# UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ACATLÁN

## "UN ALGORITMO PARA CONTAR POLÍGONOS CONVEXOS VACÍOS EN CONJUNTOS DE PUNTOS EN EL PLANO"

# T E S I S

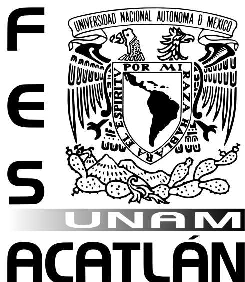
QUE PARA OBTENER EL TÍTULO DE

**LICENCIADO EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN**

P R E S E N T A

**CARLOS MIGUEL HIDALGO TOSCANO**

DIRECTOR DE TESIS: DR. RUY FABILA MONROY




Abril 2013.

**UNAM – Dirección General de Bibliotecas**
**Tesis Digitales**
**Restricciones de uso**

**DERECHOS RESERVADOS ©**
**PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL**

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

*A mi madre, Cristina*

# Agradecimientos

A mi madre, Cristina Hidalgo Toscano, por su cariño invaluable y el apoyo incondicional que me ha brindado en todo sentido y en cada momento de mi vida: infinitas gracias.

A mi asesor, el Dr. Ruy Fabila Monroy, por la oportunidad de trabajar en este tema, por su paciencia, comentarios, el tiempo dedicado en dirigir este trabajo y por todo lo que he aprendido gracias a él en el último año.

A Dení Rodríguez, con quien compartí tanto a lo largo de la carrera, por las horas de diversión, trabajo, desvelos y por estar ahí siempre. A Jaime Calles y a Jesús Hernández, con quienes tuve el gusto de compartir los últimos semestres, por todos los buenos momentos y a los tres por su amistad.

A los profesores que tuve a lo largo de la carrera, de manera especial al Fis. Manuel Valadez Rodríguez y al Lic. José Sebastián Bejos Mendoza por los consejos y orientación ofrecidos.

Al Centro de Desarrollo Tecnológico y en particular al Lic. Fernando Israel González Trejo por todo el apoyo y el espacio que ponen a disposición de los proyectos de los alumnos.

Al programa de Matemáticas Aplicadas y Computación por su ayuda a lo largo de la carrera y su valiosa orientación en los trámites y a la Universidad nacional Autónoma de México por brindarme la oportunidad de estudiar esta carrera.

Esta tesis participó en el Proyecto 153984 “CONTEO DE CONFIGURACIONES GEOMÉTRICAS EN CONJUNTOS DE PUNTOS” de Ciencia Básica, CONACyT.

# Índice general

**Introducción** **1**

**1. Hoyos Convexos** **3**

1.1. $r$-gonos convexos . . . . . 3

1.2. $r$-hoyos y el conjunto de Horton . . . . . 6

1.3. Variantes cromáticas . . . . . 10

**2. El algoritmo de Dobkin, Edelsbrunner y Overmars** **15**

2.1. Cálculo de la Gráfica de Visibilidad . . . . . 19

2.2. La Cadena Convexa más Larga . . . . . 21

2.3. Reportando los $r$-hoyos . . . . . 24

2.4. La complejidad del algoritmo . . . . . 26

**3. Modificaciones al Algoritmo** **29**

3.1. Calculando la Gráfica de Visibilidad . . . . . 30

3.2. Cadenas convexas . . . . . 33

**4. Implementación** **37**

4.1. Elección de los lenguajes . . . . . 37

4.2. Implementación en C++ . . . . . 39

4.3. Extendiendo el intérprete de Python . . . . . . . . . . . . . . . . . . . . . . . . . 40

**5. Resultados** **45**

5.1. Búsqueda Local . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

5.2. Experimentos y resultados . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**A. Conceptos y notación** **53**

**Bibliografía** **55**

# Introducción

El problema con el que inicia esta tesis fue planteado por Esther Klein alrededor de 1933. Esther Klein fue una matemática húngara nacida en Budapest, en 1910. Klein observó que para cualesquiera 5 puntos en posición general en el plano, es posible escoger 4 que forman los vértices de un cuadrilátero convexo; en base a esto planteó el siguiente problema: ¿qué cantidad de puntos en posición general en el plano se necesitan para garantizar que algún subconjunto forma los vértices de un polígono convexo de $r$ lados? ¿ese número existe para cualquier $r$?

Paul Erdős y George Szekeres (también matemáticos húngaros) demostraron un par de años después que el número en cuestión (denotado $g(r)$) existe para cualquier $r$ y dieron cotas para su valor [ES35], pero calcularlo de manera exacta resulta ser difícil. Han pasado casi 75 años de que el problema fue planteado y $g(r)$ se conoce de manera exacta sólo para $r \leq 6$.

Erdős fue un escritor prolífico, tuvo más de mil publicaciones y cientos de colaboradores y propuso numerosos problemas en diversas áreas de las matemáticas. Uno de estos problemas fue una modificación a la pregunta de Klein: pedir que los polígonos fueran vacíos [Erd84]. Esta sencilla modificación da como resultado que el número que se busca (denotado $h(r)$) no exista siempre: Horton demostró que se pueden construir conjuntos finitos arbitrariamente grandes sin polígonos vacíos de $r$-lados para $r \geq 7$ [Hor83].

Se conocen valores exactos de $h(r)$ para $r \leq 5$, pero $h(6)$ ha resultado elusivo. Fue hasta 2007 y 2008 que Nicolás [Nic07] y Gerken [Ger08] demostraron de manera independiente su existencia y desde entonces se han encontrado mejores cotas inferiores y superiores: Koshelev dio la mejor cota superior que se tiene hasta el momento en 2009 [Kos09b] mientras que la mejor cota inferior se debe a Overmars y data de 2003 [Ove03].

Existen otras modificaciones al problema y preguntas relacionadas al mismo, por ejemplo, pueden pensarse que los puntos tienen colores y buscar ahora polígonos convexos

(vacíos o no) cuyos vértices sean del mismo color [DHKS03] (a estos polígonos se les llama *monocromáticos*) o puede preguntarse cuál es la cantidad mínima de polígonos de cierto número de lados que se determinan en un conjunto de puntos.

Al existir tantas posibilidades de acomodar puntos en el plano, resulta de interés disponer de algoritmos que permitan conocer la cantidad de polígonos convexos en un conjunto. En particular, estos algoritmos son útiles para buscar cotas de valores desconocidos de $g(r)$ y $h(r)$, o demostrar sus valores exactos. A manera de ejemplo, Szekeres y Peters dieron una prueba por computadora para mostrar que $g(6) = 17$ [SP06] y las cotas inferiores que se han dado para $h(6)$ se han logrado encontrando conjuntos sin hexágonos vacíos a través de la computadora. Estos conjuntos fueron descubiertos por Overmars [OSV89] utilizando una versión modificada del algoritmo que él mismo presentó junto con Dobkin y Edelsbrunner en 1990 [DEO90].

El objetivo de este trabajo es modificar este algoritmo y conseguir una implementación eficiente que permita hacer búsquedas de polígonos convexos vacíos (ya sea monocromáticos o no) en conjuntos de puntos en posición general en el plano.

En el primer capítulo se presentan las bases teóricas sobre la existencia de $h(r)$ y $g(r)$, así como los valores y cotas que se conocen actualmente. También se presentan algunos resultados acerca de polígonos monocromáticos. En el segundo capítulo, se expone el algoritmo de Dobkin, Edelsbrunner y Overmars y su complejidad. El Capítulo 3 trata sobre las modificaciones que se hicieron al algoritmo y la motivación para hacerlas. Los lenguajes usados para implementar el algoritmo modificado y detalles sobre la implementación se encuentran en el Capítulo 4. El Capítulo 5 contiene los experimentos realizados con la implementación discutida en el Capítulo 4 y los resultados obtenidos. Finalmente, en el Apéndice A se encuentran conceptos y notación usados a lo largo de este trabajo.

# Capítulo 1

# Hoyos Convexos en Conjuntos de Puntos en Posición General

Muchos problemas en Geometría Computacional buscan, dado un conjunto de puntos, identificar subconjuntos que cumplan con características particulares.

En este capítulo se consideran conjuntos de puntos en posición general y en principio se buscan subconjuntos que formen los vértices de un polígono convexo. Surgen nuevos problemas al pedir que dichos polígonos sean vacíos o al agregar colores a los puntos y pedir que los subconjuntos tengan elementos de un solo color.

## 1.1. $r$-gonos convexos

Dado un conjunto de puntos en posición general, a un subconjunto de $r$ puntos que forman los vértices de un polígono convexo se le llama $r$-gono. Este concepto surge a partir de un problema que planteó Esther Klein alrededor de 1933 [ES35]. Klein hizo la siguiente pregunta:

> Dado un entero positivo $r$, ¿Se puede encontrar un número $g(r)$ tal que cualquier conjunto con al menos $g(r)$ puntos en posición general contiene $r$ puntos que forman los vértices de un polígono convexo?

Este problema se conocería después como “Happy Ending Problem” (Problema del final feliz) ya que Esther Klein y George Szekeres se comprometieron mientras colaboraban en

él y se casaron poco después.

Como cualesquiera 3 puntos no colineales determinan un triángulo, se tiene que $g(3) = 3$. Esther Klein demostró que $g(4) = 5$.

**Teorema 1.1.** *Dados cualesquiera 5 puntos en el plano en posición general, siempre es posible escoger 4 de tal manera que formen los vértices de un cuadrilátero convexo.*

*Demostración.* Sea $S = \{A, B, C, D, E\}$ un conjunto de puntos en posición general. Si $conv(S)$ es un pentágono o un cuadrilátero, el teorema se cumple.

Supongamos entonces que $conv(S)$ es un triángulo y que sus vértices son los puntos $A, B, C$. Entonces los puntos $D, E$ se encuentran en su interior. Como los puntos se encuentran en posición general, dos de los vértices de $conv(S)$ (digamos, $A$ y $B$) se encuentran del mismo lado de la línea $\overline{DE}$ (Figura 1.1). Entonces, $A, B, D, E$ forman los vértices de un cuadrilátero convexo. ■

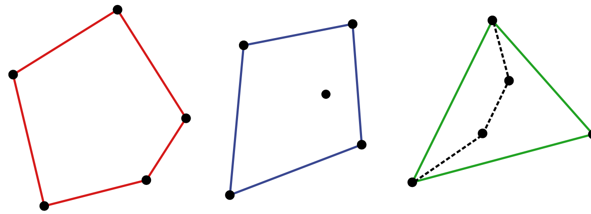


Figura 1.1: Los 3 casos de 5 puntos en posición general.

En 1935 Paul Erdős y George Szekeres respondieron de manera afirmativa la pregunta de Klein [ES35]. Erdős y Szekeres dieron 2 pruebas de este hecho, una usando el Teorema de Ramsey y otra de naturaleza geométrica y combinatoria. Aquí se sigue la exposición de [Mat02].

Antes de pasar a la prueba son necesarias unas definiciones.

**Definición 1.1.** *Se dice que un conjunto $S$ de puntos en el plano se encuentra en* ***posición convexa*** *si para cada $p \in S$ se tiene que $p \notin conv(S \setminus \{p\})$.*

**Definición 1.2.** *Sea $X$ un conjunto de puntos en posición general en el plano. Se le llama a $X$ una* ***taza (tapa)*** *si $X$ está en posición convexa y $conv(X)$ está acotado por arriba (debajo) por una sola arista. Una $k$-taza es una taza con $k$ puntos. Una $l$-tapa se define de manera análoga (Figura 1.2).*

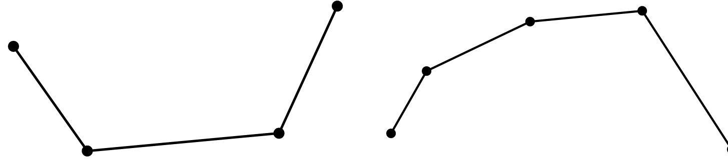


Figura 1.2: Una 4-taza y una 5-tapa.

**Teorema 1.2** (de Erdős-Szekeres)**.** *Para todo entero $r \geq 3$ existe un entero positivo $g(r)$ tal que cualquier conjunto $S$ con $g(r)$ puntos en posición general contiene un subconjunto de $r$ puntos que forman los vértices de un $r$-gono.*

*Demostración.* Sean $k, l$ enteros positivos y sea $f(k,l)$ el entero positivo más pequeño tal que cualquier conjunto con al menos $f(k,l)$ puntos en posición general en el plano contiene una $k$-taza o una $l$-tapa. A continuación se prueba siguiente desigualdad por inducción sobre $k$ y $l$:

$$f(k,l) \leq \binom{k+l-4}{k-2} + 1 \tag{1.1}$$

La desigualdad anterior se cumple tanto para $k = 2$ y cualquier $l$ como para $l = 2$ y cualquier $k$. Entonces, sean $k, l \geq 3$ y supongamos que la desigualdad se cumple para $f(k, l-1)$ y $f(k-1, l)$.

Sea $S$ un conjunto con $f(k-1,l)+f(k,l-1)-1$ puntos en posición general, entonces $S$ debe contener al menos una $k-1$-taza o una $l$-tapa. Supongamos que $S$ no contiene $l$-tapas. Sea $E$ el conjunto de puntos que se encuentran más a la derecha de una $k-1$-taza en $S$. Entonces $S \setminus E$ no contiene $k-1$-tazas, es decir

$$|S \setminus E| = |S| - |E| \leq f(k-1, l) - 1$$

Como $|S| = f(k-1,l) + f(k,l-1) - 1$, se tiene

$$|E| \geq f(k, l-1)$$

Entonces $E$ debe contener una $k$-taza o una $l-1$-tapa. Por la manera en la que se construyó $E$, si contiene una $l-1$-tapa su punto más a la izquierda es el punto más a la derecha de alguna $k-1$-taza en $S$. Por este razonamiento, la $l-1$-tapa o la $k-1$-taza se pueden extender en un punto, por lo que $S$ contiene una $k$-taza o una $l$-tapa.

Así, $f(k,l) \leq f(k-1,l) + f(k,l-1) - 1$. Usando la hipótesis de inducción:

$$\begin{aligned} f(k,l) &\leq f(k-1,l) + f(k,l-1) - 1 \\ &\leq \binom{k+l-5}{k-3} + 1 + \binom{k+l-5}{k-2} + 1 - 1 = \binom{k+l-4}{k-2} + 1 \end{aligned}$$

Con lo que la desigualdad (1.1) se cumple para cualesquiera $l, k$ enteros positivos.

Así, todo conjunto de al menos $f(k,k) \leq \binom{2k-4}{k-2} + 1$ puntos siempre contiene una $k$-taza o una $k$-tapa que, por estar en posición convexa, forma los vértices de un $r$-gono.

Por tanto, $g(r)$ existe para todo entero positivo $r$ y es menor o igual a $\binom{2r-4}{r-2} + 1$. ■

Además de probar la existencia de $g(r)$ y dar la cota superior presentada en el Teorema 1.2, Erdős y Szekeres conjeturaron que $g(r) = 2^{r-2} + 1$. En [ES61] describieron un método para acomodar $2^{r-2}$ puntos en el plano en posición general de tal manera que no aparezca un $r$-gono, demostrando así que $2^{r-2} + 1 \leq g(r)$.

En 2006, Szekeres y Peters dieron una prueba por computadora en [SP06] de que $g(6) = 17$ con lo que la igualdad en la cota inferior se da para $r \leq 6$, reforzando la conjetura. En cuanto a la cota superior, G. Toth y P. Valtr [TV98] la mejoraron a $\binom{2r-5}{r-2} + 2$ y en 2005 redujeron el +2 a +1 [TV05]. Así, las cotas actuales son:

$$2^{r-2} + 1 \leq g(r) \leq \binom{2r-5}{r-2} + 1$$

## 1.2. $r$-hoyos y el conjunto de Horton

En 1978 Erdős [Erd78] presentó un problema nuevo a partir de la pregunta hecha por Klein. Se preguntaba ahora por el menor entero $h(r)$ tal que cualquier conjunto de $h(r)$ puntos en posición general contuviera al menos un subconjunto de $r$ puntos que forme los vértices de un polígono convexo sin ningún otro punto en su interior. A tal subconjunto se le llama un *r-hoyo*.

Al igual que en el caso de los $r$-gonos, se tiene que $h(3) = 3$. Dado un conjunto $S$ de 5 puntos en posición general, si $conv(S)$ es un pentágono o un triángulo, el cuadrilátero obtenido en la prueba del Teorema 1.1 es vacío. Si $conv(S)$ es un cuadrilátero, contiene un punto $p \in S$ en su interior. Como los puntos están en posición general, si $p'$ es un vértice de $conv(S)$ debe haber 2 puntos de $S$ a un lado de la línea que pasa por $p$ y $p'$.

Esos 2 puntos, junto con $p$ y $p'$, forman un 4-hoyo. Así, $h(4) = g(4)$. En cuanto a 5-hoyos, Harborth demostró en 1978 que $h(5) = 10$ [Har78].

De manera sorprendente, Horton demostró que existen conjuntos finitos arbitrariamente grandes de puntos en posición general sin 7-hoyos [Hor83], por lo que $h(r)$ no existe para $r \geq 7$. A continuación se presenta la construcción de dichos conjuntos (llamados conjuntos de Horton) y la demostración de que no contienen 7-hoyos, como se encuentra en [Mat02].

