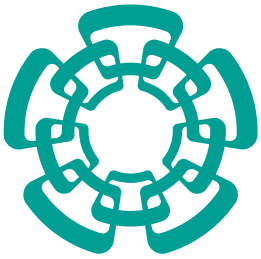

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO

DEPARTAMENTO DE
MATEMÁTICAS

"UN ALGORITMO PARA RECORRER LAS CELDAS DE UN ARREGLO DE RECTAS"

T E S I S

Que presenta

CARLOS MIGUEL HIDALGO TOSCANO

Para obtener el grado de

MAESTRO EN CIENCIAS

**EN LA ESPECIALIDAD DE
MATEMÁTICAS**

DIRECTOR DE LA TESIS: DR. RUY FABILA MONROY

MEXICO, D.F. AGOSTO 2015.

A mi madre, Cristina

Agradecimientos

Al CONACyT, por el apoyo recibido a lo largo de mis estudios de maestría, sin el cual habría sido significativamente más difícil completarlos.

Al Cinvestav, por la oportunidad de ingresar al programa de Maestría en el Departamento de Matemáticas y por los recursos y espacios que pone a disposición de los estudiantes.

Al Dr. Ruy Fabila Monroy, por introducirme a mundo de la investigación. Por su paciencia, comentarios, el tiempo dedicado en dirigir este trabajo, por el privilegio de trabajar con él y sus colegas y por todo lo que he aprendido gracias a él en los últimos años.

A mi madre, Cristina Hidalgo Toscano, por su inmenso cariño, por el apoyo incondicional que me ha brindado siempre, por sus consejos y por la confianza que me ha tenido. Gracias infinitamente.

A Amador, César, Marcy, Sttefany y Eduardo por todos los momentos de diversión, por estar ahí y por haber ayudado más de lo que se imaginan a concluir este trabajo. Su amistad es invaluable.

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

Índice general

Resumen

Varios problemas en Geometría Combinatoria tienen que ver con la estructura de conjuntos de puntos. Dos problemas de este tipo son el *número de cruce rectilíneo* y la cantidad de *hexágonos vacíos* en un conjunto de puntos en posición general en el plano. A pesar de la sencillez de su planteamiento, son problemas abiertos determinar el número de cruce rectilíneo en todo conjunto de n puntos y encontrar el entero mínimo n tal que todo conjunto con n puntos contiene un hexágono vacío.

Una heurística sencilla para buscar conjuntos con número de cruce rectilíneo bajo o conjuntos con pocos hexágonos vacíos consiste en generar un conjunto aleatorio de n puntos S, elegir un punto de S y moverlo aleatoriamente en busca de una posición que mejore el parámetro que analizamos.

En este trabajo, presentamos un algoritmo que permite recorrer las celdas de un arreglo de rectas en tiempo $O(\log^2 n)$ por cada celda, tras un preprocesamiento que toma tiempo $O(n^2 \log n)$. Se implementó este algoritmo y le heurística descrita para buscar conjuntos de puntos con número de cruce rectilíneo bajo y conjuntos de puntos con pocos hexágonos vacíos. El algoritmo permite a la heurística elegir un punto de una celda del arreglo de rectas inducido por el conjunto de puntos, evitando así la generación de conjuntos con el mismo tipo de orden a lo largo de la búsqueda.

Abstract

Many problems in Combinatorial Geometry ask about the structure of a point set. Two problems of this kind are the *rectilinear crossing number* and the number of *empty hexagons* of a set of points in general position in the plane. Despite their simple statements, finding the minimum rectilinear crossing number for every set of n points and finding the minimum integer n such that any set of n points has an empty hexagon are open problems.

A simple heuristic to search for point sets with low rectilinear crossing number or point sets with few empty hexagons is: generate a random n-point set S, choose a point from S and move it randomly, searching for a new position that improves the parameter that we are analyzing.

In this work, we present an algorithm that allows us to walk through the cells of a line arrangement in time $O(\log^2 n)$ per cell, after a $O(n^2 \log n)$ preprocessing. This algorithm was implemented along with the heuristic mentioned earlier to search for point sets with low rectilinear crossing number and point sets with few empty hexagons. This algorithm allows the heuristic to choose a point from a cell in the line arrangement induced by a set of points, avoiding in this way the generation of point sets with the same order type along the search.

Introducción

Los problemas que se estudian en Geometría Combinatoria tienen que ver en gran parte con preguntas sobre la estructura de conjuntos de objetos geométricos. En particular, muchos problemas tienen como objeto de estudio conjuntos de puntos en el plano.

Existen muchas preguntas que podemos hacer sobre las propiedades de conjunto de n puntos en el plano, y para algunas de estas propiedades las características métricas del conjunto no son importantes. Es decir, hay conjuntos distintos que combinatoriamente son indistinguibles y comparten propiedades de interés en Geometría Combinatoria. Un ejemplo de estas propiedades es la convexidad. El que un conjunto de puntos esté en posición convexa no depende de qué tan alejados estén entre sí.

Existen diversas maneras de clasificar conjuntos de puntos con el fin de agrupar los que son combinatoriamente equivalentes. Una de estas clasificaciones es el *tipo de orden*, y consiste esencialmente en considerar dos conjuntos como equivalentes si las orientaciones de las ternas de puntos coinciden.

Esta clasificación es muy conveniente desde el punto de vista computacional, pues verificar la orientación de una terna de puntos puede hacerse de manera rápida. Además, como una terna de puntos puede tener solamente dos orientaciones, existe una cantidad finita de conjuntos de n puntos con tipo de orden distinto. Esto hace posible, en principio, hacer búsquedas de conjuntos con distinto tipo de orden que cumplan alguna propiedad en particular por medio de la computadora. El inconveniente es que, a pesar de ser finita, la cantidad de conjuntos con tipo

de orden distinto es muy grande.

El uso de la computadora para resolver problemas acerca de conjuntos de puntos ha dado buenos resultados. Ejemplo de ello son la verificación de la conjetura de Erdős-Szekeres para hexágonos (esto es, todo conjunto de 17 puntos contiene un subconjunto de 6 puntos en posición convexa); la obtención de cotas inferiores sobre la cantidad de puntos necesarios para que aparezca un hexágono vacío o un cuadrilátero monocromático vacío si pintamos los puntos con dos colores; y la obtención de cotas inferiores para el número de cruces que aparecen cuando unimos todos los puntos de un conjunto con segmentos de recta. Todos estos son problemas que se han estudiado extensamente y son invariantes bajo el tipo de orden.

Para los últimos dos problemas mencionados, existen heurísticas sencillas para buscar conjuntos de puntos con pocos hexágonos vacíos o con número de cruce pequeño que han dado buenos resultados. Esencialmente consisten en, dado un conjunto de puntos S, tomar de manera aleatoria un punto y moverlo hasta encontrar una nueva posición donde el parámetro que estemos estudiando haya disminuido. Es claro que tenemos una infinidad de posibilidades para mover dicho punto, pero no todas nos darán un resultado distinto. Existen regiones muy grandes y muy pequeñas donde podemos garantizar que obtendremos el mismo resultado: las regiones del plano donde el tipo de orden del conjunto no cambia.

El objetivo de este trabajo es desarrollar un algoritmo que nos permita recorrer estas regiones con el fin de explorar conjuntos de puntos, es decir, hacer búsquedas de conjuntos de puntos con propiedades invariantes bajo tipo de orden. Está estructurado de la siguiente manera. En el Capítulo 1 se presentan las bases de la clasificación de conjuntos de puntos por tipo de orden y se introduce el concepto de dualidad geométrica, una herramienta muy útil en Geometría Combinatoria. En el Capítulo 2 describimos una estructura de datos interesante que permite mantener la cerradura convexa de un conjunto de puntos en el plano cuando añadimos o eliminamos puntos del conjunto. En el Capítulo 3 utilizamos las herramientas de los capítulos previos para desarrollar el algoritmo para explorar conjuntos de puntos. Finalmente, el Capítulo 4 habla de la implementación de dicho algoritmo, así como los resultados obtenidos.

Finalizamos esta introducción con algunos conceptos que se utilizarán a lo largo del trabajo.

Nuestro objeto principal de estudio serán conjuntos finitos de puntos en $\mathbb{R}^2$. Dado un conjunto S de n puntos en el plano, decimos que se encuentra en *posición general* si no contiene ternas de puntos colineales, es decir, ninguna línea pasa por tres puntos de S. Intuitivamente, posición general quiere decir que no hay "coincidencias poco probables". En este caso, si tomamos un conjunto de puntos en el plano de manera aleatoria, es poco probable que escojamos tres que estén sobre una línea.

Una propiedad básica de estudio en Geometría Combinatoria es la de convexidad. Un subconjunto C de $\mathbb{R}^2$ es *convexo* si para cualesquiera dos puntos en C el segmento que los une está totalmente contenido en C. Es claro que la intersección arbitraria de conjuntos convexos es convexa. Definimos el *cierre convexo* de un conjunto $X \subset \mathbb{R}^2$ como la intersección de todos los convexos que lo contienen. Denotamos el cierre convexo de X por $\text{Conv}(X)$, véase a Figura 1 para un ejemplo.

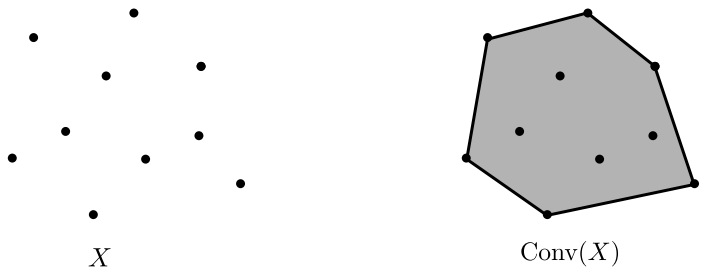

Figura 1: El cierre convexo de un conjunto de puntos.

El cierre convexo de un conjunto de puntos es un polígono cuyos vértices son puntos del conjunto, y es claro que basta con tener estos puntos ordenados en sentido contrario a las manecillas del reloj para almacenar el cierre convexo. En general no haremos distinción entre el cierre convexo de un conjunto de puntos y los vértices de su frontera.

Una *línea soporte* de un polígono es una línea que intersecta al polígono y deja a su interior de un solo lado.

El cierre convexo de un conjunto de puntos X puede descomponerse en dos

partes de manera natural, tomando las cadenas convexas que van del vértice con coordenada x más pequeña al vértice con coordenada x más grande. A la cadena que consiste de aristas cuyas líneas soporte dejan al interior de Conv(X) por debajo se le llama *cierre convexo superior* y se denota por ConvU(X). De manera análoga, a la cadena que consiste de aristas cuyas líneas soporte dejan al interior de Conv(X) por arriba se le llama *cierre convexo inferior* y se denota por ConvL(X).

Capítulo 1

Tipos de Orden

Muchos problemas de Geometría Combinatoria tienen como objetivo responder preguntas sobre conjuntos finitos de puntos, cuya respuesta es independiente de las propiedades métricas del conjunto. Un problema de este estilo que ha sido extensamente estudiado es la conjetura de Erdős-Szekeres, esta dice que todo conjunto de $2^{n-2}+1$ puntos en el plano en posición general contiene un subconjunto de n puntos en posición convexa (un subconjunto de este tipo se denomina n-gono).

Un camino para demostrar esta conjetura para cierta n, es analizar computacionalmente todo conjunto de $2^{n-2}+1$ puntos en el plano en búsqueda de un n-gono. En principio esto es imposible, ya que existe una infinidad de conjuntos con $2^{n-2}+1$ puntos en el plano. Pero las respuestas a preguntas sobre convexidad no dependen de las propiedades métricas del conjunto, por lo que es razonable pensar en agrupar conjuntos que sean combinatoriamente equivalentes y analizar sólo a un representante. Así, es útil disponer de una manera de clasificar esta infinidad de conjuntos en una cantidad finita clases, de tal manera que los conjuntos de cada clase compartan propiedades combinatorias que nos permitan responder preguntas como la asociada a la conjetura de Erdős-Szekeres.

Con esta motivación en mente, Goodman y Pollack [GP80] estudiaron diversos esquemas de clasificación de conjuntos de puntos, entre las cuales se encuentra

la clasificación por *tipo de orden*.

1.1. Tipos de orden y semiespacios

Un esquema de clasificación consiste, esencialmente, en asociar conjuntos de puntos con elementos un conjunto finito, A e identificar conjuntos con imágenes iguales. La utilidad de cualquier esquema depende de la simplicidad y precisión con que los objetos en A representan las propiedades de interés de los conjuntos de puntos y en la capacidad de distinguir qué objetos de A pueden tener asociado un conjunto de puntos (es decir, son geométricamente realizables).

Las clasificaciones de puntos estudiadas por Goodman y Pollack incluyen secuencias circulares [GP80], equivalencia combinatoria [GP84] y equivalencia por semiespacios o tipo de orden [GP83]; los dos últimos son válidos para conjuntos de puntos en $\mathbb{R}^d$. En este trabajo no enfocamos a la clasificación por tipo de orden de configuraciones de puntos (es decir, colecciones de puntos donde puede haber repeticiones) en $\mathbb{R}^2$, siguiendo la exposición de [GP83].

Definición 1.1. *Sean $p_1 = (x_1, y_1), p_2 = (x_2, y_2), p_3 = (x_3, y_3)$ puntos en el plano. Decimos que tienen* ***orientación positiva*** *(denotado por $p_1p_2p_3 > 0$) si*

$$\begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix} > 0.$$

Decimos que tienen ***orientación negativa*** *(denotado por $p_1p_2p_3 < 0$) si el determinante es negativo y decimos que tienen* ***orientación nula*** *(denotado por $p_1p_2p_3 = 0$) si el determinante es cero.*

En la definición anterior, la orientación positiva, negativa o nula es equivalente a que los tres puntos se encuentren en sentido contrario a las manecillas del reloj, en sentido de las manecillas del reloj, o sean colineales; respectivamente.

Definición 1.2. *Sea $\mathscr{C} = \{p_1, \ldots, p_n\}$ una configuración de puntos en el plano. El* ***tipo de orden*** *de $\mathscr{C}$ es una función que asigna a cada terna de puntos en $\mathscr{C}$ su*

orientación.

Decimos que dos configuraciones $\mathscr{C}_1$ y $\mathscr{C}_2$ tienen el mismo tipo de orden si existe una función biyectiva entre $\mathscr{C}_1$ y $\mathscr{C}_2$ que preserva las orientaciones de ternas de puntos (Figura 1.1).

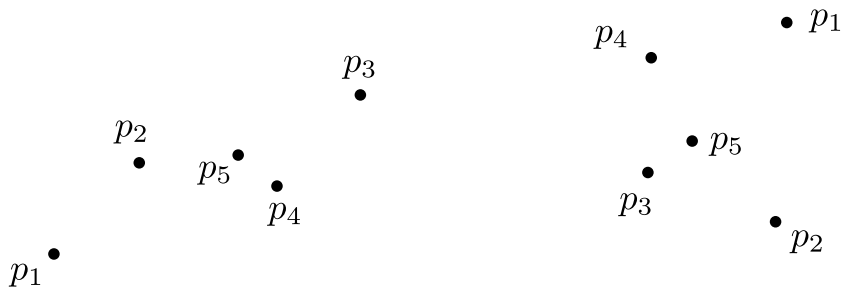

Figura 1.1: Dos configuraciones de cinco puntos con el mismo tipo de orden.

Al trabajar con representaciones de puntos en la computadora, la Definición 1.2 resulta muy conveniente. Esto es porque calcular el determinante requiere de una cantidad pequeña de operaciones simples (sumas, restas y multiplicaciones). Estas operaciones, la computadora las puede realizar en tiempo constante. Además, si se utilizan representaciones de puntos con coordenadas enteras y lenguajes que permitan la representación de enteros arbitrariamente grandes, no se introducen errores numéricos.

Dada una configuración $\mathscr{C} = \{p_1, \ldots, p_n\}$, conocer para cada par de puntos distintos p_i y p_j en $\mathscr{C}$ los puntos están a la izquierda de la recta de p_i a p_j, nos da la información necesaria para conocer el tipo de orden de $\mathscr{C}$. Estos conjuntos y sus cardinalidades juegan un papel importante la teoría de tipos de orden.

