ una gráfica. Una *subgráfica* $G' = (V', E')$ de G es una gráfica tal que $V' \subseteq V$ y $E' \subseteq E$. Un *camino* $\mathcal{W}$ de G es una subgráfica de G conformada de una sucesión de vértices y aristas, $\mathcal{W} = v_0e_0v_1, \ldots, v_ie_iv_{i+1}, \ldots, v_ne_nv_{n+1}$, donde $e_i = v_iv_{i+1}$ y $v_j \in V, \forall j, 0 \leq j \leq n+1$. $\mathcal{W}$ es un *camino cerrado*, si $v_0 = v_{n+1}$. Por comodidad adoptaremos la notación que omite a las aristas, i.e. $\mathcal{W} = v_0, v_1, \ldots, v_i, v_{i+1}, \ldots, v_n, v_{n+1}$. Si $\mathcal{W}$ es un camino que no repite vértices, lo llamaremos *trayectoria*. Si $\mathcal{W}$ es una trayectoria cerrada entonces $\mathcal{W}$ es un *ciclo*. Ver Figura 1.4.



(a) Un conjunto de puntos en posición general.

(b) Una gráfica geométrica sobre el conjunto de la Figura 1.2a.

Figura 1.2

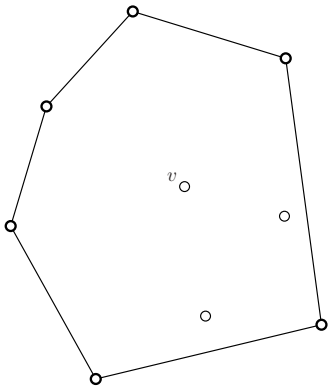
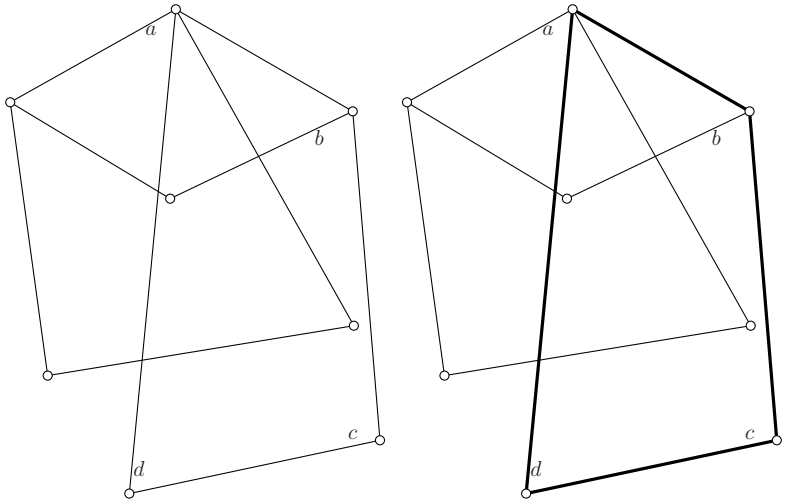


Figura 1.3: El cierre convexo del mismo conjunto de puntos de la Figura 1.2a. El punto u es un punto interior del conjunto de puntos.



(a) Una gráfica G. (b) Un ciclo $\mathcal{W} = a, b, c, d, a$ de G.

Figura 1.4: Una gráfica G y un ciclo $\mathcal{W}$ de G.

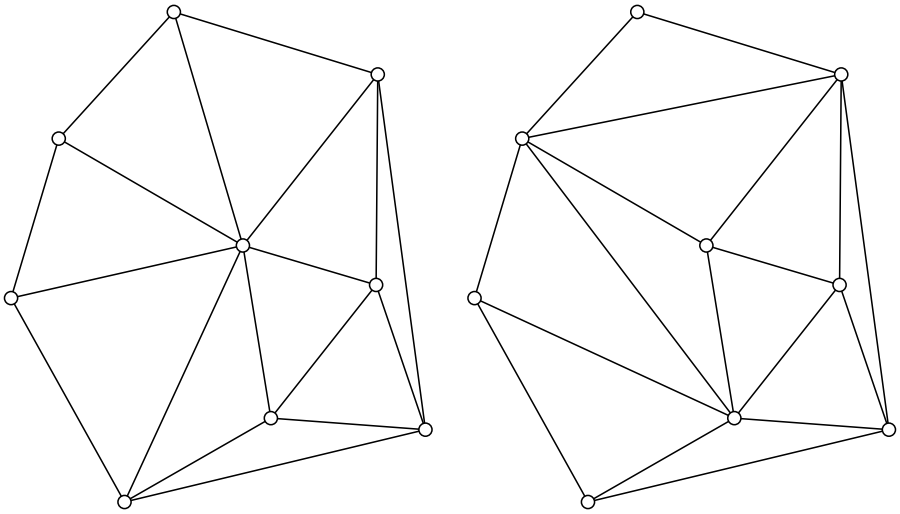
Si G es una gráfica plana, entonces G divide al plano en regiones conexas, una de estas es la *externa* o no acotada. A estas regiones se les conoce como *caras*. Cada cara es acotada por un ciclo y por lo tanto cada arista de cada ciclo separa a dos caras. En lo que resta de este trabajo llamaremos a todas las caras de G distintas de la no acotada como *caras acotadas*.

Una *triangulación* $\mathcal{T}(P) = (P, E)$ de P es una gráfica geométrica plana maximal, en cuanto al número de aristas, cuyo conjunto de vértices es P. Es decir es una gráfica geométrica tal que si $p, q \in P$ y la arista pq no está en E entonces $\mathcal{T}'(P) = (P, E \cup \{p, q\})$ no es plana. Para que la condición anterior se cumpla, es necesario que todas las caras acotadas de $\mathcal{T}(P)$ sean triángulos, de ahí el nombre de triangulación. Ver Figura 1.5.

Diremos que $\mathcal{T}(P)$ es una *triangulación pseudo par* si todo vértice en el conjunto $P \setminus CC(P)$ tiene grado par.Ver Figura 1.5b.

Los dos conceptos anteriores tienen sus equivalentes en gráficas. Diremos que $\mathscr{T} = (V, E)$ es una *triangulación* si es una gráfica plana maximal en el número de aristas. Además diremos que $\mathscr{T}$ es una *triangulación par* si todos sus vértices tienen grado par.

Sea C un conjunto de colores, $G = (V, E)$ una gráfica y $c : V \to C$ una función de los vértices de la gráfica a C. Decimos que c es una *coloración* de los vértices de G, o simplemente una coloración de G, si en c dos vértices adyacentes reciben colores



(a) Una triangulación de P. (b) Una triangulación pseudo par de P.

Figura 1.5: Dos triangulaciones para el conjunto de puntos Figura 1.2.

distintos. El *número cromático* de G, denotado por $\chi(G)$, es el mínimo número de colores necesarios para colorear G. Es decir, si $c' : V \to C'$ es una coloración de G, entonces $\chi(G) \leq |C'|$.

Decimos que una gráfica $G = (V, E)$ es *bipartita* si $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$ y para toda arista $ab \in E$, $a \in V_1$ y $b \in V_2$.

Una gráfica G es *conexa* si para cualquier par de vértices, existe un camino entre ambos. Decimos que es *disconexa* en otro caso. G es k conexa si es necesario remover al menos k vértices (y todas las aristas incidentes en ellos) de G para que lo que reste sea disconexo.

En el presente trabajo mostraremos un algoritmo que dado P construye una triangulación de $P \cup S$ ($\mathcal{T}(P \cup S)$) tal que su número cromático es tres. S es un conjunto de puntos adicionales llamados puntos Steiner (por el matemático suizo Jakob Steiner).

1.2 Trabajo previo

En [HK96] Hoffmann y Kriegel demuestran un teorema que resuelve el equivalente en gráficas al problema que en este trabajo se aborda. El teorema es el siguiente:

Teorema 1.1 *Sea G una gráfica plana, 2 conexa y bipartita. Entonces se le pueden añadir aristas a G tal que la gráfica obtenida $\mathscr{T}$ sea una triangulación 3 coloreable.*

Para demostrarlo usan dos lemas, el primero demostrado por Whitney y el segundo por ellos mismos.

Lema 1.2 *Una triangulación $\mathscr{T}$ es 3 coloreable si y sólo si todos sus vértices tienen grado par.*

Lema 1.3 *[HK96] Sea G una gráfica 2 conexa, bipartita y plana. Entonces existe un conjunto de aristas E' tales que $\mathscr{T} = (V, E \cup E')$ es una triangulación par.*

Por último los autores exponen un complicado algoritmo de complejidad $O(n^2)$ que para cualquier G, plana, 2 conexa y bipartita obtiene una triangulación $\mathscr{T}$ par.

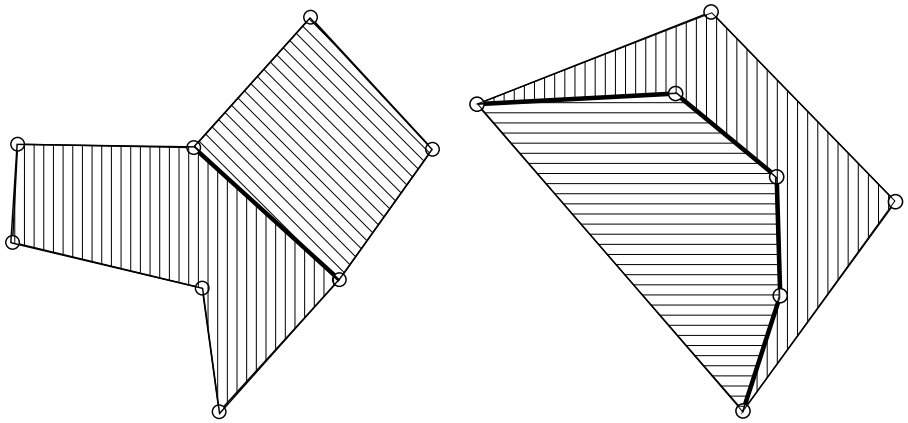
La aportación de Zhang y He en [ZH05] es la construcción de un nuevo algoritmo que en tiempo $O(n)$ construye una triangulación par. No es el algoritmo en sí el que vamos a discutir, sino las suposiciones que se hacen en ambas publicaciones sobre la gráfica inicial G. Éstas nos ayudarán a dar una cota superior a nuestro problema.

Hoffmann, Kriegel, Zhang y He suponen que las caras acotadas de G son ciclos de longitud 4 (que denotaremos como C_4) las cuales *aceptan* sus dos posibles diagonales. Un C_4 *acepta* una diagonal, si ésta divide la cara delimitada por él en dos caras[1]. El problema de encontrar una triangulación par es reducido a encontrar un conjunto de diagonales *adecuado*, es decir un conjunto de diagonales que haga que todos los vértices de G tengan grado par. Para encontrar dicho conjunto, Zhang y He hacen uso de una construcción de Hoffmann y Kriegel llamados S-*Caminos*.

Para un gráfica plana G, podemos definir el concepto de *cara vecina*. Dos caras de G son vecinas si los ciclos que las delimitan comparten al menos una arista. Diremos, también, que una cara F es una cara vecina de grado k de una cara F', si los ciclos que las delimitan comparten k aristas. Ver Figura 1.7.

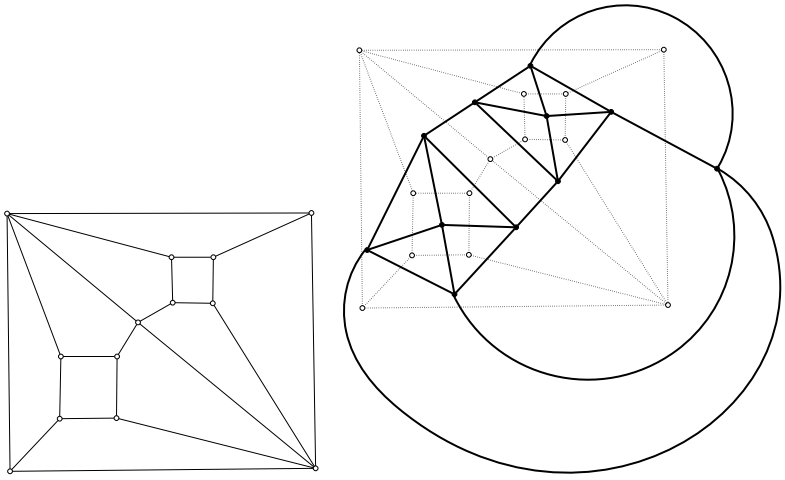
Las definiciones anteriores son útiles para definir la gráfica dual G^* de una gráfica plana G. Dada G, G^* tiene un vértice f_i por cada cara (incluida la exterior) F_i de G. Dos vértices de G^* son adyacentes si las caras que representan son vecinas. Si dos vértices de G^* representan dos caras vecinas de grado k, entonces hay k aristas entre ambos vértices. (Ver Figura 1.7). En lo que resta de esta sección supondremos que G_4 denota una gráfica donde todas sus caras son acotadas por ciclos de longitud cuatro.

[1] El lector podrá notar que esta definición puede ser generalizada para ciclos de longitud k.



(a) Caras vecinas de grado 1. (b) Caras vecinas de grado 4.

Figura 1.6: Dos ejemplos de caras vecinas.



(a) Una gráfica G. (b) La gráfica G^* de G.

Figura 1.7: Una gráfica G donde todas sus caras son ciclos de longitud cuatro y su gráfica dual.

Sea $C_i = a, b, c, d, a$ un C_4 que delimita una cara de G_4. Sean C_j, C_k los C_4 que comparten las aristas cd y ab con C_i respectivamente. Entonces, las caras delimitadas por C_j,C_k son *caras opuestas*. Ver Figura 1.8.

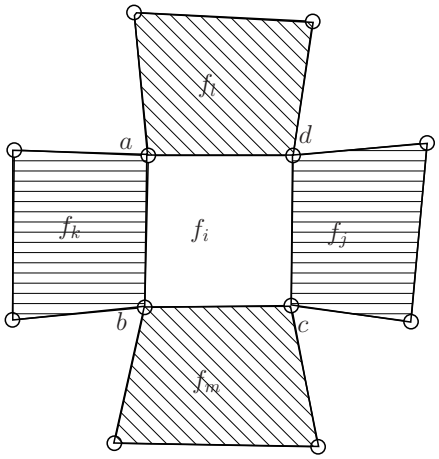


Figura 1.8: La cara f_k es opuesta a la cara f_j y la cara f_l es opuesta a la cara f_m.

Un *S-Camino* de G_4^* es un camino cerrado, $\mathcal{C} = f_1, f_2, \ldots, f_n, f_1$ que para cada tres vértices adyacentes, f_{i-1}, f_i, f_{i+1}, la cara representada por f_{i-1} es opuesta a la cara representada por f_{i+1}. Zhang y He notan que el conjunto de aristas de G_4^* puede ser particionado de forma única usando S-Caminos. (Ver Figura 1.9). Además muestran que al construir todos los S-Caminos necesarios para particionar las aristas de G^*, estos inducen una orientación en las aristas de G llamada *G-orientación*.