**Definición 1.3.** *Sean $S$ y $T$ conjuntos finitos de puntos en el plano. Se dice que $S$ está **muy arriba** de $T$ (y que $T$ está **muy abajo** de $S$) si se cumple:*

*(i) Ninguna línea determinada por dos puntos de $S \cup T$ es vertical.*

*(ii) Cada línea determinada por dos puntos de $S$ está arriba de todos los puntos de $T$.*

*(iii) Cada línea determinada por dos puntos de $T$ está debajo de todos los puntos de $S$.*

Para un conjunto de puntos en el plano $S = \{p_1, p_2, \ldots, p_n\}$ en el cual ningún par de puntos tienen la misma coordenada $x$ y con la notación elegida de tal manera que la coordenada $x$ de $p_i$ es menor que la de $p_j$ para $i < j$, se definen los conjuntos $S_0 = \{p_2, p_4, \ldots\}$ (los puntos con índice par) y $S_1 = \{p_1, p_3, \ldots\}$ (los puntos con índice impar).

**Definición 1.4.** *Se le llama conjunto de Horton a un conjunto finito $H$ de puntos en el plano si $|H| \leq 1$ o cumple las siguientes condiciones:*

- *Tanto $H_0$ como $H_1$ son conjuntos de Horton*
- *$H_0$ está muy arriba de $H_1$ o $H_0$ está muy abajo de $H_1$*

**Lema 1.1.** *Para todo entero $n \geq 1$, existe un conjunto de Horton con $n$ puntos.*

*Demostración.* Nótese que se puede obtener un conjunto de Horton a partir de uno más grande eliminando puntos con las coordenadas en $x$ más grandes.

Sea $k$ un entero no negativo. Se denota al conjunto de Horton con $2^k$ puntos como $H^k$ y se define $H^0 = (0,0)$.

Se puede obtener $H^{k+1}$ a partir de $H^k$ de la siguiente manera: sean $A = \{(2x, 2y) \mid (x, y) \in H^k\}$ y $B = \{(x+1, y+h_k) \mid (x, y) \in A\}$ donde $h_k$ es un entero lo suficientemente grande para que $B$ esté muy arriba de $A$. Nótese que tanto $A$ como $B$ siguen cumpliendo las condiciones requeridas para ser conjuntos de Horton. Entonces, tomando $H^{k+1} = A \cup B$, se obtiene un conjunto de Horton con $2^{k+1}$ puntos (Figura 1.3). ■

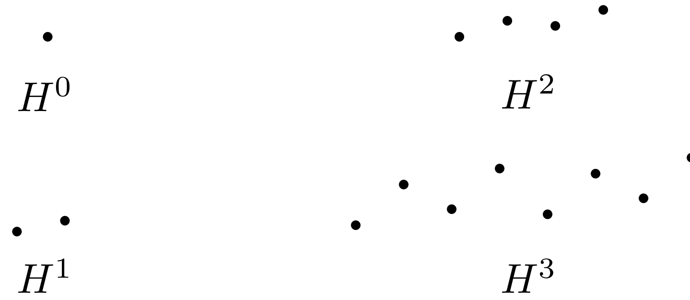


Figura 1.3: Conjuntos de Horton.

**Definición 1.5.** *Sea $S$ un conjunto finito de puntos en el plano. Se dice que $S$ es* ***r-cerrado por arriba*** *si para cualquier $r$-taza en $S$ existe un punto en $S$ que se encuentra arriba de dicha $r$-taza. Se definen conjuntos* ***r-cerrados por debajo*** *de manera análoga usando $r$-tapas. (Figura 1.4)*

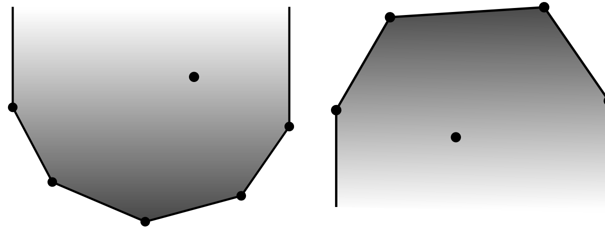


Figura 1.4: Un conjunto 5-cerrado por arriba y uno 4-cerrado por debajo

**Lema 1.2.** *Todo conjunto de Horton es* 4*-cerrado por arriba y* 4*-cerrado por debajo.*

*Demostración.* Sea $H$ un conjunto de Horton. La prueba es por inducción sobre el tamaño de $H$. Si $|H| \leq 1$, $H$ es 4-cerrado por arriba y por debajo por vacuidad. Sea $|H| > 1$ y supongamos que todo conjunto de Horton de menor cardinalidad es 4-cerrado por arriba y por debajo.

Sea $C$ una 4-taza en $H$ y asumamos, sin pérdida de generalidad, que $H_0$ está muy debajo de $H_1$.

Si $C \subseteq H_0$ o $C \subseteq H_1$, existe un punto de $H$ arriba de $C$ por hipótesis. Entonces supongamos $C \cap H_0 \neq \emptyset$ y $C \cap H_1 \neq \emptyset$.

$C$ debe tener a lo más 2 puntos en $H_1$: si hubiera 3 puntos ($a, b, c$ de izquierda a derecha), entonces $H_0$ se encuentra muy debajo de las líneas $ab$ y $bc$, por lo que el cuarto punto de $C$ que debería estar en $H_0$ no puede formar una taza con $\{a, b, c\}$. (Figura 1.5)

Entonces $C$ debe tener al menos 2 puntos (digamos, $c$ y $d$) en $H_0$. Ya que a lo largo del eje $x$ los puntos de $H$ alternan entre $H_0$ y $H_1$, existe un punto $e$ de $H$ entre $c$ y $d$. El punto $e$ está arriba del segmento $\overline{cd}$, por lo que $C$ es cerrada por arriba. Así, $H$ es 4-cerrado por arriba. La prueba es análoga para mostrar que es 4-cerrado por debajo usando 4-tapas. ■

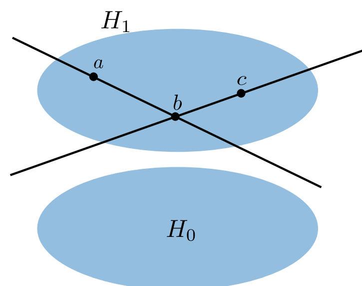



Figura 1.5: A lo más hay 2 puntos de la 4-taza en $H_1$.

**Teorema 1.3.** *Ningún conjunto de Horton contiene un 7-hoyo.*

*Demostración.* Sea $H$ un conjunto de Horton. Si $|H| \leq 6$, no contiene ningún 7-hoyo. Sea $|H| > 6$ y supongamos que ningún conjunto de Horton de menor cardinalidad contiene un 7-hoyo.

Supongamos que $C$ es un 7-hoyo en $H$. Por hipótesis de inducción, $C$ no puede estar contenido en $H_0$ ni en $H_1$. Entonces, sin pérdida de generalidad, supongamos que $H_0$ contiene la mayor cantidad de puntos de $C$ y que $H_0$ está muy debajo de $H_1$. Al menos 4 puntos de $C$ están contenidos en $H_0$ y estos puntos deben formar una 4-taza: si 3 puntos formaran una 3-tapa, ningún punto en $H_1$ no se podría añadir para formar un conjunto convexo. Por el Lema 1.2, existe un punto arriba de dicha taza, y ese punto está en el interior de $C$, una contradicción. Así, $H$ no contiene 7-hoyos. ■

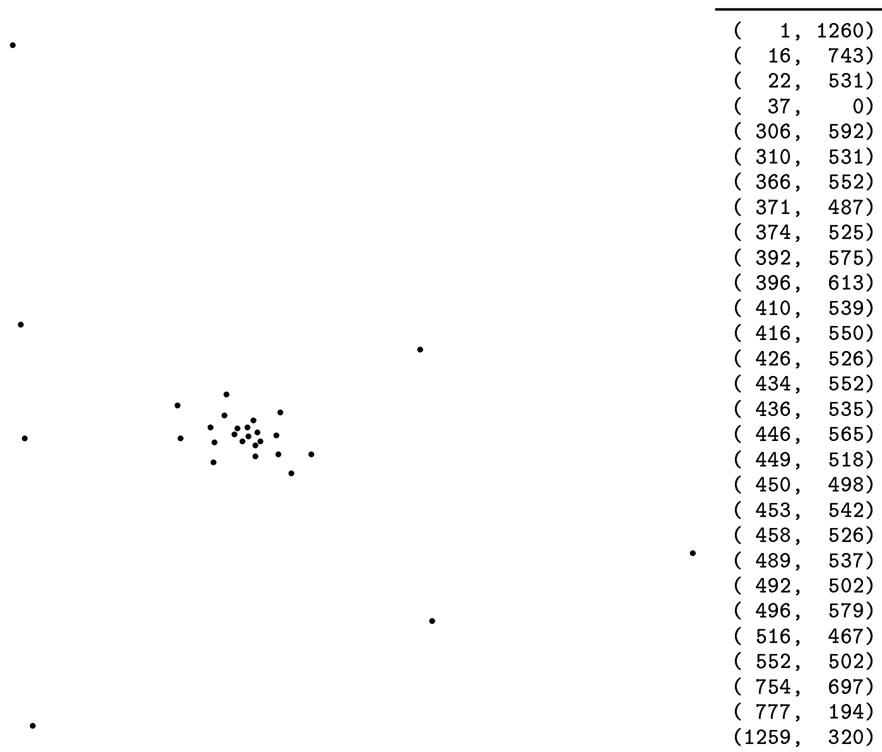
A pesar de que Horton presentó este resultado en 1983, tomó casi 25 años demostrar la existencia de $h(6)$. En 2007, Nicolás [Nic07] demostró que $h(6) \leq g(25)$ y en 2008, Gerken [Ger08] demostró que todo conjunto que contiene un 9-gono determina un hexágono vacío, por lo que $h(6) \leq g(9)$.

La cota superior más reciente se debe a Koshelev, que en 2009 demostró que $h(6) \leq \text{máx}\{g(8), 400\} \leq \binom{11}{6} + 1$ [Kos09b]. De ser cierta la conjetura de Erdős y Szekeres de que $g(r) = 2^{r-2} + 1$ la cota caería a 129 puntos usando el resultado de Gerken.

En cuanto a la cota inferior, en 1989 Overmars encontró un conjunto de 26 puntos en el plano sin hexágonos vacíos [OSV89] y en 2003 encontró conjuntos de 29 puntos sin

hexágonos vacíos [Ove03] usando una versión modificada del algoritmo presentado en [DEO90]. Uno de estos conjuntos se muestra en la Figura 1.6. Así, las mejores cotas para $h(6)$ que se conocen hasta el momento son:

$$30 \leq h(6) \leq 463$$



| | |
|---|---|
| ( 1, | 1260) |
| ( 16, | 743) |
| ( 22, | 531) |
| ( 37, | 0) |
| ( 306, | 592) |
| ( 310, | 531) |
| ( 366, | 552) |
| ( 371, | 487) |
| ( 374, | 525) |
| ( 392, | 575) |
| ( 396, | 613) |
| ( 410, | 539) |
| ( 416, | 550) |
| ( 426, | 526) |
| ( 434, | 552) |
| ( 436, | 535) |
| ( 446, | 565) |
| ( 449, | 518) |
| ( 450, | 498) |
| ( 453, | 542) |
| ( 458, | 526) |
| ( 489, | 537) |
| ( 492, | 502) |
| ( 496, | 579) |
| ( 516, | 467) |
| ( 552, | 502) |
| ( 754, | 697) |
| ( 777, | 194) |
| (1259, | 320) |

Figura 1.6: Uno de los conjuntos de 29 puntos sin 6-hoyos hallados por Overmars.

## 1.3. Variantes cromáticas

Además de los $r$-hoyos, existen numerosas variantes del problema presentado por Esther Klein. Una de ellas consiste en agregar colores a los puntos, considerada por Devillers, Hurtado, Károlyi y Seara en [DHKS03]. En esta sección se presentan algunos de sus resultados.

Sea $S$ un conjunto de puntos en el plano en posición general. Una $k$-coloración de $S$ es una partición $\{S_1, S_2, \ldots, S_k\}$ de $S$. A cada $S_i$ se le llama conjunto de puntos de color $i$ y se

dice que $S$ está coloreado con $k$ colores. A un subconjunto $T$ de $S$ se le llama *monocromático* si todos los puntos de $T$ son del mismo color.

Dado un conjunto $S$ de puntos y una coloración de $S$ se puede modificar la pregunta de Esther Klein para buscar $r$-gonos monocromáticos. En analogía con $g(r)$, se define $g_M(r,k)$ como el menor entero tal que cualquier conjunto con al menos $g_M(r,k)$ puntos en posición general en el plano y coloreados con $k$ colores contiene un $r$-gono monocromático. Se puede utilizar el Teorema de Erdős–Szekeres para determinar el valor exacto de $g_M(r,k)$.

**Teorema 1.4.** $g_M(r,k) = k \cdot (g(r) - 1) + 1$.

*Demostración.* Sean $S$ un conjunto con $k \cdot (g(r) - 1) + 1$ puntos en posición general y $\{S_1, \ldots, S_k\}$ una coloración de $S$. Entonces $\left|S_i\right| \geq g(r)$ para algún $1 \leq i \leq k$, por lo que $S_i$ contiene un $r$-gono. Así, $S$ contiene un $r$-gono monocromático. Por otro lado, existen conjuntos con $k \cdot (g(r) - 1)$ puntos sin $r$-gonos monocromáticos: $k$ copias ajenas de un conjunto de $g(r) - 1$ puntos sin $r$-gonos, cada una de un color distinto, es un ejemplo. ■

Además de $r$-gonos monocromáticos, resulta natural considerar $r$-hoyos monocromáticos. De acuerdo con la sección anterior, $h(r)$ existe para $r \leq 6$, y los conjuntos de Horton muestran que $h(r)$ no existe para $r > 6$. Se pueden obtener resultados sobre la ausencia de $r$-hoyos monocromáticos con coloraciones apropiadas de conjuntos de Horton. Los siguientes teoremas muestran que se pueden construir:

i) Conjuntos coloreados arbitrariamente grandes sin $r$-hoyos usando 3 colores y

ii) Conjuntos coloreados arbitrariamente grandes sin $r$-hoyos usando 2 colores cuando $r \geq 5$

**Definición 1.6.** *Se dice que una $r$-taza (tapa) en un conjunto $S$ es* ***vacía*** *si no existen puntos arriba (debajo) de ella.*

**Lema 1.3.** *Sea $H$ un conjunto de Horton. La diferencia entre los índices de los vértices de una 2-taza (tapa) vacía es una potencia de 2.*

*Demostración.* La prueba es por inducción sobre el tamaño de $H$. Si $|H| = 2$ el enunciado del teorema es cierto. Sea $|H| > 2$ y supongamos que el enunciado el teorema se cumple para todo conjunto de Horton de menor cardinalidad.

Sin pérdida de generalidad, supongamos que $H_1$ se encuentra muy abajo de $H_0$. Sea $C$ una 2-taza vacía en $H$. Los vértices de $C$ no pueden estar ambos en $H_1$, ya que existiría

un punto en $H_0$ que haría que $C$ no fuera vacía. Si un vértice de $C$ está en $H_1$ y el otro en $H_0$, la diferencia entre ambos debe ser 1 (de no ser así, de nuevo existen puntos en $H_0$ que evitan que $C$ sea vacía). Finalmente, si ambos vértices están en $H_0$, por hipótesis de inducción la diferencia entre sus índices en $H_0$ es una potencia de 2, digamos, $2^k$. Entonces su diferencia en $H$ es $2^{k+1}$. La prueba para las 2-tapas es análoga. ■

**Teorema 1.5.** *Sea $H$ un conjunto de Horton. Existe una 3-coloración de $H$ que no contiene 3-hoyos monocromáticos.*

*Demostración.* Sea $H$ un conjunto de Horton y coloréense los puntos de $H$ con los colores $R, G, B$ sucesivamente en el orden de su coordenada $x$, es decir, $p_1$ de color $R$, $p_2$ de color $G$, $p_3$ de color $B$, $p_4$ de color $R$, etc.

Sin pérdida de generalidad, supongamos que $H_0$ está muy arriba de $H_1$, entonces $H_0$ queda con los colores $GRBGRB\ldots$ y $H_1$ queda con los colores $RBGRBG\ldots$ Si $|H| \leq 3$ y se colorea de esta manera, no contiene ningún 3-hoyo monocromático.

Sea $|H| > 3$ y supongamos ningún conjunto de Horton de menor cardinalidad coloreado de esta manera contiene 3-hoyos monocromáticos. Considérese ahora un triángulo monocromático con 2 vértices en $H_1$ y uno en $H_0$. La diferencia entre los índices de los puntos que están en $H_1$ es un múltiplo de 3, por el Lema anterior no forman una 2-taza vacía en $H_0$ por lo que el triángulo no puede ser vacío. De manera análoga, un triángulo monocromático con 2 vértices en $H_0$ y uno en $H_1$ no puede ser vacío. Por último, un triángulo monocromático con todos sus vértices en $H_0$ o en $H_1$ no puede ser vacío por la hipótesis de inducción, lo cual completa la prueba. ■

**Teorema 1.6.** *Sea $H$ un conjunto de Horton. Existe una coloración de $H$ usando 2 colores que no contiene 5-hoyos monocromáticos.*

*Demostración.* Sea $H$ un conjunto de Horton y coloréense los puntos de $H$ con los colores $R, G, B$ sucesivamente en el orden de su coordenada $x$, es decir, $p_1$ de color $R$, $p_2$ de color $G$, $p_3$ de color $B$, $p_4$ de color $R$, etc.