Definición 1.3. *Sea $\mathscr{C} = \{p_1, \ldots, p_n\}$ una configuración de puntos en $\mathbb{R}^2$. Sean*

$$\Lambda(i,j) = \begin{cases} \{k | P_i P_j P_k > 0\} & P_i \neq P_j \\ \Omega & P_i = P_j \end{cases}$$

y

$$\lambda(i,j) = \begin{cases} |\Lambda(i,j)| & P_i \neq P_j \\ \omega & P_i = P_j \end{cases}$$

para $1 \leq i, j \leq n$. *Los conjuntos* $\Lambda(i, j)$ *se llaman* ***semiespacios*** *de* $\mathscr{C}$.

Debido a que los semiespacios de una configuración de puntos nos dan la información necesaria para conocer su tipo de orden, dos configuraciones con el mismo tipo de orden se denominan también *equivalentes por semiespacios*.

Los semiespacios permiten ver al tipo de orden de un conjunto de puntos como un generalización del problema de ordenación de números (dados n números, ordenarlos de menor a mayor). Sea $T = \{x_1, \ldots, x_n\}$ una sucesión de números. Podemos definir semiespacios para este conjunto de manera análoga a los semiespacios para conjuntos de puntos:

$$\Lambda(i) = \{j | p_i p_j > 0\}$$

y

$$\lambda(i) = |\Lambda(i)|$$

para $1 \leq i, j, \leq n$. Para el caso de estos "puntos" en $\mathbb{R}$ y su orientación, decimos que $p_i p_j > 0$ si $p_i > p_j$ (p_i está a la derecha de p_j), $p_i p_j < 0$ si $p_i < p_j$ y $p_i p_j = 0$ si $p_i = p_j$. Es claro que conocer los puntos que están a la derecha de un p_i fijo nos da información de cuántos puntos están a la derecha de ese punto. También lo contrario es cierto, basta saber *cuántos* puntos están a la derecha de cada p_i para saber *cuáles* puntos se encuentran a la derecha de cada p_i; es decir, λ determina a Λ si los puntos de la configuración son números. La primera observación es análoga en $\mathbb{R}^2$, conocer los puntos que están a la izquierda de una línea dirigida nos da de inmediato la cantidad de puntos que cumplen esta propiedad. De manera interesante, para configuraciones en $\mathbb{R}^2$ también se cumple que λ determina de manera única a Λ. Antes de demostrar esto, necesitamos el siguiente Lema.

Lema 1.1. *Sea* ℓ *una línea en* $\mathbb{R}^2$, p_0 *un punto que no está en* ℓ, *y* ℓ_0 *la línea paralela a* ℓ *que pasa por* p_0. *Sea* U *el componente conexo de* $\mathbb{R}^2 \backslash \ell_0$ *que contiene a* ℓ *y sea* $\pi : U \to \ell$ *la proyección desde* p_0. *Entonces una de las dos orientaciones de* ℓ *como espacio afín 1-dimensional coincide con la orientación estándar de* $\mathbb{R}^2$ *en el siguiente sentido: si* p_1, p_2 *son puntos de* U, *entonces* $p_0 p_1 p_2$ *tienen orientación positiva, negativa o nula si y sólo si* $\pi(p_1)\pi(p_2)$ *tiene la orientación correspondiente en* ℓ. *(Figura 1.2)*

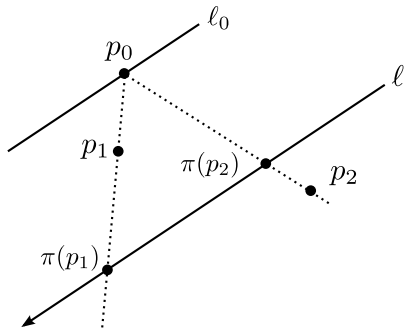

Figura 1.2: Dos puntos tales que $p_0p_1p_2 > 0$ y la orientación de ℓ que hace que $\pi(p_1)\pi(p_2) > 0$.

Demostración. Supongamos que $p_0p_1p_2 > 0$ para ciertos p_1, p_2 en U. La línea ℓ tiene solo dos orientaciones, fijemos aquella en la que $\pi(p_1)\pi(p_2) > 0$. Si q_1, q_2 son otros dos puntos de U tales que $p_0q_1q_2 > 0$, podemos mover el triángulo p_0, p_1, p_2 al triángulo p_0, q_1, q_2 de manera continua con p_0 fijo y los otros puntos manteniéndose en U de tal manera que toda configuración intermedia tenga también orientación positiva. Como para cualesquiera puntos r_1, r_2 en U se tiene que $p_0r_1r_2$ tienen orientación nula en $\mathbb{R}^2$ si y solo si $\pi(r1)\pi(r_2)$ tienen orientación nula en ℓ, se sigue del Teorema del valor intermedio que $\pi(q_1)\pi(q2) > 0$. ■

Teorema 1.1. *Dada una configuración de puntos $\mathscr{C} = \{p_1, \ldots, p_n\}$ en $\mathbb{R}^2$, si se conoce la función λ la función Λ está determinada de manera única.*

Demostración. La prueba es por inducción sobre n. Podemos suponer que $\lambda(i, j) > 0$ para algunos i, j, de lo contrario todos los puntos son colineales y $\Lambda(i, j) = \Omega$ para todo par de puntos en $\mathscr{C}$, por lo que el enunciado del Teorema se cumple.

Fijemos i, j de tal manera que $\lambda(i, j) = 0$. Este par de índices existe, basta con tomar dos puntos p_i, p_j de $\mathscr{C}$ en 1Conv($\mathscr{C}$) y elegirlos de tal manera que no haya puntos a la izquierda de la línea que va de p_i a p_j. Sea E la arista de Conv($\mathscr{C}$) que contiene a los puntos p_i, p_j. Ahora, si k es un índice tal que $\lambda(i, k) = 0$ o $\lambda(k, j) = 0$, el punto p_k debe estar en E y si p_k es un punto de $\mathscr{C}$ en E se tiene que $\lambda(i, k) = 0$ o $\lambda(k, j) = 0$. Es decir, λ determina los puntos que están en Conv($\mathscr{C}$).

Como $p_i = p_j$ si y solo si $\lambda(i, j) = \omega$, tenemos que λ determina una relación

de equivalencia $i \sim j \Leftrightarrow p_i = p_j$, por lo que podemos identificar cada grupo de puntos iguales con un solo punto, de tal manera que obtenemos una configuración $\mathscr{C}' = \{p'_1, \ldots, p'_r\}$ con $r \leq n$ sin puntos repetidos. Además, las aristas de Conv($\mathscr{C}'$) son las mismas que las de Conv($\mathscr{C}$) y, como los puntos extremos de Conv($\mathscr{C}'$) corresponden a un solo punto en $\mathscr{C}'$, podemos obtener los puntos extremos de $\mathscr{C}$. Supongamos que p_n es un punto extremo y $\{p_{m+1}, \ldots, p_n\}$ es el grupo de puntos repetidos al cual pertenece p_n. Sea ℓ_0 una línea soporte de Conv($\mathscr{C}$) en p_n y sea U el semiespacio determinado por ℓ_0 que contiene a los puntos $\{p_1, \ldots, p_m\}$. Sea ℓ una línea paralela a ℓ_0 contenida en U con la orientación dada por el Lema 1.1 y sea $\pi : U \to \ell$ la proyección desde p_n. Obtenemos entonces una nueva configuración $\mathscr{C}_n = \{\pi(p_1), \ldots, \pi(p_m)\}$ de puntos sobre una línea con funciones respectivas λ_n y Λ_n, de donde se tiene por el Lema 1.1 que

$$\lambda_n(\pi(p_i), \pi(p_j)) = \lambda(p_n, p_i, p_j)$$

para cualesquiera $1 \leq i, j \leq m$. Como ℓ es una línea, Λ_n queda determinada por λ_n. Pero esto implica que podemos saber la orientación de cualquier terna de puntos en $\mathscr{C}$ que involucra a p_n. Por inducción, la función λ'' correspondiente a la configuración $C'' = \{p_1, \ldots, p_m\}$ está determinada por λ y también la función Λ'', lo cual nos permite conocer la orientación de las ternas de puntos restantes en $\mathscr{C}$; esto completa la prueba. ■

Corolario 1.1. *Si dos configuraciones de puntos tienen la misma función λ, tienen el mismo tipo de orden.*

El Teorema 1.1 sugiere una manera de almacenar el tipo de orden de una configuración de n puntos: basta con construir una matriz de tamaño $n \times n$ cuya entrada (i, j) es $\lambda(i, j)$. Esta matriz recibe el nombre de *matriz-λ*.

Vale la pena notar que a pesar de que el tipo de orden depende de la orientación de $\binom{n}{3}$ ternas de puntos, la matriz-λ sólo requiere espacio $O(n^2 \log n)$ para almacenarlo (como $\lambda(i, j) < n$, cada entrada de la matriz requiere a lo más $\log n$ bits).

Debido a que sólo tenemos n puntos, parece tentador almacenar directamente las coordenadas de los puntos para representar el tipo de orden, escalando de tal

manera que las coordenadas no sean muy grandes para ganar un orden de magnitud en el espacio utilizado. Pero esto no es posible, como muestra el siguiente Teorema de Goodman, Pollack y Sturmfles [GPS89]:

Teorema 1.2. *Sea $\mathscr{C}$ una configuración de puntos en posición general. Sea $\nu(\mathscr{C}) = \min_{\mathscr{C}'} \max\{|x_1|, \ldots, |x_n|, |y_1|, \ldots, |y_n|\}$ donde $\mathscr{C}'$ es una configuración de puntos $p_i = (x_i, y_i)$ con coordenadas enteras y con el mismo tipo de orden de $\mathscr{C}$. Sea $\nu^*(n) = \max_{\mathscr{C}} \nu(\mathscr{C})$ sobre todas las configuraciones $\mathscr{C}$ de n puntos. Entonces, $\nu^*(n) = 2^{2^{\theta(n)}}$.*

Es decir, para toda n, existen configuraciones de n puntos que requieren coordenadas enteras doblemente exponenciales para representarlas.

Goodman y Pollack llamaron al Teorema 1.1 *Teorema básico del ordenamiento geométrico* debido a la analogía mencionada anteriormente del tipo de orden con la ordenación de números. Además, lo presentaron para configuraciones de puntos en $\mathbb{R}^d$ con una demostración análoga a la expuesta; basta utilizar doble inducción sobre n y d, para lo cual la generalización del Lema 1.1 da lo necesario. Una observación acerca de la función λ de una configuración de puntos que genera afínmente a $\mathbb{R}^d$, es que también determina el orden de conjuntos colineales, coplanares, etc. Basta con tomar uno de estos conjuntos, sustituir los puntos restantes con puntos en posición general y utilizar el Teorema 1.1. Si la configuración en cuestión no genera afínmente a $\mathbb{R}^d$ no se obtiene esta información, ya que en este caso los valores de λ son ω o 0. Pero basta añadir a la configuración los puntos $(1, 0, \ldots, 0), (0, 1, \ldots, 0), \ldots, (0, 0, \ldots, 1)$ para remediar esto.

El tipo de orden (o de manera equivalente, la función λ) depende de la manera en que etiquetemos los puntos de una configuración. Para determinar si dos configuraciones $\mathscr{C}$ y $\mathscr{C}'$ de n puntos tienen el mismo tipo de orden, necesitamos encontrar una biyección entre los puntos que respete las orientaciones, lo que implicaría en principio probar para cada una de las $n!$ etiquetaciones de una de las configuraciones si los tipos de orden coinciden. En realidad, podemos descartar muchas de estas etiquetaciones sin comparar los tipos de orden fijándonos en los vértices del cierre convexo de las configuraciones. Dado un punto p_i en $\mathrm{Conv}(\mathscr{C})$, re-etiquetemos los puntos de $\mathscr{C}$ de tal manera que p_i sea el primero y le sigan los

demás puntos ordenados a su alrededor en sentido contrario de las manecillas del reloj. Si las dos configuraciones tienen el mismo tipo de orden, la re-etiquetación adecuada de los puntos $\mathscr{C}'$ alrededor del punto p_i' correspondiente a p_i debe coincidir con la de $\mathscr{C}$. Así, basta con probar las etiquetaciones que ordenan los puntos al rededor de cada punto del cierre convexo, lo que implica que sólo necesitamos probar a lo más n de las $n!$ etiquetaciones posibles. Como la matriz λ que codifica cada etiquetación tiene tamaño cudrático, podemos responder si $\mathscr{C}$ y $\mathscr{C}'$ tienen el mismo tipo de orden en tiempo $O(n^3)$. Recientemente, Aloupis et. al. [AIL$^+$14] presentaron un algoritmo que determina si dos configuraciones de puntos en $\mathbb{R}^d$ tienen el mismo tipo de orden en tiempo $O(n^d)$.

Como se vio en la demostración del Teorema 1.1, el tipo de orden codifica la información sobre el cierre convexo de una configuración de puntos. Otra propiedad interesante que depende sólo del tipo de orden es si dos segmentos determinados por puntos de la configuración se cruzan o no. Estas propiedades dan lugar a problemas que se prestan a ser atacados utilizado los tipos de orden. En general, el tipo de orden captura propiedades de una configuración que son invariantes bajo transformaciones afines. Aichholzer y Krasser [AK01] expusieron una lista extensa de propiedades que dependen del tipo de orden y problemas que atacaron mediante el uso de una base de datos de tipos de orden que generaron para conjuntos con hasta 11 puntos.

El tipo de orden es una clasificación que funciona para configuraciones de puntos, sin importar que existan puntos colineales o repetidos. En adelante, trabajaremos solamente con conjuntos de puntos en posición general.

1.2. Arreglos de rectas

Una pregunta importante acerca de los tipos de orden es, ¿cuántos conjuntos de n puntos con tipo de orden distinto existen? Los arreglos de rectas en el plano ayudan a dar una cota inferior, además de ser útiles para los propósitos de este trabajo. Seguimos la exposición de [BCKO08].

Definición 1.4. *Sea L un conjunto de rectas en el plano. La subdivisión plano en*

vértices, aristas y caras (no necesariamente acotadas) inducida por L recibe el nombre de ***arreglo*** *inducido por L y se denota por $A(L)$. Si no hay tres líneas en L que se intersecten en un mismo punto, $A(L)$ se llama* ***simple****. La* ***complejidad*** *de $A(L)$ es el número total de vértices, aristas y caras que contiene. (Figura 1.3).*

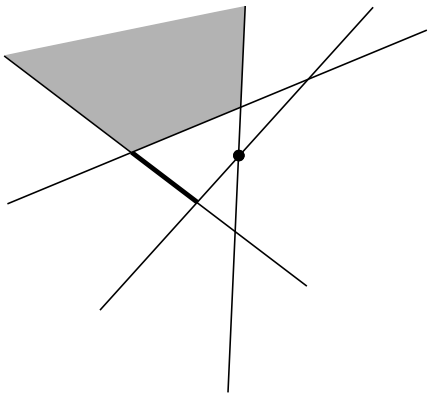

Figura 1.3: Un arreglo simple de rectas. Se resaltan un vértice, una arista y una cara no acotada.

Si un vértice es el punto final de una arista, decimos que son incidentes. De manera análoga, si una arista está en la frontera de una cara, también decimos que son incidentes. No es difícil acotar la complejidad de un arreglo de rectas. Si L tiene n rectas, estas se intersectan en a lo más $\binom{n}{2}$ puntos distintos, es decir, hay a lo más $(n^2-n)/2$ vértices en $A(L)$. Cada recta de L es intersectada en a lo más $n-1$ puntos distintos, por lo que en total hay a lo más n^2 aristas. En cuanto al número de caras, sean $L = \{\ell_1, \ldots, \ell_n\}$ y $L_i = \{\ell_1, \ldots, \ell_i\}$, veamos cuántas caras más tiene $A(L_{i+1})$ respecto a $A(L_i)$. Cada arista en ℓ_{i+1} divide una cara de $A(L_i)$ en dos, y como ℓ_{i+1} tiene a lo más i aristas en $A(L_{i+1})$, el número de caras en $A(L)$ es $1 + \sum_{i=1}^{n} i = (n^2+n)/2 + 1$. Estas cotas se alcanzan si $A(L)$ es simple. Así, la complejidad de un arreglo de rectas es cuadrática.

Un conjunto de n puntos en posición general en el plano determina $\binom{n}{2}$ rectas que a su vez, definen un arreglo no simple. Este arreglo tiene al menos cn^4 caras para cierta $c < 1/8$, y podemos extender nuestra configuración de tamaño n una configuración de tamaño $n+1$ poniendo un punto en alguna de estas caras. Obtendremos una configuración con un tipo de orden distinto por cada cara, ya que habrá ternas con orientaciones diferentes. Podemos hacer este pro-

cedimiento con cualquier configuración de n puntos, por lo que tenemos una cota inferior para la cantidad de tipos de orden de n puntos de, aproximadamente, $c(n!)^4$. Utilizando la fórmula de Stirling, obtenemos una cota inferior de $\exp(4(1+O(1/\log n)n\log n)) = n^{4n+O(n/\log n)}$. Este conteo no da la cantidad exacta de tipos de orden, ya que arreglos de diferentes configuraciones con el mismo tipo de orden pueden tener caras no equivalentes. Véase la Figura 1.4 para un ejemplo: un punto en la celda triangular de cada configuración da lugar a un tipo de orden distinto.