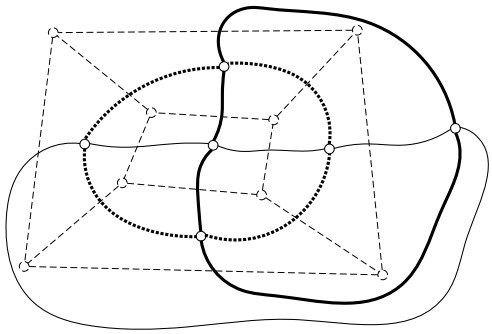


Figura 1.9: Una gráfica G, su gráfica dual G^* y la partición de las aristas de G^* usando S-Caminos.

Una *G-orientación* descompone cada C_4 que define una cara de G_4 en dos trayectorias dirigidas, una en contra de las manecillas del reloj y la otra a favor de ellas.

Los vértices iniciales y finales de las dos trayectorias dirigidas son llamados vértices principales. Los otros dos vértices restantes son llamados vértices secundarios. La *diagonal princial* de un C_4 es la arista que incide en sus dos vértices principales. Ver Figura 1.10.

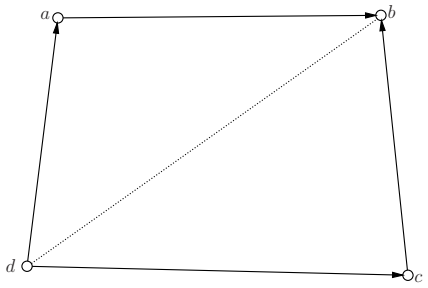


Figura 1.10: Una G-Orientación para el $C_4 = a, b, c, d, a$ y su diagonal principal db.

Finalmente en [ZH05] se demuestra el siguiente teorema:

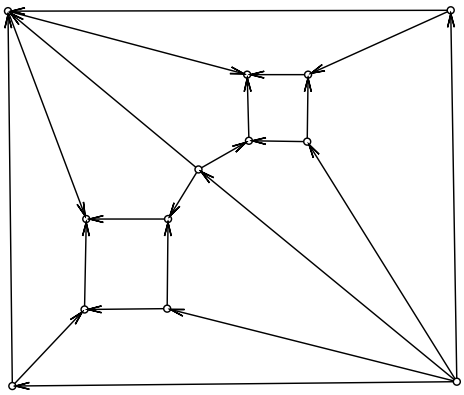
Teorema 1.4 *Sea G una gráfica plana, bipartita y conexa y sea $\mathcal{G}$ una G-orientación de G. Si a cada ciclo se le añade su diagonal princial, entonces lo resultante es una triangulación par.*

Un posible acercamiento para resolver el problema que aborda este trabajo es el intentar aplicar el algoritmo de Zhang y He a gráficas geométricas. Como veremos más adelante esto nos llevará a determinar una cota superior para nuestro problema.

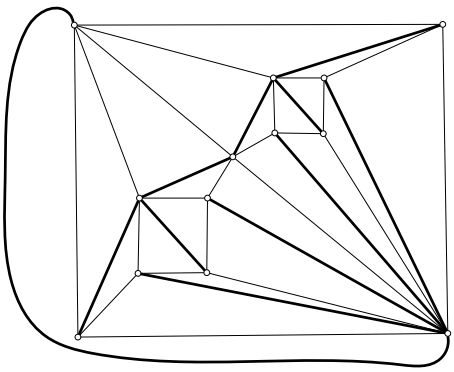
1.2.1 Gráficas geométricas

Las gráficas geométricas que cumplen con las condiciones del Teorema 1.1 reciben el nombre de cuadrilaterizaciones. En [TB97] se demuestra que P admite una cuadrilaterización si y sólo si, $|CC(P)|$ es par. Por este motivo en este apartado supondremos que P cumple con esta condición.

Sea P el conjunto de la Figura 1.12a y sea $\mathcal{C}(P)$ una cuadrilaterización de P. En la Figura 1.12c podemos ver la dirección de las aristas inducidas por un S-camino. El lector podrá observar que el algoritmo de Zhang y He insertaría la arista que une a los vértices opuestos del cierre convexo de P. En este caso, el algoritmo no construye una triangulación pseudo par, es más, el algoritmo no construye una triangulación.



(a) G-orientación

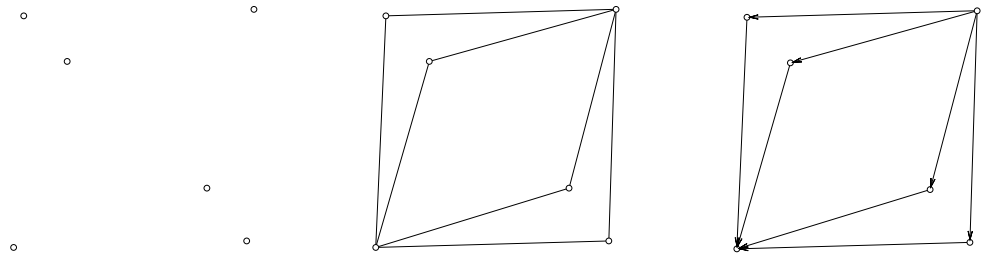


(b) Las aristas principales resultantes de la G-Orientación

Figura 1.11: Una G-orientación para la gráfica G de la figura 1.7a y sus aristas principales.

Por otro lado si P es el conjunto de la Figura 1.13a el mismo algoritmo sí construye una triangulación pseudo par. Nótese que en este último la triangulación resultante no cumple con el Lema 1.2, esto es porque no estamos considerando cuadriáteros en la cara exterior, pues la planaridad no se mantendría. Esta observación se hará más clara con el Lema 1.5.

El motivo por el que la cuadrilaterización de la Figura 1.12b falla con el algoritmo de Zhang y He es porque éste asume que todo C_4 acepta ambas diagonales. Esto no es necesariamente cierto para gráficas geométricas. Para poderlo aplicar es necesario que para un conjunto de puntos podamos encontrar su cuadrilaterización convexa. Heredia y Urrutia demuestran en [HU07] que para construir una cuadrilaterización convexa de un conjunto de puntos basta con introducir $\frac{4n}{5} + 2$ puntos Steiner al conjunto.



(a) Conjunto de puntos

(b) Una cuadrilaterización del conjunto de puntos

(c) Las direcciones de las aristas asociadas a un S-Camino

Figura 1.12: Un conjunto de puntos al cual no podemos aplicar el teorema 1.1

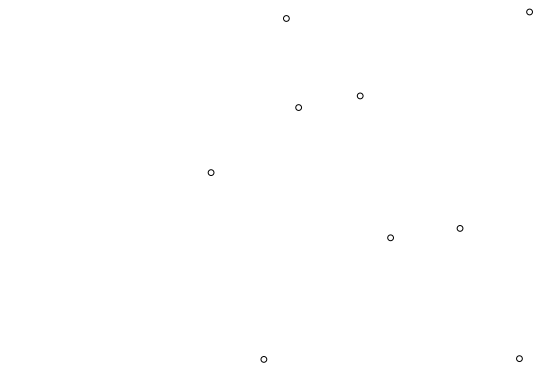
De lo anterior se desprende la meta del presente trabajo. Dado un conjunto de n puntos en posición general, construir una triangulación par insertando, en el caso promedio, un número menor a $\frac{4n}{5} + 2$ puntos Steiner.

Una gráfica $G = (V, E)$ es *extraplana*[2] si existe un dibujo de ella tal que sus vértices pertenezcan a su cara exterior. Toda gráfica extraplana es plana, pero el inverso no es correcto. Un ejemplo es la gráfica *completa*[3] de cuatro vértices denotada por K_4. No importa cómo se represente, K_4 siempre va a tener un vértice que no está en la cara exterior. Ver Figura 1.14.

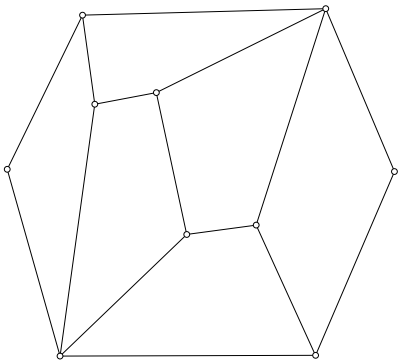
Un *pseudo-triángulo* es un polígono simple (un polígono cuyas aristas no se cruzan) tal que tiene exactamente tres vértices cuyos ángulos interiores son menores que π. El lector podrá observar que todo triángulo es un pseudo-triángulo. Una *pseudo-triangulación* de un conjunto de puntos en posición general P es una des-

[2]En inglés *outerplanar*

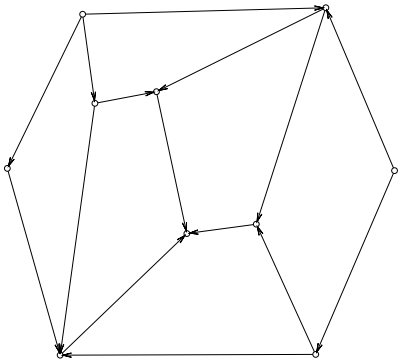
[3]Una gráfica es *completa* si todo par de vértices es adyacente.



(a) Conjunto de puntos



(b) Una cuadrilaterización del conjunto de puntos



(c) Las direcciones de las aristas asociadas a un S-Camino

Figura 1.13: Un conjunto de puntos al cual podemos aplicar el Teorema 1.1

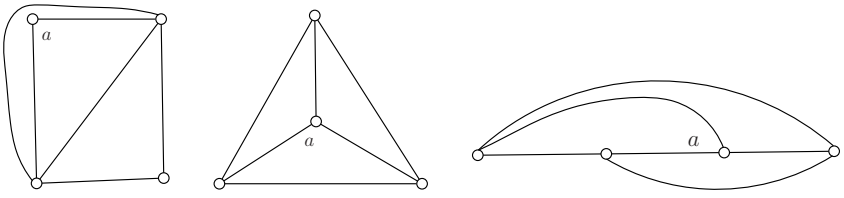


Figura 1.14: Tres formas de dibujar K_4. En todas el vértice a no pertenece a la cara exterior.

composición de la región del plano acotada por $CC(P)$ tal que cada región de la descomposición es acotada por un pseudo-triángulo. Una pseudo-triangulación de P es *puntual*, si es una pseudo-triangulación de P y además cada vértice $p \in P$ tiene una cara que incide en él tal que su ángulo en p es mayor de π. Una cara F incide en un punto p, si éste forma parte de un ciclo que delimita a F de otra cara F'; el ángulo de F en p es el ángulo formado por las aristas del ciclo incidentes en p del lado de F. Ver Figura 1.15.

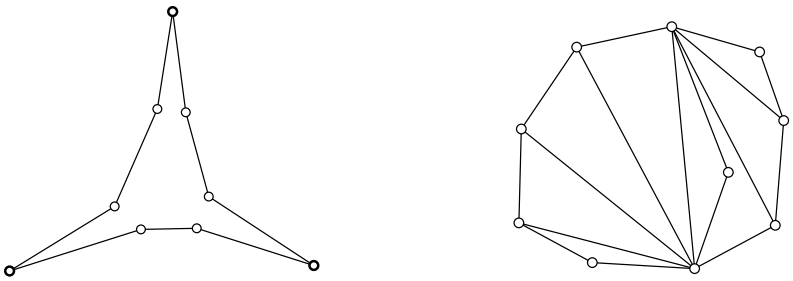


Figura 1.15: Un pseudo triángulo (izquierda) y una pseudo-triangulación de un conjunto de puntos (derecha).

Aichholzer, *et. al.* demuestran en [AHH$^+$09] varios resultados relacionados con el presente trabajo. Dado P, supóngase que se tiene una función $r : P \to \{par, impar\}$ que asigna restricciones de paridad a los elementos de P. Se dice que G satisface la restricción de paridad de un vértice u si la paridad de su grado es igual a $r(u)$.

En [AHH$^+$09] se muestra una forma de construir una gráfica extra plana y una pseudo-triangulación puntual para un conjunto de puntos P con restricciones de paridad. Tales gráficas cumplen que todos sus vértices, con excepción de a lo más tres, satisfacen sus restricciónes de paridad. El resultado obtenido por los autores más relevante para este trabajo dice que siempre existe una triangulación que satisface 2/3 de todas las restricciones.

Por último, al momento de la escritura de este trabajo dos artículos fueron publicados. En [PRU10], Peláez, *et.al.* mejoran la cota demostrada por Aichholzen, *et. al.* en [AHH$^+$09] usando una técnica distinta. En [Alv10] Álvarez muestra que $P \cup S$ siempre admite una triangulación pseudo par tal que $|S| \leq \frac{n}{3}$, donde $n = |P|$.

El siguiente Lema demostrado por Diks, et. al. en [DKK02] relaciona las triangulaciones 3 coloreables con las triangulaciones pseudo pares.

Lema 1.5 *Cada* casi triangulación *internamente par es 3 coloreable.*

Una *casi triangulación* es una gráfica plana G que es biconexa y que todas las caras excepto por la exterior están acotadas por ciclos de longitud 3. El lector podrá notar que toda gráfica geométrica que además es una triangulación, es una *casi triangulación*.

Capítulo 2

La heurística

En este capítulo mostraremos la construcción de una heurística que construye una triangulación pseudo par de un conjunto de puntos en posición general usando puntos Steiner.

2.1 Vuelta a la izquierda y derecha, triangulaciones abanico y rueda

2.1.1 Vuelta a la izquierda y derecha

Si $p \in \mathbb{R}^2$, entonces podemos asociarle coordenadas x, y. A la coordenada x de p la denotaremos por $p.x$ y denotamos por $p.y$ a su coordenada y. Sean $a, b, c \in \mathbb{R}^2$ tres puntos en el plano y sean $\overline{ba}$, $\overline{ac}$ dos segmentos de recta con puntos finales b, a y a, c respectivamente. Si $a = (0, 0)$, entonces el segmento $\overline{ba}$ puede ser visto como el vector $\vec{b}$. Análogamente, el segmento $\overline{ac}$ lo podemos ver como el vector $\vec{c}$ (ver Figura 2.1). Dada la configuración anterior, podemos contestar la pregunta de si el vector $\vec{c}$ está *en contra* o *a favor* de las manecillas del reloj del vector $\vec{b}$. El vector $\vec{c}$ está en contra de las manecillas del reloj del vector $\vec{b}$ si el ángulo entre ellos, medido de $\vec{c}$ a $\vec{b}$, es menor que π. El vector $\vec{c}$ está a favor de las manecillas del reloj del vector $\vec{b}$ si el ángulo de $\vec{c}$ a $\vec{b}$ es mayor que π. Los vectores $\vec{b}$ y $\vec{c}$ son colineales si su ángulo es π.