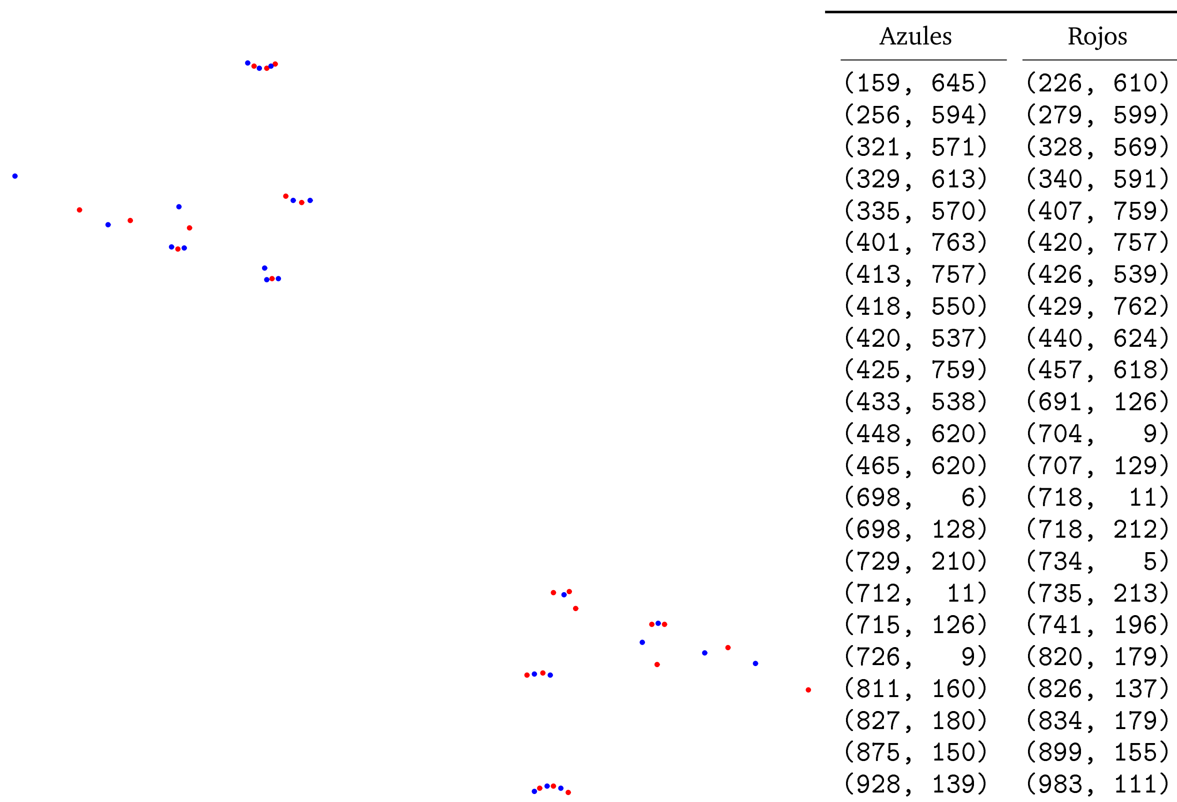
Se puede construir una bicoloración de $H$ identificando los colores $G$ y $B$ con el color $GB$. Sea $P$ un pentágono monocromático en $H$. Con esta coloración, $P$ puede ser de color $R$ o de color $GB$. $P$ contiene al menos 3 vértices del mismo color en la coloración inicial, por lo que forman un triángulo monocromático en la coloración inicial. Dicho triángulo no es vacío por el Teorema anterior, lo cual implica que $P$ tampoco es vacío. Así, $H$ no contiene 5-hoyos. ■

En base a estos 2 teoremas, resulta interesante buscar 3-hoyos y 4-hoyos en conjuntos bicromáticos.

Puede preguntarse por la cantidad mínima de 3-hoyos monocromáticos generados en este tipo de conjuntos. Como $h(5) = 10$, cualquier conjunto bicoloreado de 10 puntos contiene al menos un 3-hoyo vacío: cualquier pentágono vacío debe tener al menos 3 puntos del mismo color. De esta observación, se puede concluir que hay al menos una cantidad lineal de triángulos monocromáticos vacíos en cualquier conjunto bicoloreado de puntos, ya que es posible partir cualquier conjunto grande en conjuntos pequeños de cardinalidad constante. La primer cota supralineal que se dio a conocer para la cantidad de este tipo de hoyos es $\Omega(n^{5/4})$, dada por Aichholzer, Fabila-Monroy, Flores-Peñaloza, Hackl, Huemer y Urrutia [AFMFP$^+$08]. Esta cota se mejoró a $\Omega(n^{4/3})$ en [PT08], pero en [AFMFP$^+$08] se conjetura que todo conjunto bicromático determina una cantidad cuadrática de 3-hoyos.

Si un conjunto bicromático no contiene 4-hoyos monocromáticos, tampoco puede contener 7-hoyos, ya que cualquier heptágono vacío tendría al menos 4 puntos del mismo color sin importar la coloración del conjunto, determinando así un 4-hoyo monocromático. Por ello, los conjuntos de Horton son buenos candidatos para intentar probar que existen conjuntos bicromáticos arbitrariamente grandes sin 4-hoyos monocromáticos, pero en [DHKS03] se demuestra que cualquier bicoloración de un conjunto de Horton con 64 o más puntos determina un 4-hoyo monocromático.

Esto lleva a la conjetura de que todo conjunto bicromático suficientemente grande contiene algún 4-hoyo monocromático. En [DHKS03] los autores presentaron un conjunto bicoloreado de 18 puntos sin 4-hoyos monocromáticos. Desde entonces se han encontrado conjuntos más grandes, los más recientes se encontraron en 2009: Huemer y Seara encontraron un conjunto de 36 puntos con estas características [HS09] y poco después Koshelev encontró uno con 46 puntos [Kos09a]. Este conjunto se ilustra en la Figura 1.7.



| Azules | Rojos |
|---|---|
| (159, 645) | (226, 610) |
| (256, 594) | (279, 599) |
| (321, 571) | (328, 569) |
| (329, 613) | (340, 591) |
| (335, 570) | (407, 759) |
| (401, 763) | (420, 757) |
| (413, 757) | (426, 539) |
| (418, 550) | (429, 762) |
| (420, 537) | (440, 624) |
| (425, 759) | (457, 618) |
| (433, 538) | (691, 126) |
| (448, 620) | (704, 9) |
| (465, 620) | (707, 129) |
| (698, 6) | (718, 11) |
| (698, 128) | (718, 212) |
| (729, 210) | (734, 5) |
| (712, 11) | (735, 213) |
| (715, 126) | (741, 196) |
| (726, 9) | (820, 179) |
| (811, 160) | (826, 137) |
| (827, 180) | (834, 179) |
| (875, 150) | (899, 155) |
| (928, 139) | (983, 111) |

Figura 1.7: El conjunto de 46 puntos sin 4-hoyos monocromáticos encontrado por Koshelev

# Capítulo 2

# El algoritmo de Dobkin, Edelsbrunner y Overmars

Una vez definido lo que es un $r$-hoyo, surge el problema de encontrar los $r$-hoyos en un conjunto de puntos en el plano. De manera precisa:

> Sea $S$ un conjunto de $n$ puntos en el plano en posición general. Dado un entero positivo $r$, $3 \leq r \leq n$, encontrar todos los $r$-hoyos en $S$.

Dobkin, Edelsbrunner y Overmars [DEO90] presentaron en 1990 un algoritmo para resolver este problema, el cual se expone en este capítulo.

El conjunto de todos los $r$-hoyos de $S$ se denota por $\Gamma_r(S)$, una solución a este problema se llama una enumeración del conjunto $\Gamma_r(S)$. La cardinalidad de este conjunto se representa por $\gamma_r(S)$. Un algoritmo trivial para resolver el problema consiste en:

1. Enumerar todos los subconjuntos $U$ de $S$ con $r$ elementos $[O(n^r)]$
2. Determinar si los puntos de $U$ son vértices de $conv(U)$ $[O(r \log r)]$
3. Determinar si algún punto de $S \backslash U$ se encuentra en el interior de $conv(U)$ $[O(n \log r)]$

La complejidad de dicho algoritmo es $O(n^{r+1} \log r)$, la del algoritmo expuesto en este capítulo encuentra $\Gamma_r(S)$ en tiempo proporcional a $\gamma_r(S)$ cuando $r = 3, 4$ y en tiempo proporcional a $\gamma_3(S) + r\gamma_r(S)$ cuando $r \geq 5$. La salida puede ser tan grande como $\binom{n}{r}$

(cuando los puntos de $S$ se encuentran en posición convexa) o tan pequeña como 0 de acuerdo a los valores que tomen $n$ y $r$.

Sea $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ el conjunto con $n$ puntos en el plano en posición general del cual se quieren encontrar los $r$-hoyos, con $x_i \neq x_j \; \forall \; i, j \in 1, 2, \ldots, n$. Para cada punto $p \in S$, se localizan los $r$-hoyos que tienen como vértice más a la izquierda a $p$. De esta manera, cada $r$-hoyo se cuenta sólo una vez.

El algoritmo para encontrar $\Gamma_r(S)$ tiene 3 pasos:

1. Para cada $p \in S$, ordenar los demás elementos de $S$ alrededor de $p$ por ángulo, obteniendo así el conjunto ordenado $S_p$. Eliminar de $S_p$ los puntos a la izquierda de $p$ y agregar $p$. Esto da como resultado el polígono estrellado $P_p$, y $p$ pertenece a su kernel.

2. Para cada $p \in S$, calcular la gráfica de visibilidad $VG_p$ dentro de $P_p$, incluyendo las aristas de $P_p$ e ignorando las aristas que involucran a $p$.

3. Para cada $p \in S$, calcular todas las cadenas convexas en $VG_p$ con $r - 2$ aristas. Cada una de estas forma, junto con $p$, un $r$-hoyo. [*]

El paso 1 (ordenar todos los puntos de $S$ alrededor de cada punto en $S$) se puede completar en $O(n^2 \log n)$ usando algoritmos de ordenación bien conocidos. Usando los resultados de [CGL85] y [EOS83], se puede hacer en $O(n^2)$. Eliminar los puntos que se encuentran a la izquierda de $p$ para obtener $P_p$ puede hacerse en $O(n^2)$. En las siguientes secciones se detallan los pasos 2 y 3.

Que el algoritmo es correcto se sigue de las siguientes observaciones:

1. Cualquier $r$-hoyo tiene un único vértice que se encuentra más a la izquierda ya que ninguna línea vertical pasa por un par de puntos de $S$.

2. El hecho de que $r$ puntos formen un $r$-hoyo cuyo vértice más a la izquierda es $p$ es independiente de los puntos a la izquierda de $p$.

3. Cualquier $r$-hoyo cuyo vértice más a la izquierda es $p$ debe encontrarse dentro de $P_p$ y sus lados son aristas de $VG_p$.

[*] Las definiciones de *polígono estrellado, kernel* y *gráfica de visibilidad* se encuentran en el Apéndice A.

4. El punto $p$ es visible desde cualquier punto del polígono $P_p$ ya que pertenece a su kernel. Así, si $p_1, \ldots, p_{r-1}$ forman una cadena en $VG_p$, $p, p_1, \ldots, p_{r-1}$ es un polígono de $r$ lados sin ningún punto de $S$ en su interior.

5. Si $p_1, \ldots, p_{r-1}$ es una cadena convexa en $VG_p$, por lo argumentado en el punto 4 $p, p_1, \ldots, p_{r-1}$ es un $r$-hoyo.

Dados 3 puntos $p_i = (x_i, y_i), p_j = (x_j, y_j), p_k = (x_k, y_k)$, el algoritmo usa como operación básica determinar si los segmentos $\overline{p_i p_j}$ y $\overline{p_j p_k}$ forman una vuelta izquierda o derecha en el punto $p_j$. El procedimiento para este cálculo se basa en el Lema 2.1.

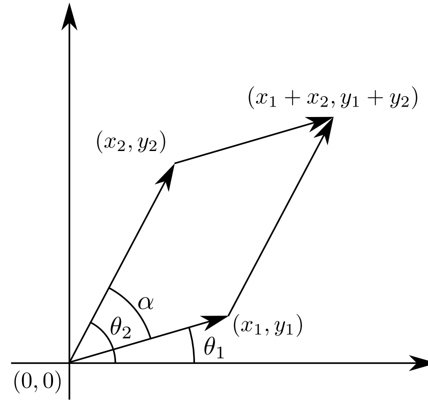



Figura 2.1: El paralelogramo formado por $O$, $p$, $q$ y $p + q$.

**Lema 2.1.** *Sean $p = (x_1, y_1)$ y $q = (x_2, y_2)$ puntos en el plano y $O = (0,0)$ con $p \neq q \neq O$. El determinante*

$$\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1 y_2 - x_2 y_1$$

*representa el área dirigida del paralelogramo cuyos vértices son O, p, $p + q$ y q. Es positivo si y solo si p esta en sentido de las manecillas del reloj de q respecto al origen, negativo si y solo si p esta en sentido contrario a las manecillas del reloj de q respecto al origen y es* 0 *si y solo su p, q y O son colineales.*

*Demostración.* Utilizando coordenadas polares, $x_1 = r_1 \cos \theta_1$ y $y_1 = r_1 \operatorname{sen} \theta_1$, donde $r_1 = \left|\overline{Op}\right|$ y $\theta_1$ es el ángulo que se forma entre $\overline{Op}$ y la parte positiva del eje $x$. Entonces el paralelogramo tiene área

$$A = \left|\overline{Op}\right| \left|\overline{Oq}\right| \operatorname{sen} \alpha \tag{2.1}$$

donde $\alpha = \angle pOq$ (Figura 2.1).

Nótese que $\alpha$ es positivo a menos que $p$ esté en sentido contrario a las manecillas del reloj de $q$ respecto al origen, por lo que en la ecuación (2.1) $A$ tendrá valor positivo si y sólo si $p$ está en sentido de las manecillas del reloj de $q$, tendrá valor negativo si y sólo si $p$ está en sentido contrario de las manecillas del reloj de $q$, y será 0 si y sólo si $\operatorname{sen}\alpha = 0$. Como $p$, $q$ y $O$ son distintos, $\operatorname{sen}\alpha = 0$ sólo si $\alpha = 0$ o $\alpha = 180$, es decir, si los 3 puntos son colineales.

Utilizando coordenadas polares para representar a $q$, $x_2 = r_2 \cos\theta_2$ y $y_2 = r_2 \operatorname{sen}\theta_2$, donde $r_2 = \left|\overline{Oq}\right|$ y $\theta_2 = \theta_1 + \alpha$. Entonces:

$$\begin{aligned}
A &= \left|\overline{Op}\right| \left|\overline{Oq}\right| \operatorname{sen}\alpha \\
&= \left|\overline{Op}\right| \left|\overline{Oq}\right| \operatorname{sen}(\theta_2 - \theta_1) \\
&= \left|\overline{Op}\right| \left|\overline{Oq}\right| (\operatorname{sen}\theta_2 \cos\theta_1 - \cos\theta_2 \operatorname{sen}\theta_1) \\
&= \left|\overline{Op}\right| \operatorname{sen}\theta_2 \left|\overline{Oq}\right| \cos\theta_1 - \left|\overline{Oq}\right| \cos\theta_2 \left|\overline{Op}\right| \operatorname{sen}\theta_1 \\
&= x_1 y_2 - x_2 y_1 \\
&= \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}
\end{aligned}$$

Lo cual completa la prueba. ■

Para determinar el sentido de la vuelta de $\overline{p_i p_j}$ y $\overline{p_j p_k}$ en el punto $p_j$, basta con tomar $p_i$ como el origen y verificar si $p_k$ está en sentido de las manecillas del reloj de $p_j$ usando el Lema 2.1 (Figura 2.2). Esto queda expresado en el Algoritmo 2.1.

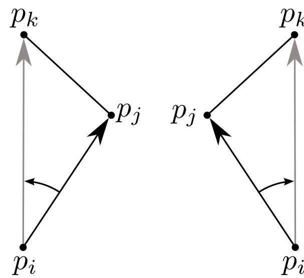



Figura 2.2: Determinar el sentido de una vuelta.

**Algoritmo 2.1** Algoritmo para calcular el sentido de una vuelta

1: **function** TURN($\overline{p_i p_j}$, $\overline{p_j p_k}$)
2: $t = (x_k - x_i) \cdot (y_j - y_i) - (x_j - x_i) \cdot (y_k - y_i)$
3: **if** $t < 0$ **then**
4: **return** *LEFT*
5: **else if** $t > 0$ **then**
6: **return** *RIGHT*
7: **else**
8: **return** *COLLINEAR*
9: **end if**
10: **end function**

## 2.1. La Gráfica de Visibilidad de un Polígono Estrellado

Ahora se tiene un polígono estrellado $P_p$ con $N$ vértices y un vértice $p$ que pertenece a su kernel. El objetivo de esta sección es obtener la gráfica de visibilidad de $P_p$, denotada $VG_p$.

Los vértices de $P_p$ se encuentran ordenados por ángulo alrededor de $p$, numerados $p_1, p_2, \ldots, p_{N-1}$. $VG_p$ se construye como una gráfica dirigida donde una arista en $VG_p$ parte de $p_i$ a $p_j$ si $i < j$; dicha arista se denota $p_i p_j$. En $VG_p$ se incluyen las aristas de $P_p$ pero no las aristas que involucran a $p$ (Figura 2.3).