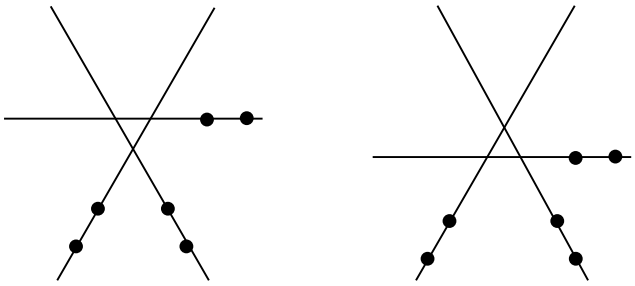

Figura 1.4: Dos conjuntos con el mismo tipo de orden, pero con arreglos que tienen caras no equivalentes.

Esta manera simple de contar tipos de orden puede utilizarse para configuraciones de puntos en $\mathbb{R}^d$ y da una cota muy cercana a la mejor cota superior conocida. Goodman y Pollack [GP86] mostraron el siguiente Teorema:

Teorema 1.3. *Sea $f(n,d)$ el número de tipos de orden distintos de configuraciones simples de n puntos en $\mathbb{R}^d$. Entonces* $\exp(d^2(1+O(1/\log n)n\log n)) \leq f(n,d) \leq \exp(d^2(1+O(1/\log n)n\log n))$.

Dado un arreglo de rectas es conveniente disponer de una manera de representarlo en la computadora que nos permita, por ejemplo, conocer las aristas de una cara o recorrer las caras cruzando aristas que comparten. Una estructura de datos que nos permite esto, es la *lista doblemente conexa de aristas* (LDCA), descrita por Muller y Preparata [MP78]. A continuación describimos la complejidad de almacenamiento y construcción de una LDCA. Para una exposición detallada, se pueden consultar [BCKO08] y [PS85].

Una LDCA contiene una registro por cada vértice, un registro por cada cara y

dos registros por cada arista del arreglo (un registro guarda la arista con puntos extremos a, b como segmento dirigido de a a b y el otro registro la guarda como segmento dirigido de b a a). Esto da como resultado que se guarden dos conjuntos de aristas que acotan una cara: uno que la recorre en sentido contrario a las manecillas del reloj y uno que la recorre en sentido de las manecillas del reloj. Llamaremos a estos conjuntos *frontera interior* y *frontera exterior*, respectivamente. Para evitar caras y aristas no acotadas, podemos pensar un rectángulo $B(L)$ que contiene a todos los vértices del arreglo, y añadir los vértices y aristas que se forman al intersectar el rectángulo con las líneas de L. Los registros contienen la siguiente información:

- Cada registro de un vértice v contiene las coordenadas de v y un apuntador a una arista cualquiera que se encuentre dirigida desde v.
- Cada registro de una arista e almacena el vértice el cual parte, un apuntador a la arista dirigida en sentido contrario y un apuntador a la cara f con la que incide y se encuentra a su izquierda. También almacena un apuntador a las aristas que inciden en f que se encuentran antes y después, respectivamente, al recorrer la frontera interior de f en sentido contrario a las manecillas del reloj.
- Cada registro de una cara f almacena un apuntador a una arista cualquiera de su frontera interior.

Así, el tamaño de una LDCA que representa un arreglo de rectas es proporcional a la complejidad del arreglo. Podemos construir la LDCA de manera incremental, es decir, procesar las líneas $\ell_1, \ldots, \ell_n$ en orden y se actualizar la LDCA con los vértices, aristas y caras generados al añadir ℓ_i a $A(L_{i-1})$ (por simplicidad, asumimos que ninguna ℓ_i es vertical y $A(L)$ es simple). Al añadir la línea ℓ_i, debemos dividir las caras que intersecta. La línea ℓ_i intersecta la frontera de $B(L)$ en dos aristas, hay $2i + 2$ aristas en la frontera de $B(L)$, por lo que podemos encontrar la arista e que ℓ_i intersecta primero al recorrerla de izquierda a derecha en tiempo lineal. Sea f la cara a la cual entra ℓ_i por e. Podemos recorrer f en sentido contrario a las manecillas del reloj para encontrar la arista e' por donde ℓ_i sale de

f. Una vez encontradas e y e', así como sus intersecciones con ℓ_i, podemos crear los registros nuevos en la LDCA para las caras, aristas y vértices nuevos; esto toma tiempo proporcional a la complejidad de f. Hace falta acotar el número de caras de $A(L_{i-1})$ que ℓ_i intersecta. Al conjunto de estas caras se le conoce como la *zona* de ℓ_i. El siguiente Teorema nos dice que la zona de ℓ_i tiene tamaño lineal.

Teorema 1.4 (Teorema de zona)**.** *Sea L un conjunto con n lineas en posición general y ℓ una línea que no pertenece a L. La complejidad de la zona de ℓ en $A(L)$ es $O(n)$.*

Demostración. Dada una arista e en la frontera de una cara f en $A(L)$, diremos que e ve a ℓ o que e es visible desde ℓ si existe un punto $x \in e$ y un punto $y \in \ell$ tales que el segmento abierto xy no intersecta ninguna línea en L. Nótese que la elección de x no es importante: o todos los puntos de e pueden ver a ℓ o ninguno puede.

Consideremos la zona de ℓ. Las caras que cruza son polígonos convexos, y como un polígono convexo tiene la misma cantidad de aristas y de vértices, basta con acotar el número de aristas que ven a ℓ.

Supongamos que ℓ es una línea horizontal (si este no es el caso, basta con hacer un cambio de coordenadas). Consideremos las aristas visibles desde ℓ que se encuentran arriba de ℓ. A lo más n de estas aristas intersectan a ℓ, ya que cada línea de L da lugar a lo más a una arista de este tipo.

Sea uv visible desde ℓ que no la intersecta. Sea ℓ_1 la línea en L que contiene a uv y sea a la intersección de ℓ y ℓ_1.

Escojamos la notación de tal manera que u esté más cerca de a que v. Sea ℓ' la línea que intersecta a ℓ_1 en u y denotemos por b al punto donde se intersectan ℓ y ℓ'. Diremos que la arista uv es una arista derecha de ℓ' si el punto b está a la derecha del punto a y diremos que es una arista izquierda de ℓ' en caso contrario.

Veamos ahora que para cada línea ℓ' en L existe a lo más una arista derecha. Si este no fuera el caso, existirían dos aristas, uv y xy, donde u está más abajo que x, tales que ambas son aristas derechas de ℓ', como en la Figura 1.5. La arista xy vería un punto de ℓ, pero la parte de ℓ a la derecha de a no es visible ya que ℓ_1

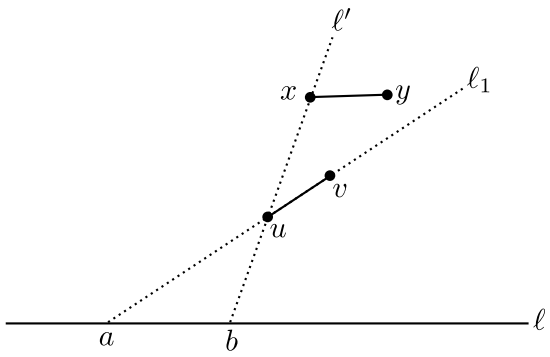

Figura 1.5: Sólo puede haber una arista derecha para cada línea del arreglo.

se interpone y la parte de ℓ a la izquierda de a no es visible ya que ℓ' se interpone. Esto es una contradicción, lo que implica que hay a lo más n aristas derechas.

De manera similar, existen a lo más n aristas izquierdas. Se pueden obtener las mismas cotas para las aristas que se encuentran debajo de ℓ, por lo que en total hay $O(n)$ aristas en la zona de ℓ. A su vez, esto implica que hay $O(n)$ caras en la zona de ℓ. ■

Con el resultado anterior, podemos ver que necesitamos tiempo $O(i)$ para actualizar la LDCA al procesar ℓ_i, por lo que la complejidad total de construir la lista es $\sum_{i=1}^{n} O(i) = O(n^2)$.

Los arreglos de rectas tienen una relación estrecha con los conjuntos de puntos. A través la dualidad geométrica, podemos asociar a cada conjunto de puntos un arreglo de rectas.

1.3. Dualidad geométrica

Un punto en $\mathbb{R}^2$ queda determinado por dos parámetros: sus coordenadas. De manera similar, una recta no vertical en $\mathbb{R}^2$ que da determinada por dos parámetros: su pendiente y su intersección con el eje y. Así, podemos asociar a todo conjunto de puntos un conjunto de rectas y a todo conjunto de rectas no verticales podemos asociarle un conjunto de puntos. Esta asignación recibe el nombre de

dualidad.

Definición 1.5. *Sea $p = (p_x, p_y)$ un punto en el plano. El dual de p, denotado por p^* es la línea definida como*

$$p^* = (y = p_x x - p_y)$$

Sea $\ell = (y = mx + b)$ una línea en el plano. El dual de ℓ, denotado por ℓ^* es el punto definido como

$$\ell^* = (m, -b)$$

La transformación dual no está definida para líneas verticales. Decimos que los objetos a los que aplicamos esta transformación se encuentran en el *plano primal* y sus imágenes se encuentran en el *plano dual*. La transformación dual tiene la ventaja de preservar ciertas propiedades. Si p y ℓ son un punto y una línea no vertical, respectivamente, se tiene que:

- La transformación dual preserva incidencias: $p \in \ell$ si y sólo si $\ell^* \in p^*$
- La transformación dual preserva orden: p se encuentra arriba de ℓ si y sólo si ℓ^* se encuentra arriba de p^*

La transformación de la Definición 1.5 tiene la siguiente interpretación geométrica. Consideremos la parábola $U : y = x^2/2$ y un punto p en el plano. Si $p = (p_x, p_y)$ se encuentra sobre U, su dual es la tangente a U en p. Si $p = (p_x, p_y + c)$ no está en U, su dual es la recta paralela a la tangente en U en el punto (p_x, p_y) que pasa por $(p_x, p_y - c)$ (Figura 1.6). Por esta razón, esta transformación recibe el nombre de *dualidad parabólica*.

La dualidad nos permite relacionar el cierre convexo de un conjunto de puntos con las fronteras de ciertas caras de un arreglo de rectas. Sea P un conjunto de puntos en el plano que no comparten coordenadas x y denotemos por P^* el conjunto de líneas que corresponden a los duales de los puntos en P. Un punto $p \in P$ aparece en ConvU(P) si y sólo si existe una línea no vertical ℓ que pasa por p tal que todos los puntos de P quedan debajo de ℓ. Entonces, en el plano dual existe

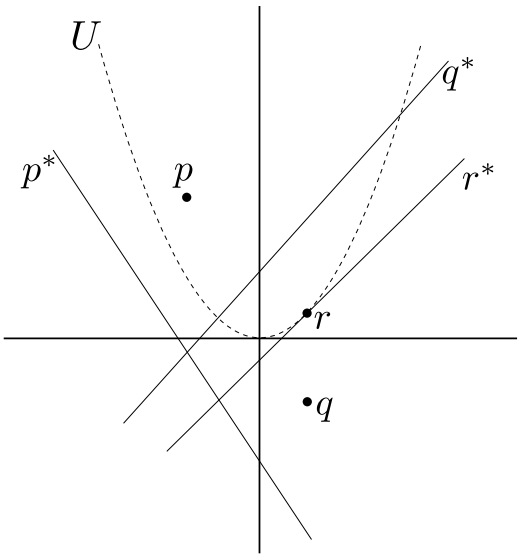

Figura 1.6: Dualidad parabólica.

un punto ℓ^* en la línea p^* tal que ℓ^* se encuentra debajo de todas las demás líneas de P^*. Es decir, en el arreglo $A(P^*)$, p^* contribuye con una arista de la frontera de la celda inferior no acotada del arreglo. La frontera de esta celda (la intersección de los semiespacios definidos por los puntos que se encuentran debajo de alguna recta en P^*) recibe el nombre de *envolvente inferior* del arreglo, y se denota por LE(P^*). La frontera de la celda superior no acotada (la intersección de los semiespacios definidos por los puntos que se encuentran arriba de alguna recta en P^*) recibe el nombre de *envolvente superior* del arreglo, y se denota por UE(P^*). Los puntos del ConvU(P) ordenados de manera creciente por coordenada x corresponden a las líneas que acotan LE(P^*), recorridas de derecha a izquierda. De manera análoga, los vértices de ConvL(P) ordenados por coordenada creciente x corresponden a las líneas que acotan UE(P^*), recorridas de izquierda a derecha. (Figura 1.7)

De manera similar, la dualidad nos permite asociar la intersección de semiespacios con el cierre convexo inferior y superior de un conjunto de puntos. Dado un conjunto H de semiespacios, denotemos por H_- el subconjunto de semiespacios inferiores de H (es decir, aquellos definidos como el conjunto de puntos abajo de una recta) y por H_+ el subconjunto de semiespacios superiores. Por un razon-

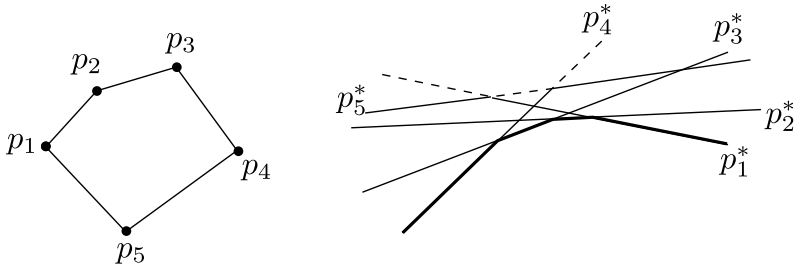

Figura 1.7: La relación entre cierre convexo y envolventes superior/inferior.

amiento análogo al del párrafo anterior, $\cup H_-$ corresponde a ConvL($H_-{}^*$) y $\cup H_+$ corresponde a ConvU($H_+{}^*$), los cuales pueden calcularse de manera eficiente. (Figura 1.8)

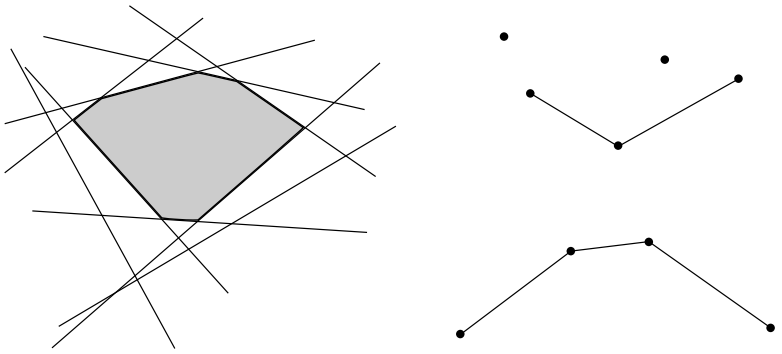

Figura 1.8: La relación entre cierre intersección de semiespacios y cierres convexos superior/inferior.

Así, la dualidad nos permite calcular la frontera de la cara de un arreglo si sabemos qué líneas la acotan. Esto jugará un papel importante en el Capítulo 3.

Para finalizar esta sección, la dualidad parabólica no es la única transformación dual. Existen otras transformaciones entre puntos y líneas que también preservan incidencias y orden, un ejemplo es la dualidad circular, que es análoga a la dualidad parabólica pero utiliza un círculo unitario. Más aún, estas transformaciones pueden definirse para puntos en $\mathbb{R}^d$. Más información acerca de dualidad geométrica puede encontrarse en [Mat02].

Capítulo 2

Cerradura convexa dinámica

Dado un conjunto de puntos P, encontrar Conv(P) es un problema que se ha estudiado extensamente y se conocen varios algoritmos óptimos para resolverlo. En general, estos algoritmos asumen que todos los puntos de P se conocen desde un inicio. Un problema relacionado es mantener el cierre convexo de un conjunto de puntos. Es decir, dados un conjunto P inicialmente vacío y una sucesión de N puntos en el plano $p_1, p_2, \ldots, p_N$ cada uno de los cuales corresponde a una inserción o borrado en P, mantener el cierre convexo de P. En esta sección describimos un algoritmo que resuelve este problema. Antes de presentarlo son necesarios algunos conceptos sobre estructuras de datos. Seguimos la exposición de [CLRS09] y [MR95].