Si el vector $\vec{c}$ está en contra de las manecillas del reloj del vector $\vec{b}$, decimos que los segmentos de recta $\overline{ba}$ y $\overline{ac}$ dan *vuelta a la izquierda* en a. Si el vector $\vec{c}$ está a favor de las manecillas del reloj del vector $\vec{b}$, decimos que $\overline{ba}$ y $\overline{ac}$ dan *vuelta a la derecha* en a. Si $\vec{b}$ y $\vec{c}$ son colineales, entonces $\overline{ba}$ y $\overline{ac}$ también lo son. Ver Figura 2.2.

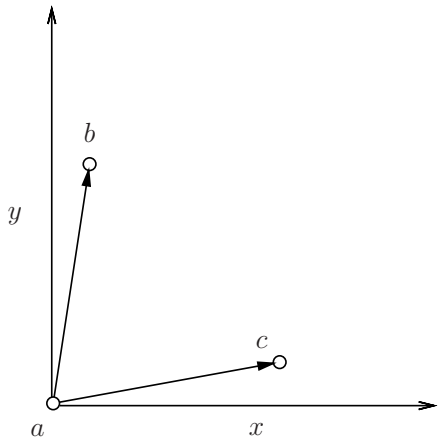
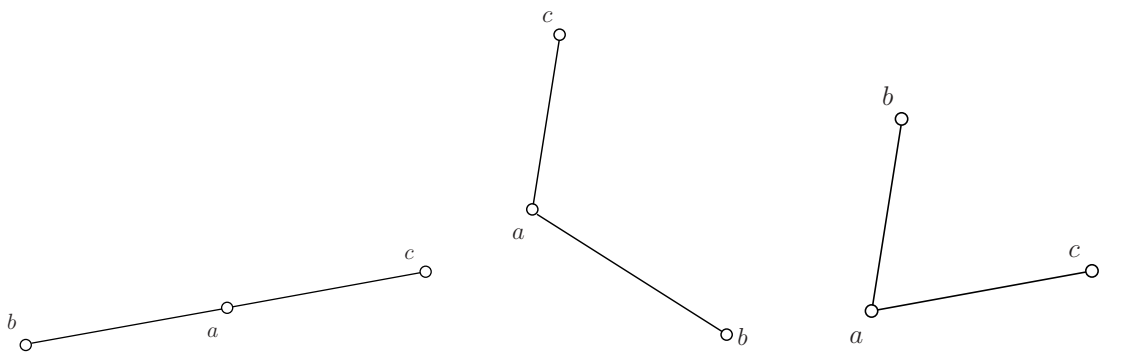


Figura 2.1



(a) $\overline{ba}$ y $\overline{ac}$ son colineales

(b) $\overline{ba}$ y $\overline{ac}$ dan vuela a la derecha en a

(c) $\overline{ba}$ y $\overline{ac}$ dan vuela a la izquierda en a

Figura 2.2: Las tres posibles configuraciones para los segmentos ab y ac

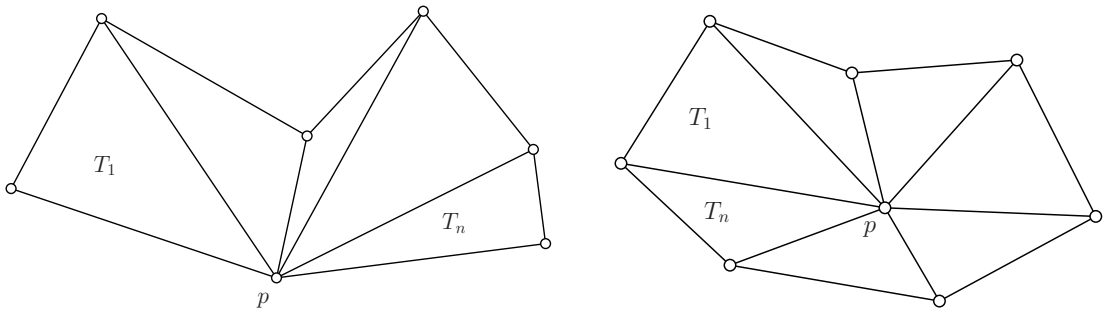
En [CLRS01] se muestra una rutina llamada DIRECTION, expuesta a continuación, que calcula cuándo dos segmentos $\overline{ba}$ y $\overline{ac}$ dan vuelta a la izquierda o derecha.

DIRECTION regresa un número negativo si $\overline{ac}$ está orientado en contra de las manecillas del reloj respecto a $\overline{ba}$; un número positivo si $\overline{ac}$ está orientado en sentido de las manecillas del reloj respecto a $\overline{ba}$; cero cuando a, b y c son colineales.

DIRECTION(a, b, c)
1 **return** $(c - a) \times (b - a)$

2.1.2 Triangulaciones abanico y rueda

Sean $T_1, T_2, \ldots, T_n$ n triángulos tales que todos comparten un mismo vértice p, T_i, T_{i+1}, $1 \leq i < n$ comparten una arista y si $i \neq j$, los interiores de T_i y T_j son ajenos. Si consideramos a los triángulos $T_1, \ldots, T_n$ como una gráfica geométrica H, entonces, si p está en la cara exterior de H, H recibe el nombre de *triangulación abanico*. Si p no está en la cara exterior de H entonces llamamos a H una *triangulación rueda*. Nótese que en una triangulación rueda también se cumple que T_n comparte una arista con T_1. Ver Figura 2.3. En las triangulaciones rueda llamamos a p su *vértice central*. En las triangulaciones abanico, p recibe el nombre de *vértice común* y los vértices de T_1 y T_n que no comparten con T_2 y T_{n-1} respectivamente reciben el nombre de *vértices extremos*.



(a) Abanico (b) Rueda

Figura 2.3: Triangulaciones abanico y rueda con vértice común y central p

Las triangulaciones abanico y rueda nos ayudan a definir el concepto de *vértice triangulado*. Sea G una gráfica geométrica de un conjunto de puntos en posición general P y q uno de sus vértices. Sea Δ el conjunto de vecinos de q. Sea G_q la subgráfica generada por los vértices del conjunto $\Delta \cup \{q\}$. Si q es un vértice interior

de G, decimos que q está triangulado si existe H subgráfica de G_q tal que H es una triangulación rueda con q como su vértice central y cada uno de los triángulos de H no contiene en su interior algún vértice de G. Si por otro lado, q es un vértice del cierre convexo de P, decimos que q está triangulado si existe una subgráfica H de G_q tal que H es una triangulación abanico con q como su vértice común, cada uno de los triángulos de H no contienen en su interior ningún vértice de G, y los vértices extremos de H son los vecinos de q en el cierre convexo de P.

2.2 Espirales convexas

En esta sección vamos a construir una gráfica geométrica Tr de un conjunto de puntos en posición general P que recibe el nombre de *espiral convexa*.

Sea a el punto con la coordenada x menor, en caso de empate, a será el punto que también tenga la coordenada y menor. Sea $\mathcal{L}$ la recta paralela al eje Y que pasa por x. Marquemos x como *visto*. Rotemos $\mathcal{L}$ a favor de las manecillas del reloj hasta encontrar el siguiente punto de P no visitado. Sea b tal punto. Incrementar las aristas de Tr en la arista ab. Continuemos por inducción rotando $\mathcal{L}$ en sentido de las manecillas del reloj pero ahora desde el último ́vertice de Tr marcado (b) hasta encontrar el siguiente vértice no marcado.

La construcción anterior es introducida por Toussaint en [TB97]. Como el lector podrá notar, el conjunto P puede ser rotado de tal forma que cualquier vértice de su cierre convexo cumpla con las características del punto a. Por otro lado la recta $\mathcal{L}$ puede ser rotada tanto en sentido de las manecillas del reloj como en sentido contrario. A todas las gráficas geométricas Tr que se puedan construir para un conjunto P mediante rotaciones y elecciones de rotación para la recta $\mathcal{L}$ las llamaremos *espirales convexas*.

2.3 La heurística

La heurística propuesta recorre la espiral convexa de P en orden contrario, es decir, comienza en el último punto añadido a ésta. En cada paso revisamos si el punto que se quiere triangular, p, cumple con dos condiciones: si se puede triangular[1] y si tiene grado par. Si p no se puede triangular, entonces el algoritmo aumenta la triangulación abanico de la que p es vértice común hasta que p pueda ser triangulado. Si p ya se puede triangular pero no tienen grado par, se introducen el número necesario de

[1] p se puede triangular si es vértice común de una triangulación abanico, tal que puede ser convertida en una triangulación rueda, con p como vértice común, solamente añadiendo una arista.

puntos Steiner para que el grado de p sea par. Por último, si p se puede triangular y tiene grado par, entonces se triangula y se continúa con el siguiente punto en la espiral convexa.

El algoritmo lo podemos resumir en los siguientes pasos.

1. Sea p el siguiente vértice de la espiral convexa a triangular y q su vecino en la espiral no triangulado.

2. Revisar si p puede ser triangulado.

   (a) Si p puede ser triangulado, revisar si su grado es par. Si el grado de p es par, entonces triangular p y hacer $p \leftarrow q$ y volver al paso 1.

   (b) Si p no puede ser triangulado, agregar aristas a la espiral convexa tales que formen una triangulación abanico con p como su vértice común hasta que p pueda ser triangulado.

3. Una vez que p puede ser triangulado, revisar si p tiene grado par.

   (a) Si p tiene grado par, entonces triangular p y hacer $p \leftarrow q$ y volver al paso 1.

   (b) Si p tiene grado impar, revisar si la triangulación abanico se puede *extender* un vértice más para que p tenga grado par. Si la triangulación se puede extender entonces se triangula p extendiendo primero la triangulación abanico y luego se hace $p \leftarrow q$ y volvemos al paso 1.

   (c) Si la triangulación abanico no se puede extender, entonces se insertan los puntos Steiner que se necesiten (uno para p y otro posiblemente para q). Se triangula p (y posiblemente q) y se hace $p \leftarrow q$ ($p \leftarrow q'$ donde q' es el vecino de q en la espiral convexa que no ha sido triangulado) volvemos al paso 1.

La rutina que inserta puntos Steiner a P es sencilla, simplemente toma dos puntos y regresa su punto medio.

STEINER(a, b)
1 // Recibe dos puntos en $\mathbb{R}^2$, a, b.
2 // Regresa el punto medio entre a y b.
3 **return** $(\frac{a.x-b.x}{2}, \frac{a.y-b.y}{2})$

TRIANGULACIÓN-ESPIRAL(Tr)

1 // Recibe: La espiral convexa de P en orden inverso, Tr.
2 // Regresa: Tr tal que $V(Tr) = P \cup S$ y Tr es una triangulacion pseudo par de $P \cup S$.
3 $j = 2$
4 $i = 0$
5 **while** $p_i \in Int(CC(P))$
6 **while** DIRECTION$(p_{i+1}, p_i, p_j) > 0$
7 $E(Tr) = E(Tr) \cup \{p_i p_j\}$
8 $j = j + 1$
9 $E(Tr) = E(Tr) \cup \{p_i p_j\}$
10 **if** $\delta(p_i)$ es par
11 $i = i + 1$
12 **continue**
13 **if** $p_{j+1} \notin V(Tr)$
14 // Los nuevos puntos Steiner van en el cierre convexo
15 // *verticeCc* es el ultimo vertice de $CH(P)$ visto por la espiral
16 s_i = STEINER$(p_j, verticeCc)$
17 $S = S \cup \{s_i\}$
18 // Nos aseguramos que $p_{j+1} = s_i$
19 // Revisamos si p_i ve a p_{j+1} en tal caso no se necesita de un pto. Steiner
20 **if** DIRECTION$(p_{i+1}, p_i, p_{j+1}) < 0$
21 $E(Tr) = E(Tr) \cup \{p_i p_{j+1}\}$
22 $i = i + 1$
23 **continue**
24 // Revisamos que p_{i+1} vea a p_{j+1}
25 **if** DIRECTION$(p_{i+1}, p_{i+2}, p_{j+1}) > 0$
26 s_i = STEINER(p_j, p_{i+1})
27 $E(Tr) = E(Tr) \cup \{s_0 p_i, s_0 p_{i+1}, s_0 p_j, s_0 p_{j+1}\}$
28 **else**
29 // En caso contrario vemos cuantos puntos Steiner se necesitan
30 // dependiendo de la paridad de p_{i+1}
31 **if** $\delta(p_{i+1})$ es par
32 s_i = STEINER(p_{i+1}, p_j)
33 s_{i+1} = STEINER(p_{i+2}, p_j)
34 $E(Tr) = E(Tr) \cup \{s_i p_i, s_i p_{i+1}, s_i p_j, s_i s_{i+1}, s_{i+1} p_j, s_{i+1} p_{i+1}, s_{i+1} p_{i+2}\}$
35 **else**
36 s_i = STEINER(p_{i+1}, p_j)
37 $E(Tr) = E(Tr) \cup \{s_i p_i, s_i p_{i+1}, s_i p_j, s_i p_{i+2}\}$
38 $S = S \cup \{s_i\}$
39 $i = i + 1$
40 $j = j - 1$
41 $i = i + 1$
42 // Los triangulos con vertices en el cierre convexo
43 **while** $p_j \in S$
44 // *verticeCC* es el punto de P mas a la izquierda y mas abajo (en caso de empate).
45 $E(Tr) = E(Tr) \cup \{p_j verticeCc\}$

Proposición 2.1 *Al finalizar la primera iteración del* ***while*** *de la línea 5, el grado de p_0 es par. Además p_0 es el vértice común de una triangulación en abanico o el vértice central de una triangulación rueda.*

Demostración

La condición del **while** de la línea 6 es verdadera pues $p_j = p_2$ y p_0, p_1, p_2 son tres vértices consecutivos de la espiral convexa, por lo que los segmentos $\overline{p_1p_0}$ y $\overline{p_0p_2}$ dan vuelta a la derecha.

Sea p_k el vértice en el que se detiene el **while** de la línea 6. TRIANGULACIÓN-ESPIRAL después inserta la arista p_0p_k y revisa la paridad de p_0 en la línea 10.
Caso 1: $\delta(p_0)$ es par

En este caso, p_0 ya tiene grado par y puede ser triangulado en la siguiente iteración del **while** de la línea 5. Notemos que el **while** de la línea 6 introduce las aristas $p_0p_2, \ldots, p_0p_{k-1}$. Como $p_2, \ldots, p_k$ son vértices consecutivos de la espiral convexa, entonces las aristas introducidas forman una triangulación abanico con p_0 su vértice común.

Caso 2: $\delta(p_0)$ es impar

En este caso TRIANGULACIÓN-ESPIRAL primero revisa que S cuente con puntos para poder triangular, después intenta triangular sin uso de puntos Steiner. De no ser esto posible, revisa que p_1 y p_{k+1} se vean y actúa de acuerdo. Todo esto lo demostramos en los siguientes subcasos.