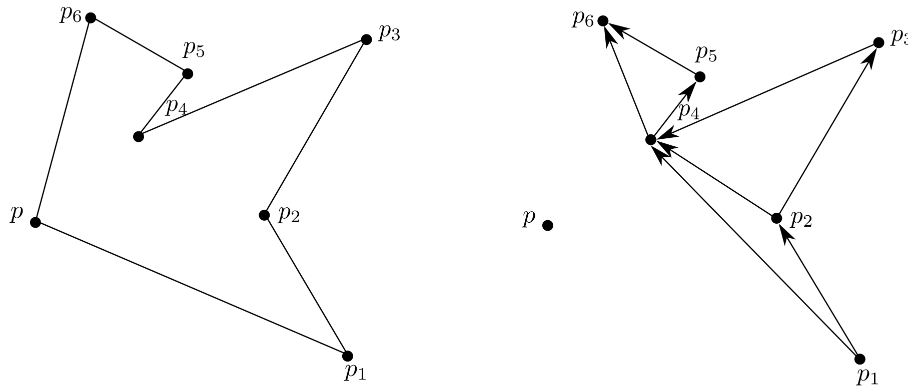



Figura 2.3: El polígono estrellado y la gráfica de visibilidad correspondientes al punto $p$.

El algoritmo para calcular $VG_p$ se basa en el siguiente Lema.

**Lema 2.2.** *Sean $i, j \in \{1, 2, \ldots, N-1\}$. $p_i p_j \in VG_p$ si, y sólo si $j = i + 1$ o existe un vértice $p_k$, $i < k < j$, tal que $p_i p_k \in VG_p$, $p_k p_j \in VG_p$ y $p_i, p_k$ y $p_j$ forman una vuelta izquierda.*

*Demostración.* Nótese que $p_i p_j \in VG_p$ si, y sólo si $\triangle pp_i p_j$ está vacío: Supongamos que $p_i p_j \in VG_p$ y que $\triangle pp_i p_j$ no está vacío y sea $p_m$ un punto en su interior. Por la manera en que se construyó $P_p$, debe suceder que $i < m < j$, pero entonces $\overline{p_i p_j}$ no está contenido en $P_p$, lo cual es una contradicción. Por otro lado, como $p$ pertenece al kernel de $P_p$, si $\triangle pp_i p_j$ está vacío $\overline{p_i p_j}$ debe estar contenido en $P_p$, por lo que $p_i p_j$ pertenece a la gráfica de visibilidad.

Ahora, para probar la parte "sólo si" del teorema, supongamos que $p_i p_j \in VG_p$. Si $j \neq i+1$, sea $p_k$ $(i < k < l)$ el punto más cercano al segmento $\overline{p_i p_j}$. Ya que $\triangle pp_i p_j$ está vacío, $p_i, p_k$ y $p_j$ forman una vuelta izquierda. Entonces tanto $\triangle pp_i p_k$ como $\triangle pp_k p_j$ son vacíos. Se sigue que $p_i p_k \in VG_p$ y $p_k p_j \in VG_p$.

Resta probar la otra implicación. Si $j = i+1$, $\triangle pp_i p_j$ es vacío y $p_i p_j \in VG_p$. Supongamos entonces que $j > i+1$ y existe $p_k$, $i < k < j$, tal que $p_i p_k \in VG_p$, $p_k p_j \in VG_p$ y $p_i, p_k$ y $p_j$ forman una vuelta izquierda. Entonces $\triangle pp_i p_k$ y $\triangle pp_i p_k$ están vacíos y $p_k$ se encuentra a la derecha de $\overline{p_i p_j}$. Así, $\triangle pp_i p_j$ debe estar vacío, por lo que $p_i p_j \in VG_p$. ■

Se construye $VG_p$ a través de un recorrido de $P_p$ en sentido contrario a las manecillas del reloj respecto a $p$, iniciando en $p_1$. El Lema 2.2 da las condiciones que deben verificarse para agregar una arista a la gráfica.

Cuando se visita $p_i$, se construyen todas sus aristas entrantes. Para hacer esto, se mantiene una cola $Q_l$ para cada vértice $p_l$, $l \leq i$. $Q_l$ guarda, en orden contrario al sentido de las manecillas del reloj, los vértices $p_j$ tales que $p_j p_l$ es una arista de $VG_p$ y aún no se alcanza un punto $p_k$ $(l < k)$ que forme una arista de $VG_p$ junto con $p_j$. $Q_l$ contiene puntos desde donde $p_l$ es visible pero no han formado aristas nuevas porque $p_l$ los bloquea. El Algoritmo 2.2 detalla esto.

PROCEED añade una arista que parte de $p_i$ hacia $p_j$ después de verificar si algún punto en $Q_i$ es visible desde $p_j$. Si esto sucede, se llama de manera recursiva. Ya que los puntos en $Q_i$ se encuentran ordenados en sentido contrario a las manecillas del reloj sólo se necesita examinar una porción de la cola: si $p_l$ está al frente de la cola y $p_l p_k p_i$ forman una vuelta derecha, cualquier vértice que esté después de $p_l$ en la cola formará también una vuelta derecha.

Nótese que $p_i p_j$ se añade una vez que finalizan todas las llamadas recursivas, lo cual garantiza que los puntos en las colas estén ordenados en sentido contrario a las manecillas del reloj. De igual manera, para todo vértice se guardan las aristas entrantes y salientes en sentido contrario a las manecillas del reloj.

**Algoritmo 2.2** Cálculo de la Gráfica de visibilidad

```
1: procedure VISIBILITY
2:     for i = 1 to N − 1 do
3:         Q_i = ∅
4:     end for
5:     for i = 1 to N − 2 do
6:         PROCEED(i, i + 1)
7:     end for
8: end procedure
9: procedure PROCEED(i, j)
10:    while Q_i ≠ ∅ and TURN(Q_i.front(),p_i,p_j)==LEFT do
11:        PROCEED(Q_i.front(), p_j)
12:        Q_i.dequeue()
13:    end while
14:    VG_p.add(p_i p_j)
15:    Q_j.enqueue(p_i)
16: end procedure
```

**Lema 2.3.** *Toma tiempo* $O(\left|VG_p\right|)$ *calcular la gráfica de visibilidad de* $P_p$.

*Demostración.* Esto se sigue del hecho de que con cada llamada a PROCEED, se añade una arista a $VG_p$. ■

## 2.2. Encontrando la Cadena Convexa más Larga

Ahora, dada $VG_p$, el objetivo es encontrar la cadena convexa más larga. Esto es equivalente a encontrar el subconjunto convexo más grande fijando un punto como su vértice más a la izquierda. En el algoritmo, a cada arista $e$ de $VG_p$ se le etiqueta con la longitud de la cadena convexa más larga que comienza con $e$ y va en sentido contrario a las manecillas del reloj. Esta etiqueta se denota $L_e$ (Figura 2.4).

Para lograrlo se tratan los vértices en sentido de las mancillas del reloj, comenzando por el vértice con índice más grande. Supongamos que nos encontramos en el vértice $q$. Sean $\{i_1, i_2, \ldots, i_{imax}\}$ las aristas entrantes y $\{o_1, o_2, \ldots, o_{omax}\}$ las aristas salientes de $q$, ordenadas en sentido contrario a las manecillas del reloj (tal como las da el Algoritmo 2.2). Se tratan las aristas entrantes en orden inverso, comenzando por $i_{imax}$. Para esta arista, se verifican todas las aristas salientes con las que forma un ángulo convexo. Sean

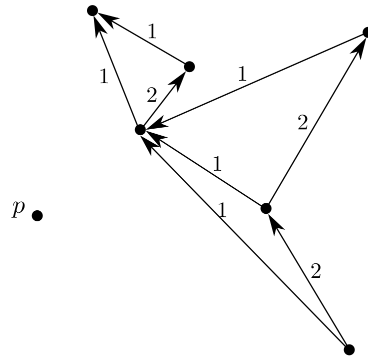



Figura 2.4: La gráfica de visibilidad con las etiquetas $L_e$ correspondientes a cada arista.

$O = \{o_l, \ldots, o_{omax}\}$ el conjunto ordenado de dichas aristas. Si $O = \emptyset$, $L_{i_{imax}} = 1$ de otra manera, sea $m = max\{L_o | o \in O\}$. Entonces $L_{i_{imax}} = m + 1$.

Debido al orden en que se encuentran, todas las aristas salientes que forman un ángulo convexo con $i_j$ forman un ángulo convexo con $i_{j-i}$. Por ello, al momento de examinar $i_{j-1}$ no es necesario procesarlas de nuevo: la longitud de la cadena convexa más larga que comienza en $i_{j-1}$ es $m+1$ o existe una arista saliente $o_k$ con $k < l$ y $L_{o_k} > m$. Así, comenzando en $l-1$ se examinan las aristas salientes que forman un ángulo convexo con $i_{j-1}$; $l$ se convertirá en el nuevo índice mínimo y $m$ el nuevo valor máximo $L_e$. (Figura 2.5).

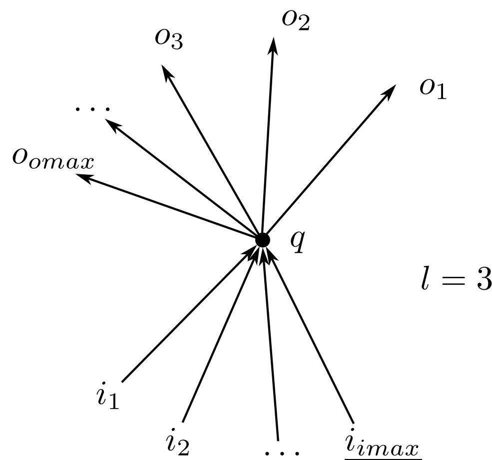



Figura 2.5: Al analizar aristas entrantes con índice menor a $i_{imax}$, sólo se procesarán aristas salientes con índice menor a 3.

Se continúa de esta manera con los vértices restantes. Nótese $L_e$ ya ha sido calculado para las aristas salientes del vértice que se está analizando.

El Algoritmo 2.3 precisa lo anterior. Que es correcto se sigue de lo argumentado en los

párrafos anteriores. En el algoritmo, para una arista $e$, $e[0]$ denota el vértice inicial de la arista y $e[1]$ denota el vértice final.

**Algoritmo 2.3** Algoritmo para encontrar la cadena más larga en $VG_p$

```
1:  procedure MAX_CHAIN
2:      for k = N − 1 down to 1 do
3:          Sean {i_1, ..., i_imax} las aristas entrantes de p_k,
4:          y {o_1, ..., o_omax} las aristas salientes de p_k ordenadas
5:          en sentido contrario a las manecillas del reloj
6:          l = omax
7:          m = 0
8:          for j = imax down to 1 do
9:              L_{i_j} = m + 1
10:             while l > 0 and TURN(i_j[0], p_k, o_l[1])==LEFT do
11:                 if L_{o_l} > m then
12:                     m = L_{o_l}
13:                     L_{i_j} = m + 1
14:                 end if
15:                 l = l − 1
16:             end while
17:         end for
18:     end for
19: end procedure
```

En el algoritmo, $l$ es el índice más chico tal que $o_l$ no forma una vuelta izquierda con la arista entrante que se examina y por tanto $o_l$ es la primera arista que se debe analizar al pasar a la siguiente arista entrante.

**Lema 2.4.** *Calcular el valor de todos los campos $L$ toma tiempo $O(\left|VG_p\right|)$.*

*Demostración.* Por cada vértice de $P_p$, las aristas entrantes se examinan una vez. Para cada arista entrante, se examinan las aristas salientes a partir de $o_l$, de donde cada arista saliente se trata una vez.

Así, las aristas de $VG_p$ se examinan exactamente 2 veces, la cota se sigue ya que $\left|VG_p\right|$ es mayor o igual que la cantidad de vértices de $P_p$. ∎

## 2.3. Reportando los $r$-hoyos

Ahora tenemos, para cada arista en $VG_p$ la longitud de la cadena convexa más larga que comienza ahí. Usando esta información, ahora se determinan todas las cadenas convexas de longitud $r-2$. Cada una de estas cadenas forma, junto con $p$, un $r$-hoyo. Las cadenas se determinan durante un recorrido de los vértices de $P_p$ en sentido contrario a las manecillas del reloj.

Para cada arista $e$ de $VG_p$, se mantiene un conjunto $C_e$ de todas las cadenas de longitud menor a $r-2$ que terminan en $e$ y que pueden extenderse a una cadena de longitud $r-2$ ($L_e$ determina si esto es posible). Las cadenas se guardan como listas de puntos, junto con su tamaño. En el algoritmo se asume que una cadena $ch$ tiene los siguientes métodos y se utilizan las siguientes operaciones:

1. ch.Length(): regresa la longitud de $ch$ $[O(1)]$
2. ch.Extend($e$): extiende la cadena $ch$ con la arista $e$ $[O(1)]$
3. ch.Report(): reporta $ch$ como respuesta [Lineal en el tamaño de $ch$]
4. new ch($e$): crea una cadena de longitud 1 que comienza con la arista $e$ $[O[1]]$

Para formar y extender las cadenas de manera eficiente, para cada punto $q$ de $P_p$ se ordenan las aristas salientes de manera decreciente por el valor de su etiqueta $L$. Ya que los valores de $L$ están entre 1 y $N-2$, se pueden ordenar con un radix sort global en tiempo $O(\left|VG_p\right|)$.

Sea $S_o{}' = \{o'_1, \ldots, o'_{omax}\}$ el conjunto de aristas salientes ordenadas de la manera descrita. Como en la sección anterior, $S_o = \{o_1, o_2, \ldots, o_{omax}\}$ es el conjunto de aristas salientes y $S_i = \{i_1, i_2, \ldots, i_{imax}\}$ el conjunto de aristas entrantes, ordenadas en sentido contrario a las manecillas del reloj. El Algoritmo 2.4 da el proceso para reportar los $r$-hoyos de $S$.

El procedimiento TREAT crea los conjuntos de cadenas para cada arista saliente de $q$. Al momento de visitar un punto, todas sus aristas entrantes tienen sus conjuntos de cadenas construidos. Para cada arista saliente con etiqueta $L \geq r-2$ se crea una cadena que consiste sólo de esa arista. Debido al valor de $L$, sabemos que se puede extender a una cadena de longitud $r-2$.

Después se extienden las cadenas de las aristas entrantes moviéndolas a las aristas salientes apropiadas. Esto se hace borrando todas las aristas salientes que no forman una

**Algoritmo 2.4** Algoritmo para encontrar los $r$-hoyos

1: **procedure** TREAT($q$)
2: Sean $S_i = \{i_1, i_2, \ldots, i_{imax}\}$ el conjunto de aristas entrantes de $q$,
3: $S_o = \{o_1, o_2, \ldots, o_{omax}\}$ el conjunto de aristas salientes de $q$,
4: y $S_o{}' = \{o'_1, \ldots, o'_{omax}\}$ el conjunto de aristas salientes de $q$
5: ordenadas por campo $L$
6: **for** $j = 1$ **to** $omax$ **do**
7: **if** $L_{o_j} \geq r-2$ **then**
8: $C_{o_j} = \{$ **new** $ch(o_j)\}$
9: **else**
10: $C_{o_j} = \emptyset$
11: **end if**
12: **end for**
13: $m = 1$
14: **for** $j = 1$ **to** $imax$ **do**
15: **while** $m \leq omax$ **and** TURN($i_j[0], q, o_m[1]$)==*RIGHT* **do**
16: Borrar $o_m$ de $S'_o$
17: $m = m+1$
18: **end while**
19: **for each** $ch \in C_{i_j}$ **do**
20: $t = 1$
21: $l = ch$.length()
22: **while** $t \leq S'_o$.size() **and** $L_{o'_t} \geq r-2-l$ **do**
23: $ch' = ch$; $ch'$.extend($o'_t$)
24: **if** $l == r-3$ **then**
25: $ch'$.report()
26: **else**
27: $C_{o'_t} = C_{o'_t} \cup \{ch'\}$
28: **end if**
29: $t = t+1$
30: **end while**
31: **end for**
32: **end for**
33: **end procedure**
34: **procedure** CHAINS
35: **for** $i = 1$ **to** $N-2$ **do**
36: TREAT($q_i$)
37: **end for**
38: **end procedure**

vuelta izquierda con la arista entrante que se está examinando. Debido al orden en el que están almacenadas, las aristas borradas tampoco forman una vuelta izquierda con las aristas entrantes que faltan de procesar.

Cada cadena $ch$ de la arista entrante actual se extiende con todas las aristas salientes posibles (existe al menos una, ya que ése es el criterio para crear las cadenas). Para extenderla se procesan las aristas salientes en orden decreciente en cuanto al campo $L$. Mientras el valor de dicho campo sea mayor o igual a $r-2-ch.length()$, se extiende $ch$ con esa arista. Si se obtiene una cadena con longitud $r-2$ se reporta, si no, se guarda en la arista saliente.

**Lema 2.5.** *Reportar las cadenas de longitud $r$ toma tiempo y espacio $O(\left|VG_p\right|+rk)$ donde $k$ es el número de cadenas reportadas.*

*Demostración.* Para cada punto $q$ se realiza lo siguiente:

1. Inicializar los conjuntos de cadenas para cada arista saliente, lo cual toma $O(\left|VG_p\right|)$ en total.
2. Por cada arista entrante, se eliminan algunas aristas salientes. Ya que cada arista saliente se elimina a lo más una vez, esto toma $O(\left|VG_p\right|)$ en total.
3. Para cada cadena en una arista entrante, se tratan de encontrar aristas con las cuales pueda ser extendida y sabemos que debe existir al menos una. Por cada cadena, se usa tiempo proporcional al número de aristas de este tipo encontradas, por lo que se usa hasta $O(rk)$ tiempo.

■

Es sencillo modificar el algoritmo 2.4 para encontrar $r$-hoyos monocromáticos en conjuntos coloreados: sólo se necesita verificar que la arista con la que se crea una cadena tenga vértices extremos del mismo color y siempre se debe extender una cadena con puntos del mismo color que el último que tiene.