2.1. Árboles binarios de búsqueda

Un problema fundamental en estructuras de datos es mantener un conjunto $S = \{s_1, \ldots, s_n\}$ de objetos de tal manera que se puedan realizar ciertos tipos de operaciones y consultas de forma eficiente. A cada elemento s de la colección se le asocia una *llave* $k(s)$ que lo representa. Las operaciones usuales deseadas son:

- *Crear* un conjunto vacío S.

- *Insertar* el elemento s en S.
- *Borrar* el elemento con llave k de S.
- *Buscar* el elemento con llave k en S.
- Dados conjuntos S_1 y S_2 donde $k(s) < k(t)$ para todos $s \in S_1$ y $t \in S_2$, *unir* S_1 y S_2, dando como resultado el conjunto $S = S_1 \cup S_2$.
- Dados un conjuntos S y una llave k, *dividir* S en conjuntos S_1 y S_2 tales $S_1 = \{s \in S \mid k(s) < k\}$ y $S_2 = \{s \in S \mid k(s) > k\}$.

Una estructura estándar que permite las operaciones mencionadas es el *árbol binario de búsqueda*. Los vértices de un árbol binario de búsqueda T almacenan los objetos del conjunto junto con su llave. Dado un vértice v en T, denotamos a la llave que almacena por $v.k$. Cada vértice v tiene asociado un padre denotado por $\pi(v)$ (excepto por un vértice distinguido llamado *raíz*) y puede tener asociado un hijo izquierdo (denotado por $v.lson$) y un hijo derecho (denotado por $v.rson$).

La propiedad que cumplen las llaves en un árbol binario de búsqueda T es la siguiente: dados un vértice v en T, un vértice u en el subárbol izquierdo de v y un vértice w en el subárbol derecho de v, siempre se tiene que $u.k \leq v.k$ y $v.k \leq w.k$. Si ordenamos de manera creciente las llaves de los vértices de T, el vértice que contiene la llave que precede a $v.k$ en este orden se denomina *antecesor* de v y el vértice que contiene la llave que sucede a $v.k$ en este orden se denomina *sucesor* de v. Véase la Figura 2.1 para un ejemplo.

Veamos que un árbol binario de búsqueda nos permite realizar las operaciones mencionadas para conjuntos de objetos.

Para crear un conjunto vacío S, basta inicializar un árbol vacío T.

Dado un árbol binario de búsqueda T con n vértices, la búsqueda del vértice que contiene la llave k se realiza de la siguiente manera. Comenzando por la raíz r, si $r.k = k$, hemos encontrado el vértice. Si $k < r.k$, exploramos recursivamente el subárbol izquierdo de r; de otra manera, exploramos el subárbol derecho de r. El tiempo necesario para determinar si k está en T depende de su altura, ya que

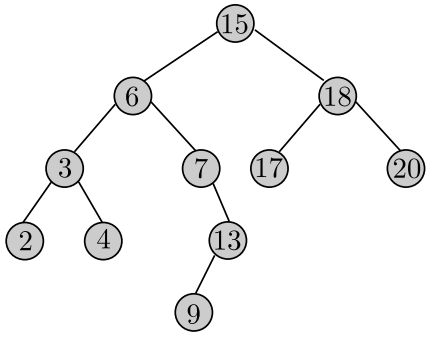

Figura 2.1: Un árbol binario de búsqueda. El antecesor y el sucesor del vértice 13 son los vértices 9 y 15, respectivamente

en el peor de los casos k no se encuentra en T y debemos recorrer la trayectoria más larga de la raíz a una hoja. Denotaremos esta operación por $T.search(k)$.

Para insertar un vértice v en T, basta con hacer una búsqueda de v en T y hacerlo hijo (izquierdo o derecho, dependiendo de $v.k$) de la hoja a la que lleguemos. De nuevo, esto toma tiempo proporcional a la altura de T. Denotaremos esta operación por $T.insert(k)$.

Si se desea borrar el vértice v que tiene llave k y éste es una hoja, basta con eliminarlo del árbol. Si v tiene un solo hijo, basta con borrar v y reemplazarlo por su hijo. Si v tiene dos hijos, reemplazamos v con su sucesor w y repetimos el proceso de borrado con w. Nótese que si v tiene un hijo derecho, w es el vértice más a la izquierda del subárbol derecho de x, de otra manera, w es el ancestro más cercano de v cuyo hijo izquierdo es también un ancestro de v. Como el tiempo requerido para borrar v depende del tiempo que nos toma encontrar su sucesor y encontrar al sucesor implica seguir una trayectoria hacia arriba o abajo de v, el borrado toma tiempo proporcional a la altura de T. Denotaremos esta operación por $T.delete(k)$.

Dados dos árboles binarios de búsqueda T_1 y T_2 tales que $v.k < u.k$ para cualesquiera $v \in T_1$ y $u \in T_2$, el árbol $T = T_1 \cup T_2$ se construye de la siguiente manera. Buscamos el vértice w en T_1 con la llave más grande y lo borramos de T_1. Inicializamos T con el vértice w como raíz, y hacemos T_1 y T_2 sus hijos izquierdo y derecho, respectivamente. Denotaremos esta operación por $T.join(T_1, T_2)$.

Dada una llave k, dividir un árbol binario T en T_1 y T_2 donde $v.k < k$ para todo $v \in T_1$ y $u.k > k$ para todo $u \in T_2$ es sencillo si el vértice v que contiene a k es la raíz: basta con hacer T_1 igual al subárbol izquierdo de v y T_2 igual al subárbol derecho de v. De otra manera, podemos hacer movimientos llamados *rotaciones* en T que modifiquen la estructura de T de tal manera que v se convierta en la raíz y se mantenga la propiedad de árbol binario de búsqueda. Las rotaciones se ilustran en la Figura 2.2, una rotación derecha en v consiste en pasar de a) a b), una rotación izquierda en u consiste en pasar de b) a a). Denotaremos esta operación por $T.split(k)$.

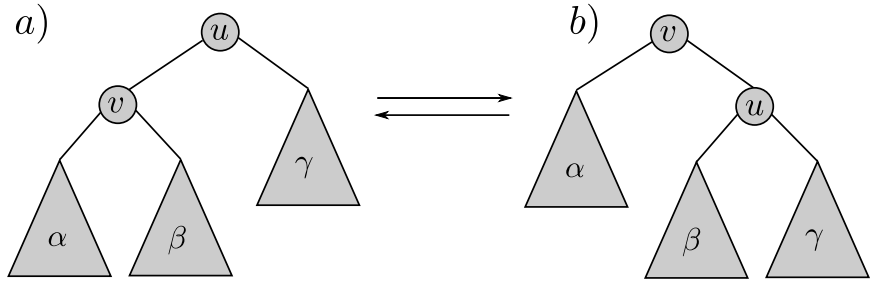

Figura 2.2: Rotaciones en un árbol binario.

Es claro que todas las operaciones dependen de las llaves de los vértices, por lo que en general no haremos distinción entre la llave y el objeto que un vértice almacena.

Sea T un árbol binario de búsqueda con n vértices. Todas las operaciones mencionadas toman tiempo proporcional a la altura de T. Desafortunadamente, puede suceder que el orden de inserción de los vértices sea tal que T resulte ser una trayectoria, es decir, que tenga altura $O(n)$ (por ejemplo si los vértices se insertan en orden creciente respecto a sus llaves). En el mejor de los casos, cada vértice interior tiene dos hijos; entonces T tiene altura $O(\log n)$ y decimos que T está *balanceado*. Si los objetos se conocen desde el inicio, se puede elegir un orden de inserción que de como resultado un árbol binario de búsqueda balanceado, pero en general, esto no sucede.

Las rotaciones que se utilizan para llevar a un vértice a la raíz tienen el efecto de modificar la altura del árbol respetando la propiedad de búsqueda binaria. Así, una estrategia para mantener T balanceado es realizar las rotaciones necesarias

para mantener una altura logarítmica tras cada inserción o borrado. Este tipo de árboles se denominan *autobalanceables*.

Una estrategia alterna muy sencilla que da buenos resultados es insertar los vértices en orden aleatorio. Decimos que un árbol que se construye insertando los vértices en orden aleatorio, donde cada una de las $n!$ permutaciones del orden de los vértices es igual de probable, es *construido aleatoriamente*.

Teorema 2.1. *Sea T un árbol binario de búsqueda con n vértices construido aleatoriamente. La altura esperada de T es $O(\log n)$.*

Demostración. Sea X_n una variable aleatoria que denota la altura de T y definamos la altura exponencial como la variable aleatoria $Y_n = 2^{X_n}$. Al construir T, un vértice se convierte en la raíz; sea R_n la variable aleatoria que representa la posición que ocupa la llave de la raiz de T en el conjunto ordenado de llaves (a esto se le llama el *rango* de la llave). El valor de R_n puede ser cualquiera entre $\{1,\ldots,n\}$ con la misma probabilidad. Si $R_n = i$, el subárbol izquierdo es un árbol binario de búsqueda construido aleatoriamente con $i-1$ vértices y el subárbol derecho es un árbol binario de búsqueda construido aleatoriamente con $n-i$ vértices. Como la altura de un árbol binario es 1 más que la altura más grande de sus subárboles, la altura exponencial es el doble de la más grande de las alturas exponenciales de los subárboles. Entonces, si $R_n = i$, se tiene que $Y_n = 2 \cdot \text{máx}(Y_{i-1}, Y_{n-i})$. Para los casos base, tenemos que $Y_1 = 1$ y definimos $Y_0 = 0$.

Ahora, definamos variables aleatorias $Z_{n,1},\ldots,Z_{n,n}$, donde $Z_{n,i} = I\{R_n = i\}$. Tenemos que $\Pr\{R_n = i\} = 1/n$, por lo que $\mathrm{E}[Z_{n,i}] = 1/n$ para $1 \leq i \leq n$. Además, como sólo una $Z_{n,i}$ es 1 y las demás son 0, tenemos que:

$$Y_n = \sum_{i=1}^{n} Z_{n,i}(2 \cdot \text{máx}(Y_{i-1}, Y_{n-i}))$$

Una vez escogido $R_n = i$, el subárbol izquierdo es un árbol binario de búsqueda construido aleatoriamente con $i-1$ vértices, cuyas llaves tienen rango menor a i. Además de la cantidad de vértices que tiene, la estructura de este subárbol

no se afecta por la elección de $R_n = i$, por lo que las variables $Z_{n,i}$ y Y_{i-1} son independientes. Un análisis análogo muestra que $Z_{n,i}$ y Y_{n-i} son independientes. Entonces, tenemos:

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\big[Z_{n,i}]E[2 \cdot \max(Y_{i-1}, Y_{n-i})\big] \\
&= \frac{2}{n}\sum_{i=1}^{n} \mathrm{E}\left[\max(Y_{i-1}, Y_{n-i})\right] \\
&\leq \frac{2}{n}\sum_{i=1}^{n} (\mathrm{E}\left[Y_{i-1}\right] + \mathrm{E}\left[Y_{n-i}\right])
\end{aligned}$$

Como cada $\mathrm{E}[Y_i]$ para $0 \leq i \leq n-1$ aparece dos veces en la última suma, obtenemos la recurrencia:

$$\mathrm{E}[Y_n] \leq \frac{4}{n}\sum_{i=0}^{n-1} \mathrm{E}[Y_i]$$

Veamos que esta recurrencia cumple

$$\mathrm{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}$$

Para los casos base, tenemos las cotas $0 = Y_0 = \mathrm{E}[Y_0] \leq (1/4)\binom{3}{3} = 1/4$ y $1 = Y_1 = \mathrm{E}[Y_1] \leq (1/4)\binom{1+3}{3} = 1$. Sustituyendo en la recurrencia:

$$\begin{aligned}
\mathrm{E}[Y_n] &\leq \frac{4}{n}\sum_{i=0}^{n-1} \mathrm{E}[Y_i] \\
&\leq \frac{4}{n}\sum_{i=1}^{n} \frac{1}{4}\binom{i+3}{3}
\end{aligned}$$

$$= \frac{1}{n}\binom{n+3}{4}$$
$$= \frac{1}{4}\binom{n+3}{3}$$

Ahora, como la función $f(x) = 2^x$ es convexa, de la desigualdad de Jensen obtenemos $2^{\mathrm{E}[X_n]} \leq \mathrm{E}[2^{X_n}] = \mathrm{E}[Y_n]$. Ahora podemos acotar $\mathrm{E}[X_n]$:

$$\begin{aligned} 2^{\mathrm{E}[X_n]} &\leq \frac{1}{4}\binom{n+3}{3} \\ &= \frac{n^3 + 6n^2 + 11n + 6}{24} \end{aligned}$$

Tomando logaritmos de ambos lados, obtenemos que $\mathrm{E}[X_n] = O(\log n)$.

■

Esta estrategia permite construir árboles binarios de búsqueda con altura esperada logarítmica si se dispone de todos los objetos en un inicio. Una estructura que sigue una estrategia similar y que permite inserciones y borrados manteniendo una altura esperada logarítmica es el *treap*.

2.2. Treaps

Un treap es una estructura de datos introducida por Aragon y Seidel [SA96] que combina las propiedades de un árbol binario de búsqueda y de un *heap* (un heap es un árbol binario donde las llaves que almacenan sus vértices están en orden creciente en cualquier trayectoria de la raíz a una hoja). Un treap es un árbol binario que almacena en cada vértice v un par de valores: una llave denotada por *v.key* y una *prioridad* denotada por *v.priority*. Es un árbol binario de

búsqueda con respecto a las llaves y simultáneamente un heap con respecto a las prioridades (Figura 2.3). Es decir:

- Si v está en el subárbol izquierdo de u, entonces $v.key < u.key$.
- Si v está en el subárbol derecho de u, entonces $v.key > u.key$.
- Si v es un descendiente de u, entonces $v.priority > u.priority$.

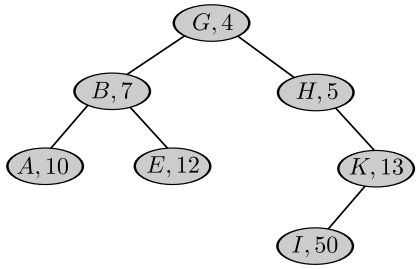

Figura 2.3: Un treap. Cada vértice tiene una etiqueta de la forma *llave,prioridad*.

El siguiente teorema muestra que, dado un conjunto de pares de llaves y prioridades, existe un único treap que los representa.

Teorema 2.2. *Sea $S = \{(k_1, p_1), \ldots, (k_n, p_n)\}$ un conjunto de pares de llaves-prioridades. Entonces existe un único treap $T(S)$ que lo representa.*

Demostración. La prueba es por construcción y la construcción es recursiva. Si $n = 0$ o $n = 1$ el teorema se cumple. Supongamos entonces que $n \geq 2$ y que (k_1, p_1) tiene la prioridad más baja en S. La raíz de $T(S)$ debe contener este par, y podemos construir recursivamente el subárbol izquierdo con los pares que tienen llaves menores a k_1 y el subárbol derecho con los pares que tienen llaves mayores a k_1. Este procedimiento para construir $T(S)$ es determinista, por lo que $T(S)$ es único. ■

Como se vio en la demostración del teorema anterior, la forma de $T(S)$ está determinada por las prioridades de los pares de S y podemos pensar en un treap como un árbol binario de búsqueda que fue construido de acuerdo al orden ascendente de las mismas. Así, si tomamos prioridades aleatorias podemos verlo como un árbol construido aleatoriamente, por lo que su altura esperada es $O(\log n)$.

Debido a que un treap es un árbol binario de búsqueda respecto a sus llaves, la búsqueda de un vértice se hace de la misma manera que en un árbol binario de búsqueda normal. La inserción de un vértice nuevo, por otro lado, puede romper con la propiedad de heap si se realiza de la misma manera que en un árbol binario de búsqueda normal. Esto puede remediarse haciendo rotaciones: como las rotaciones preservan la propiedad de árbol binario e invierten el orden de un vértice y su padre, podemos cambiar la posición del vértice nuevo hasta que su prioridad sea mayor a la de su padre. Así, el tiempo requerido para insertar un vértice es igual al tiempo requerido para insertar un vértice en un árbol binario de búsqueda mas el tiempo requerido para hacer las rotaciones. Cada rotación toma tiempo constante, y en el peor de los casos un vértice pasa de ser una hoja a ser la raíz. Como después de cada rotación el vértice está un nivel más cerca de la raíz y la altura esperada del treap es $O(\log n)$, se hacen a lo más $O(\log n)$ rotaciones y el tiempo esperado total de inserción es $O(\log n)$.