Caso 2.1: p_{k+1} no existe en P

Si p_{k+1} no existe en P entonces $p_k = p_n$ y el punto Steiner que se tiene que insertar se hace en el cierre convexo entre p_n y *verticeCc*. Esto lo hace la línea 16. Los segmentos $\overline{p_1p_0}$ y $\overline{p_0p_{k+1}}$ dan vuelta a la izquierda, por lo que el algoritmo inserta la arista p_0p_{k+1} en la línea 21. p_0 queda con paridad par y como vértice común de una triangulación abanico. Nótese que en la siguiente iteración la condición de la línea 5 evalúa a falso y la triangulación abanico de p_0 se transforma en una triangulación de rueda en el ciclo de la línea 43.

Caso 2.2: p_{k+1} existe en P y p_0 ve a p_{k+1}

Si $p_{k+1} \in V(Tr)$ y p_0 lo ve. Entonces, la condición del **if** de la línea 20 evalúa a verdadero y el procedimiento inserta la arista p_0p_{k+1}. p_0 queda con grado par y como vértice común de una triangulación abanico.

Caso 2.3: p_{k+1} existe en P y p_0 no ve a p_{k+1}

Cuando $p_{k+1} \in S$ y p_0 no lo ve, entonces tenemos dos subcasos. El primero es cuando $p_1 = p_{i+1}$ ve a p_{k+1}; el segundo cuando p_1 no ve a p_{k+1}.

Caso 2.3.1: p_1 ve a p_{k+1}

Si p_1 ve a p_{k+1} entonces el cuadrilátero $C = \square p_0, p_1, p_k, p_{k+1}$ es cóncavo, porque de ser convexo entonces p_0 ve a p_{k+1} contradiciendo la hipótesis. Para resolver este caso TRIANGULACIÓN-ESPIRAL inserta un punto Steiner s_0 y hace s_0 adyacente a todos los vértices de C. Esto lo hace en la líneas 26 y 27. En este caso p_0 queda con paridad par y es el vértice central de una triangulación de rueda.

Caso 2.3.2: p_1 no ve a p_{k+1}

Si p_1 no ve a p_{k+1} es porque $\overline{p_2p_1}$ y $\overline{p_1p_{k+1}}$ da una vuelta a la derecha. Entonces $\overline{p_1p_2}$ y $\overline{p_2p_k}$ da una vuelta a la izquierda. Entonces p_2 ve a p_k.

Caso 2.3.2.1: p_1 tiene grado par

Si p_1 tiene grado par, entonces son necesarios dos puntos Steiner, uno para p_0 y otro para p_1. TRIANGULACIÓN-ESPIRAL hace esto en el **if** de la línea 31, además agrega las aristas $\{s_0p_0, s_0p_1, s_0p_k, s_0s_1, s_1p_k, s_1p_1, s_1p_2\}$ En este caso p_0 y p_1 quedan triangulados de forma par. p_0 es el centro de una triangulación de rueda.

Caso 2.3.2.2: p_1 tiene grado impar

Si p_1 tiene grado impar, entonces el insertar s_0 es suficiente para triangular p_0 y p_1 de forma par. Esto se hace en la líneas 36 y 37. Al igual que el caso anterior p_0 y p_1 quedan triangulados de forma par y p_0 es el centro de una triangulación rueda.

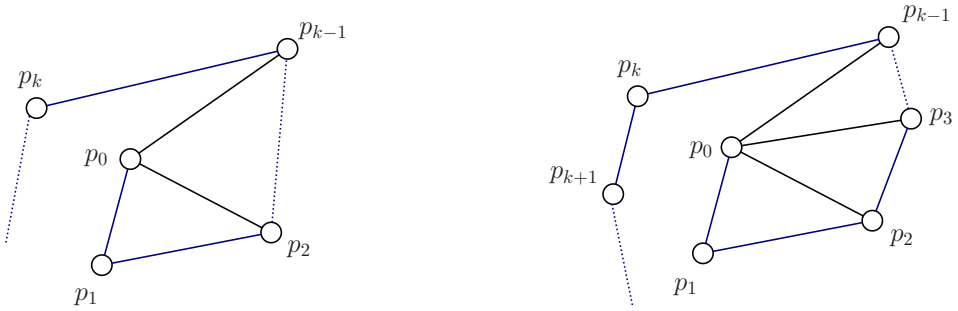
Al finalizar la iteración p_0 tiene grado par y es vértice común de una triangulación en abanico.■

En las figuras 2.4 y 2.5 podemos ver los casos de la demostración de la proposición 2.1.

Proposición 2.2 *Después de la primera iteración del ciclo de la línea 5, el siguiente vértice a triangular es p_1 o p_2. Si el siguiente punto a triangular es p_2 es porque p_1 ya lo está.*

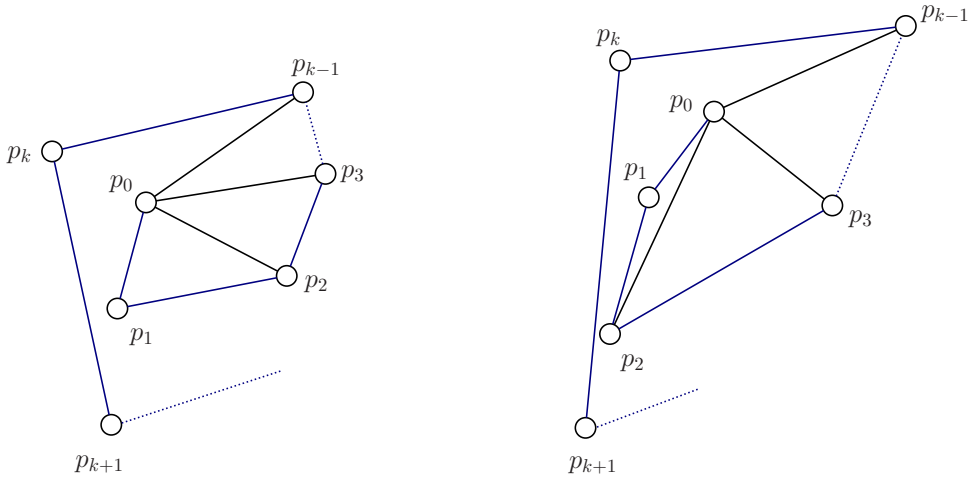
Demostración

Examinando el procedimiento TRIANGULACIÓN-ESPIRAL y el caso 1 de la demostración de la proposición 2.1 podemos observar que lo siguiente que hace TRIANGULACIÓN-ESPIRAL es incrementar la variable i en uno y regresar a la condición booleana del ciclo en la línea 6, entonces, el siguiente vértice a triangular es p_1.



(a) p_0 impar (b) p_0 par

Figura 2.4: Las posibles opciones para triangular p_0 sin necesidad de puntos Steiner.



(a) p_0 par (b) p_0 par

Figura 2.5: Las posibles opciones para triangular p_0 usando uno o dos puntos Steiner.

El siguiente punto en donde se modifica la variable i es cuando la condición del **if** de la línea 20 evalúa a verdadero. En este caso el contador i es incrementado en uno y se regresa al inicio del **while** de la línea 6. Entonces, el siguiente punto a triangular es p_1.

Si la condición booleana del **if** de la línea 25 evalúa a verdadero, entonces, después de ejecutar el código correspondiente el contador i se incrementa en uno y se regresa a la primera condición booleana; p_1 será el siguiente punto a triangular.

Por otro lado, si la condición evalúa a falso entonces sabemos que p_0 y p_1 van a ser triangulados de forma par. El contador i debe incrementarse en uno para que el incremento de la línea 41 haga que i valga 2. Entonces el siguiente punto a triangular es p_2.■

Proposición 2.3 *Si $p_i \neq verticeCC$ en la línea 5 de* Triangulación-Espiral *entonces, al finalizar la iteración, el grado de p_i es par y p_i es el vértice común en una triangulación en abanico*

Demostración

Ya demostramos qué pasa cuando $p_i = p_0$. Ahora nos ocuparemos cuando $i > 0$.

El **while** de la línea 6 hace que el punto p_i pueda ser triangulado, insertando, de ser necesario, las aristas p_ip_j. Sea p_k el vértice en el cual la condición booleana de la línea 6 es falsa.

Saliendo del **while** se inserta la arista p_ip_k. Entonces tenemos dos casos, ya sea que $\delta(p_i)$ sea par o no. Si resulta que $\delta(p_i)$ es par, entonces Triangulación-Espiral incrementa i y continúa con el siguiente punto. Sean $p_l, p_{l+1}, \ldots, p_{k-1}, p_k$ los puntos que se hicieron adyacentes a p_i. Entonces, $p_l, p_{l+1}, \ldots, p_{k-1}, p_k$ son vértices consecutivos en la espiral convexa. Los triángulos $T = \{\triangle p_l p_i p_{l+1}, \ldots, \triangle p_{k-1} p_i p_k\}$ están en posición de abanico, todos compartiendo a p_i.

Si resulta que $\delta(p_i)$ es impar, entonces, Triangulación-Espiral determina si p_{k+1} es visto por p_i; en caso afirmativo el algoritmo introduce la arista p_ip_{k+1}, incrementa i y continúa. Si p_i no ve a p_{k+1} etonces hay dos casos.
Caso 1: p_{i+1} ve a p_{k+1}.

Si p_{i+1} ve a p_{k+1} entonces un sólo punto Steiner, s_i, es necesario para triangular p_i de forma par. Este punto se coloca en el punto medio del segmento $p_{i+1}p_k$. Nótese que el cuadrilátero $C = \square p_i, p_{i+1}, p_k, p_{k+1}$ es cóncavo, pues de ser convexo entonces p_{k+1} ve a p_i.

Triangulación-Espiral hace s_i adyacente a todos los vértices del cuadrilátero C y salta a la línea 41. Los triángulos $T \cup \triangle\{p_i s_i p_k\}$ están en posición

abanico.
Caso 2: p_{i+1} no ve a p_{k+1}.

Si p_{i+1} no ve a p_{k+1} entonces p_{i+2} ve a p_k entoces triangularemos los puntos p_i y p_{i+1} que están en el cuadrilátero $C = \square p_i, p_{i+1}, p_{i+2}, p_k$.
Caso 2.1: $\delta(p_{i+1})$ es par.

Si p_{i+1} es par, entonces necesitamos dos puntos Steiner,s_i y s_{i+1}, para triangular a p_i y p_{i+1}. El algoritmo inserta las aristas $\{s_i p_i, s_i p_{i+1}, s_i p_k, s_i s_{i+1}, s_{i+1} p_k, s_{i+1} p_{i+1}, s_{i+1} p_{i+2}\}$ que hacen que los vértices p_i, p_{i+1}, s_i y s_{i+1} sean pares. Al finalizar salta a la línea 41 para incrementar i en uno.
Caso 2.1: $\delta(p_{i+1})$ es impar.

Si p_{i+1} es impar entonces un sólo punto en C basta para triangular a p_i y p_{i+1} de forma par. Triangulación-Espiral introduce el vértice s_i y lo hace adyacente a todos los vértices del cuadrilátero C. Después salta a la línea 41 para incrementar el contador i en uno.

Ya sea el caso 2.1 o 2.2, los tríangulos de $T \cup \{\triangle p_i s_i p_k\}$ forman una triangulación abanico con p_i su vértice común.

Por lo tanto al finalizar la iteración p_i es vértice común de una triangulación en abanico o está triangulado de forma par.■

Proposición 2.4 *Si $p_i \neq verticeCC$ en la línea 5 de* Triangulación-Espiral. *Entonces, el siguiente vértice a triangular es p_{i+1} o p_{i+2}. Si es p_{i+2} es porque p_{i+1} ya está triangulado*

Demostración

Para demostrar esta proposición sólo hace falta revisar las líneas 11, 22, 39 y 41 donde se modifica la variable i.

Los incrementos de las líneas 11 y 22 suceden cuando p_i ya puede ser triangulado y cuando p_i puede ser triangulado sin uso de puntos Steiner respectivamente. Por lo tanto el siguiente punto a triangular es p_{i+1}.

Para que Triangulación-Espiral llegue a la línea 39 es porque p_i fue impar y p_{i+1} no veía a p_{j+1}. Entonces, p_i y p_{i+1} son triangulados, por lo que el índice al terminar la iteración necesita valer $i+2$. La línea 39 hace que i valga $i+1$. Siguiendo el flujo, el procedimiento llega a la línea 41 lo que hace que i valga de nuevo $i+1$ logrando que al final de la iteración el valor de i se incremente en dos.

Por otro lado, también podemos llegar a la línea 41 si p_{i+1} si ve a p_{j+1}. En este caso sólo p_i es triangulado por lo que al final de la iteración i valdrá $i+1$.

Por lo tanto, El siguiente vértice a triangular es p_{i+1} o p_{i+2}.■

Proposición 2.5 *Sea $p_i = p_k$, $k \neq 0$ en la línea 5 de* Triangulación-Espiral. *Si p_{k-1} no está triangulado, entonces,* Triangulación-Espiral *lo triangula.*

Demostración

Por inducción.

B.I. $k = 1$

Por la proposición 2.1 sabemos que p_0 es vértice común de una triangulación abanico. Si los segmentos $\overline{p_2p_1}$ y $\overline{p_1p_j}$ dan una vuelta a la derecha, entonces, p_j no ve a p_2 y el **while** de la línea 6 aumenta a Tr en la arista p_1p_j. Entonces, p_0 es el vértice central de los triángulos $\triangle p_1p_0p_2, \triangle p_2p_0p_3, \ldots, \triangle p_1p_0p_j$.

Por lo tanto p_0 está triangulado.

H.I.

Supongamos que $p_i = p_l$ entonces Triangulación-Espiral triangula a p_{l-1} si no está ya triangulado.

P.I. **P.D.** Si $p_i = p_{l+1}$ entonces Triangulación-Espiral triangula a p_l si no está ya triangulado.

Si $\overline{p_{l+2}p_{l+1}}$ y $\overline{p_{l+1}p_m}$ dan una vuelta a la derecha entonces el **while** de la línea 6 es verdadero y Triangulación-Espiral inserta la arista $p_{l+1}p_j$.

Por la hipótesis de inducción sabemos que cualquier arista que conecta a p_l con cualquier vértice de índice menor viola la planaridad de Tr. Por la proposición 2.3 sabemos que p_l es vértice común de una triangulación abanico. Entonces, los únicos vértices, p_h, posibles a los que p_l puede ser adyacente cumplen que los segmentos $\overline{p_lp_{l+1}}$, $\overline{p_{l+1}p_h}$ dan una vuelta a la derecha.

Por lo tanto al añadir la arista $p_{l+1}p_j$ el vértice p_l queda triangulado.■

Proposición 2.6 *Sea $p_i = verticeCC$ en el **while** de la línea 43.* Triangulación-Espiral *triangula a p_{i-1} y forma triángulos con vértices en el cierre convexo de P.*

Demostración

Caso 1: p_{i-1} necesitó un punto Steiner

Si se introdujo un punto Steiner s_{i-1} para triangular a p_{i-1}, entonces, s_{i-1} está en el cierre convexo de S y Triangulación-Espiral sólo añade la arista $verticeCCs_{i-1}$, triangulando a p_{i-1}.