## 2.4. La complejidad del algoritmo

Antes de dar la complejidad total del algoritmo, se presenta una cota inferior de $\gamma_4(S)$ en términos de $\gamma_3(S)$ y una cota inferior de $\gamma_3(S)$.

**Lema 2.6.** *Sea $S$ un conjunto de puntos en posición general en el plano.* $\gamma_4(S) \geq \gamma_3(S) - \binom{n-1}{2}$

*Demostración.* Sea $p$ un punto de $S$. Por cada arista $e = p_i p_j$ de $VG_p$ que no es también una arista de $P_p$ existe al menos un cuadrilátero convexo con $e$ como diagonal: existe al menos un punto de $S$ a la derecha de $e$ cuya coordenada $y$ queda entre las coordenadas $y$ de $p_i$ y $p_j$. A la vez, $p_i$ y $p_j$ forman un triángulo con $p$. Entonces, por cada triángulo vacío (excepto los definidos por $p$ y una arista de $P_p$) existe al menos un cuadrilátero convexo vacío y ninguno se cuenta más de una vez.

Sean $q_1, q_2, \ldots q_n$ los puntos de $S$ ordenados por su coordenada en $x$. Entonces $VG_{q_i}$ tiene $n - i - 1$ aristas que pertenecen también a $P_{q_i}$ para $i \in \{1, 2, \ldots, n-2\}$. La cantidad total de este tipo de aristas es:

$$\sum_{k=1}^{n-2} k = \frac{(n-2)(n-1)}{2} = \frac{(n-1)!}{2(n-3)!} = \binom{n-1}{2}$$

Como cada arista de $VG_p$ forma un triángulo con $p$, de acuerdo con el párrafo anterior se tiene $\gamma_4(s) \geq \gamma_3(S) - \binom{n-1}{2}$. ■

Es fácil dar una cota superior para $\gamma_3(S)$: si los puntos de $S$ forman los vértices de un polígono convexo, $\gamma_3(S) = \binom{n}{3}$. La cota inferior que se da en el Lema 2.7 se debe a Katchalski [KM88].

**Lema 2.7.** *Sea $S$ un conjunto de $n$ puntos en posición general en el plano.* $\gamma_3(S) \geq \frac{(n-2)(n-1)}{2}$

*Demostración.* Si $|S| = 3$ la desigualdad se cumple. Sea entonces $|S| > 3$ y supongamos que la desigualdad se cumple para cualquier conjunto de menor cardinalidad. Sea $S' = \{p_1, \ldots p_n\}$ el conjunto con los puntos de $S$ ordenados de manera creciente por su coordenada $x$. Por cada arista de $P_{p_1}$ (sin contar las aristas que tienen como extremo a $p_1$) se tiene un triángulo vacío. Así, $S'$ tiene al menos $n-2$ triángulos con $p_1$ como vértice. Si se elimina $p_1$ de $S'$ no se generan triángulos vacíos nuevos, por lo que $\gamma_3(S') \geq (n-2) + \gamma_3(S \setminus \{p_1\})$. Usando la hipótesis de inducción, se tiene:

$$\gamma_3(S) = \gamma_3(S') \geq (n-2) + \frac{(n-3)(n-2)}{2} = \frac{(n-2)(n-1)}{2}$$

■

**Teorema 2.1.** *Dado un conjunto $S$ de $n$ puntos en el plano en posición general, es posible obtener una enumeración de $\Gamma_r(S)$ en tiempo y espacio $O(\gamma_3(S) + r\gamma_r(S))$. Para $r = 3, 4$, esto se simplifica a $O(\gamma_r(S))$.*

*Demostración.* Se pueden ordenar todos los puntos de $S$ alrededor de cada punto en $S$ en $O(n^2)$ usando los resultados de [CGL85] y [EOS83]. No es necesario incluir esto en la complejidad, ya que $\gamma_3(S) = \Omega(n^2)$. Para cada $p$ en $S$, toda arista de $VG_p$ forma un triángulo con $p$, por lo que $\sum_{p \in S} \left|VG_p\right| = \gamma_3(S)$.

Juntando las complejidades obtenidas en los Lemas 2.2, 2.4 y 2.5, la complejidad del algoritmo es $O(\gamma_3(S) + r\gamma_r(S))$. Ya que $\gamma_4(S)$ es al menos proporcional a $\gamma_3(S)$, para $r = 3, 4$ la complejidad es $O(\gamma_r(S))$. ■

# Capítulo 3

# Modificaciones al Algoritmo

Ahora, con el objetivo de encontrar conjuntos de puntos sin $r$-hoyos, se considera el problema de calcular los $r$-hoyos de un conjunto cuando un punto cambia de posición.

Sea $S$ un conjunto de $n$ puntos en posición general en el plano y $p$ un punto que no está en $S$ tal que $S_p = S \cup \{p\}$ se encuentra en posición general. Dado un entero $r \geq 3$, se denota $H_r(S_p)$ como la cantidad de $r$-hoyos en $S_p$, $A_r(S_p)$ como la cantidad de $r$-hoyos en $S_p$ que tienen a $p$ como vértice y $B_r(S_p)$ como la cantidad de $r$-gonos de $S$ que tienen sólo a $p$ en su interior.

Si se cambia la posición de $p = (x, y)$, $H_r(S_p)$ puede tomar un valor distinto. Supongamos que esto sucede y sea $q = (x', y')$ la nueva posición de $p$. Los $r$-gonos de $S_p$ que contenían sólo a $p$ en su interior son ahora $r$-hoyos en $S_q = S \cup \{q\}$ y se generan $r$-hoyos en $S_q$ que contienen a $q$ como vértice. Por otro lado, los $r$-hoyos que tienen a $p$ como vértice en $S_p$ no pertenecen a $S_q$ y se generan $r$-gonos en $S_q$ con sólo $q$ en su interior. Así, se tiene que

$$H_r(S_q) = H_r(S_p) + A_r(S_q) - A_r(S_p) + B_r(S_p) - B_r(S_q)$$

El objetivo de este capítulo es presentar modificaciones hechas al algoritmo de Dobkin, Edelsbrunner y Overmars con el fin de calcular $A_r(S_p)$ y $B_r(S_p)$. Nótese que si $p$ es un vértice de $conv(S \cup \{p\})$, entonces $B_r(S_p) = 0$ y $A_r(S_p)$ se puede calcular con el algoritmo del Capítulo 2.

En general, las modificaciones son:

1. Ordenar los elementos de $S$ alrededor de $p$ por ángulo, obteniéndose el conjunto $S'$.

Este conjunto da un polígono estrellado $P_p$, y $p$ pertenece a su kernel.

2. Calcular la gráfica de visibilidad $VG_p$ dentro de $P_p$, incluyendo las aristas de $P_p$ e ignorando las aristas que involucran a $p$.

3. Calcular todas las cadenas convexas en $VG_p$ con de longitud $r-2$ que forman un $r$-hoyo con $p$ como vértice.

4. Calcular las cadenas de longitud $r-1$ forman un $r$-gono con sólo $p$ en su interior.

Es posible encontrar $S' = \{p_0, \ldots, p_{n-1}\}$ en $O(n \log n)$. Se puede verificar si $p$ es un vértice de $conv(S \cup \{p\})$ calculando el sentido de las vueltas de $p_i p p_{(i+1) \pmod n}$ para cada punto $p_i$ de $S'$: si alguna vuelta es izquierda, $p$ es un vértice de $conv(S \cup \{p\})$. En adelante se asume que esto no sucede.

## 3.1. Calculando la Gráfica de Visibilidad

Para poder encontrar las cadenas que den los valores de $A_r(S_p)$ y $B_r(S_p)$ es necesario calcular la gráfica de visibilidad de $P_p$, denotada $VG_p$. Para conservar la estructura de las gráficas de visibilidad obtenidas en el capítulo 2, las aristas se orientan de tal forma que si $p_i p_j$ es una arista de $VG_p$, $p_i p_j p$ forma una vuelta izquierda.

Las aristas donde intervienen puntos que se encuentran a la derecha de $p$ se pueden calcular usando el mismo procedimiento que en el Algoritmo 2.2, pero no se pueden calcular correctamente las demás aristas ya que el Lema 2.2 no se cumple: por ejemplo, en la Figura 3.1 tanto $p_1 p_4$ como $p_4 p_5$ pertenecen a $VG_p$ y forma una vuelta izquierda, pero $p_1 p_5 \notin VG_p$. Además, pueden existir aristas $p_i p_j \in VG_p$ con $i > j$.

Por ello, es necesario agregar otra condición para encontrar las aristas de $VG_p$ para tomar en cuenta casos como el anterior. Obsérvese que, como en el capítulo anterior, $p_i p_j \in VG_p$ si y sólo si el triángulo $p p_i p_j$ es vacío.

**Lema 3.1.** *Sean $i, j \in \{0, 1, ..., n-1\}$. $p_i p_j \in VG_p$ si y sólo si $j \equiv (i+1) \pmod n$ o $p p_i p_j$ forman una vuelta izquierda y existe $k \in \{(i+1) \pmod n, (i+2) \pmod n, \ldots, j-1\}$ tal que $p_i p_k \in VG_p, p_k p_j \in VG_p$ y $p_i p_k p_j$ forman una vuelta izquierda.*

*Demostración.* Sólo es necesario demostrar que $p_i p_j \in VG_p$ si y sólo si $p p_i p_j$ es una vuelta izquierda ya que, salvo los cambios en la notación para tomar en cuenta las aristas que

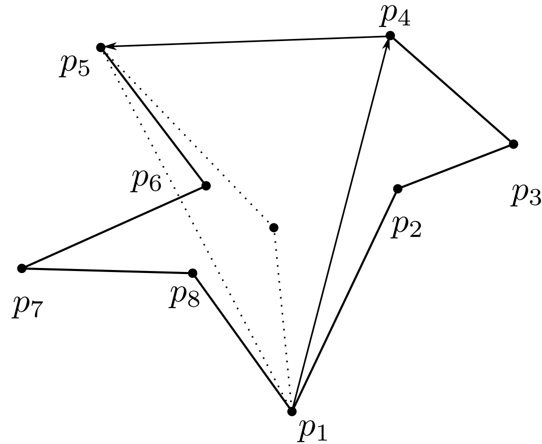



Figura 3.1: El Lema 2.2 no es cierto al calcular la gráfica de visibilidad alrededor de un punto.

ahora pueden tener vértice final con índice mayor al de su vértice inicial, la prueba es idéntica a la del Lema 2.2.

El que $p_i p_j \in VG_p$ implica que $pp_i p_j$ es una vuelta izquierda, se sigue de la manera en la que se orientan las aristas de la gráfica ($p$ queda a la izquierda de la arista).

En cuanto a la otra implicación, como $p_i p_k, p_k p_j \in VG_p$, los triángulos $pp_i p_k$ y $pp_k p_j$ son vacíos. Más aún, tanto $p_i p_k p_j$ como $pp_i p_j$ son vueltas izquierdas, por lo cual el triángulo $pp_i p_j$ es vacío. De ello se concluye que $p_i p_j \in VG_p$. ■

Así, se debe verificar una condición más al momento de llamar a *PROCEED* (Algoritmo 3.1).

$VG_p$ se construye mediante un recorrido de los vértices de $P_p$ en sentido contrario a las manecillas del reloj más un recorrido de los puntos que se encuentran a la derecha de $p$. Para cada $p_i$ se mantiene una lista $Q_i$ que cumple la misma función de las colas en el Algoritmo 2.2, guardar vértices que ven a $p_i$ en $VG_p$ pero que no se han vuelto a observar.

PROCEED añade la arista $p_i p_j$ y revisa si los vértices en la lista correspondiente cumplen con las condiciones del Lema 3.1. De ser así, se llama recursivamente. Los puntos que se encuentran a la derecha de $p$ se recorren sin añadir las aristas correspondientes, pero se guarda la información de los vértices que los ven en las listas (línea 21). De esta manera, al procesar a los puntos a la izquierda de $p$ se construyen todas sus aristas entrantes. Se usan listas porque pueden existir vértices en la parte inicial que no cumplan con la nueva condición del Lema 3.1, seguidos de vértices que si la cumplan. En la Figura 3.2 se

**Algoritmo 3.1** Algoritmo para construir $VG_p$

```
1: procedure PROCEED(i,j, first)
2:     k = 0
3:     while k < Q_i.size() and TURN(Q_i[k],p_i,p_j)==LEFT do
4:         if TURN(p,Q_i[k], j==LEFT) then
5:             PROCEED(Q_i[k], j, first)
6:             Q_i.delete(k)
7:         else
8:             k = k + 1
9:         end if
10:    end while
11:    Q_j.add(i)
12:    if first ≠ true then
13:        VG_p.add(ij)
14:    end if
15: end procedure
16: procedure VISIBILITY
17:    Right=cantidad de puntos de S a la derecha de p
18:    for i = 0 to n − 1 do
19:        Q_i = ∅
20:    end for
21:    for i = 0 to Right-2 do
22:        PROCEED(i,i + 1,true)
23:    end for
24:    for i = Right − 1 to n − 1 do
25:        PROCEED(i,i + 1,false)
26:    end for
27:    outgoing=∅
28:    for i = 0 to Right-1 do
29:        outgoing.add({aristas salientes de p_i})
30:        Borrar de VG_p las aristas salientes de p_i
31:        Q_i = ∅
32:    end for
33:    PROCEED(n − 1, 0, false)
34:    for i = 0 to Right-2 do
35:        PROCEED(i,i + 1,false)
36:    end for
37:    for i = 0 to Right-1 do
38:        Agregar a VG_p las aristas salientes de p_i guardadas en outgoing
39:    end for
40: end procedure
```

muestra un ejemplo de ello, la arista $p_1p_4 \notin VG_p$ pero $p_2p_4 \in VG_p$. Nótese que al momento de llamar `PROCEED(3,4)` las aristas punteadas no han sido añadidas a $VGp$.

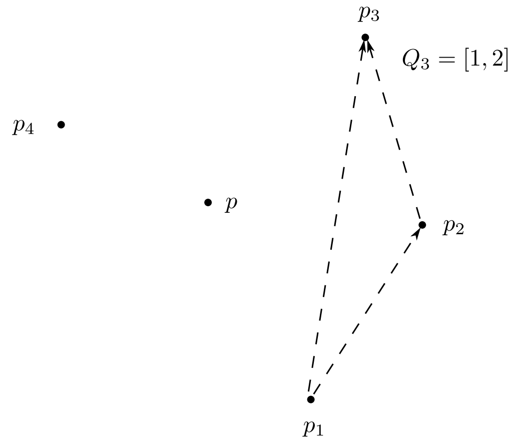



Figura 3.2: Uso de listas en vez de colas al almacenar los vértices.

Una vez completado el primer recorrido, `outgoing` guarda las aristas salientes de los puntos a la derecha de $p$. Estas aristas se añaden al final para preservar el orden (las aristas entrantes y salientes de cada punto se guardan en sentido contrario a las manecillas del reloj). Sólo resta construir las aristas que faltan de los puntos a la derecha de $S$ (línea 34).

Debido a que se añade cada arista de $VG_p$ a lo más dos veces, la complejidad del Algoritmo 3.1 es $O(\left|VG_p\right|)$. Esto es lo mismo que la cantidad de triángulos vacíos en $S_p$ que tienen a $p$ como vértice.

## 3.2. Cadenas convexas

Ahora se buscan cadenas convexas adecuadas para obtener $A_r(S_p)$ y $B_r(S_p)$. La idea principal se mantiene sin cambios: primero, para cada arista $e$ de $VG_p$, se encuentra la longitud de la cadena convexa más larga que comienza en $e$ y van en sentido contrario a las manecillas del reloj ($L_e$). A diferencia de las gráficas de visibilidad del capítulo 2, $VG_p$ contiene al menos un ciclo (las aristas de $P_p$), por lo que es necesario definir un punto donde terminen las cadenas.

Después se buscan cadenas convexas de longitud $r-2$ que forman junto con $p$ un $r$-hoyo y cadenas de longitud $r-1$ que forman, al unir sus puntos inicial y final, un $r$-gono con sólo $p$ en su interior. En adelante se referirá a estas como cadenas de tipo 1 y de tipo 2, respectivamente.

Sean $q$ y $q'$ puntos con la misma coordenada $x$ que $p$ tales que la coordenada $y$ de $q$ es mayor que la de cualquier punto de $S$ y la coordenada $y$ de $q'$ es menor que la de cualquier punto de $S$. Para que el cálculo de las etiquetas $L_e$ se pueda hacer en forma análoga a como se realiza en el Algoritmo 2.3, se fuerza a que las etiquetas de las aristas que cruzan $\overline{pq}$ tomen el valor de cero. Se inicia el recorrido en el punto con índice más grande que se encuentra a la derecha de $p$. Las etiquetas $L_e$ se calculan de la misma manera que en el algoritmo original, excepto para las aristas que cruzan $\overline{pq}$. A cada paso del algoritmo se calculan las etiquetas $L_e$ de las aristas entrantes al punto que se está tratando, pero pueden haber aristas salientes que aún no tengan $L_e$ calculado. Por ello es necesario verificar su existencia.

Con las etiquetas $L$ calculadas de esta forma, no se tiene información de las cadenas en las que intervienen aristas que cruzan $\overline{pq}$. Para obtener esta información, se calcula un segundo conjunto de etiquetas, denotadas $L'_e$. Para calcular estas etiquetas, se fuerza ahora a que las aristas que cruzan $pq'$ tomen valores de cero y se comienza el recorrido en el punto con índice más grande de $S'$. Un ejemplo de los valores que toman las etiquetas de cada conjunto se muestra en la Figura 3.3.