Aunque la búsqueda e inserción de un vértice en un treap tienen el mismo tiempo esperado, los costos en la práctica son distintos. Una búsqueda efectúa sólo operaciones de lectura, pero una inserción cambia apuntadores al realizar las rotaciones necesarias. Debido a que las operaciones de lectura son mucho más rápidas que las de escritura, es deseable que las operaciones de inserción efectúen pocas rotaciones. Resulta que el número esperado de rotaciones que se efectúan al insertar un vértice está acotado por una constante. Para demostrar esto, introducimos algunas definiciones.

La *espina izquierda* de un árbol binario es la trayectoria que inicia en la raíz y consiste solamente de aristas izquierdas. La *espina derecha* de un árbol binario es la trayectoria que inicia en la raíz y consiste solamente de aristas derechas (Figura 2.4). La longitud de una espina es el número de vértices que contiene.

Lema 2.1. *Sea T un treap y v un nodo que acaba de ser insertado. Sean C la longitud de la espina derecha del subárbol izquierdo de v y D la longitud de la espina izquierda del subárbol derecho de v. El número de rotaciones realizadas durante la inserción de v es igual a $C + D$.*

Demostración. La prueba es por inducción sobre el número de rotaciones real-

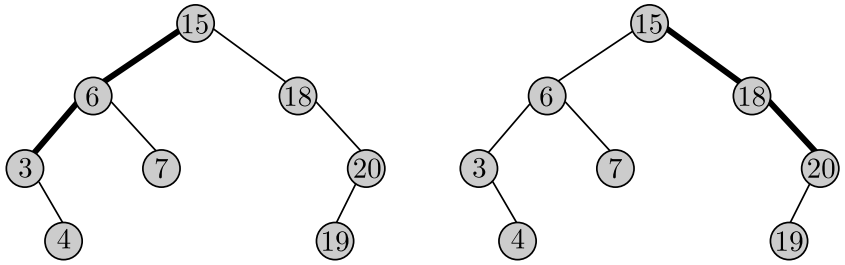

Figura 2.4: Las espinas izquierda y derecha de un árbol binario, respectivamente.

izadas k. Si no se realizaron rotaciones, v es una hoja de T y el teorema se cumple. Supongamos entonces que se realizaron $k > 0$ rotaciones.

Sea u el padre de v y supongamos que v es hijo izquierdo y que se han realizado $k-1$ rotaciones. Por hipótesis de inducción, tenemos que $C + D = k - 1$ hasta este momento. La rotación que falta es una rotación derecha en v, al realizarla, u se convierte en el hijo derecho de v y el vértice que era hijo derecho de v se convierte en el hijo izquierdo de u. Es decir, la longitud de espina izquierda del subárbol derecho de v tras la rotación se incrementa en uno, mientras que la espina derecha del subárbol izquierdo de v queda sin cambios.

Si v es el hijo derecho de u, un argumento análogo muestra que una rotación izquierda en v aumenta la longitud de la espina derecha del subárbol izquierdo en uno, dejando la longitud de la espina izquierda del subárbol derecho sin cambios. Así, tras hacer la última rotación, $C + D = k$. ■

En vista del Lema 2.1, para calcular el número de rotaciones esperado tras insertar un vértice v en un treap T, basta con calcular el valore esperado de $C+D$. En adelante asumimos que las llaves de los vértices son $\{1, \ldots, n\}$. Esto no supone pérdida de generalidad, ya que las operaciones que involucran a las llaves son sólo comparaciones.

Para vértices distintos u y v de T, sean $k = u.key$ e $i = v.key$. Definimos variables indicadoras $X_{ik} = \mathrm{I}\{v$ está en la espina derecha del subárbol izquierdo de $u\}$.

Lema 2.2. *$X_{ik} = 1$ si y sólo si se cumplen las siguientes condiciones:*

1. *v.priority* $>$ *u.priority*
2. *v.key* $<$ *u.key*
3. *Para todo vértice w tal que v.key* $<$ *w.key* $<$ *u.key, se cumple v.priority* $<$ *w.priority*

Demostración. Supongamos primero que $X_{ik} = 1$. Entonces, 1 se cumple porque v es descendiente de u y T es un heap respecto a las prioridades; 2 se cumple porque v está en el subárbol izquierdo de u y T es un árbol binario de búsqueda respecto a las llaves. Sea w un vértice tal que $v.key < w.key < u.key$. Un recorrido en inorden de T debe encontrar a v, w y u en ese orden. Más aún, como v está en la espina derecha del subárbol izquierdo de u, tras encontrar a v y recorrer su subárbol derecho, el siguiente vértice a visitar debe ser u. Esto implica que w está en el subárbol derecho de v, por lo que *v.priority* $<$ *w.priority*, es decir, 3 se cumple.

Ahora supongamos que 1, 2 y 3 se cumplen. El vértice v debe ser descendiente de u, de no ser así, tienen un ancestro común w (que no es v, ya que *v.priority* $>$ *u.priority*). Como *v.key* $<$ *u.key*, v está en el subárbol izquierdo de w y u está en el subárbol derecho, por lo que *v.key* $<$ *w.key* $<$ *u.key*, pero entonces *v.priority* $>$ *w.priority*, contradiciendo 3. Ahora, v debe estar en el subárbol izquierdo de u, ya que *v.key* $<$ *u.key*. Supongamos que v no está en la espina derecha del subárbol izquierdo de u. Entonces existe un vértice w en dicha espina tal que v está en el subárbol izquierdo de w. Esto quiere decir que *v.key* $<$ *w.key* $<$ *u.key* y *v.priority* $>$ *w.priority*, contradiciendo 3. Así, $X_{ik} = 1$. ■

Las condiciones del Lema 2.2 nos permiten calcular $\Pr\{X_{ik} = 1\}$. Como las llaves de los vértices son $\{1, \ldots, n\}$, las llaves involucradas en la probabilidad anterior son $\{i, i+1, \ldots, k\}$. Además, las prioridades de u y v deben ser las más bajas entre los vértices con dichas llaves. Así, hay $(k-i-1)!$ posibles ordenes de estas prioridades, por lo que:

$$\Pr\{X_{ik} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}$$

Ahora podemos calcular $\mathrm{E}[C]$, que es igual a la suma del valor esperado de X_{ik} para todo i en el árbol. Pero $X_{ik} = 0$ si $k \leq i$, por lo que:

$$\begin{aligned}
\mathrm{E}[C] &= \mathrm{E}\left[\sum_{i=1}^{k-1} X_{ik}\right] \\
&= \sum_{i=1}^{k-1} \mathrm{E}[X_{ik}] \\
&= \sum_{i=1}^{k-1} \Pr\{X_{ik} = 1\} \\
&= \sum_{i=1}^{k-1} \frac{1}{(k-i+1)(k-i)} \\
&= \sum_{j=1}^{k-1} \frac{1}{j(j+1)} \\
&= 1 - \frac{1}{k}
\end{aligned}$$

El valor esperado de C depende sólo del rango de la llave de v. Si construimos un treap T' con los mismos vértices que T pero asignando la llave $n-k+1$ al vértice con llave k, obtendremos un treap simétrico a T en el sentido de que T' es T reflejado. La longitud de la espina derecha del subárbol izquierdo de un vértice en T' es $1-1/(n-k+1)$, por lo que $\mathrm{E}[D] = 1-1/(n-k+1)$. Así, tenemos que:

$$\mathrm{E}[C+D] = \mathrm{E}[C] + \mathrm{E}[D] = 1 - \frac{1}{k} + 1 - \frac{1}{n-k+1} \leq 2$$

es decir, el valor esperado de rotaciones realizadas al insertar un vértice en un treap es a lo más 2.

Falta analizar las operaciones restantes que se realizan en árboles binarios de búsqueda para los treaps. El borrado de un nodo se realiza de manera inversa a la inserción: una vez encontrado el vértice, se hacen las rotaciones necesarias para llevarlo a una hoja (estas rotaciones pueden realizarse respetando las propiedades de árbol binario de busqueda y de heap) y después se borra del treap. El análisis realizado para el número de rotaciones esperado en una operación de inserción

también aplica para la operación de borrado.

Unir dos treaps T_1 y T_2 se hace de la misma manera que en un árbol binario de búsqueda, la diferencia es que la raíz del treap resultante puede tener prioridad mayor a la de alguno de sus hijos. Si esto sucede, basta con hacer rotaciones hasta que la prioridad de dicho vértice no viole la propiedad de heap. Finalmente, dividir un treap T en T_1 y T_2 dada una llave k se consigue borrando el vértice v cuya llave es k, insertándolo con prioridad $-\infty$ y haciendo T_1 igual al subárbol izquierdo de esta nuevo treap y T_2 igual al subárbol derecho. Así, todas las operaciones que se hacen en árboles binarios de búsqueda se pueden hacer en treaps en el mismo tiempo esperado. El siguiente teorema resume lo presentado en esta sección.

Teorema 2.3. *Sea T un treap con n vértices.*

- *Se puede buscar, insertar o borrar un vértice en tiempo esperado $O(\log n)$ y el número esperado de rotaciones que se realizan es a lo más 2.*
- *Se pueden hacer las operaciones de unión y división en tiempo esperado $O(\log n + \log m)$, donde m es el número de vértices del otro treap involucrado en la operación.*

Cabe mencionar que estructuras de árboles autobalanceabless como los árboles AVL [AH74] o los árboles Rojo-Negros [CLRS09] tienen la misma complejidad para operaciones analizadas, pero son deterministas. La elección de los treaps en este trabajo se debe a que su implementación es considerablemente menos complicada.

2.3. Mantenimiento del cierre convexo de un conjunto de puntos

Retomemos ahora el problema mencionado al inicio de este capítulo: dados un conjunto P inicialmente vacío y una sucesión de N puntos en el plano $p_1, p_2, \ldots, p_N$, cada uno de los cuales corresponde a una inserción o borrado en P,

mantener el cierre convexo de P. Una solución trivial es calcular, para cada punto de N que procesamos, el cierre convexo de los puntos en P. Esto da un algoritmo que mantiene el cierre convexo y toma tiempo $O(n \log n)$ por inserción/borrado. Por otro lado, de la cota de $\Omega(n \log n)$ en el tiempo requerido para calcular el cierre convexo de un conjunto de n puntos, tenemos que cada operación de inserción borrado debe tomar tiempo $\Omega(\log n)$.

Preparata [Pre79] diseñó un algoritmo que permite hacer inserciones a P y mantener el cierre convexo en tiempo $O(\log n)$ por inserción. El algoritmo funciona de la siguiente manera. Se mantienen los puntos extremos de $\{p_1, \ldots, p_i\}$ ordenados por ángulo al rededor de algún punto p_0 al interior de su cierre convexo. Cuando se desea insertar p_{i+1}, se determina si está o no en el interior del cierre convexo actual analizando el sector al que pertenece (esto puede lograrse mediante una búsqueda binaria al rededor de p_0). Si p_{i+i} está en el interior del cierre convexo actual, no se necesitan hacer cambios. En otro caso, se encuentran las tangentes $\overline{p_{i+1}q}$ y $\overline{p_{i+1}r}$ al cierre convexo actual, se eliminan los puntos que se encuentran entre q y r y se inserta p_{i+1} en su lugar (Figura 2.5).

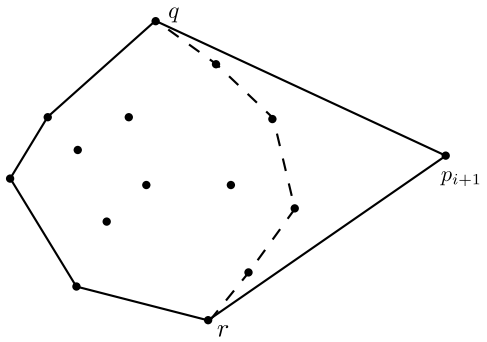

Figura 2.5: Añadiendo un vértice al cierre convexo.

En este algoritmo, la parte más importante es diseñar una estructura de datos que permita realizar la búsqueda binaria en tiempo $O(\log n)$ además de permitir la búsqueda, inserción y borrado de puntos en el cierre convexo en $O(\log n)$.

A pesar de poder realizar inserciones para actualizar el cierre convexo en tiempo óptimo, el algoritmo anterior no da soporte para borrado de puntos. Un hecho que utilizan a lo largo del algoritmo es que los puntos que quedan en algún momento al interior del cierre convexo pueden ser descartados, pues ya no pueden

formar parte del mismo en ningún momento. Esto no se cumple si se desea borrar puntos: al eliminar un punto del conjunto, muchos puntos del interior pueden pasar a formar parte del cierre convexo (la figura 2.5 da un ejemplo de esto). Así, es necesario mantener todos los puntos almacenados.

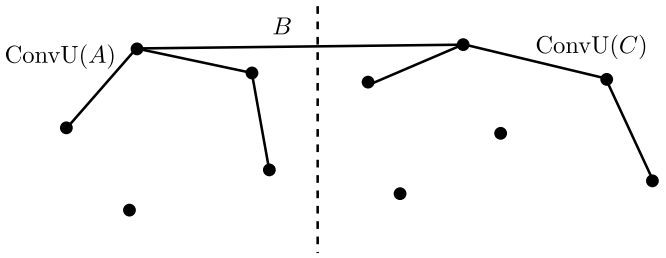

Figura 2.6: La descomposición del cierre convexo superior de un conjunto de puntos.

Overmars y Van Leeuwen [OvL81] presentaron un algoritmo que resuelve el problema en cuestión; seguimos su exposición. El algoritmo aprovecha el hecho de que el cierre convexo de un conjunto de puntos es igual a la unión del cierre convexo superior (ConvU) y el cierre convexo inferior (ConvL). Como el cierre convexo se obtiene de una simple concatenación de ConvL y ConvU, restringimos el análisis al mantenimiento de ConvU. Nótese que los puntos P que definen a ConvU aparecen en orden creciente de coordenada x.

Para un conjunto de puntos S, podemos descomponer ConvU(S) de la siguiente manera. Dividamos a S con una línea vertical en dos subconjuntos A y C, como en la Figura 2.6. Entonces ConvU(S) está compuesto por un segmento inicial de ConvU(A), un segmento final de ConvU(C) y una arista B llamada *puente* que conecta dichas partes. El siguiente Teorema nos muestra que, dados ConvU(A) y ConvU(C) podemos encontrar ConvU(S) de manera eficiente.

Teorema 2.4. *Sean $p_1, \ldots, p_n$ puntos en el plano ordenados por coordenada x. Si se conocen* ConvU($\{p_1, \ldots, p_i\}$) *y* ConvU($\{p_{i+1}, \ldots, p_n\}$) *para algún $1 \leq i \leq n$, se puede construir* ConvU($\{p_1, \ldots, p_n)\}$) *en $O(\log n)$ pasos.*

Demostración. Sean $S = \{p_1, \ldots, p_n\}$, $A = \{p_1, \ldots, p_i\}$ y $C = \{p_{i+1}, \ldots, p_n\}$ y supongamos que ConvU(A) y Conv(C) están almacenados en árboles autobalanceables Q_A y Q_C, respectivamente. Para encontrar ConvU(S), sólo necesitamos

encontrar el puente B entre ConvU(A) y ConvU(C). Supongamos que tenemos a B y que sus extremos en ConvU(A) y ConvU(C) son los puntos l y r, respectivamente. Entonces, podemos construir el árbol autobalanceable Q que contiene a ConvU(S) de la siguiente manera. Obtenemos $Q_{A1}, Q_{A2} := Q_A.split(l.key)$ (haciendo que Q_{A1} contenga a l) y obtenemos $Q_{C1}, Q_{C2} := Q_C.split(r.key)$ (haciendo que Q_{C2} contenga a r). Finalmente, obtenemos $Q := Q.join(Q_{A1}, Q_{C2})$, lo cual toma tiempo $O(\log n)$ como se vio en la sección anterior.

Así, necesitamos determinar de manera eficiente el puente B. Para este fin, debemos poder efectuar una búsqueda binaria de l y r en los cierres convexos superiores guardados en Q_A y Q_C. Para lograr esto, hacemos que cada vértice de Q tenga apuntadores a las hojas de sus subárboles con la coordenada x más chica y más grande, respectivamente. Esto no afecta la complejidad de las operaciones para los árboles. Los apuntadores hacen posible que, al hacer una búsqueda binaria y llegar a un vértice que contiene el "segmento" $[p, r]$ de un cierre convexo superior, sólo necesitemos inspeccionar dichos apuntadores (digamos, a q_1 y q_2) para saber en qué "subsegmento" $[p, q_1]$ o $[q_2, r]$ continuar.