Caso 2: En Triangulación-Espiral, $j \leq n$

Sea $p_{i-1}p_k$ la última arista introducida por Triangulación-Espiral para p_{i-1}. Entonces, $k \leq n$ y Tr es aumentada en las aristas $p_jVerticeCC$ para $k = j \leq n$, todas uniendo vértices del cierre convexo. En particular la arista $p_{i-1}p_k$ triangula a p_{i-1}.

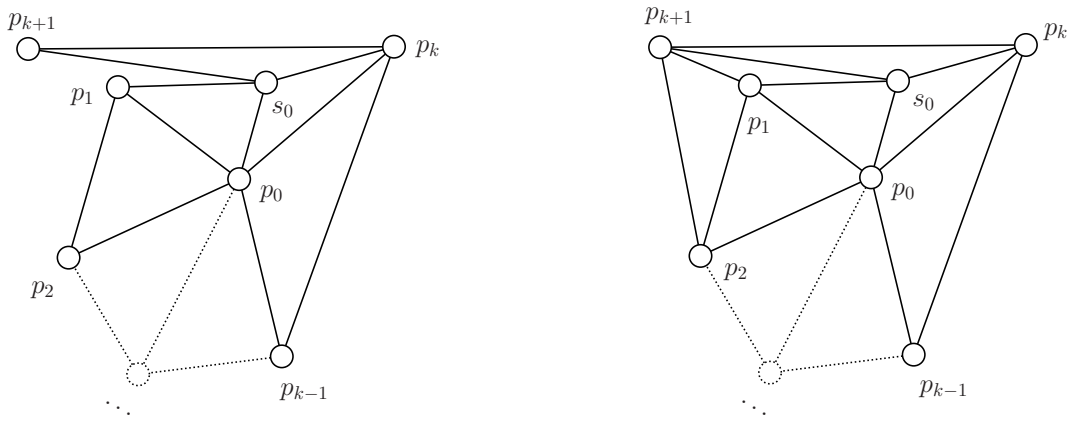
Por lo tanto la proposición se cumple. ■

Hemos demostrado la siguiente proposición.

Proposición 2.7 Triangulación-Espiral *construye una triangulación pseudo par de* $S \cup S$ *partiendo de su espiral convexa.*

2.4 Cardinalidad de S

Supongamos que queremos construir un conjunto de puntos P tal que al ejecutar Triangulación-Espiral S sea de cardinalidad máxima.



(a) p_0 impar (b) p_1 queda par

Figura 2.6: Caso en donde p_0 necesita un punto Steiner.

Para que Triangulación-Espiral inserte un punto s_0, p_0 tiene que estar distribuido como en la Figura 2.6a. Sin embargo podemos ver que p_1 queda triangulado de forma par y p_2 se hace adyacente a p_{k+1} (ver Figura 2.6b).

Partiendo de la configuración de la Figura 2.6b podemos construir S tal que los vértices $p_2, p_3, ..., p_{k-1}$ necesiten de un punto Steiner cada uno. Notemos que los vértices p_i, $2 < i < k$ tienen grado tres.

Sean p_{k+2}, p_{k+3} tales que p_2 vea a p_{k+2} pero no a p_{k+3}. Entonces p_2 necesitará un punto Steiner y nótese que p_3 tendrá grado impar. Siguiendo este proceso, encontramos puntos p_{k+j}, $1 < j < k$ tales que Triangulación-Espiral inserte puntos Steiner para triangular a los vértices $p_2, p_3, \ldots, p_{k-1}$(ver Figura 2.7). Nótese que p_{2k-1} no ve a p_{2k} y p_k quedará con grado par. Notemos que la misma construcción puede ser aplicada para los vértices $p_{k+1}, p_{k+2}, \ldots, p_{2k-1}$.

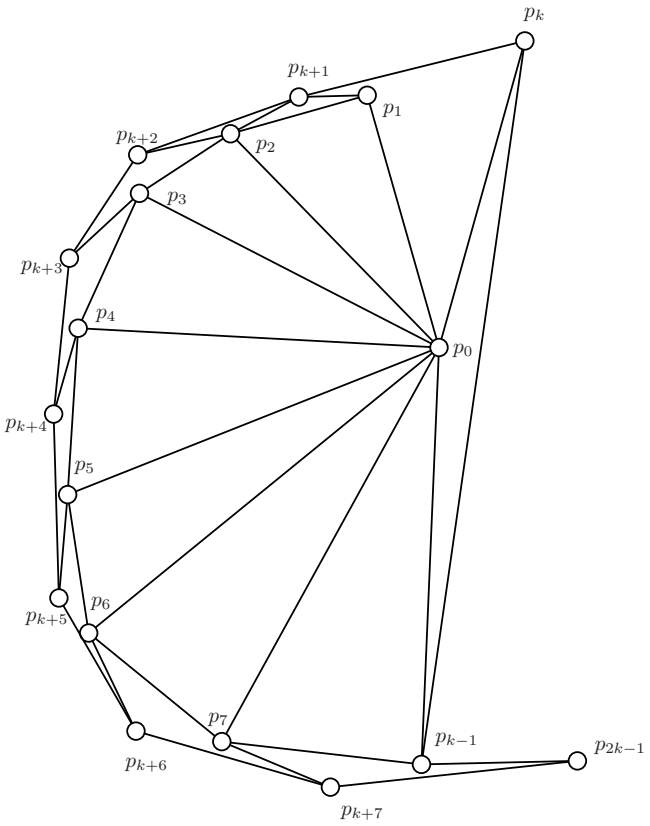


Figura 2.7: Configuración de P para que $p_2, \ldots, p_{k-1}$ necesiten puntos Steiner.

Sea $p_i \in S$, como sólo nos interesa la paridad de p_i, entonces a lo más se necesita un punto Steiner para hacer que p_i tenga la paridad deseada. Entonces, con la construcción anterior hemos demostrado la siguiente proposición.

Proposición 2.8 *Al finalizar la ejecución de* TRIANGULACIÓN-ESPIRAL *para una espiral convexa,* $|S| < n$.

2.5 La complejidad de TRIANGULACIÓN-ESPIRAL

En [Tou85] Toussaint analiza la complejidad del algoritmo SRC que construye una triangulación serpentina de un conjunto de puntos en posición general.

El primer paso de SRC es calcular la espiral convexa, para esto se usan un par de algoritmos bien conocidos. El primer algoritmo calcula las capas convexas y el segundo construye la espiral convexa a partir de éstas.

Para calcular las capas convexas de un conjunto de puntos de forma óptima, se puede usar el algoritmo de Chazelle expuesto en [Cha85], el cual lo hace en $\mathrm{O}(n \log n)$. Para calcular la espiral convexa a partir de las capas convexas optimamente se puede emplear el algoritmo de Preparata y Shamos expuesto en [PS85] que lo hace en $\mathrm{O}(n)$; en ambos casos n es la cardinalidad del conjunto de puntos en el plano.

De lo anterior se concluye que la espiral convexa de un conjunto de n puntos se puede calcular en $\mathrm{O}(n \log n)$.

Cada punto interior de TRIANGULACIÓN-ESPIRAL es visitado a lo más dos veces, una para triangular un punto de su capa convexa inmediata anterior, y otra cuando es triangulado de forma par. Entonces TRIANGULACIÓN-ESPIRAL tiene complejidad $\mathrm{O}(n)$. Por lo tanto la complejidad de construir $P \cup S$ es $\mathrm{O}(n \log n)$.

2.6 Algunos resultados experimentales

Aunque en la sección 2.4 demostramos que es posible construir un ejemplo para el cual TRIANGULACIÓN-ESPIRAL inserta n puntos nuevos una pregunta interesante es ¿cuántos puntos se insertan en promedio?

Para tratar de dar una idea de la respuesta, se armó un experimento similar al reportado por Toussaint y Bose en [TB97]. Se consideraron 11 cardinalidades distintas, $50, 100, 200, 300, \ldots, 1000$; por cada cardinalidad de generaron 10 conjuntos de puntos distribuidos uniformemente en el círculo unitario. La primera columna del

cuadro 2.1 es la cardinalidad de los conjuntos de puntos, la segunda columna es la media del conjunto S y la tercera la proporción de la media de la cardinalidad de S y la cardinalidad de P.

$\lvert P\rvert$	$\overline{\lvert S\rvert}$	$\frac{\overline{\lvert S\rvert}}{\lvert P\rvert}$
50	5.11	0.1022
100	13.8	0.138
200	24.8	0.124
300	37.3	0.1243
400	56.8	0.142
500	64.5	0.129
600	81.7	0.1361
700	96.1	0.1372
800	107.5	0.1343
900	116.0	0.128
1000	136.6	0.1366

Cuadro 2.1: Relación entre $|P|$ y $|S|$.

2.7 Conclusiones y trabajo futuro

Los resultados experimentales, sugieren que el número necesario de puntos Steiner no es tan grande. Ejemplos de entre 50 y 70 puntos hechos *a mano*, es decir sin la ayuda de algoritmo alguno, sugieren que la cota es en realidad muy menor a la que los resultados experimentales sugieren. Sin embargo, la forma en la que estos ejemplos se triangularon no dejaron entrever un algoritmo. Dos ventajas pueden ser mencionadas del algoritmo que en este trabajo se expone. La primera es la rapidez con la que construye $P \cup S$, la segunda, que en el caso promedio, la cota superior para S parece ser difícil que pueda alcanzarse.

La razón principal porque se decidió usar la espiral convexa es porque esta da cierto *orden* al conjunto de puntos. Sin embargo el precio de este orden es introducir n aristas a la triangulación. Sin embargo de todas las aristas con las que comenzamos la heurística, las del cierre convexo y las de la espiral convexa, solamente estamos seguros que las del cierre convexo siempre van a estar en toda triangulación pseudo par del conjunto de puntos ya sea que este admita una o sea necesario agregar puntos Steiner. Al momento de la escritura de este trabajo, se pensaron un par de algoritmos para mejorar la cota expuesta aquí. Ambos algoritmos se basan en triangular el conjunto de puntos asignando colores a los puntos que se quieren triangular en lugar

de fijarse en su paridad. El primer algoritmo se basa en la técnica de barrido de línea y esencialmente triangula de izquierda a derecha insertando puntos Steiner cuando un vértice ya no puede ser adyacente a ningún otro vértice del conjunto. Una debilidad que se observó en este algoritmo es que en la mayoría de los casos cuando un vértice necesita un punto Steiner fue porque en pasos anteriores se le restó visibilidad y este hecho no era tomado en cuenta.

El segundo algoritmo intenta subsanar la deficiencia del primer algoritmo mediante la idea de *compromiso*. Esta idea se basa en que dado un conjunto de aristas E que forman parte de una triangulación del conjunto de puntos, se agregan a la gráfica todas las aristas que están presentes en cualquier triangulación del conjunto de puntos que su conjunto de aristas tiene como subconjunto a E. La idea subyacente es crecer el conjunto E mediante un barrido de línea de tal manera que al terminar quede una triangulación con un *hoyo* al cual se le agreguen pocos puntos Steiner.

Para finalizar, planteamos la siguiente conjetura: Para cualquier conjunto de puntos P en posición general en el plano, son suficientes un número constante de puntos Steiner para que $P \cup S$ acepte una triangulación 3 coloreable. Conjeturamos que son suficientes dos puntos Steiner en el interior del cierre convexo de P o uno en éste.

Apéndice A

Código para generar $P \cup S$

A continuación se presenta una implementación del algoritmo Triangulacion-Espiral en Objective-C. El programa es parte de un conjunto experimental de programas para geometría computacional llamado CGR (Computational Geometry's Rover) desarrollado por el autor. Al finalizar se muestran tres ejemplos de conjuntos de puntos distribuidos uniformemente en el disco unitario, para los cuales se construyó $P \cup S$. También se presenta un código que para cualquier conjunto de puntos y una triangulación base, se calculan todas sus triangulaciones y reporta la que maximiza el número de vértices pares. Un problema interesante es: para un $k \in \mathbb{N}$ dado, encontrar la configuración de puntos P tal que P acepta una triangulación que maximiza (minimiza) el número de vértices con grado par.