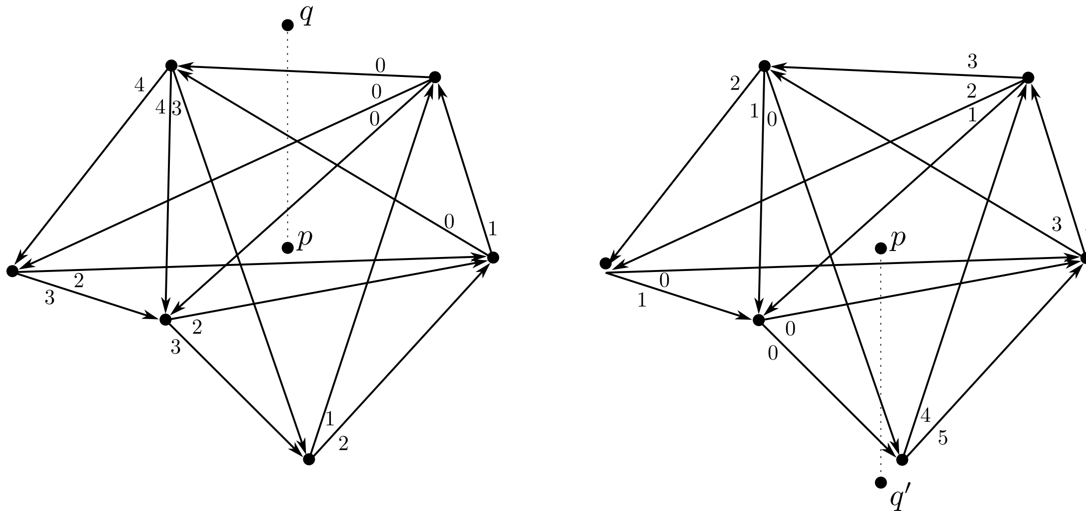



Figura 3.3: Los valores de las etiquetas $L$ (izquierda) y $L'$ (derecha).

Las modificaciones discutidas dan como resultado el Algoritmo 3.2.

Ahora, a partir de los conjuntos de etiquetas, se encuentran las cadenas de tipo 1 y 2.

Si $ch = \{p_{k_1}, p_{k_2}, \ldots, p_{k_m}\}$ es una cadena de tipo 1, cruza $\overline{pq}$, $\overline{pq'}$ o ningún segmento, pero no ambos; si fuera así, $p_{k_m} p p_{k_1}$ formaría una vuelta derecha y la cadena no podría formar un $r$-hoyo con $p$. Por otro lado, si $ch$ es una cadena de tipo 2, el triángulo $p p_{k_1} p_{k_m}$

**Algoritmo 3.2** MAX_CHAIN

```
1:  procedure MAX_CHAIN
2:      i = índice más grande de los puntos a la derecha de p
3:      for k in {i,...,0} ∪ {N−1,...,i+1} do
4:          Sean {i_0,...,i_imax} las aristas entrantes de p_k,
5:          y {o_0,...,o_omax} las aristas salientes de p_k
6:          ordenadas en sentido contrario a las manecillas del reloj
7:          l = omax
8:          m = 0
9:          for j = imax down to 0 do
10:             L_{i_j} = m + 1
11:             if p_{i_j}p_k cruza pq then
12:                 L_{i_j} = 0
13:             end if
14:             while l ≥ 0 and TURN(i_j[0], p_k, o_l[1])==LEFT do
15:                 if L_{o_l} ya fue calculado and L_{i_j} > 0 and L_{o_l} > m then
16:                     m = L_{o_l}
17:                     L_{i_j} = m + 1
18:                 end if
19:                 l = l − 1
20:             end while
21:         end for
22:     end for
23: end procedure
```

debe ser vacío y por tanto $p_{k_1}p_{k_m}$ pertenece a $VG_p$. Además el ciclo $p_{k_1},\ldots,p_{k_m},p_{k_1}$ cruza $\overline{pq}$ y $\overline{pq'}$ de no ser así, $p$ no estaría en el interior del $r$-gono que forma.

Sea $P$ un $r$-gono en $S\cup\{p\}$ con sólo $p$ en su interior. Al quitarle el lado que cruza $\overline{pq}$ se obtiene una cadena convexa de longitud $r-1$ y esta cadena se puede encontrar con los valores $L$ obtenidos en este primer recorrido. Así, con la información que se tiene hasta el momento, se pueden encontrar todas las cadenas de tipo 2 y en consecuencia, $B_r(S_p)$. También es posible encontrar las cadenas de tipo 1 que cruzan sólo $\overline{pq}$ y las que no cruzan ni $\overline{pq}$ ni $\overline{pq'}$.

Hace falta encontrar las cadenas de tipo 1 que cruzan sólo $\overline{pq'}$, es aquí donde se utilizan las etiquetas $L'_e$. En este caso se buscan cadenas de longitud $r-2$ que no crucen $\overline{pq}$.

Las modificaciones para reportar las cadenas con la información que se tiene son menos que las necesarias para el cálculo de la $VG_p$ y los valores $L_e$. Los cambios son:

- Se buscan cadenas en 3 ocasiones: en la primera se buscan cadenas de tipo 1 y en la segunda se buscan cadenas de tipo 2, ambas con el conjunto de valores $L$. En la tercera ocasión se buscan cadenas de tipo 1 con el conjunto de valores $L''$. Todas las búsquedas se hacen mediante un recorrido de los puntos en orden inverso al seguido en el Algoritmo 3.2.

- Sea $ch = \{p_{k_1}, \ldots, p_{k_{r-2}}\}$. Para que $ch$ sea una cadena de tipo 1 se debe cumplir que $p_{k_1}p_{k_{r-1}} \in VG_p$: esto implica que $p_{k_{r-2}}p_{k_{r-1}}p$, $p_{k_{r-1}}pp_{k_1}$ y $pp_{k_1}p_{k_2}$ son vueltas izquierdas. Esto es necesario ya que pueden ocurrir casos como el de la Figura 3.4 (a).

- Sea $ch = \{p_{k_1}, \ldots, p_{k_{r-1}}\}$. Para que $ch$ sea una cadena de tipo 2 se deben cumplir dos condiciones. La primera es $p_{k_{r-1}}p_{k_1} \in VG_p$, esto implica que el triángulo $p_{k_{r-1}}p_{k_1}p$ es vacío. La segunda condición es que $p_{k_{r-2}}p_{k_{r-1}}p_{k-1}$ y $p_{k_{r-1}}p_{k_1}p_{k_2}$ deben ser vueltas izquierdas. Con esto se toma en cuenta casos como el de la Figura 3.4 (b).

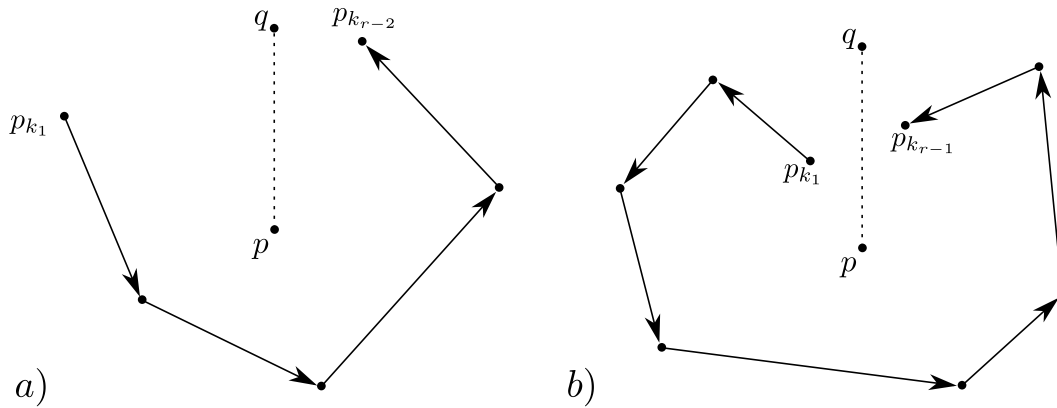



Figura 3.4: Casos que se deben verificar para las cadenas de tipo 1 y 2.

Los demás pasos del Algoritmo 2.4 quedan sin cambios. Una vez hechas estas modificaciones, la cantidad de cadenas encontradas de tipo 1 es $A_r(S_p)$ y la cantidad de cadenas encontradas de tipo 2 es $B_r(S_p)$.

La complejidad sigue dependiendo del tamaño de la gráfica de visibilidad, pero en este caso se calcula una sola gráfica, por lo que en la práctica es más rápido calcular $A_r(S_p)$ y $B_r(S_p)$ para calcular los $r$-hoyos cuando cambia la posición de $p$ que usar el algoritmo del Capítulo 2.

# Capítulo 4

# Implementación

En este capítulo se describe la implementación de los algoritmos expuestos en los capítulos 2 y 3.

Al implementar dichos algoritmos, se busca obtener un medio con el cual hacer búsquedas de conjuntos de puntos con pocos $r$-hoyos. El uso de un lenguaje compilado provee un mejor desempeño comparado a lenguajes interpretados; pero hacer pruebas, modificaciones o depurar el código es más complicado y toma más tiempo. Mediante la extensión de un intérprete es posible tener las ventajas de ambos tipos de lenguaje.

## 4.1. Elección de los lenguajes

Hay muchas posibilidades para realizar búsquedas de conjuntos con pocos $r$-hoyos, por lo que es deseable elegir un lenguaje en el cual sea sencillo modificar las estrategias de búsqueda. Los lenguajes interpretados proveen esta característica.

En el desarrollo de este trabajo, se optó por partir de conjuntos aleatorios de puntos y reducir el número de $r$-hoyos que contienen cambiando de posición elementos del conjunto. El algoritmo resultante de las modificaciones presentadas en el capítulo 3 permite decidir si la nueva posición de un punto da un conjunto con menos $r$-hoyos (conociendo la cantidad de $r$-hoyos que tiene el conjunto con el que se inicia). Por ello, es necesario implementar este algoritmo (y el del capítulo 2) de tal forma que se ejecuten rápidamente. En este caso un lenguaje interpretado no es muy útil; a pesar de que la codificación es sencilla, la ejecución es muy lenta para los propósitos de este trabajo. Así, es conveniente

usar un lenguaje compilado para esta parte.

Existe una gran cantidad de lenguajes interpretados. Lo que se busca en este caso, es un lenguaje con herramientas útiles para programar diversas estrategias de búsqueda de los conjuntos mencionados y con la capacidad de comunicarse de manera sencilla y eficiente con un lenguaje compilado.

Python es un lenguaje interpretado de alto nivel, con tipado dinámico, multiparadigma (orientado a objetos, funcional, imperativo) y cuenta con una sintaxis que produce códigos cortos y sencillos de leer que se asemejan mucho a un pseudocódigo. Estas características, junto con su extensa librería estándar han propiciado su uso en cómputo científico [Lan09].

La mayor desventaja de Python es su velocidad. Las pruebas realizadas en [Ful12] muestran que el tiempo de ejecución de un programa escrito en Python puede ser desde 4 hasta 77 veces mayor que su equivalente escrito en C++ (hasta 80 veces mayor, comparado con C). Cabe notar que la longitud del código en Python puede ser hasta $\frac{1}{7}$ de su equivalente en C/C++.

Existen implementaciones de Python en varios lenguajes de programación, en particular, CPython es la implementación en C. Es multiplataforma, de código abierto y gratuita, y permite el uso de código escrito en C/C++ a través del uso de librerías compartidas, lo cual lo hace muy conveniente para el presente trabajo.

En base a esto, la elección del lenguaje compilado queda entre C y C++. C es un lenguaje de propósito general, de alto nivel (también permite hacer operaciones de nivel bajo), imperativo, y de tipado estático. El primer estándar surgió en 1989 y la revisión más reciente se dio en 2011. C ha influenciado a varios lenguajes de programación posteriores, intérpretes de otros lenguajes (como Python, Perl y PHP) son a menudo implementados en C.

C++ desciende de C y ofrece compatibilidad para la mayor parte de los programas escritos en C. Es también de propósito general, alto nivel y de tipado estático. A diferencia de C, es multiparadigma (orientado a objetos, imperativo, genérico). Su desarrollo comenzó en 1979 y la última revisión fue en 2011. Una característica importante de C++ es su librería de plantillas estándar (STL por sus siglas en inglés) que provee de contenedores, iteradores y algoritmos.

El estándar más reciente (referido como C++11) contiene numerosos cambios, en particular, los que resultan más útiles para implementar los algoritmos de los capítulos anteriores son: la disponibilidad de nuevos contenedores en la STL que implementan tablas hash, el método `emplace` en los contenedores de la STL, que permite añadir objetos sin

hacer llamadas al constructor de copia y soporte para funciones lambda. Por estas características, se eligió a C++ sobre C.

## 4.2. Implementación en C++

En esta sección se detalla la precisión a usar para representar los puntos y las estructuras usadas al implementar los algoritmos.

Ya que el tamaño del tipo de dato entero más grande que se puede representar en C++ es 64 bits, es necesario limitar los valores que pueden tomar las coordenadas de los puntos a fin de evitar problemas desbordamiento en las operaciones del algoritmo 2.1. El entero más grande que se puede representar es $2^{63} - 1$ y el menor es $-2^{63}$. Si $2^k$ representa el valor más grande que se puede usar en el algoritmo 2.1, entonces

$$2^{2k+2} \quad < 2^{62} \quad < 2^{63} - 1$$

de donde es necesario restringir las coordenadas a no más de $2^{30}$ en valor absoluto para garantizar que el algoritmo 2.1 dé valores correctos. Python permite realizar operaciones con enteros de longitud arbitraria, pero este aumento en la precisión no parece ayudar en las búsquedas.

Cada punto queda representado por una estructura con tres enteros, dos para guardar las coordenadas del punto y uno para guardar el color del punto. Dados $n$ puntos $p_0, \ldots, p_{n-1}$, se forma un vector `sorted_points` que contiene $n$ vectores. El vector en la $i$-ésima posición contiene los puntos a la derecha de $p_i$, ordenados por ángulo a su alrededor.

Las gráficas de visibilidad se mantienen también en vectores. Para cada punto $p_i$, se mantiene un vector $V_{p_i}$. En la $j$-ésima posición de $V_{p_i}$ se guarda un par de vectores, el primero con los índices de los puntos que forman una arista entrante del $j$-ésimo punto a la derecha de $p_i$ y el segundo con los índices de los puntos que forman una arista saliente del $j$-ésimo punto a la derecha de $p_i$. Se eligieron los vectores porque brindan acceso aleatorio a sus elementos, lo cual es útil en el algoritmo 2.4.

En la implementación del algoritmo 2.3, los campos $L_e$ se guardan por medio de una tabla hash donde la llave es un par de enteros (índices dentro del vector correspondiente en `sorted_points`) y el valor es $L_e$. C++ dispone del contenedor `map`, que permite asociar una llave con un valor. Está implementado como un árbol rojo-negro, con lo que insertar

y buscar un valor tiene complejidad $O(\log n)$. Esto resulta muy lento. En C++11 se define un nuevo tipo de contenedor, `unordered_map`. Al igual que `map`, permite asociar una llave con un valor, pero está implementado como una tabla hash. `unordered_map` provee especializaciones de funciones hash para tipos de datos básicos, pero no para pares de enteros, que es lo que se desea hacer.

Como los valores de entrada de la función hash son pares de enteros no negativos, se puede usar una función de emparejamiento. Esta es una función hash perfecta (es decir, sin colisiones). La función

$$f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$f(x, y) := \begin{cases} y^2 + x & : x \geq y \\ x^2 + x + y & : x < y \end{cases}$$

es una de tales funciones. En la implementación, esta función provee un desempeño ligeramente menor que

$$g(x, y) := (x << 16) \quad XOR \quad y$$

donde *XOR* es la operación *OR* exclusivo a nivel de bits y << es el desplazamiento de bits hacia la izquierda. Esta es una función hash perfecta siempre que $x, y \leq 2^{16}$.

## 4.3. Extendiendo el intérprete de Python

Existen 2 maneras en las que Python y C/C++ pueden interactuar, denominadas *incrustar* y *extender*. Incrustar Python se refiere a la inserción de llamadas para inicializar el intérprete Python y llamadas a código de Python dentro de una aplicación de C/C++. Extender se refiere a la creación de una librería compartida escrita en C++ que el intérprete de Python puede cargar como un módulo cualquiera.

A pesar de que los propósitos son distintos, extender e incrustar Python son actividades muy similares. De manera general, el proceso que se lleva a cabo cuando se utilizan funciones de una librería compartida es:

1. Convertir tipos de datos de Python a tipos de datos de C/C++
2. Realizar una llamada a una rutina escrita en C/C++ con los valores convertidos
3. Convertir los tipos de datos que regresa la rutina de C/C++ a tipos de datos de Python

Cuando se incrusta Python, se realizan los mismos pasos en orden inverso (en el paso 2 se llama a una rutina escrita en Python).