Dados un punto p en A y un punto q en C, necesitamos un criterio para eliminar las partes de Q_A y Q_C que van antes o después de p y q al hacer la búsqueda binaria de l y r. Existen 9 posibilidades para la forma en que $\overline{pq}$ intersecta a A y C, ilustradas en la Figura 2.7. Llamamos a p o q *cóncavo*, *soporte* o *reflejo* dependiendo de cómo intersecta al cierre convexo correspondiente.

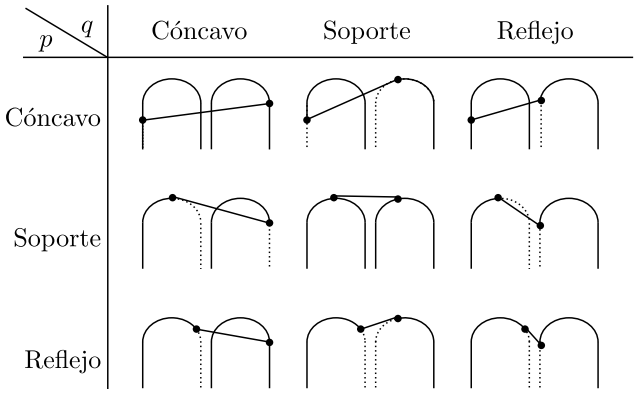

Figura 2.7: Los casos que se pueden presentar al buscar el puente entre dos cierres convexos superiores.

En el caso soporte-soporte, hemos terminado. En los demás casos, las partes punteadas de los cierres convexos son las que podemos descartar en la búsqueda binaria. Por ejemplo, en los casos reflejo-reflejo, soporte-reflejo y reflejo-soporte, podemos eliminar la parte de Q_A que está después de p y la parte de Q_C que está después de q. El único caso en el cual no es inmediato saber qué parte de qué cierre convexo podemos descartar es el caso cóncavo-cóncavo. Por ejemplo, si r se encuentra en la parte de ConvU(C) a la izquierda de q, l puede estar tanto a la derecha como a la izquierda de p en ConvU(A).

Sean ℓ una línea vertical que separa a A y a C, ℓ_p una tangente a ConvU(A) por p, ℓ_q una tangente a ConvU(C) por q y m la línea que pasa por p y q. Ocurren dos casos. Supongamos que el punto donde se intersectan ℓ_p y ℓ_q se encuentra a la izquierda de ℓ (primer caso de la Figura 2.8). Entonces r sólo puede estar a en el área sombreada o la derecha de q. Esto implica que podemos descartar la parte de ConvU(A) a la izquierda de p. Si el punto donde se intersectan ℓ_p y ℓ_q se encuentra a la derecha de ℓ ocurre algo análogo y podemos descartar la parte de ConvU(C) a la derecha de q.

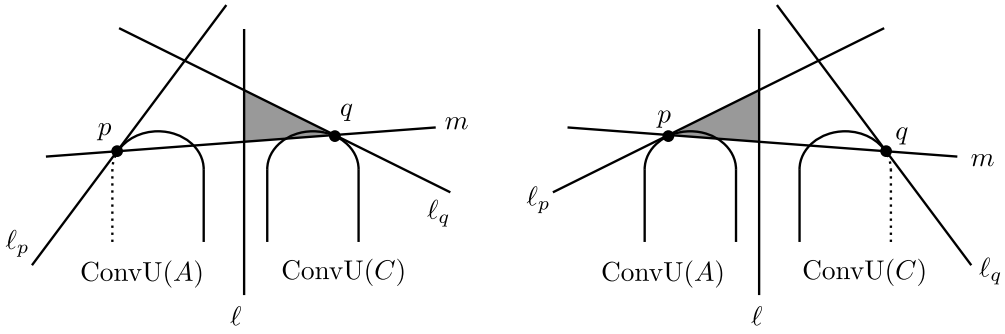

Figura 2.8: Los subcasos del caso cóncavo-cóncavo.

Podemos clasificar a p y q como vértices soporte, feflejo o cóncavos en tiempo constante y en todos los casos podemos localizar al menos un segmento de ConvU(A) o ConvU(C) que se puede descartar también en tiempo constante, por lo que podemos encontrar el puente B en tiempo $O(\log n)$. ■

Veamos ahora cómo utilizar el Teorema 2.4 para mantener ConvU(P) con inserciones y borrados. Una opción es mantener un árbol autobalanceable T que

guarde en sus hojas los puntos del conjunto y que que en cada vértice interno v guarde el cierre convexo superior de los vértices en las hojas del subárbol de v (abusando de la notación, nos referiremos a este cierre convexo como ConvU(v)), representado por a su vez por un árbol autobalanceable Q_v. Esta estrategia tiene el siguiente problema de eficiencia. Supongamos que u y w son los hijos de v y que Q_u y Q_w son los árboles correspondientes. Podemos construir Q_v a partir de Q_u y Q_w, pero en el proceso debemos dividir estos últimos árboles, perdiendo la información que guardaban. Si deseamos conservar esta información, necesitaremos más tiempo que $O(\log n)$ para copiar los segmentos adecuados de Q_u y Q_w y después unirlos para formar Q_v. Es aquí donde podemos explotar la estructura de ConvU(v). Como ConvU(v) está formado por una parte inicial de ConvU(u), una parte final de ConvU(w) y un puente entre dicha partes, bien podríamos "cortar" dichas partes de Q_u y Q_w y pasarlas a Q_v, dejando en los vértices u y w sólo los fragmentos que no contribuyen a ConvU(v). Si mantenemos un registro en v con la información del lugar donde se puso el puente al construir Q_v, simplemente tenemos que dividir Q_v en dicha parte para recuperar las piezas que, al ser unidas con Q_u y Q_w, forman ConvU(u) y ConvU(w).

Con esta estrategia, podemos recorrer T hacia abajo y reconstruir los árboles Q en cada vértice por el que pasemos haciendo las divisiones adecuadas y pasando las piezas correspondientes de un vértice a sus hijos. Además, podemos regresar por la misma trayectoria y reconstruir los árboles Q haciendo el proceso inverso, es decir, haciendo divisiones en un hijo y pasando la pieza adecuada a su padre. Descender en búsqueda de un vértice es sencillo y sólo requiere $O(\log n)$ operaciones de división/unión de árboles. Subir desde un vértice una vez hechas las operaciones de un descenso también toma $O(\log n)$ pasos, pero requiere de encontrar el puente entre pares de cierres convexos superiores. Antes de detallar los procesos de ascenso y descenso en búsqueda de un vértice, resumamos la información que contienen los vértices de T. Cada vértice v de T almacena:

- Un árbol autobalanceable Q_v con la parte de ConvU(v) que no aparece en ConvU($\pi(v)$) (en particular, si v es la raíz de T, Q_v contiene a ConvU(P)).
- La coordenada x más grande de los puntos en las hojas del subárbol con raíz v. Esta coordenada hará la función de llave de v.

- Un punto b denotado por $v.b$: el punto que es el extremo izquierdo del puente entre ConvU($v.lson$) y ConvU($v.rson$) en ConvU(v)
- Si v es una hoja de T, contiene un punto del P que también es su llave.

En la Figura 2.9 se encuentra un ejemplo del árbol T para un conjunto de puntos.

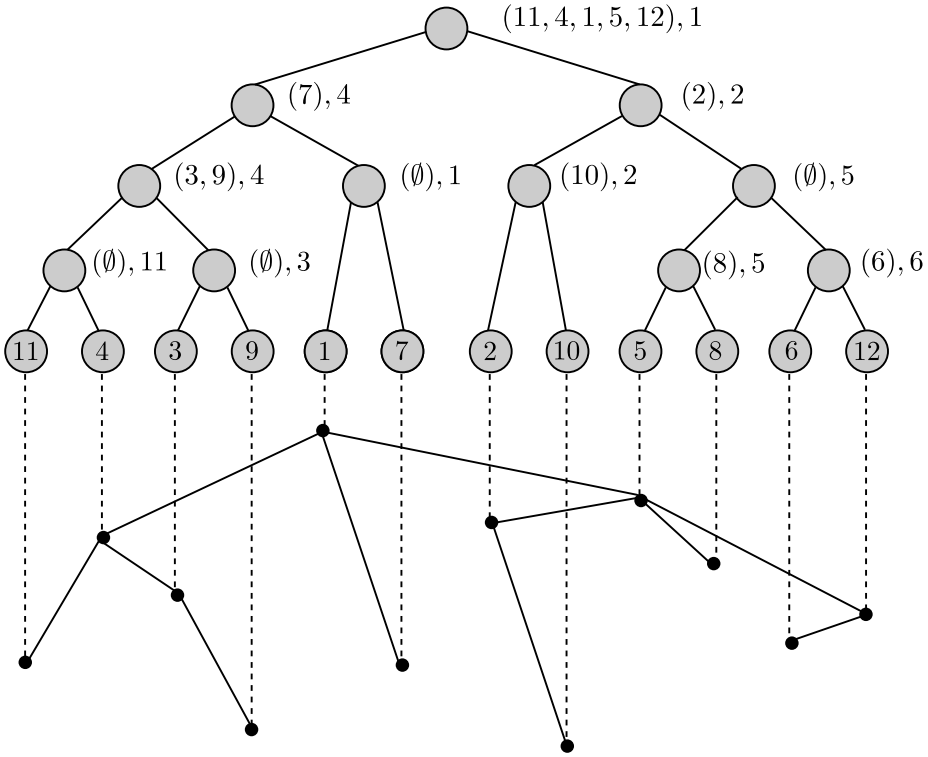

Figura 2.9: Un ejemplo de T. En cada vértice interior v, los puntos en Q_v aparecen entre paréntesis, seguidos de $v.b$.

La función DESCEND da los detalles del descenso en T en búsqueda de un vértice.

En una llamada a DESCEND, u es el vértice en el cual nos encontramos y k la llave del vértice que buscamos. Así, basta con llamar a DESCEND($T.root, p$) para encontrar la hoja que contiene al punto p (o la hoja donde debería insertarse el punto p). Como T está balanceado, se recorren $O(\log n)$ vértices, y en cada

Algoritmo 2.1 Algoritmo para descender por T en búsqueda de un vértice

```
1:  procedure DESCEND(u,k)
2:      if u es una hoja then
3:          return u
4:      else
5:          Q_u1, Q_u2 = Q_u.split(u.b)
6:          Q_u1.insert(u.b)
7:          Q_u.lson = Q_u1.join(Q_u.lson)
8:          Q_u.rson = Q_u.rson.join(Q_u2)
9:          if k < u.key then
10:             DESCEND(u.lson, k)
11:         else
12:             DESCEND(u.rson, k)
13: end procedure
```

uno se hacen operaciones que requieren tiempo $O(\log n)$. Así, una llamada a DESCENT toma tiempo $O(\log^2 n)$. Es importante notar una concecuencia de llamar a DESCEND: los árboles Q_v de cada vértice v con padre en la trayectoria recorrida contienen la representación completa de ConvU(v). Los árboles en vértices que aparecen en la trayectoria recorrida (a excepción del último vértice) quedan temporalmente vacíos.

La utilidad de DESCEND es que nos permite buscar el lugar apropiado para insertar un punto o nos da la hoja que contiene un punto que queremos eliminar. Una vez hecha la inserción/borrado, basta con subir por la trayectoria recorrida en el descenso, reconstruyendo los árboles en cada vértice hasta llegar a la raíz. Tenemos toda la información necesaria, ya que los árboles Q que contribuyen a los cierres convexos superiores se encuentran completos. El único inconveniente es que tras la inserción/borrado, T puede haber quedado desbalanceado. El procedimiento ASCEND es el encargado de esto.

En vista del Teorema 2.4, asumiremos que contamos con una función BRIDGE(Q_u, Q_w) que recibe las representaciones por arboles ConvU(A) y ConvU(B), donde A y B son conjuntos de puntos que pueden ser separados por una línea vertical, y devuelve una tupla (Q_1, Q_2, Q_3, Q_4, b) donde Q_1 es la porción de Q_u que participa en Conv($A \cup B$), Q_2 es la porción restante de Q_u, Q_4 es la porción de Q_w que participa en Conv($A \cup B$), Q_3 es la porción restante de Q_w, y finalmente b es el

extremo izquierdo del puente entre ConvU(A) y ConvU(B).

Algoritmo 2.2 Algoritmo para ascender por T desde un vértice v

```
1: procedure ASCEND(v)
2:     if v no es la raíz de T then
3:         Rebalancear T, de ser necesario
4:         (Q_1, Q_2, Q_3, Q_4, b) = BRIDGE(Q_v, Q_u)
5:         Q_{π(v).lson} = Q_2
6:         Q_{π(v).rson} = Q_3
7:         Q_{π(v)} = Q_1.join(Q_4)
8:         π(v).b = b
9:         ASCEND(π(v))
10: end procedure
```

En cada llamada a ASCEND, v es el vértice en el cual nos encontramos y Q_v y Q_u (donde u es el hermano de v) contienen las representaciones completas de ConvU(v) y ConvU(u). Una llamada a ASCEND recorre $O(\log n)$ vértices hasta que llega a la raíz de T. En cada paso, se requiere tiempo $O(\log n)$ para encontrar el puente entre dos cierres convexos superiores y hacer las operaciones de unión con los árboles adecuados. Por último, se efectúan las operaciones de rebalanceo necesarias en T. Así, una llamada a ASCEND requiere tiempo $O(\log^2 n+R)$, donde R es el costo de los rebalanceos a lo largo del ascenso.

El proceso de rebalanceo depende de la estructura en particular que se haya utilizado para T. Además de las operaciones usuales, hay que tener en cuenta los árboles Q en cada vértice de T al momento de rebalancear. Overmars y Van Leuween [OvL81] analizan el caso de árboles AVL y demuestran que un rebalanceo en una llamada a ASCEND puede realizarse en tiempo $O(\log n)$, con lo que el tiempo total de ASCEND es $O(\log^2 n)$. Utilizaremos este resultado para analizar la complejidad de las inserciones en T. Sin embargo, debido a que más adelante utilizaremos un treap como estructura para T, a continuación describimos las operaciones de rebalanceo asumiendo que T es un treap.

Supongamos que nos encontramos en el vértice v, que u es su hermano y sea w el padre de v. En una llamada a ASCEND. Si v.*priority* es menor a $\pi(v)$.*priority*, debemos hacer una rotación en v. Supongamos que tenemos que hacer una rotación derecha (el caso de una rotación izquierda es análogo). ASCEND garantiza que

los hijos de v tienen en sus árboles Q la información completa de los cierres convexos superiores respectivos. Tras realizar la rotación, esto ya no se cumple, ya que w ahora es el hijo izquierdo de v y tiene un árbol Q sin información válida. Sin embargo, los hijos de w tras la rotación sí tienen los árboles Q con toda la información requerida para hacer una llamada a ASCEND. Así, basta con llamar a ASCEND desde w para continuar con el ascenso a la raíz. Como el número de rotaciones esperadas es menor a dos, ASCEND toma tiempo esperado $O(\log^2 n)$ si T es un treap.

Ahora tenemos todo lo necesario para probar el siguiente teorema.

Teorema 2.5. *El cierre convexo de un conjunto de puntos en el plano puede mantenerse a un costo de $O(\log^2 n)$ por cada inserción y borrado.*

Demostración. Sólo necesitamos analizar los costos de insertar y borrar un punto en T. Para insertar un punto, una llamada a DESCEND nos indica la hoja donde debemos insertar el punto nuevo, a la vez que prepara los árboles Q a lo largo de la trayectoria recorrida para su reconstrucción. Una vez insertado el vértice nuevo, basta con llamar a ASCEND desde dicho vértice para reconstruir todos los árboles Q necesarios. Al llegar a la raíz, el árbol Q en este vértice contiene el cierre convexo superior de todo el conjunto. Basta con mantener otro árbol T' con el cierre convexo inferior para tener el cierre convexo del conjunto de puntos. El borrado de un vértice es análogo: tras llamar a DESCEND, eliminamos la hoja encontrada y llamamos a ASCEND desde el hermano de la hoja eliminada. De los análisis de ASCEND y DESCEND, se sigue que cada inserción y borrado toma tiempo $O(\log^2 n)$. ■

Finalizamos con algunos detalles a tomar en cuenta cuando T es un treap. Debido a que las T debe tener tantas hojas como puntos en el conjunto, al insertar un vértice v en T debemos asegurarnos que su prioridad sea lo suficientemente grande para que no sea necesario hacer una rotación, ya que esto podría cambiar el número de hojas. En cambio, en cada inserción creamos un vértice auxiliar (que tal vez necesite ser rotado) que será el padre de v y de la hoja que nos indica dónde insertar a v. El borrado de vértices siempre será sobre hojas, por lo que no hay rotaciones involucradas. Nótese que en cada borrado se eliminan dos vértices:

una hoja y su padre, el hermano de la hoja eliminada toma el lugar del padre. Nótese también que en todo momento, cada vértice interior tiene dos hijos.