En la versión electrónica de este trabajo, se anexa el código fuente completo de CGR, y unos cuantos *plug-ins*.

```
//
//  EvenTriangulation.m
//  EvenTriangulationCSFinal
//
//

#import "EvenTriangulationInterface.h"
#import "CGRAlgorithmFramework/AlgorithmModificationsHolder.h"
#import "CGRAlgorithmFramework/CGRNotification.h"
#import "CGRAlgorithmFramework/CGREdge.h"
#import "CGRAlgorithmFramework/CGRPoint.h"

@implementation EvenTriangulationInterface
@synthesize helpWindow;
@synthesize typeOfAlgorithm;
@synthesize modifications;
@synthesize simulationNumbering;


-(id)init
{
    self = [super init];
    helpWindow = [[NSPanel alloc] init];
    modifications = [[AlgorithmModificationsHolder alloc]init];
    typeOfAlgorithm = @"Structural";
    simulationNumbering = @"PointsIncidentOnEdge";
    //We look for the Convex Hull Bundle
    NSString *convexSpiralPath = [NSHomeDirectory()
        stringByAppendingPathComponent:
                                    @"Library/Application Support/CGR/PlugIns
                                        /Algorithms/ConvexSpiralFinal.bundle"
                                        ];
    NSBundle *convexSpiralBundle = [NSBundle bundleWithPath:
        convexSpiralPath];
    Class BundleInterface = [convexSpiralBundle principalClass];
    convexSpiral = [[[BundleInterface alloc]init] retain];
    return self;
}
-(BOOL)isUpdatable
{
    return NO;
}
-(void)setUp:(NSArray *)points edges:(NSArray *)edges selectedPoints:
    (NSArray *)selectedPoints selectedEdges:(NSArray *)selectedEdges
{
    //We setup the modifications originals...
    if([selectedPoints count] != 0){
        [modifications setOriginalPoints:selectedPoints];
    }else{
        [modifications setOriginalPoints:points];

    }
    if([selectedEdges count] != 0){
```

```
            [modifications setOriginalEdges:selectedEdges];
        }else{
            [modifications setOriginalEdges:edges];
        }
    }
    -(double) crossProduct:(NSPoint)p0 p1:(NSPoint)p1{
        return p0.x * p1.y - p1.x*p0.y;
    }
    -(unsigned short int) leftRightTurn:(CGRPoint *)p0 p1:(CGRPoint *)p1 p2:
        (CGRPoint *)p2
    {
        NSPoint p2p0,p1p0;
        p2p0.x = [p2 origin].x - [p0 origin].x;
        p2p0.y = [p2 origin].y - [p0 origin].y;

        p1p0.x = [p1 origin].x - [p0 origin].x;
        p1p0.y = [p1 origin].y - [p0 origin].y;

        double turn = [self crossProduct:p2p0 p1:p1p0];
        if( turn < 0 )
            return 0;
        else if( turn > 0)
            return 1;
        else
            return 2;

    }
    //Sorts the points following the spiral from the inside out
    -(void) sortPoints
    {
        if([[modifications newEdges] count] == 0)
            return;
        NSMutableArray *sortedPoints = [NSMutableArray arrayWithCapacity:
            [[modifications originalPoints]count]];
        for(unsigned int i = [[modifications newEdges] count] - 1; ; i--){
            CGREdge *edge = [[modifications newEdges] objectAtIndex:i];
            if(i == 0){
                [sortedPoints addObject:[edge point2]];
                [sortedPoints addObject:[edge point1]];
                break;
            }
            [sortedPoints addObject:[edge point2]];
        }
        [modifications setOriginalPoints:sortedPoints];
    }
    //Finds the last convex hull point in the spiral
    -(unsigned int) lastCHPoint
    {
        CGRPoint *p0 = [[modifications originalPoints]lastObject];
        for(unsigned int i = [[modifications originalPoints] count] - 2; ; i--)
            {
            if(i-1 > i) //We are dealing with unsigned ints
                return -1;
```

```
            CGRPoint *p1 = [[modifications originalPoints] objectAtIndex:i];
            CGRPoint *p2 = [[modifications originalPoints] objectAtIndex:i-1];
            if([self leftRightTurn:p1 p1:p2 p2:p0] == 0)
                return i;

        }
        return -1;
    }
    //Returns the first point index that can triangulate "pointInd" (not
        necessary evenly)
    //starting from lastPoint
    -(unsigned int) firstPoint: (unsigned int) pointInd lastPoint:(unsigned int
        ) lastPoint
    {
        CGRPoint *p0 = [[modifications originalPoints] objectAtIndex:pointInd];
        CGRPoint *p1 = [[modifications originalPoints] objectAtIndex:pointInd +
            1];
        unsigned int i = lastPoint;
        for(; i < [[modifications originalPoints] count]; i++){
            CGRPoint *p2 = [[modifications originalPoints] objectAtIndex:
                lastPoint++];
            if([self leftRightTurn:p0 p1:p2 p2:p1] == 0)
                break;
        }
        return i;
    }
    //Triangulates the point with index pointInd evenly if possible, returning
        a reference
    //to the last point used if sucessful. Returns nil otherwise.
    -(CGRPoint *) canBeEvenlyTriangulated:(unsigned int) pointInd lastPoint:
        (unsigned int) lastPoint
    {
        CGRPoint *point = [[modifications originalPoints] objectAtIndex:
            pointInd];
        unsigned int firstPoint = [self firstPoint:pointInd lastPoint:lastPoint
            ];
        //new degree of point once we do the adyacencies
        unsigned int newDegree = [point degree] + (firstPoint - lastPoint) + 1;
        for(unsigned int i = lastPoint; i <= firstPoint; i++){
            CGRPoint *q = [[modifications originalPoints]
                           objectAtIndex:i];
            CGREdge *e = [[CGREdge alloc]initWithPoints:point
                                              andPoint:q];
            if(e == nil){
                e = [point giveEdgeForNeighbor:q];
            }
        }
        //If we could do it
        if(newDegree % 2 == 0)
            return [[modifications originalPoints] objectAtIndex:firstPoint];

        return nil;
```

```
}

-(AlgorithmModificationsHolder *)executeAlgorithm
{
    [convexSpiral setUp:[modifications originalPoints] edges:nil
        selectedPoints:nil selectedEdges:nil];
    AlgorithmModificationsHolder *spiral = [convexSpiral executeAlgorithm];
    NSMutableArray *newEdges = [NSMutableArray arrayWithArray:[spiral
        newEdges]];
    NSMutableArray *newPoints = [NSMutableArray arrayWithCapacity:
        [[modifications originalPoints]count]/10];
    [modifications setNewEdges:[spiral newEdges]];
    [self sortPoints];
    unsigned int lastCHPoint = [self lastCHPoint];
    if(lastCHPoint == -1){
        CGRPoint *p = [[modifications originalPoints] objectAtIndex:0];
        for(unsigned int i=2; i < [[modifications originalPoints] count]; i
            ++){
            CGRPoint *q = [[modifications originalPoints] objectAtIndex:i];
            CGREdge *e = [[CGREdge alloc]initWithPoints:p andPoint:q];
            if(e == nil){
                e = [p giveEdgeForNeighbor:q];
            }
            [newEdges addObject:e];
        }
        [modifications setNewEdges:newEdges];
        return modifications;
    }
    //Just to make things simpler...
    NSArray *points = [modifications originalPoints];
    //The index of the point being triangulated
    unsigned int curntIndex = 0;
    //The index of the last point seen
    unsigned int lastSeenIndex = 2;
    //The point being triangulated
    CGRPoint *curntPoint = [points objectAtIndex:curntIndex];
    //The last seen point
    CGRPoint *lastPointSeen = [points objectAtIndex:lastSeenIndex];
    //While we are seeing interior points
    while(curntIndex < lastCHPoint){
        unsigned int firstPointIndex = [self firstPoint:curntIndex
            lastPoint:lastSeenIndex];
        for(; lastSeenIndex <= firstPointIndex; lastSeenIndex++){
            lastPointSeen = [[modifications originalPoints]
                            objectAtIndex:lastSeenIndex];
            CGREdge *e = [[CGREdge alloc]initWithPoints:curntPoint
                                                andPoint:lastPointSeen];
            if(e == nil){
                e = [curntPoint giveEdgeForNeighbor:lastPointSeen];
            }
            [newEdges addObject:e];
        }
        unsigned int newDegree = [curntPoint degree];
```

```
lastSeenIndex--;
//If we could do it
if(newDegree % 2 == 0){
    curntPoint = [points objectAtIndex:++curntIndex];
    continue;
}
//If the parity of curntPoint is odd we check if it sees another
    point in the CS
if(++lastSeenIndex < [points count]){
    lastPointSeen = [points objectAtIndex:lastSeenIndex];
    CGRPoint *nextCurntPoint = [points objectAtIndex:curntIndex + 1
        ];
    if([self leftRightTurn:curntPoint p1:lastPointSeen p2:
        nextCurntPoint] == 0){
        //if curntPoint sees the next point in the cs we just make
            the adjecency
        CGREdge *ne = [[CGREdge alloc] initWithPoints:curntPoint
            andPoint:lastPointSeen];
        if(ne == nil)
            ne = [curntPoint giveEdgeForNeighbor:lastPointSeen];
        [newEdges addObject:ne];
        curntPoint = [points objectAtIndex:++curntIndex];
        continue;
    }
    //if we weren't so lucky...
    //Check that the point "curntPoint + 1" sees "lastPointSeen +
        1"
    CGRPoint *next2CurntPoint = [points objectAtIndex:curntIndex +
        2];
    if(lastSeenIndex + 1 < [points count]){
        CGRPoint *nextLastPointSeen = [points objectAtIndex:
            lastSeenIndex + 1];
        if([self leftRightTurn:nextCurntPoint p1:nextLastPointSeen
            p2:next2CurntPoint] == 0){
            CGRPoint *tmpLastPointSeen = [points objectAtIndex:
                lastSeenIndex - 1];
            NSPoint middle;
            middle.x = [tmpLastPointSeen origin].x -
                        ([tmpLastPointSeen origin].x -
                            [nextCurntPoint origin].x)/2;
            middle.y = [tmpLastPointSeen origin].y -
                        ([tmpLastPointSeen origin].y -
                            [nextCurntPoint origin].y)/2;
            NSLog(@"STEINER");
            CGRPoint *middlePoint = [[CGRPoint alloc]
                initWithCorePoint:middle];
            [newPoints addObject:middlePoint];
            //We make all the adyacencies
            CGREdge *e1 = [[CGREdge alloc]initWithPoints:curntPoint
                andPoint:middlePoint];
            CGREdge *e2 = [[CGREdge alloc]initWithPoints:
                nextCurntPoint andPoint:middlePoint];
            CGREdge *e3 = [[CGREdge alloc]initWithPoints:
```

```
                tmpLastPointSeen andPoint:middlePoint];
            CGREdge *e4 = [[CGREdge alloc]initWithPoints:
                lastPointSeen andPoint:middlePoint];
            [newEdges addObject:e1];
            [newEdges addObject:e2];
            [newEdges addObject:e3];
            [newEdges addObject:e4];
            curntPoint = [points objectAtIndex:++curntIndex];
            [tmpLastPointSeen release];
            [nextLastPointSeen release];
            continue;
        }else{
            //If curntPoint + 1 doesn't see "lastPointSeen + 1"
            //We check "curntPoint + 1"'s degree
            if([nextCurntPoint degree] % 2 == 0){
                //its even
                CGRPoint *tmpLastPointSeen = [points objectAtIndex:
                    lastSeenIndex - 1];
                NSPoint middle1;
                middle1.x = [tmpLastPointSeen origin].x -
                ([tmpLastPointSeen origin].x - [nextCurntPoint
                    origin].x)/2;
                middle1.y = [tmpLastPointSeen origin].y -
                ([tmpLastPointSeen origin].y - [nextCurntPoint
                    origin].y)/2;
                NSLog(@"STEINER");
                CGRPoint *s1 = [[CGRPoint alloc] initWithCorePoint:
                    middle1];

                NSPoint middle2;
                middle2.x = [lastPointSeen origin].x -
                ([lastPointSeen origin].x - [nextCurntPoint origin]
                    .x)/2;
                middle2.y = [nextLastPointSeen origin].y -
                ([lastPointSeen origin].y - [nextCurntPoint origin]
                    .y)/2;
                NSLog(@"STEINER");
                CGRPoint *s2 = [[CGRPoint alloc] initWithCorePoint:
                    middle2];

                //We make the adyacencies..
                CGREdge *e = [[CGREdge alloc]initWithPoints:s1
                    andPoint:s2];
                CGREdge *e1 = [[CGREdge alloc]initWithPoints:
                    curntPoint andPoint:s1];
                CGREdge *e2 = [[CGREdge alloc]initWithPoints:
                    nextCurntPoint andPoint:s1];
                CGREdge *e3 = [[CGREdge alloc]initWithPoints:
                    tmpLastPointSeen andPoint:s1];

                CGREdge *e11 = [[CGREdge alloc]initWithPoints:
                    nextCurntPoint andPoint:s2];
                CGREdge *e12 = [[CGREdge alloc]initWithPoints:
```

```
                    next2CurntPoint andPoint:s2];
                CGREdge *e13 = [[CGREdge alloc]initWithPoints:
                    lastPointSeen andPoint:s2];

                [newEdges addObject:e];
                [newEdges addObject:e1];
                [newEdges addObject:e2];
                [newEdges addObject:e3];
                [newEdges addObject:e11];
                [newEdges addObject:e12];
                [newEdges addObject:e13];
                [newPoints addObject:s1];
                [newPoints addObject:s2];
                curntIndex++;
                curntPoint = [points objectAtIndex:++curntIndex];
                continue;
                NSLog(@"OOOOPSSS");


            }else{
                //its odd
                CGRPoint *next2CurntPoint = [points objectAtIndex:
                    curntIndex + 2];
                NSPoint middle;
                middle.x = [lastPointSeen origin].x -
                ([lastPointSeen origin].x - [nextCurntPoint origin]
                    .x)/2;
                middle.y = [lastPointSeen origin].y -
                ([lastPointSeen origin].y - [nextCurntPoint origin]
                    .y)/2;
                NSLog(@"STEINER");
                CGRPoint *middlePoint = [[CGRPoint alloc]
                    initWithCorePoint:middle];
                CGREdge *e1 = [[CGREdge alloc]initWithPoints:
                    curntPoint andPoint:middlePoint];
                CGREdge *e2 = [[CGREdge alloc]initWithPoints:
                    nextCurntPoint andPoint:middlePoint];
                CGREdge *e3 = [[CGREdge alloc]initWithPoints:
                    lastPointSeen andPoint:middlePoint];
                CGREdge *e4 = [[CGREdge alloc]initWithPoints:
                    next2CurntPoint andPoint:middlePoint];
                [newEdges addObject:e1];
                [newEdges addObject:e2];
                [newEdges addObject:e3];
                [newEdges addObject:e4];
                curntIndex++;
                curntPoint = [points objectAtIndex:++curntIndex];


            }

        }
    }else{
```

```
            }
        }else{
            //We need to insert a Steiner point in the convex hull..
            lastSeenIndex--;
            lastPointSeen = [points objectAtIndex:lastSeenIndex];
            NSPoint middle;
            middle.x = [lastPointSeen origin].x -
            ([lastPointSeen origin].x - [[points objectAtIndex:lastCHPoint]
                origin].x)/2;
            middle.y = [lastPointSeen origin].y -
            ([lastPointSeen origin].y - [[points objectAtIndex:lastCHPoint]
                origin].y)/2;
            NSLog(@"STEINER");
            CGRPoint *middlePoint = [[CGRPoint alloc] initWithCorePoint:
                middle];
            [newPoints addObject:middlePoint];
            //We make all the adyacencies
            CGREdge *e1 = [[CGREdge alloc]initWithPoints:curntPoint
                andPoint:middlePoint];
            CGREdge *e2 = [[CGREdge alloc]initWithPoints:[points
                objectAtIndex:lastCHPoint] andPoint:middlePoint];
            CGREdge *e3 = [[CGREdge alloc]initWithPoints:lastPointSeen
                andPoint:middlePoint];
            [newEdges addObject:e1];
            [newEdges addObject:e2];
            [newEdges addObject:e3];
            curntPoint = [points objectAtIndex:++curntIndex];
            lastPointSeen = nil;
            continue;
        }

    }

    CGRPoint *lchp = [points objectAtIndex:lastCHPoint];
    while(lastPointSeen){
        CGREdge *e = [[CGREdge alloc]initWithPoints:lchp andPoint:
            lastPointSeen];
        if(e == nil){
            e = [lchp giveEdgeForNeighbor:lastPointSeen];
        }
        [newEdges addObject:e];
        if(lastPointSeen == [points lastObject])
            break;
        lastPointSeen = [points objectAtIndex:++lastSeenIndex];
    }
    [modifications setNewEdges:newEdges];
    [modifications setNewPoints:newPoints];
    return modifications;
}

-(NSString *)menuItemName
{
```

```
    return @"EvenTriangulation";
}
-(void) handlePointDidMoveNotification:(NSNotification *) note
{
    //If the algorithm has not been executed then we ignore the
        notification
    return;

}
-(void) handlePointWasAddedNotification:(NSNotification *) note
{
    //If the algorithm has not been executed then we ignore the
        notification
    return;

}
-(void) handlePointWasDeletedNotification:(NSNotification *) note
{
    //If the algorithm has not been executed then we ignore the
        notification
    return;

}



@end
```