Lo que se desea hacer es extender el intérprete de Python para poder hacer uso de los algoritmos implementados en C++. Para hacer esto, es necesario crear funciones *wrapper* que se encargan del proceso descrito en el párrafo anterior. Dichas funciones hacen uso de la interfaz que ofrece Python (Python/C API), en donde se define un conjunto de funciones, macros y variables que proveen acceso al entorno de Python [Pyt12]. El siguiente código ilustra una función en C y su correspondiente wrapper:

```
//Función en C, calcula
//el máximo común divisor de a y b
int gcd(int a, int b)
{
    if(b == 0)
        return a;
    return gcd(b, a%b);
}
//wrapper
static PyObject* gcd(PyObject* self, PyObject* args)
{
    int x, y;
    //Conversión de datos de Python a datos de C
    if (!PyArg_ParseTuple(args, "ii", &x, &y))
        return NULL;
    //Llamada a la función en C y conversión del
    //resultado a dato de Python
    return Py_BuildValue("i", gcd(x, y));
}
```

La dificultad de escribir el código para los wrappers, es que es muy extenso (se deben crear wrappers para cada función que se quiera utilizar desde Python) y propenso a errores, ya que se debe mantener un conteo de referencias de los objetos que expone Python. Existen varias herramientas que automatizan hasta cierto punto la generación de este código, como CXX, SCXX, SIP, SWIG y Boost.Python. Se probaron las 2 últimas en este trabajo.

Boost.Python forma parte de Boost, un conjunto de más de 80 librerías que extienden la funcionalidad de C++. La mayor parte de las librerías consisten en archivos de cabecera,

aunque algunas (entre ellas Boost.Python) deben ser compiladas de manera separada. Boost.Python provee funciones para exponer clases y funciones usando la API de Python y una herramienta llamada Bjam para compilar el código a una librería compartida.

SWIG (Simplified Wrapper and Interface Generator) es una herramienta que genera el código necesario para conectar C/C++ con 19 lenguajes de programación, entre ellos Python. Está implementado en C y C++ y es libre y gratuito. SWIG utiliza un archivo de interfaz en el cual recibe instrucciones para exponer funciones y objetos. A partir de dicho archivo, SWIG puede generar el código necesario para compilar la librería compartida.

Tanto Boost.Python como SWIG proveen soporte para exponer los contenedores de la STL a Python, y poseen una buena documentación. En las pruebas realizadas, las librerías generadas tienen el mismo rendimiento. Se eligió SWIG sobre Boost.Python porque el proceso de compilación no depende de librerías adicionales. Es posible utilizar módulo `distutils` de Python para generar scripts que automatizan la compilación de la librería.

Para finalizar este capítulo se presenta una comparación de los tiempos de ejecución de los algoritmos implementados en C++, en Python y llamados desde Python a la librería compartida generada con SWIG.

| | 6-hoyos | | | $A_6$ y $B_6$ | | |
|---|---|---|---|---|---|---|
| Tamaño de $S$ | C++ | L.C. | Python | C++ | L.C. | Python |
| 100 puntos | 0.02 | 0.04 | 0.25 | 0.01 | 0.02 | 0.13 |
| 500 puntos | 0.89 | 1.23 | 8.48 | 0.19 | 0.34 | 1.85 |
| 1000 puntos | 3.56 | 4.91 | 36.46 | 0.56 | 0.96 | 5.52 |
| 1500 puntos | 8.22 | 11.29 | 86.80 | 0.80 | 1.33 | 7.91 |

Tabla 4.1: Comparación de tiempos de ejecución en segundos.

En la tabla 4.1 se muestran los tiempos de ejecución en segundos. Los conjuntos $S$ de puntos se generaron utilizando el módulo `random` de Python, sus coordenadas toman valores entre $-2^{30}$ y $2^{30}$. Las primeras 3 columnas muestran el tiempo tomado para encontrar la cantidad de 6-hoyos en $S$. Las siguientes 3 columnas muestran el tiempo usado para encontrar $A_6(S_p)$ y $B_6(S_p)$, donde $p$ es el punto más cercano al origen.

El tiempo usado por la implementación en C++ es en promedio 10 veces menor que el tiempo usado por la implementación en Python tanto para la búsqueda de 6-hoyos como para el cálculo de $A_6(S_p)$ y $B_6(S_p)$. Por otro lado, al usar la librería compartida el tiempo usado en las mismas pruebas es 7 y 6 veces menor respectivamente que la implementación en Python.

El uso de la librería compartida resulta en pérdida de eficiencia en comparación al sólo uso de C++, pero las ganancias en cuanto a facilidades para usar los algoritmos programados son mucho mayores.

# Capítulo 5

# Resultados

En este capítulo se presentan los experimentos realizados con la implementación del algoritmo del capítulo 3 y los resultados obtenidos. Antes de ello se da una breve descripción de algoritmos de búsqueda local con base en [RN10].

## 5.1. Búsqueda Local

En general, algoritmos de búsqueda como DFS, BFS, A*, IDA*, etc., parten de un estado inicial y realizan una búsqueda sistemática en el espacio de estados del problema que se está resolviendo. Esto se consigue manteniendo registro de los estados que se han explorado y los caminos que se han seguido. Cuando se alcanza un estado óptimo, el camino seguido hasta él constituye una solución al problema.

Sin embargo, hay problemas en los que no es relevante el camino seguido para llegar a una solución óptima. Los algoritmos de búsqueda local, se mantienen en un sólo estado del problema y exploran vecinos de dicho estado. A pesar de que estos algoritmos no son sistemáticos, utilizan poca memoria y generalmente encuentran soluciones razonables en problemas que tienen espacios de estados muy grandes, donde un algoritmo de búsqueda completa no es factible.

*Hill Climbing* es un algoritmo de este tipo. Es simplemente un ciclo que se mueve a estados donde la función objetivo toma valores mejores que en el estado actual, va “colina arriba” (Figura 5.1). Termina cuando llega a una “cima”, a un estado donde no hay vecinos que mejoren la función objetivo. Típicamente el siguiente estado se escoge de manera

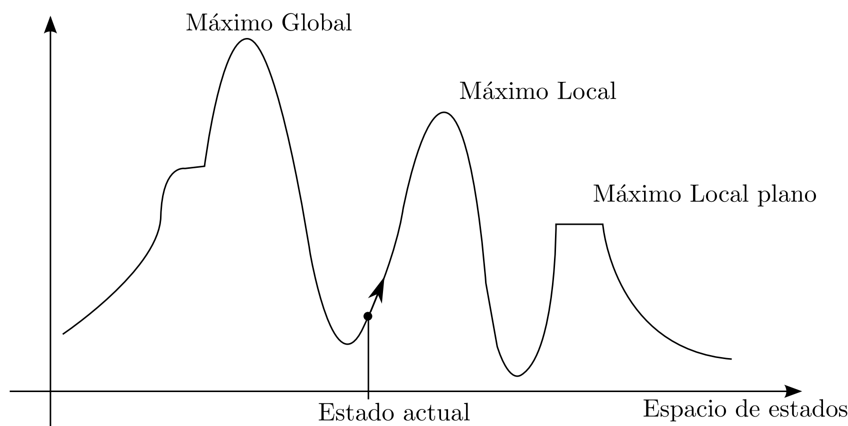



Figura 5.1: Hill-climbing.

aleatoria si se encuentran varios vecinos con valores igualmente buenos. Los algoritmos de tipo Hill-climbing generalmente encuentran estados mejores de manera muy rápida, pero pueden quedar atrapados en un máximo local (estados sin vecinos que mejoren la función objetivo) o en mesetas (estados donde todos los vecinos dan el mismo valor a la función objetivo). En ambos casos, el algoritmo llega a un punto donde no puede hacer mejoras. En las mesetas puede optarse por seguir explorando estados, aunque si es un máximo local plano no se conseguirán mejoras.

Existen numerosas variantes de este algoritmo. Una variante es escoger vecinos de manera aleatoria y escoger el primero que represente una mejora. Esta es una buena estrategia cuando un estado tiene muchos vecinos en el espacio de búsqueda [RN10].

Esta variante es la usada en los experimentos que se realizaron.

## 5.2. Experimentos y resultados

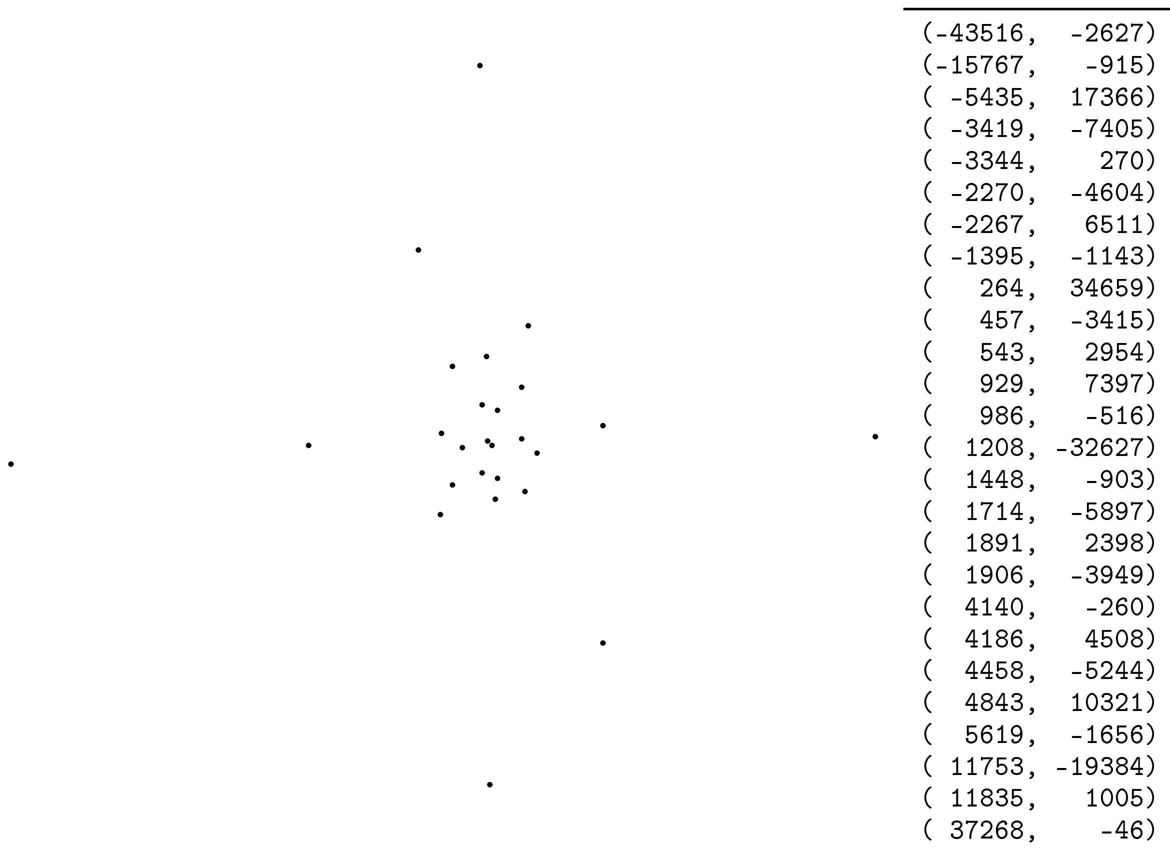
La búsqueda de conjuntos sin $r$-hoyos se realizó de la siguiente manera:

1. Generar un conjunto de $S$ con $n$ puntos en posición general y calcular $H_r(S)$. Las coordenadas de los puntos se generan de manera aleatoria.
2. Elegir de un punto $p$ de $S$ y eliminarlo de $S$. Calcular $A_r(S_p)$ y $B_r(S_p)$.
3. Hacer $q = (p_x + \Delta_x, p_y + \Delta_y)$ y calcular $A_r(S_q)$ y $B_r(S_q)$. Con estos valores, se obtiene la cantidad $H_r(S_q)$.

4. Si $H_r(S_q) = 0$ se ha obtenido un conjunto sin $r$-hoyos y la búsqueda se detiene. Si $H_r(S_q) \leq H_r(S_p)$, hacer $p = q$ e ir al paso 3.

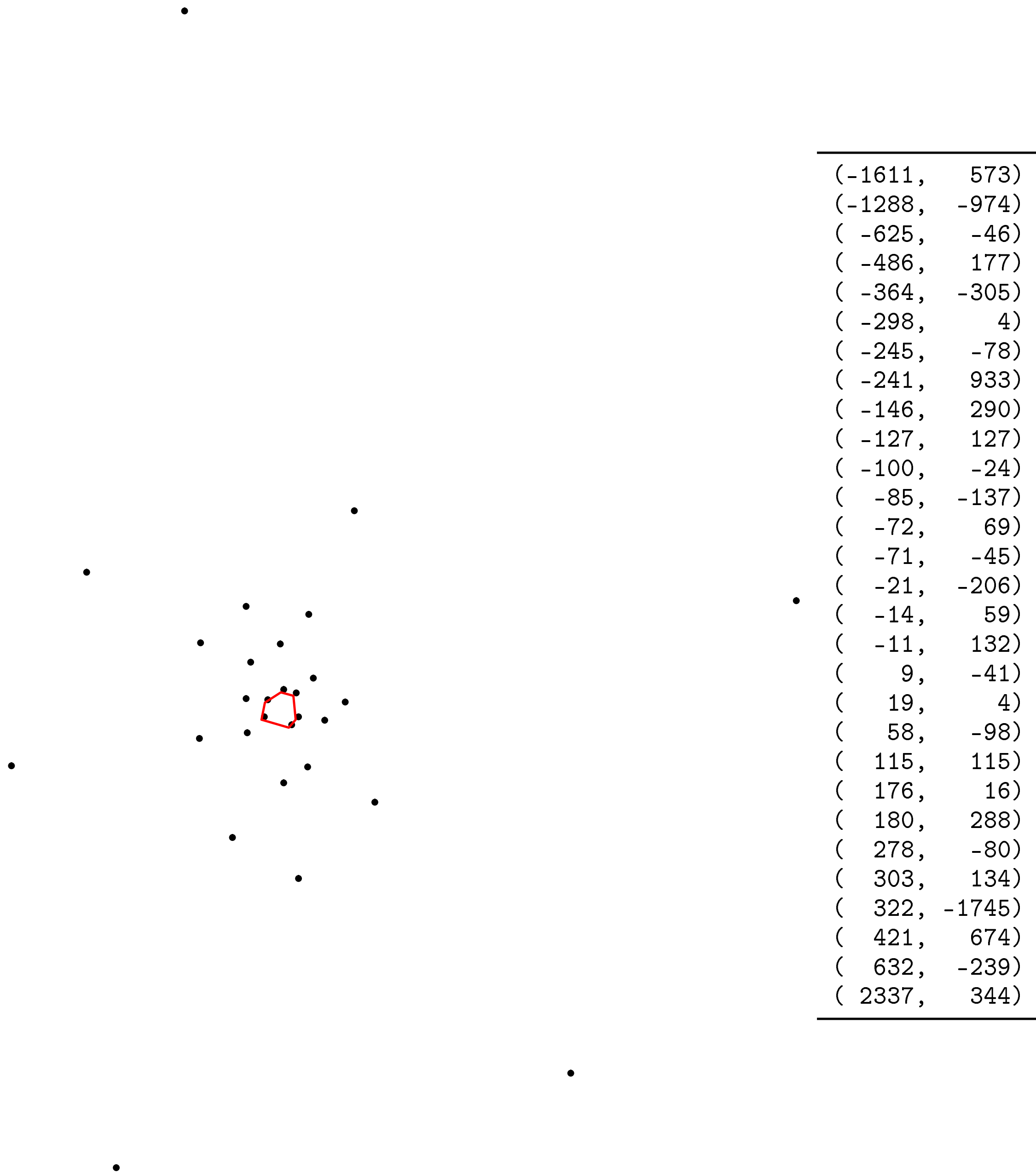
Los números $\Delta_x$ y $\Delta_y$ se generaron de manera aleatoria siguiendo una distribución exponencial. La razón para elegir esta distribución es tener la posibilidad de que las coordenadas de los puntos cambien muy poco en ciertas ocasiones y mucho en otras. Siguiendo estos pasos, se realizaron búsquedas de conjuntos sin 6-hoyos y conjuntos bicoloreados sin 4-hoyos monocromáticos.

Para los conjuntos sin 6-hoyos, se encontraron conjuntos de 26 puntos sin hexágonos vacíos (la misma cardinalidad del conjunto reportado en [OSV89]). No fue posible reproducir los resultados de [Ove03], aunque se encontró un conjunto de 29 puntos con sólo 1 hexágono vacío y un conjunto de 30 puntos con 2 hexágonos vacíos. Estos conjuntos se muestran en las Figuras 5.2, 5.3 y 5.4, respectivamente. En cuanto a 4-hoyos monocromáticos, no se logró reproducir el resultado de [Kos09a], pero se obtuvo un conjunto de 47 puntos con un 4-hoyo monocromático, del cual se puede obtener un conjunto de 41 puntos sin 4-hoyo monocromático. Este conjunto se muestra en la Figura 5.5.