Capítulo 3

Recorriendo un arreglo de rectas

El objetivo de este capítulo es presentar un algoritmo que facilite la búsqueda de conjuntos de puntos en posición general con distinto tipo de orden que cumplan con ciertas propiedades. Presentamos también dos problemas que son buenos candidatos para ser atacados con dicho algoritmo.

Iniciamos presentando el problema que motiva el diseño de este algoritmo, el cual está relacionado con el problema mencionado al inicio del Capítulo 1, el llamado "Problema del Final Feliz".

3.1. r-gonos y r-hoyos

El "Problema del Final Feliz" consiste en lo siguiente: dado un entero positivo r, ¿existe un número $g(r)$ tal que todo conjunto con $g(r)$ puntos en posición general en el plano contiene un subconjunto de r puntos en posición convexa? A un subconjunto de r puntos con las caracterísitas mencionadas se le conoce como r-gono. Este problema fue planteado por Esther Klein al rededor de 1933 [ES35]. Erdős nombró a a este problema "El Problema del Final Feliz" ya que Esther Klein y George Szekeres se comprometieron mientras trabajaban en él y se casaron poco después. Erdős y Szekeres demostraron que $g(r)$ existe para todo entero positivo r y dieron cotas para su valor.

Años más tarde, Edrős planteó una variante del problema. Dado un entero positivo r, ¿existe un número $h(r)$ tal que todo conjunto con $h(r)$ puntos en posición general en el plano contiene un r-gono vacío? Un r-gono vacío en un conjunto de puntos S es aquel que no contiene puntos de S en su interior, se conoce también como r-hoyo. Denotaremos al conjunto de r-hoyos de un conjunto S por $\Gamma_r(S)$.

Es inmediato que $h(3) = 3$ y puede obtenerse $h(4) = 5$ mediante un análisis del tamaño de la cerradura convexa de un conjunto de 5 puntos. Poco después de que se planteara esta variante, Harborth [Har78] demostró que $h(5) = 10$. De manera sorprendente, Horton [Hor83] demostró que $h(r)$ no existe para $r \geq 7$ al exhibir conjuntos de puntos arbitrariamente grandes en posición general sin 7-hoyos. Estos conjuntos se denominan *conjuntos de Horton*, seguimos la exposición de Matoušek [Mat02].

Definición 3.1. *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 3.2. *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 3.1. *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 3.1). ■

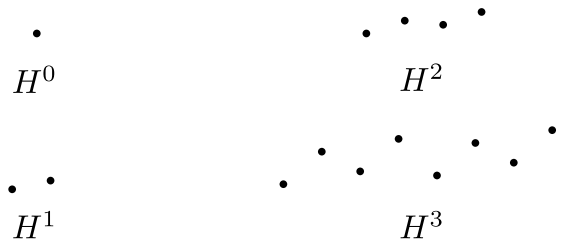

Figura 3.1: Conjuntos de Horton.

Los conjuntos de Horton son utilizados en varios problemas combinatorios sobre puntos en el plano, por lo que es de interés disponer de una manera eficiente de representarlos en la computadora. En la construcción presentada, $h_k = 3^{2^k}$ funciona (Bárány y Füredi [BF87]) pero las coordenadas de los puntos crecen de manera muy rápida conforme k aumenta. Recientemente, con Barba, Duque y Fabila-Monroy [BDMH14] mostramos que el conjunto de Horton de n puntos puede dibujarse en el plano con coordenadas enteras cuyo valor absoluto es a lo más $1/2 \cdot n^{1/2\log(n/2)}$ y cualquier conjunto con el mismo tipo de orden y coordenadas enteras contiene un punto con una coordenada en valor absoluto al menos $c \cdot n^{1/24\log(n/2)}$, donde c es una constante positiva.

El problema de determinar la existencia de $h(6)$ permaneció sin respuesta hasta 2006, cuando Nicolás [Nic07] y Gerken [Ger08] demostraron de manera

independiente su existencia. Sin embargo, el valor exacto de $h(6)$ no se conoce hasta el momento. Las mejores cotas que se conocen son $30 \leq h(6) \leq 463$, la cota superior se debe a Koshelev [Kos09b] y la inferior a Overmars [Ove03].

La cota inferior se obtuvo a través de una búsqueda por computadora, utilizando una variación del algoritmo presentado por Donkin, Edelsbrunner y Overmars [DEO90] para enumerar r hoyos en un conjunto de puntos. En esencia es una búsqueda incremental: al encontrar un conjunto de n puntos sin hexágonos vacíos se trata se extender a un conjunto de $n+1$ puntos sin hexágonos vacíos al añadir un punto que no genere 6-hoyos nuevos.

Hay numerosas variaciones de este problema. Una de ellas es colorear los puntos del conjunto y buscar r-gonos o r-hoyos monocromáticos, es decir, subconjuntos de puntos en posición convexa de un mismo color. En analogía con $g(r)$, definimos $g_m(r,k)$ como el entero mas pequeño tal que cualquier conjunto de $g_m(r,k)$ puntos en posición general coloreado con k colores contiene un r-gono monocromático. Devillers et al. [DHKS03] estudiaron esta variación y demostraron que $g_m(r,k) = k \cdot (g(r)-1)+1$. Para r-hoyos monocromáticos, exhibieron coloraciones del conjunto de Horton con tres y dos colores sin 3-hoyos monocromáticos y sin 5-hoyos monocromáticos, respectivamente. Estos resultados motivan la búsqueda de 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 et al. [AFMFP$^+$08]. Pach y Thót [PT08] mejoraron esta cota a $\Omega(n^{4/3})$, 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. Por ello, los conjuntos de Horton son buenos candidatos para intentar probar que existen conjuntos bicromáticos arbitrariamente grandes sin 4-hoyos monocromáticos, pero en Devillers et al. [DHKS03] mostraron 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 [HS09] encontraron un conjunto de 36 puntos con estas características y poco después Koshelev [Kos09a] encontró uno con 46 puntos.

Una estrategia sencilla para buscar minimizar la cantidad de r-hoyos en un conjunto S de puntos en posición general es la siguiente:

Heurística 3.1 Heurística para minimizar el número de r-hoyos

```
1: procedure MINIMIZE_RHOLES(S,min)
2:     while Γ_r(S) > min do
3:         Escoger aleatoriamente un punto p ∈ S
4:         Escoger aleatoriamente un punto q el plano (esta elección puede depender de p)
5:         if Γ_r(S \ {p} ∪ {q}) ≤ Γ_r(S) then
6:             S := S \ {p} ∪ {q}
7: end procedure
```

Podemos generar puntos aleatoriamente de manera sencilla con la computadora, por lo que si se cuenta con un algoritmo que actualice de manera eficiente la cantidad de r-hoyos en S al mover un punto, esta heurística puede implementarse fácilmente. El autor [HT13] implementó esta heurística junto con un algoritmo que actualiza Γ_r al cambiar p por q en tiempo proporcional al número de r-hoyos que contienen a p y a q (este algoritmo es una modificación del presentado por Dobkin, Edelsbrunner y Overmars [DEO90]). Aunque se encontraron conjuntos de puntos con pocos 6-hoyos y pocos 4-hoyos monocromáticos en conjuntos bicoloreados, no fue posible mejorar las cotas conocidas.

Como se vio en el Capítulo 1, el número de r-hoyos de conjuntos de puntos con el mismo tipo de orden no cambia. Nótese que debido a esto, la heurística anterior puede utilizarse para extender conjuntos de puntos con propiedades invariantes bajo el tipo de orden.

3.2. Número de cruce rectilíneo

El segundo problema que presentamos en esta sección es el de *número de cruce rectilíneo*. Este problema tiene origen en el llamado "Problema de la fabrica de ladrillos" de Turán, que preguntaba por el menor número de cruces de un dibujo de $K_{m,n}$ en el plano. Poco después, Erdős y Guy [EG73] se preguntaron por el menor número de cruces de un dibujo rectilíneo de K_n en el plano (es decir, un dibujo donde los vértices son puntos en posición general y las aristas son segmentos de líneas rectas). A este número, denotado por $\overline{cr}(K_n)$ se le conoce como número de cruce rectilíneo.

Este problema se relaciona con la búsqueda de r-gonos de la siguiente manera. Dados cuatro puntos en el plano en posición general, los segmentos de recta que los unen generan un cruce si y sólo si los puntos están en posición convexa. Así, dado un conjunto de n puntos en el plano, el número de cruce rectilíneo del dibujo de K_n que generan es igual al número de cuadriláteros en el conjunto. Es claro que dos conjuntos de puntos con el mismo tipo de orden tienen el mismo número de cruce rectilíneo. Utilizando este hecho, Aichholzer y Kraser [AK01] determinaron el valor exacto de $\overline{cr}(K_n)$ para $n \leq 11$ utilizando la base de datos de tipos de orden.

Como el número de cruce es invariante para conjuntos de puntos con el mismo tipo de orden, es posible utilizar la Heurística 3.1 para buscar conjuntos de puntos con número de cruce pequeño. Una motivación para encontrar conjuntos con número de cruce pequeño es el siguiente Teorema, mostrado por Ábrego et al. [ÁCFM$^+$10].

Teorema 3.1. *Sea S un conjunto de m puntos en el plano en posición general, con*

m impar. Entonces

$$\overline{cr}(K_n) \leq \frac{24cr(S) + 3m^3 - 7m^2 + (30/7)m}{m^4}\binom{n}{4} + \Theta(n^3)$$

Es decir, conjuntos con número de cruce pequeño son buenos candidatos para encontrar una mejor cota de $\overline{cr}(K_n)$. Fabila-Monroy y López [FL14] utilizaron la Heurística 3.1 con un algoritmo que actualiza el número de cruce de un conjunto tras mover un punto en $O(n^2)$ en los conjuntos de puntos con número de cruce pequeño disponibles en la página de Aichholzer `www.ist.tugraz.at/aichholzer/research/rp/triangulations/crossing/` y lograron disminuir el número de cruce de la mayoría de los 100 conjuntos disponibles. Más aún, utilizando el conjunto de 75 puntos que mejoraron y el Teorema 3.1 obtuvieron la siguiente cota, que es la mejor hasta el momento:

$$\overline{cr}(K_n) \leq \frac{9363184}{24609375}\binom{n}{4} + \Theta(n^3) < 0.380473\binom{n}{4} + \Theta(n^3)$$

Finalizamos esta sección con los resultados obtenidos por Duque [Duq14]. Mediante el uso de un algoritmo que calcula el cambio en el número de cruce de un conjunto al mover un punto en tiempo $O(n)$ amortizado y la Heurística 3.1 logró mejorar varios los conjuntos de puntos disponibles en la página de Aichholzer, además de dar conjuntos de n puntos con número de cruce pequeño para $101 \leq n \leq 350$.

3.3. Caminata sobre las celdas de un arreglo de rectas

En la Heurística 3.1 puede ocurrir que al momento de cambiar p por q, éste último se encuentre en la misma celda del arreglo de rectas inducido por $S \setminus \{p\}$ que p. También puede suceder el caso contrario, es decir, que nunca encontremos candidatos que pertenezcan a celdas de área pequeña en el arreglo de rectas inducido por $S \setminus \{p\}$. Un resultado de Goodman, Pollack y Sturmfels [GPS90] nos permite formalizar esta situación. Dada una configuración de n puntos $\mathscr{C}$, su *extensión* se define como la razón entre el triángulo de área más grande con vértices

en $\mathscr{C}$ y el triángulo de área más pequeña con vértices en $\mathscr{C}$. La *extensión intrínseca* de S es la extensión mínima sobre las configuraciones con el mismo tipo de orden de $\mathscr{C}$.

Teorema 3.2. *Sea $\sigma(n)$ el máximo de la extensión intrínseca sobre todas las configuraciones con n puntos en posición general en el plano. Entonces $\sigma(n) = \Theta(2^{2^n})$.*

Es decir, la razón entre el área del triángulo más grande y el área del triángulo más chico con vértices en un conjunto de puntos es doblemente exponencial. Esto implica que existen conjuntos de puntos donde la razón entre el área de la celda más grande y la celda más pequeña del arreglo de rectas inducido por el conjunto es doblemente exponencial.

Esto motiva una manera distinta de escoger a q: en vez de generar un punto aleatorio, podemos escoger una celda aleatoria del arreglo de rectas de $S \setminus \{p\}$ y tomar cualquier punto q dentro de dicha celda.

Una opción para lograr esto es construir la LDCA del arreglo de rectas correspondiente. Aunque esto nos permite recorrer el arreglo en tiempo constante por cada paso, necesitamos tiempo y espacio $O(n^4)$ para construir la LDCA. Esto es poco deseable si se desea recorrer sólo una porción de las celdas del arreglo.

Una manera de hacer esta elección utilizando las herramientas de los capítulos anteriores es la siguiente. Sea S un conjunto de n puntos en posición general sin puntos con la misma coordenada x y p un punto del plano que no pertenece a S. Sea L el conjunto de rectas definidas por dos puntos de S, $L_u(p)$ el subconjunto de rectas de L que se encuentran arriba de p y L_l el subconjunto de rectas de L que se encuentran debajo de p. La celda f de $A(L)$ donde se encuentra p está acotada por $\mathrm{LE}(L_u) \cap \mathrm{UE}(L_l)$. Podemos movernos a una celda adyacente a f cruzando una de las aristas que acotan a f, digamos, e. Supongamos que la línea de ℓ que contiene a e pertenece a L_u. Entonces basta con hacer $L_u = L_u \setminus \{\ell\}$ y $L_l = L_l \cup \{\ell\}$ y recalcular $\mathrm{LE}(L_u)$ y $\mathrm{UE}(L_l)$ para obtener la celda nueva.

Esto sugiere de inmediato utilizar el algoritmo del Capítulo 2. Las envolturas $\mathrm{UE}(L_l)$ y $\mathrm{LE}(L_u)$ corresponden a $\mathrm{ConvL}(L_l^*)$ y $\mathrm{ConvU}(L_u^*)$, por lo que podemos utilizar dos estructuras de datos T_u y T_l para mantener dichos cierres convexos.

A diferencia de mantener un cierre convexo de puntos, no todos los puntos en ConvL(L_l^*) ∪ ConvU(L_u^*) nos servirán para determinar la frontera de la celda que estamos construyendo. Es decir, no necesariamente todas las rectas de UE(L_l) y LE(L_u) contribuyen a la frontera de f, véase la Figura 3.2 para un ejemplo.

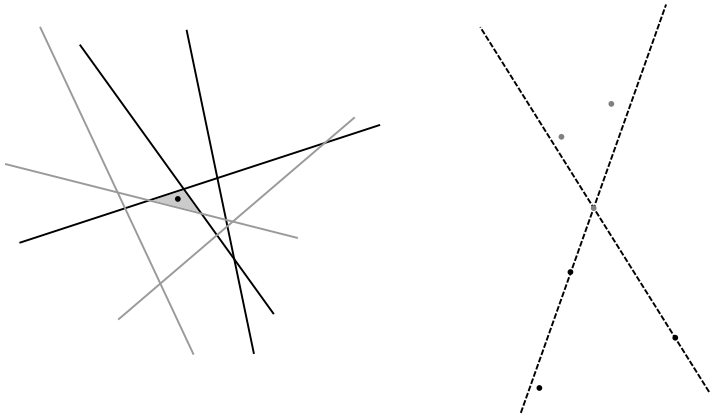

Figura 3.2: Las rectas que forman la frontera de una celda y sus duales.

Los puntos donde se intersectan las fronteras de UE(L_l) y LE(L_u) cumplen que se encuentran debajo de toda línea de UE(L_l) y se encuentran arriba de toda línea de LE(L_u). En el dual, esto se traduce en (a lo más) dos líneas que pasan cada una por un punto de ConvL(L_U^*) y ConvU(L_l^*) y separan a L_U^* y L_l^*. Estas líneas reciben el nombre de *líneas críticas de soporte*. Toussaint [Tou83] presentó una aplicación de un algoritmo de Shamos [Sha78] llamado *rotating callipers* para encontrar las líneas de soporte críticas de dos polígonos convexos en tiempo proporcional al número de vértices de los polígonos. En nuestro caso, podemos aprovechar que nuestros polígonos son cierres convexos superiores e inferiores, además de estar representados en por un árbol binario de búsqueda, para encontrar las líneas de soporte críticas de manera más rápida.