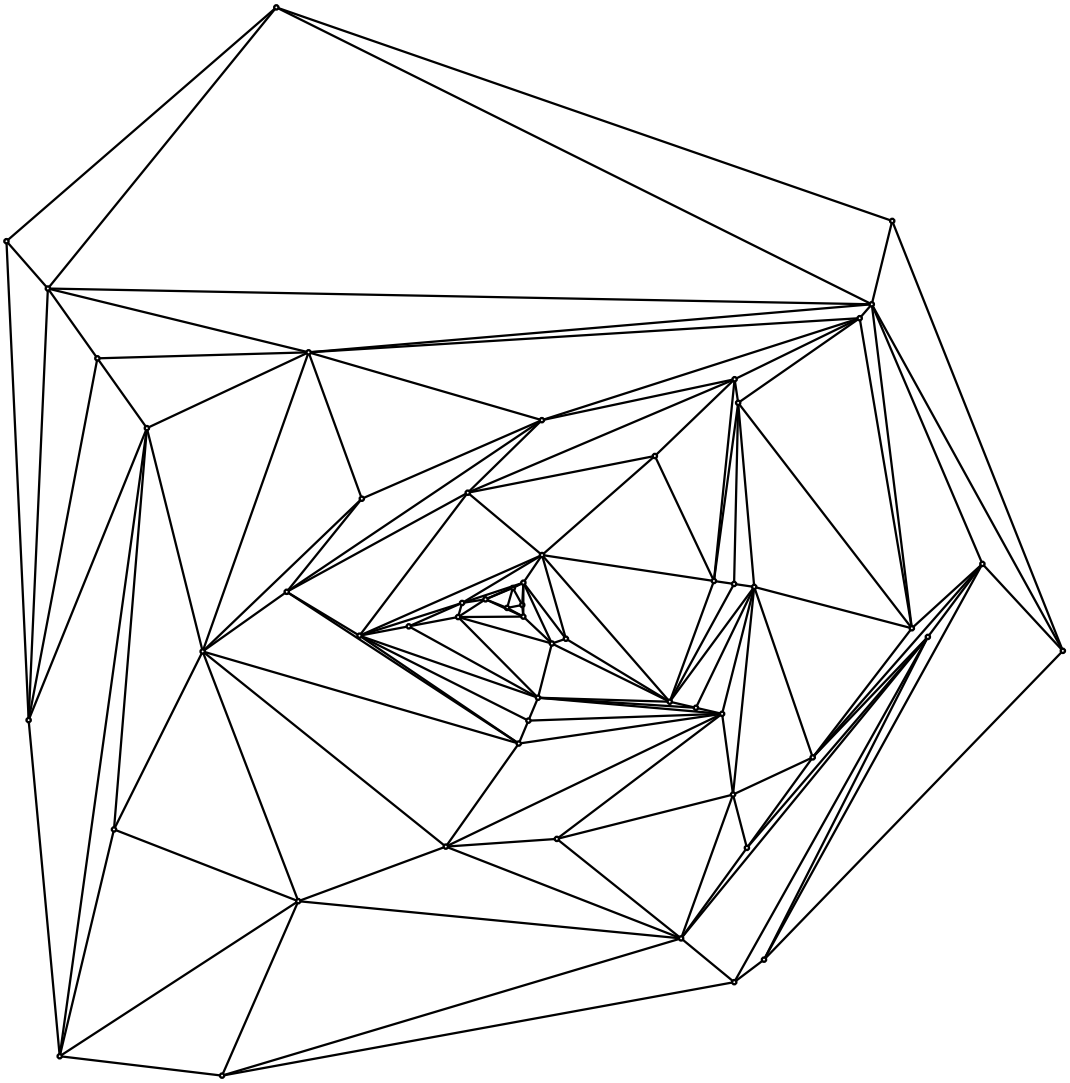


Figura A.1: Configuración de 50 puntos distribuidos uniformemente en el círculo unitario. Triangulacion-Espiral insertó 6 puntos steiner.

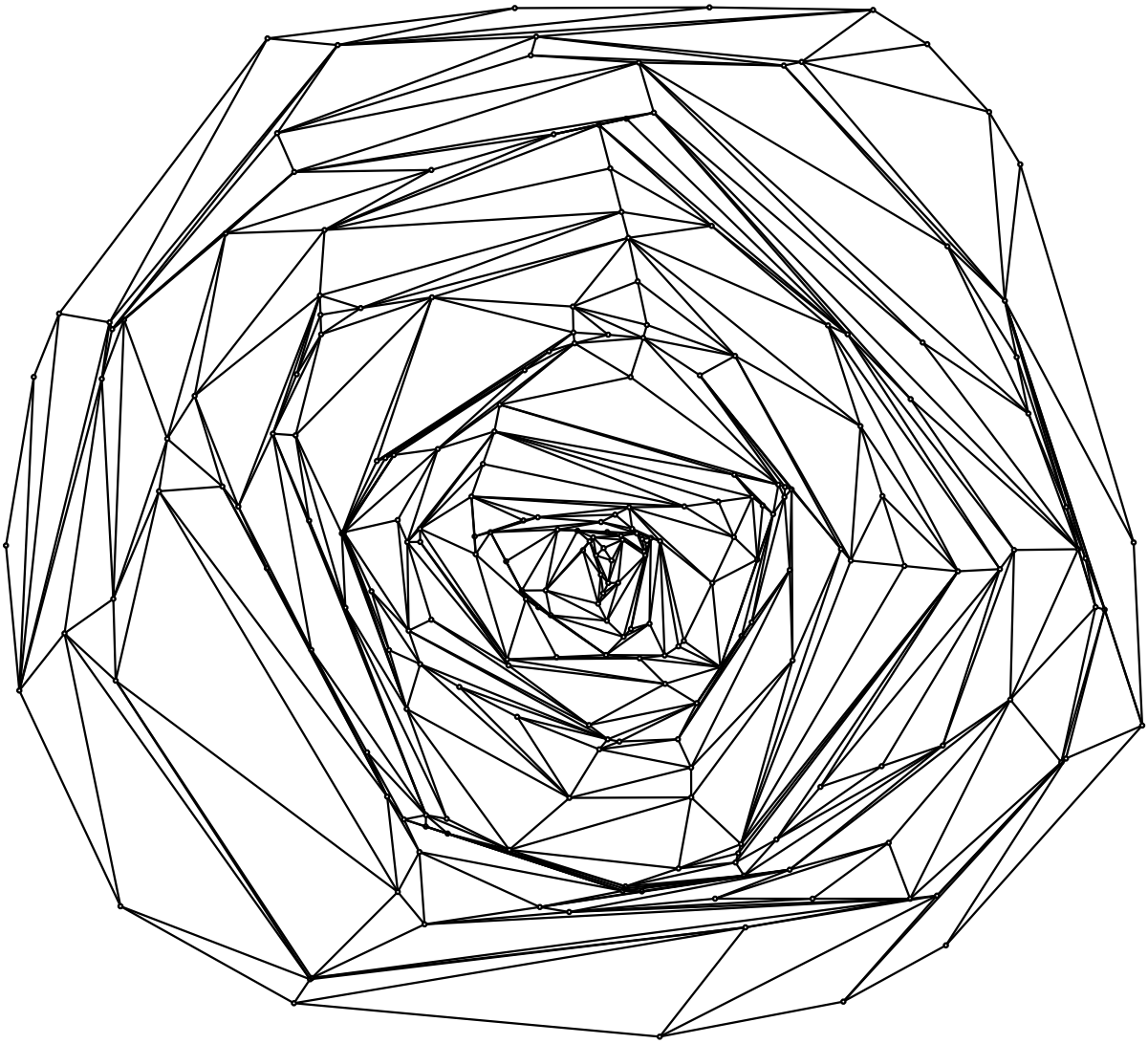


Figura A.2: Configuración de 200 puntos distribuidos uniformemente en el círculo unitario. TRIANGULACION-ESPIRAL insertó 33 puntos steiner.