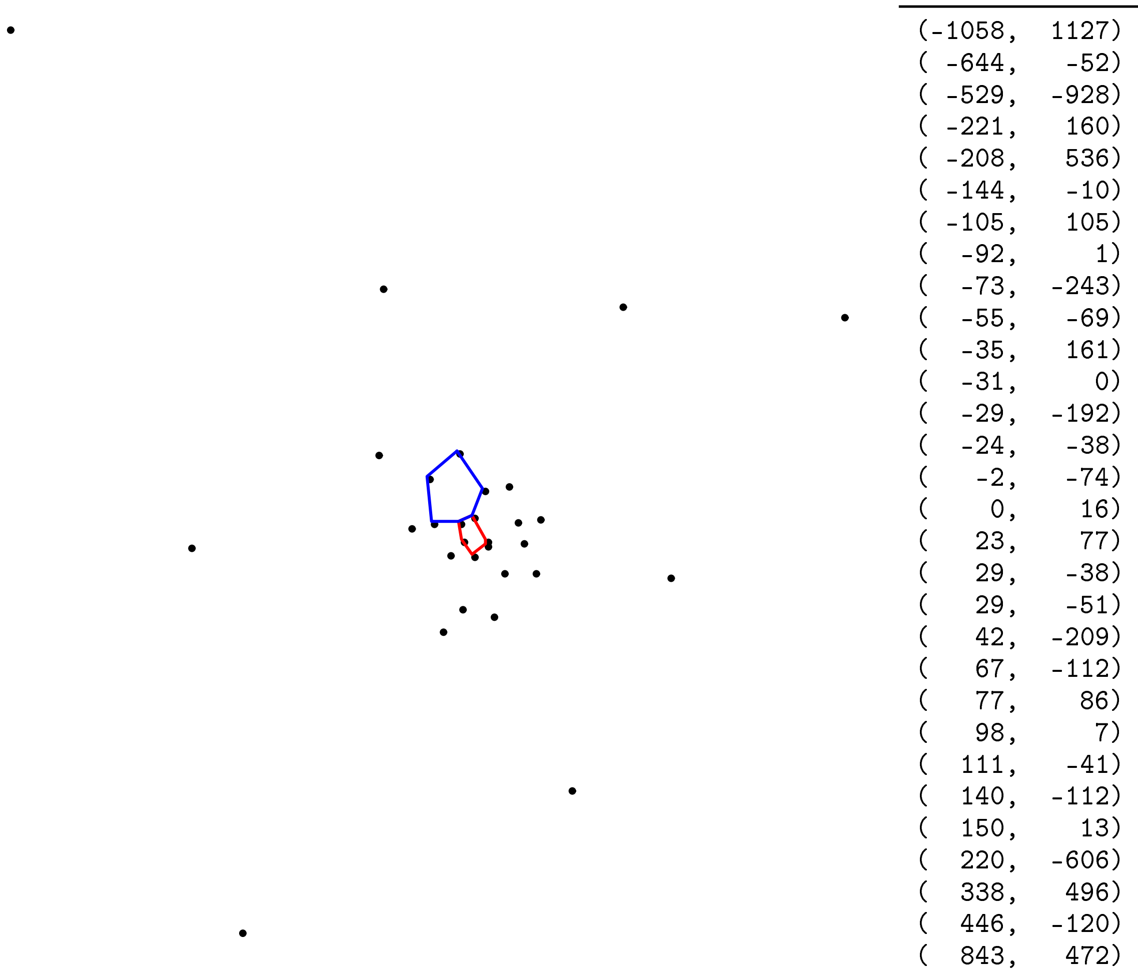
| | |
|---|---|
| (-43516, | -2627) |
| (-15767, | -915) |
| ( -5435, | 17366) |
| ( -3419, | -7405) |
| ( -3344, | 270) |
| ( -2270, | -4604) |
| ( -2267, | 6511) |
| ( -1395, | -1143) |
| ( 264, | 34659) |
| ( 457, | -3415) |
| ( 543, | 2954) |
| ( 929, | 7397) |
| ( 986, | -516) |
| ( 1208, | -32627) |
| ( 1448, | -903) |
| ( 1714, | -5897) |
| ( 1891, | 2398) |
| ( 1906, | -3949) |
| ( 4140, | -260) |
| ( 4186, | 4508) |
| ( 4458, | -5244) |
| ( 4843, | 10321) |
| ( 5619, | -1656) |
| ( 11753, | -19384) |
| ( 11835, | 1005) |
| ( 37268, | -46) |

Figura 5.2: Un conjunto de 26 puntos sin 6-hoyos.



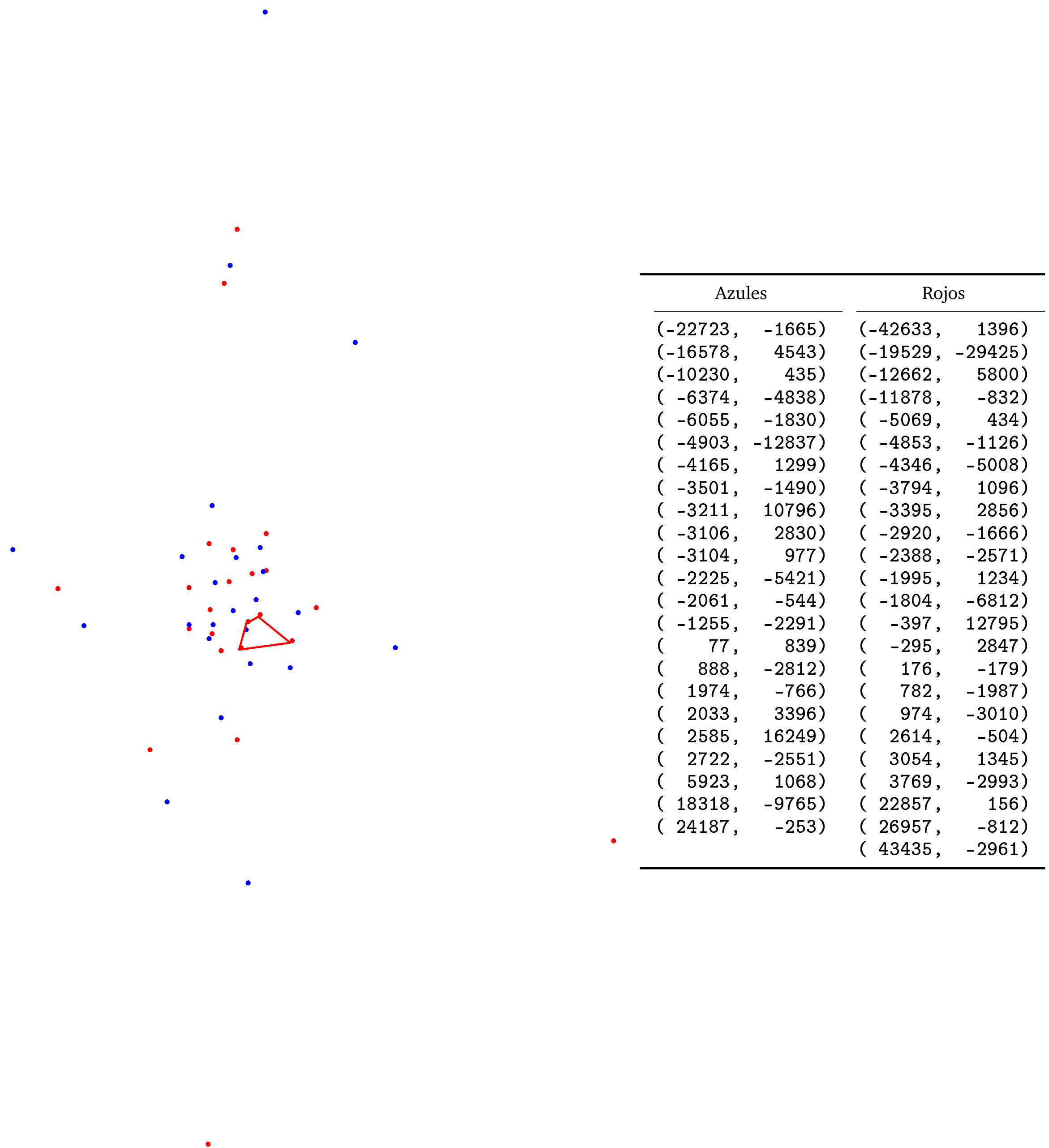
| | |
|---|---|
| (-1611, | 573) |
| (-1288, | -974) |
| ( -625, | -46) |
| ( -486, | 177) |
| ( -364, | -305) |
| ( -298, | 4) |
| ( -245, | -78) |
| ( -241, | 933) |
| ( -146, | 290) |
| ( -127, | 127) |
| ( -100, | -24) |
| ( -85, | -137) |
| ( -72, | 69) |
| ( -71, | -45) |
| ( -21, | -206) |
| ( -14, | 59) |
| ( -11, | 132) |
| ( 9, | -41) |
| ( 19, | 4) |
| ( 58, | -98) |
| ( 115, | 115) |
| ( 176, | 16) |
| ( 180, | 288) |
| ( 278, | -80) |
| ( 303, | 134) |
| ( 322, | -1745) |
| ( 421, | 674) |
| ( 632, | -239) |
| ( 2337, | 344) |

Figura 5.3: Un conjunto de 29 puntos con un 6-hoyo (rotado 90° a la izquierda).



| x | y |
|---|---|
| -1058 | 1127 |
| -644 | -52 |
| -529 | -928 |
| -221 | 160 |
| -208 | 536 |
| -144 | -10 |
| -105 | 105 |
| -92 | 1 |
| -73 | -243 |
| -55 | -69 |
| -35 | 161 |
| -31 | 0 |
| -29 | -192 |
| -24 | -38 |
| -2 | -74 |
| 0 | 16 |
| 23 | 77 |
| 29 | -38 |
| 29 | -51 |
| 42 | -209 |
| 67 | -112 |
| 77 | 86 |
| 98 | 7 |
| 111 | -41 |
| 140 | -112 |
| 150 | 13 |
| 220 | -606 |
| 338 | 496 |
| 446 | -120 |
| 843 | 472 |

Figura 5.4: Un conjunto de 30 puntos con dos 6-hoyos.



| Azules | Rojos |
|---|---|
| (-22723, -1665) | (-42633, 1396) |
| (-16578, 4543) | (-19529, -29425) |
| (-10230, 435) | (-12662, 5800) |
| ( -6374, -4838) | (-11878, -832) |
| ( -6055, -1830) | ( -5069, 434) |
| ( -4903, -12837) | ( -4853, -1126) |
| ( -4165, 1299) | ( -4346, -5008) |
| ( -3501, -1490) | ( -3794, 1096) |
| ( -3211, 10796) | ( -3395, 2856) |
| ( -3106, 2830) | ( -2920, -1666) |
| ( -3104, 977) | ( -2388, -2571) |
| ( -2225, -5421) | ( -1995, 1234) |
| ( -2061, -544) | ( -1804, -6812) |
| ( -1255, -2291) | ( -397, 12795) |
| ( 77, 839) | ( -295, 2847) |
| ( 888, -2812) | ( 176, -179) |
| ( 1974, -766) | ( 782, -1987) |
| ( 2033, 3396) | ( 974, -3010) |
| ( 2585, 16249) | ( 2614, -504) |
| ( 2722, -2551) | ( 3054, 1345) |
| ( 5923, 1068) | ( 3769, -2993) |
| ( 18318, -9765) | ( 22857, 156) |
| ( 24187, -253) | ( 26957, -812) |
| | ( 43435, -2961) |

Figura 5.5: Un conjunto bicoloreado de 47 puntos con un 4-hoyo monocromático (rotado 90° a la izquierda).

# Apéndice A

# Conceptos y notación

En este apéndice se presentan conceptos usados en este trabajo, así como su notación.

La frase "en posición general" se utiliza para decir que no hay coincidencias poco posibles. Dependiendo del contexto en el que se trabaje, estas coincidencias querrán decir algo distinto. En este trabajo, un conjunto $S$ de puntos en el plano se encuentra *en posición general* si para cualesquiera 3 puntos de $S$ no existe una línea que pase por ellos.

Una *gráfica* es un par ordenado $G = (V, E)$ de conjuntos, donde $E \subset V \times V$. Los elementos de $V$ se llaman vértices y los elementos de $E$ se llaman aristas. El conjunto de vértices de $G$ se denota $V(G)$ y el conjunto de aristas se denota $E(G)$. Si $\{x, y\}$ es un elemento de $E(G)$, generalmente se escribe $xy$. No se distingue estrictamente entre una gráfica y su conjunto de vértices o su conjunto de aristas. Por ejemplo, se puede hablar de un vértice $v$ de $G$ (en vez de un vértice $v$ de $V(G)$) o de una arista $xy$ de $G$.

Se dice que un conjunto $S$ de puntos en el plano es *convexo* si para cualesquiera 2 puntos de $S$, el segmento de línea que los une está totalmente contenido en $S$. La *cerradura convexa* de $S$ es la intersección de todos los conjuntos convexos que contienen a $S$ y se denota *conv(S)*.

Un *polígono* $P$ es una región cerrada del plano acotada por una colección finita de segmentos de línea que forman una curva cerrada que no se autointersecta. Los segmentos de línea se llaman *aristas* o *lados* y los extremos de los segmentos se llaman *vértices*. Se representa un polígono $P$ por un conjunto ordenado $\{v_1, \ldots, v_n\}$, donde cada $v_i$ es un vértice de $P$ y $\overline{v_i v_{i+1}}$ es una arista de $P$ para $i \in \{1, \ldots, n-1\}$, al igual que $\overline{v_n v_1}$.

Un punto $x$ en un polígono $P$ es *visible* desde un punto $y$ en $P$ si el segmento de línea

$\overline{xy}$ está contenido en $P$. Se dice que $P$ es un *polígono estrellado* si existe un punto $z$ en $P$ tal que $z$ es visible desde cualquier otro punto de $P$. El conjunto de todos los puntos que cumplen con esta propiedad se llama el *kernel* de $P$.

La *gráfica de visibilidad* de un polígono $P$ es la gráfica cuyo conjunto de vértices consta de los vértices de $P$ y cuyo conjunto de aristas consta de los pares de vértices de $P$ que son visibles entre sí.

# Bibliografía


[AFMFP+08] Oswin Aichholzer, Ruy Fabila-Monroy, David Flores-Peñaloza, Thomas Hackl, Clemens Huemer, and Jorge Urrutia. Empty monochromatic triangles. In *Proceedings of the 20th Canadian Conference on Computational Geometry (CCCG2008)*, pages 75–79, August 2008.

[CGL85] Bernard Chazelle, Leo J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25(1):76–90, June 1985.

[DEO90] David Dobkin, Herbert Edelsbrunner, and Mark Overmars. Searching for empty convex polygons. *Algorithmica*, 5:561–571, 1990.

[DHKS03] Olivier Devillers, Ferran Hurtado, Gyula Károlyi, and Carlos Seara. Chromatic variants of the Erdős-Szekeres theorem on points in convex position. *Comput. Geom.*, 26(3):193–208, 2003.

[EOS83] Herbert Edelsbrunner, Joseph O'Rourke, and Raimund Seidel. Constructing arrangements of lines and hyperplanes with applications. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, SFCS '83, pages 83–91, Washington, DC, USA, 1983. IEEE Computer Society.

[Erd78] P. Erdős. Some more problems on elementary geometry. *Austral. Math. Soc. Gaz.*, 5(2):52–54, 1978.

[Erd84] Paul Erdős. *Some old and new problems in combinatorial geometry*, volume 87 of *North-Holland Mathematics Studies*, pages 129–136. North-Holland, 1984.

[ES35] Paul Erdős and George Szekeres. A combinatorial problem in geometry. *Compositio Math.*, 2:463–470, 1935.

[ES61] Paul Erdős and George Szekeres. On some extremum problems in elementary geometry. *Ann. Univ. Sci. Budapest. Eötvös Sect. Math.*, 3(4):53–62, 1961.

[Ful12] Brent Fulgham. The computer language benchmarks game. `http://shootout.alioth.debian.org/`, September 2012.

[Ger08] Tobias Gerken. Empty convex hexagons in planar point sets. *Discrete Comput. Geom.*, 39(1-3):239–272, 2008.

[Har78] Heiko Harborth. Konvexe Fünfecke in ebenen Punktmengen. *Elem. Math.*, 33(5):116–118, 1978.

[Hor83] J. D. Horton. Sets with no empty convex 7-gons. *Canad. Math. Bull.*, 26(4):482–484, 1983.

[HS09] Clemens Huemer and Carlos Seara. 36 two-colored points with no empty monochromatic convex fourgons. *Geombinatorics*, XIX, July 2009.

[KM88] M. Katchalski and A. Meir. On empty triangles determined by points in the plane. *Acta Math. Hungar.*, 51(3-4):323–328, 1988.

[Kos09a] V. A. Koshelev. On Erdős–Szekeres problem and related problems. `http://arxiv.org/abs/0910.2700`, 2009.

[Kos09b] V.A. Koshelev. On Erdős–Szekeres problem for empty hexagons in the plane. *Modeling and analysis of information systems*, 16(2):22–74, 2009.

[Lan09] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer Publishing Company, Incorporated, 3rd edition, 2009.

[Mat02] Jiří Matoušek. *Lectures on Discrete Geometry*, volume 212 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2002.

[Nic07] Carlos M. Nicolás. The empty hexagon theorem. *Discrete Comput. Geom.*, 38(2):389–397, 2007.

[OSV89] M.H. Overmars, B. Scholten, and I. Vincent. Sets without empty convex 6-gons. *Bull.of the EATCS*, 1989.

[Ove03] Mark Overmars. Finding sets of points without empty convex 6-gons. *Discrete Comput. Geom.*, 29(1):153–158, 2003.

[PT08] Janos Pach and Géza Tóth. Monochromatic empty triangles in two-colored point sets. In *Geometry, Games, Graphs and Education: the Joe Malkevitch Festschrift*, November 2008.

[Pyt12] Python/C API reference manual. `http://docs.python.org/c-api/index.html#c-api-index`, September 2012.

[RN10] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2010.

[SP06] George Szekeres and Lindsay Peters. Computer solution to the 17-point erdős-szekeres problem. *The ANZIAM Journal*, 48(02):151–164, 2006.

[TV98] G. Tóth and P. Valtr. Note on the Erdős-Szekeres theorem. *Discrete and Computational Geometry*, 1998.

[TV05] Géza Tóth and Pavel Valtr. The Erdős-Szekeres theorem: upper bounds and related results. In *Combinatorial and computational geometry*, volume 52 of *Math. Sci. Res. Inst. Publ.*, pages 557–568. Cambridge Univ. Press, Cambridge, 2005.