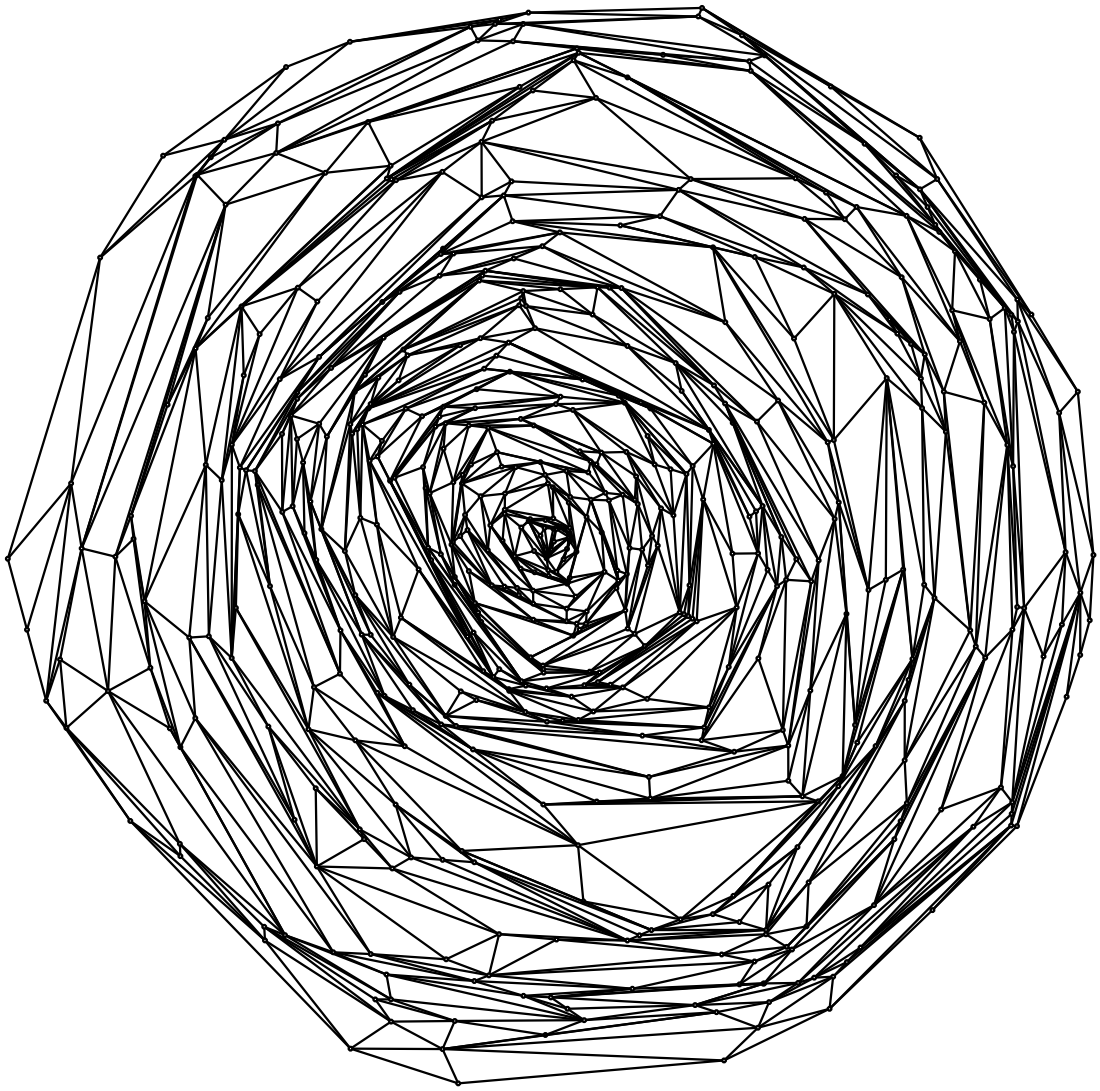


Figura A.3: Configuración de 500 puntos distribuidos uniformemente en el círculo unitario. Triangulacion-Espiral insertó 60 puntos steiner.

```
//
//  AllTriangulationsInterface.m
//  AllTriangulations
//
//

#import "AllTriangulationsInterface.h"
#import "CGRAlgorithmFramework/AlgorithmModificationsHolder.h"
#import "CGRAlgorithmFramework/CGRNotification.h"
#import "CGRAlgorithmFramework/CGREdge.h"
#import "CGRAlgorithmFramework/CGRPoint.h"

@interface ConvexQuadrilateral:NSObject{
   NSArray *points;
   NSArray *edges;
}
@property (readonly) NSArray *points;
@property (readonly) NSArray *edges;
-(id) initWithPoints:(NSArray *)p andEdges:(NSArray *)e;
-(BOOL) isEmpty:(NSArray *) points;
-(CGREdge *)diagonal;

@end
@implementation ConvexQuadrilateral
@synthesize points;
@synthesize edges;
-(id) initWithPoints:(NSArray *)p andEdges:(NSArray *)e
{
   points = [[NSArray arrayWithArray:p] retain];
   edges = [[NSArray arrayWithArray:e] retain];
   return self;
}
-(BOOL) isEmpty:(NSArray *)pointSet
{
   NSBezierPath *newPath = [[NSBezierPath alloc]init];
   [newPath moveToPoint:[[points objectAtIndex:0]origin]];
   [newPath lineToPoint:[[points objectAtIndex:1]origin]];
   [newPath lineToPoint:[[points objectAtIndex:2]origin]];
   [newPath lineToPoint:[[points objectAtIndex:3]origin]];
   for(CGRPoint *p in pointSet){
      if([newPath containsPoint:[p origin]])
         return NO;
   }
   return YES;
}

-(CGREdge *)diagonal
{
   return [edges lastObject];

}
-(NSArray *)flipDiagonal
{
```

```
   NSMutableArray *p = [NSMutableArray arrayWithArray:points];
   [p removeObject:[[edges lastObject] point1]];
   [p removeObject:[[edges lastObject] point2]];
   NSMutableArray *a = [NSMutableArray arrayWithCapacity:2];
   [a addObject:[p objectAtIndex:0]];
   [a addObject:[p objectAtIndex:1]];
   return a;

}
-(BOOL) isEqual:(id)object
{
   if([object class] != [self class])
      return NO;
   for(CGRPoint *p in points){
      if(![[(ConvexQuadrilateral *)object points] containsObject:p])
         return NO;
   }
   return YES;

}

@end

@interface Triangulation: NSObject
{
   NSArray *points, *edges;
   BOOL **adyacencyMatrix;
   unsigned int ident;
}
@property (readonly) BOOL** adyacencyMatrix;
@property (readonly) NSArray *points;
@property (readonly) NSArray *edges;
-(id) initWithPoints:(NSArray *)p edges:(NSArray *)e andIdent:(unsigned int
    )a;
@end

@implementation Triangulation
@synthesize adyacencyMatrix;
@synthesize points;
@synthesize edges;
-(id) initWithPoints:(NSArray *)p edges:(NSArray *)e andIdent:(unsigned int
    )a
{
   NSMutableArray *newPoints = [NSMutableArray arrayWithCapacity:[p count]]
      ;
   NSMutableArray *newEdges = [NSMutableArray arrayWithCapacity:[e count]];
   unsigned int i=0;
   for(; i < [p count];i++){
      CGRPoint *point = [p objectAtIndex:i];
      CGRPoint *newPoint = [[CGRPoint alloc]initWithCorePoint:[point origin
          ]];
      [newPoint setCurrentColor:[point currentColor]];
      [newPoints insertObject:newPoint atIndex:i];
```

```
        }
        for(CGREdge *ed in e){
           unsigned int p1Index, p2Index;
           p1Index = [p indexOfObject:[ed point1]];
           p2Index = [p indexOfObject:[ed point2]];
           CGREdge *newEdge = [[CGREdge alloc]initWithPoints:[newPoints
               objectAtIndex:p1Index]
                                          andPoint:[newPoints objectAtIndex:
                                              p2Index]];

           [newEdges addObject:newEdge];

        }
        points = [[NSArray arrayWithArray:newPoints] retain];
        edges = [[NSArray arrayWithArray:newEdges]retain];
        adyacencyMatrix = (BOOL **)malloc(sizeof(BOOL *)*[points count]);
        unsigned int j=0;
        for(i=0; i < [points count];i++){
           adyacencyMatrix[i] = malloc(sizeof(BOOL)*(i+1));
        }
        for(i=0; i < [points count]; i++){
           for(j=0; j < i+1; j++){
              CGRPoint *p1, *p2;
              p1 = [points objectAtIndex:i];
              p2 = [points objectAtIndex:j];
              if([p1 isNeighborOf:p2]){
                 adyacencyMatrix[i][j] = YES;
              }else{
                 adyacencyMatrix[i][j] = NO;
              }
           }
        }
        ident = a;
        return self;

    }
    -(void) addEdge:(unsigned int)p1Index p2Index:(unsigned int) p2Index
    {
        CGREdge *newEdge = [[CGREdge alloc]initWithPoints:[points objectAtIndex:
            p1Index]
                                       andPoint:[points objectAtIndex:p2Index]];
        NSMutableArray *tmp = [NSMutableArray arrayWithArray:edges];
        [tmp addObject:newEdge];
        [edges release];
        edges = [[NSArray arrayWithArray:tmp] retain];
        if(p1Index > p2Index){
           adyacencyMatrix[p1Index][p2Index] = YES;
        }else{
           adyacencyMatrix[p2Index][p1Index] = YES;
        }

    }
    -(BOOL)isEqual:(id)object
```

```
{
   if([object class] != [self class])
      return NO;

   BOOL **matrix = [(Triangulation *)object adyacencyMatrix];
   unsigned int i,j;
   for(i=0; i < [points count]; i++){
      for(j=0; j < i+1; j++){
         if(adyacencyMatrix[i][j] != matrix[i][j])
            return NO;
      }


   }
   return YES;
}

-(NSString *)description
{
   NSString *desc = [NSString stringWithString:@"ident: "];
   desc = [desc stringByAppendingString:[[NSNumber numberWithInt:ident]
       stringValue]];
   desc = [desc stringByAppendingString:@"\n"];
   unsigned int i,j;
   for(i=0; i < [points count]; i++){
      for(j=0; j < i+1; j++){
         if(adyacencyMatrix[i][j]){
            desc = [desc stringByAppendingString:[[NSNumber numberWithInt:1
                ]stringValue]];
         }else{
            desc = [desc stringByAppendingString:[[NSNumber numberWithInt:0
                ]stringValue]];
         }
      }
      desc = [desc stringByAppendingString:@"\n"];
   }
   desc = [desc stringByAppendingString:@"\n"];
   return desc;
}



@end


@implementation AllTriangulationsInterface
@synthesize helpWindow;
@synthesize typeOfAlgorithm;
@synthesize modifications;
@synthesize simulationNumbering;


-(id)init
```

```
{
   self = [super init];
   helpWindow = [[NSPanel alloc] init];
   modifications = [[AlgorithmModificationsHolder alloc]init];
   typeOfAlgorithm = @"NewWindows";
   simulationNumbering = @"PointsIncidentOnEdge";
   //We register for PointWasAdded, PointWasDeleted, PointDidMove
       notifications
   return self;
}
-(BOOL)isUpdatable
{
   return NO;
}
-(void)setUp:(NSArray *)points edges:(NSArray *)edges selectedPoints:
    (NSArray *)selectedPoints selectedEdges:(NSArray *)selectedEdges
{
   //We setup the modifications originals...
   /*if([selectedPoints count] == 1){
      [modifications setOriginalPoints:[NSArray arrayWithArray:points]];
      return;
   }
   else if([selectedPoints count] > 0){
      [modifications setOriginalPoints:[NSArray arrayWithArray:
          selectedPoints]];
      return;
   }*/
   [modifications setOriginalPoints:[NSArray arrayWithArray:points]];
   [modifications setOriginalEdges:[NSArray arrayWithArray:edges]];
   [modifications setNewPoints:[[NSMutableArray alloc]init]];
   [modifications setNewEdges:[[NSMutableArray alloc]init]];

}
-(float) leftRightTurn:(CGRPoint *)p0 p1:(CGRPoint *)p1 p2:(CGRPoint *)p2
{
   NSPoint p2p0,p1p0;
   p2p0.x = [p2 origin].x - [p0 origin].x;
   p2p0.y = [p2 origin].y - [p0 origin].y;
   p1p0.x = [p1 origin].x - [p0 origin].x;
   p1p0.y = [p1 origin].y - [p0 origin].y;

   return p2p0.x*p1p0.y - p1p0.x*p2p0.y;
}
-(BOOL) checkConvexity: (CGRPoint *)p1 p2:(CGRPoint *)p2 p3:(CGRPoint *)p3
    p4:(CGRPoint *)p4
{
   if([self leftRightTurn:p1 p1:p2 p2:p3] < 0){
      if([self leftRightTurn:p2 p1:p3 p2:p4] < 0){
         if([self leftRightTurn:p3 p1:p4 p2:p1] < 0){
            if([self leftRightTurn:p4 p1:p1 p2:p2] < 0){
               return YES;
            }else{
               return NO;
```

```
            }
         }else {
            return NO;
         }

      }else{
         return NO;
      }
   }
   if([self leftRightTurn:p1 p1:p2 p2:p3] > 0){
      if([self leftRightTurn:p2 p1:p3 p2:p4] > 0){
         if([self leftRightTurn:p3 p1:p4 p2:p1] > 0){
            if([self leftRightTurn:p4 p1:p1 p2:p2] > 0){
               return YES;
            }else{
               return NO;
            }
         }else{
            return NO;
         }

      }else{
         return NO;
      }
   }
   return NO;
}
-(NSArray *)findAllEmptyConvexCuadrilaterals:(NSArray *)points edges:
    (NSArray *)edges
{
   NSMutableArray *convexQuadrilaterals = [[NSMutableArray alloc]init];

   for(CGRPoint *p in points){
      NSArray *pNeighbors = [p giveNeighbors];
      for(CGRPoint *q in pNeighbors){
         NSMutableArray *qNeighbors = [NSMutableArray arrayWithArray:[q
             giveNeighbors]];
         [qNeighbors removeObject:p];
         for(CGRPoint *o in qNeighbors){
            NSMutableArray *oNeighbors = [NSMutableArray arrayWithArray:[o
                giveNeighbors]];
            [oNeighbors removeObject:p];
            [oNeighbors removeObject:q];
            for(CGRPoint *r in oNeighbors){
               if([pNeighbors containsObject:r]){
                  if([self checkConvexity:p p2:q p3:o p4:r]){
                     NSMutableArray *quadrilateralPoints = [[NSMutableArray
                         alloc]init];
                     NSMutableArray *quadrilateralEdges = [[NSMutableArray
                         alloc]init];
                     [quadrilateralPoints addObject:p];
                     [quadrilateralPoints addObject:q];
                     [quadrilateralPoints addObject:o];
```

```
                        [quadrilateralPoints addObject:r];

                        [quadrilateralEdges addObject:[p giveEdgeForNeighbor:q
                            ]];
                        [quadrilateralEdges addObject:[q giveEdgeForNeighbor:o
                            ]];
                        [quadrilateralEdges addObject:[o giveEdgeForNeighbor:r
                            ]];
                        [quadrilateralEdges addObject:[r giveEdgeForNeighbor:p
                            ]];
                        if([p isNeighborOf:o]){
                           [quadrilateralEdges addObject:[p
                               giveEdgeForNeighbor:o]];
                        }
                        if([q isNeighborOf:r]){
                           [quadrilateralEdges addObject:[q
                               giveEdgeForNeighbor:r]];
                        }
                        ConvexQuadrilateral *nq = [[ConvexQuadrilateral alloc]
                            initWithPoints:quadrilateralPoints
                                                                andEdges:
                                                                    quadr
                                                                    ilate
                                                                    ralEd
                                                                    ges];
                        NSMutableArray *ps = [NSMutableArray arrayWithArray:
                            points];
                        [ps removeObject:p];
                        [ps removeObject:q];
                        [ps removeObject:r];
                        [ps removeObject:o];

                        if([convexQuadrilaterals containsObject:nq] | ![nq
                            isEmpty:ps] )
                           continue;
                        else
                           [convexQuadrilaterals addObject:nq];

                     }

                  }

               }

            }

         }
      }
      return convexQuadrilaterals;

   }

   -(void)showFlip:(Triangulation *)t1 to:(Triangulation *)t2
```

```
{
   NSString *transf = [NSString stringWithString:[t1 description]];
   transf = [transf stringByAppendingString:@" |\n |\n V\n"];
   transf = [transf stringByAppendingString:[t2 description]];
   NSLog(transf);
}

-(AlgorithmModificationsHolder *)executeAlgorithm
{
   NSMutableArray *allTriangulations = [[NSMutableArray alloc]
       initWithCapacity:100];
   NSMutableArray *queue = [[NSMutableArray alloc] initWithCapacity:100];
   unsigned int ident = 0;
   Triangulation *curntTriangulation = [[Triangulation alloc]
       initWithPoints:[modifications originalPoints]
                                                        edges:[modifications
                                                            originalEdges]
                                                            andIdent:ident++]
                                                            ;
   [allTriangulations addObject:curntTriangulation];
   [queue addObject:curntTriangulation];

   while([queue count] != 0){
      curntTriangulation = [[queue lastObject]retain];
      //NSLog([curntTriangulation description]);
      [queue removeLastObject];
      NSArray *convexQuadrilaterals = [self
          findAllEmptyConvexCuadrilaterals:[curntTriangulation points]
                                                             edges:
                                                                 [
                                                                 curntTriangulati
                                                                 on edges]];

      for(ConvexQuadrilateral *cq in convexQuadrilaterals){
         NSArray *ne = [cq flipDiagonal];
         NSArray *points = [NSMutableArray arrayWithArray:
             [curntTriangulation points]];
         NSMutableArray *edges = [NSMutableArray arrayWithArray:
             [curntTriangulation edges]];
         [edges removeObject:[cq diagonal]];
         Triangulation *nt = [[Triangulation alloc] initWithPoints:points
             edges:edges andIdent:ident];
         unsigned int p1Index = [[curntTriangulation points] indexOfObject:
             [ne objectAtIndex:0]];
         unsigned int p2Index = [[curntTriangulation points] indexOfObject:
             [ne objectAtIndex:1]];
         [nt addEdge:p1Index p2Index:p2Index];
         if(![allTriangulations containsObject:nt]){
            [self showFlip:curntTriangulation to:nt];
            ident++;
            [allTriangulations addObject:nt];
            [queue addObject:nt];
         }
```

```
        }
    }
    for(Triangulation *t in allTriangulations){
        [[modifications newPoints] addObject:[t points]];
        [[modifications newEdges] addObject:[t edges]];
    }
    return modifications;
}
-(NSString *)menuItemName
{
    return @"All Triangulations";
}
-(void) handlePointDidMoveNotification:(NSNotification *) note
{
    //If the algorithm has not been executed then we ignore the notification
    return;

}
-(void) handlePointWasAddedNotification:(NSNotification *) note
{
    //If the algorithm has not been executed then we ignore the notification
    return;

}
-(void) handlePointWasDeletedNotification:(NSNotification *) note
{
    //If the algorithm has not been executed then we ignore the notification
    return;

}

@end
```

Bibliografía

[AHH+09] Oswin Aichholzer, Thomas Hackl, Michael Hoffmann, Alexander Pilz, Günter Rote, Bettina Speckmann, and Birgit Vogtenhuber. Plane graphs with parity constraints. In Frank K. H. A. Dehne, Marina L. Gavrilova, Jörg-Rüdiger Sack, and Csaba D. Tóth, editors, *WADS*, volume 5664 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2009.

[Alv10] Victor Alvarez. Even triangulation of planar set of points with steiner points. In *Proceeding of the 26th European Workshop on Computational Geometry*, 2010.

[Cha85] Bernard Chazelle. On the convex layers of a planar set. *IEEE Transactions on Information Theory*, 31:509–517, 1985.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* McGraw-Hill, 2001.

[DKK02] Krzysztof Diks, Łukasz Kowalik, and Maciej Kurowski. A new 3-color criterion for planar graphs. In Ludek Kucera, editor, *Proc. 28th Int. Worksh. Graph-Theoretic Concepts in Computer Science (WG 2002)*, number 2573 in Lecture Notes in Computer Science, pages 138–149. Springer-Verlag, 2002.

[Har94] F. Harary. *Graph Theory.* Westview Press, 1994.

[HK96] Frank Hoffmann and Klaus Kriegel. A graph-coloring result and its consequences for polygon-guarding problems. *SIAM J. Discret. Math.*, 9(2):210–224, 1996.

[HU07] V. Heredia and J. Urrutia. On convex quadrangulations of point sets on the plane. In Jin Akiyama, William Chen, Mikio Kano, Xueliang Li, and Qinglin Yu, editors, *Discrete Geometry, Combinatorics and Graph Theory*, volume 4381 of *Lecture Notes in Computer Science*, pages 38–46. Springer Berlin / Heidelberg, 2007.

[PRU10] Canek Peláez, Adriana Ramírez, and Jorge Urrutia. Triangulations with many points of even degree. In *Proceedings of the 22nd Canadian Conference on Computational Geometry*, 2010.

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[TB97] Godfried Toussaint and Prosenjit Bose. Characterizing and efficiently computing quadrangulations of planar point sets. *Comput. Aided Geom. Des*, 14:14–763, 1997.

[Tou85] Godfried Toussaint. Quadrangulations of planar sets. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pages 218–227. Springer-Verlag, 1985.

[ZH05] Huaming Zhang and Xin He. On even triangulations of 2-connected embedded graphs. *SIAM J. Comput.*, 34(3):683–696, 2005.