Sea ℓ una de estas líneas y sean p_i, q_j los puntos de ConvL(L_l^*) y ConvU(L_u^*) por los que pasa, respectivamente. Supongamos sin pérdida de generalidad que ℓ está dirigida de p_i a q_j y que ConvU(L_u^*) se encuentra a la derecha de ℓ. Si p_i, q_j no son los puntos con coordenada x más chica o más grande del cierre convexo al que pertenecen, la línea que pasa por p_{i-1} y p_i intersecta a ConvU(L_u^*) y la línea que pasa por q_i y q_{i+1} intersecta a ConvL(L_l^*). Además, debe ocurrir al menos uno

de los siguientes casos:

- La línea dirigida de p_i a p_{i+1} deja a ConvU(L_u^*) a la derecha.
- La línea dirigida de q_{i-1} a q_i deja a ConvL(L_l^*) a la izquierda.

Entonces podemos encontrar las líneas de soporte críticas en tiempo $O(\log^2 n)$ de la siguiente manera. Recordemos que los T_u y T_l almacenan los puntos de los cierres convexos correspondientes ordenados por coordenada x. Hacemos una búsqueda binaria sobre T_l para encontrar el punto p_i tal que la línea dirigida de p_{i-1} a p_i intersecta a ConvU(L_u^*) y la línea dirigida de p_i a p_{i+1} deja a ConvU(L_u^*) a la derecha (podemos determinar si las líneas dirigidas intersectan o no a ConvU(L_u^*) con una búsqueda binaria sobre T_u). Si ningún punto de T_l tiene las características mencionadas, repetimos el proceso en T_u. Las búsquedas binarias son posibles gracias a los apuntadores que cada elemento del árbol tiene a su actecesor y sucesor. Este proceso toma tiempo $O(\log^2 n)$, y tras repetirlo a lo más 4 veces, obtenemos los puntos que definen a las líneas de soporte críticas. Nótese que si sólo existe una línea de soporte crítica es porque la celda que estamos analizando no es acotada.

Así, podemos construir la frontera de la celda f que contiene a p en tiempo $O(n^2 \log^2 n)$, después de esto podemos determinar cualquier celda adyacente a f en tiempo $O(\log^2 n)$ y utilizamos espacio $O(n^2)$.

El procedimiento descrito para recorrer las celdas del arreglo mantiene los duales de todas las rectas del arreglo en las estructuras T_u y T_l, pero no se necesitan todas las rectas para calcular la frontera de la celda f. Dado un punto q en S, las rectas que pasan por q y los demás puntos de S dividen el plano en cuñas. El punto p se encuentra en una de estas cuñas, y los rayos que la definen pertenecen a las únicas rectas que pasan por q y pueden formar parte de la frontera de f (Figura 3.3).

Sea $S_q = \{S \setminus \{q\}\} \cup \{p' | p'$es antipodal a algún punto de S respecto a $q\}$. Entonces podemos identificar la cuña que contiene a p por los puntos de S_q que definen los rayos de su frontera. Llamemos *antecesor* al punto que aparece primero en el orden radial de S_q alrededor de q y *sucesor* al punto que aparece después

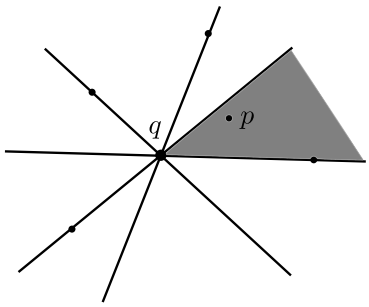

Figura 3.3: La cuña donde se encuentra p respecto a q.

en el orden radial de S_q alrededor de q. Entonces, al cruzar una arista de f podemos pensar que estamos moviendo a p a una cuña adyacente, por lo que sólo necesitamos actualizar al antecesor y sucesor de p.

Así, sólo necesitamos almacenar en cualquier momento a lo más $2n$ líneas en T_u y T_l: las definidas por el antecesor y sucesor de p respecto a cada punto de S.

El algoritmo luce ahora como sigue. Primero construimos los conjuntos S_q para cada q en S, y determinamos los antecesores y sucesores de p en cada uno de los órdenes radiales. Ordenar los conjuntos toma tiempo $O(n^2 \log n)$ y podemos determinar los antecesores y sucesores en tiempo $O(n \log n)$ haciendo búsquedas binarias. Para cada antecesor y sucesor de p, determinamos en tiempo constante si la recta que define se encuentra arriba o abajo de p, e insertamos su dual en T_u o T_l, lo cual lleva tiempo $O(\log^2 n)$. Una vez hecho esto, obtenemos las líneas que definen la frontera de f mediante las líneas de soporte críticas de los cierres convexos almacenados en T_u y T_l en tiempo $O(\log^2 n)$. Por último, obtenemos los puntos de intersección de dichas líneas. Este último paso toma tiempo esperado constante ya que el número de aristas de una celda en $A(L)$ es en promedio menor a 6. Esto es porque podemos ver a $A(L)$ como una gráfica plana, donde cada celda es un vértice y el grado promedio de una gráfica plana es menor a 6.

Podemos actualizar los antecesores y sucesores necesarios al cruzar una arista de f en tiempo constante: basta con guardar los conjuntos S_q ordenados y actualizar apuntadores a los antecesores y sucesores. Supongamos que la línea ℓ que contiene a la arista que cruzamos contiene a un antecesor respecto al punto q y está almacenada en T_u. Entonces borramos ℓ^* de T_u y lo insertamos en T_l. Tam-

bién debemos borrar al dual del sucesor actual de T_u o T_l e insertar al antecesor nuevo en la estructura adecuada. Una vez completadas estas inserciones y borrados, obtenemos la frontera de la celda nueva como se hizo anteriormente. Así, utilizamos tiempo $O(\log^2 n)$ cada vez que cruzamos una arista. Esto prueba el siguiente teorema:

Teorema 3.3. *Sea S un conjunto de puntos en posición general en el plano, todos con coordenada x distinta. Sea p un punto que no pertenece a S. Podemos construir la celda del arreglo de rectas inducido por S contiene a p en tiempo esperado $O(n^2 \log n)$. Una vez construida dicha celda, podemos construir cualquier celda adyacente en tiempo esperado $O(\log^2 n)$.*

Vale la pena hacer algunos comentarios al respecto de estas modificaciones. A pesar de que el tiempo utilizado al cruzar una arista es el mismo asintóticamente en ambas maneras de recorrer el arreglo y que en la segunda manera se hacen dos inserciones y dos borrados en las estructuras T_u y T_l contra una inserción y un borrado en la primera, en la segunda forma las estructuras T_u y T_l tienen tamaño lineal. Al ser estructuras un tanto complejas, en una implementación es deseable que tengan tamaño pequeño para que las búsquedas, inserciones y borrados tomen poco tiempo.

Es claro que podemos seguir cruzando aristas cada que obtenemos una celda, por lo que si realizamos una búsqueda en profundidad es posible recorrer todo el arreglo en tiempo $O(n^4 \log^2 n)$. Esto se compara de manera desfavorable con el uso de una LDCA, pero si la cantidad de celdas a recorrer es $O(n^3)$, se obtiene una mejoría en tiempo y espacio.

Capítulo 4

Implementación y resultados

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

El objetivo es conseguir una herramienta que permita hacer búsquedas de puntos con distinto tipo de orden, en particular, conjuntos con pocos hexágonos vacíos y con número de cruce pequeño.

4.1. Lenguaje e implementación

Debido a que las búsquedas descritas emplean mucho la generación de números de números aleatorios (para generar puntos aleatorios en el plano, generar prioridades de los vértices de un treap, elegir qué arista cruzar en una celda, etc.) y las coordenadas de los puntos pueden crecer muy rápidamente al moverlos, es deseable elegir un lenguaje de programación con un buen soporte para la generación de números aleatorios y que ofrezca precisión arbitraria de números enteros.

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 biblioteca estándar han

propiciado su uso en cómputo científico [Lan09]. Estas características, junto con su soporte para enteros de longitud arbitraria y generación de números aleatorios lo hacen un buen candidato para la implementación.

Otra razón para la elección de Python, es la disponibilidad de la librería *PyDCG* (Python's Discrete and Combinatorial Geometry Library) desarrollada por Fabila-Monroy y en la cual ha colaborado el autor (`http://www.PyDCG.org`). En esta librería se encuentran implementados los algoritmos mencionados en el Capítulo 3 para búsqueda de r-hoyos y cálculo de número de cruce, junto con varias primitivas geométricas útiles para los propósitos de este trabajo.

La estructura elegida para implementar el algoritmo de mantenimiento dinámico de cerradura convexa fue el treap, debido a la poca dificultad que representa implementar esta estructura comparada con otras que cumplen la misma función, como árboles AVL, árboles rojo-negros o árboles AA.

Debido a que los conjuntos de puntos sobre los cuales realizamos búsquedas tienen coordenadas enteras, las líneas del arreglo tendrán pendiente e intersección con el eje y racionales, por lo que sus duales tendrán coordenadas racionales. Afortundamente Python provee soporte para operaciones con números racionales a través de su biblioteca `fractions`.

La caminata aleatoria sobre las celdas del arreglo de rectas se implementó como una búsqueda en profundidad. Como el algoritmo del capítulo 3 sólo guarda información de la celda que visita actualmente, es necesaria una manera de recordar las celdas que ya se han visitado. Para esto se utilizó una tabla hash: como las celdas del arreglo comparten a lo más dos vértices, basta con elegir como llave tres vértices de la celda (por ejemplo, los tres vértices más a la izquierda de su cierre convexo superior). Así, obtenemos una llave de tamaño pequeño que nos permite verificar en tiempo esperado constante si ya hemos visitado una celda.

Las implementaciones de los algoritmos para mantenimiento dinámico de cerradura convexa y para recorrido aleatorio de las celdas de un conjunto de rectas se encuentran disponibles en la librería PyDCG.

4.2. Resultados

Como ya se mencionó en capítulos anteriores, la búsqueda sobre las celdas de un conjunto rectas es una herramienta que permite buscar conjuntos de puntos con propiedades invariantes bajo tipos de orden. El problema que motivó el desarrollo e implementación de este explorador de puntos fue el de búsqueda de puntos con pocos 6-hoyos. Sin embargo, el uso de este algoritmo no produjo mejoras a los conjuntos que proveen la mejor cota conocida.

El caso contrario ocurrió para búsquedas de conjuntos con número de cruce rectilíneo pequeño. Se tomaron los conjuntos disponibles en `http://www.ist.tugraz.at/aichholzer/research/rp/triangulations/crossing/` y se utilizó la Heurística 3.1 junto con el explorador de puntos para intentar mejorarlos. Los valores de n para los cuales se obtuvieron conjuntos con menor número de cruce se encuentran en la Tabla 4.1.

n	Cota anterior	Cota nueva	n	Cota anterior	Cota nueva
44	49370	49368	80	587280	587278
49	77420	77416	81	617930	617926
52	99161	99158	83	682976	682974
55	125207	125199	84	717276	717267
57	145164	145162	88	867887	867880
58	156042	156041	89	908940	908936
59	167506	167502	90	951379	951376
61	192265	192263	91	995478	995464
62	205634	205633	92	1040946	1040945
65	249962	249960	93	1087899	1087893
72	380925	380923	94	1136586	1136582
73	403180	403178	95	1186887	1186879
74	426398	426391	96	1238662	1238652
76	475773	475766	97	1292312	1292294
78	529278	529273	98	1347559	1347546
79	557743	557741	100	1463457	1463426

Tabla 4.1: Conjuntos encontrados con número de cruce pequeño de tamaño a lo más 100.

El mismo procedimiento se llevó a cabo con los conjuntos de tamaño mayor

a 100 encontrados por Duque [Duq14], los valores de n para los cuales se obtuvieron conjuntos con menor número de cruce se ilustran en las Tablas 4.2 a 4.5.

n	Cota anterior	Cota nueva	n	Cota anterior	Cota nueva
101	1524363	1524129	226	39967932	39943222
102	1587379	1586592	227	40690202	40661714
103	1651701	1651033	228	41393002	41388455
104	1718190	1717472	229	42148691	42127243
105	1786348	1785770	230	42904959	42874851
106	1857476	1856188	231	43646897	43631903
107	1932014	1928648	232	44457791	44400745
108	2003474	2003085	233	45201980	45177432
109	2080837	2079817	234	45994665	45964682
110	2159583	2158648	235	46819654	46762928
111	2240295	2239573	236	47591238	47570861
112	2324364	2323027	237	48429507	48388793
113	2409440	2408650	238	49255211	49218166
114	2498096	2496492	239	50108343	50057535
115	2588274	2586978	240	50929213	50907518
116	2681130	2679732	241	51792676	51768904
117	2777239	2774957	242	52662503	52640674
118	2874020	2872784	243	53548518	53522838
119	2974297	2973129	244	54448550	54417596
120	3076889	3075945	245	55372365	55322457
121	3182896	3181708	246	56254725	56238381
122	3291158	3290006	247	57219945	57166791
123	3402484	3400956	248	58155830	58105463
124	3516383	3514885	249	59086520	59056046
125	3634075	3631535	250	60045485	60018286
126	3752531	3750953	251	61057738	60992298
127	3875191	3873499	252	62011095	61977595
128	4001638	3998992	253	63022207	62975247
129	4129382	4127226	254	64024394	63985390
130	4261216	4258949	255	65047915	65006643
131	4396636	4393652	256	66071975	66041085
132	4533324	4531243	257	67125136	67087298
133	4675375	4672417	258	68178300	68145644

Tabla 4.2: Conjuntos encontrados con número de cruce pequeño de tamaño a lo más 350.

n	Cota anterior	Cota nueva	n	Cota anterior	Cota nueva
134	4819577	4816682	259	69234568	69216412
135	4968225	4964133	260	70338388	70300463
136	5118296	5115219	261	71496995	71396883
137	5272750	5269624	262	72563923	72506569
138	5430179	5427333	263	73662671	73628293
139	5591372	5588912	264	74796672	74762844
140	5757277	5753825	265	75958712	75911480
141	5925299	5922291	266	77088834	77072947
142	6095659	6094693	267	78303346	78246754
143	6275343	6270773	268	79474190	79434995
144	6456243	6450326	269	80672129	80636709
145	6637090	6634156	270	81895879	81851095
146	6825602	6821619	271	83180837	83079681
147	7017254	7012964	272	84391834	84321795
148	7213170	7208448	273	85662920	85578768
149	7413187	7408085	274	86875801	86848655
150	7615789	7611560	275	88174330	88132992
151	7821117	7819447	276	89464024	89432196
152	8034808	8031520	277	90970203	90743785
153	8251288	8247536	278	92119047	92072486
154	8471159	8468410	279	93467607	93414133
155	8702792	8693498	280	94791085	94769882
156	8927970	8922772	281	96219476	96142702
157	9160623	9156969	282	97553293	97529051
158	9420129	9395396	283	98983218	98929724
159	9641579	9638513	284	100392995	100347470
160	9890798	9886520	285	101833899	101778449
161	10143846	10139308	286	103289131	103225498
162	10401152	10396521	287	104750511	104689155
163	10662682	10659086	288	106226420	106168314
164	10932453	10926393	289	107703076	107662716
165	11204313	11198421	290	109189474	109174202
166	11488239	11475987	291	110783682	110700740
167	11761915	11758475	292	112325073	112242836
168	12056361	12045811	293	113886359	113799009

Tabla 4.3: Conjuntos encontrados con número de cruce pequeño de tamaño a lo más 350.

n	Cota anterior	Cota nueva	n	Cota anterior	Cota nueva
169	12344936	12338994	294	115421397	115372313
170	12646681	12637145	295	117105135	116960697
171	12949212	12940428	296	118612533	118565856
176	14548328	14541227	301	126913315	126840617
177	14889078	14878168	302	128727584	128546391
178	15229501	15221354	303	130419217	130268167
179	15585550	15570236	304	132119300	132010744
180	15939174	15924727	305	133814406	133767652
181	16292667	16285919	306	135660719	135543563
182	16661561	16652855	307	137470862	137338058
183	17032887	17025703	308	139258419	139148959
184	17415844	17405333	309	141096217	140978539
185	17809092	17791058	310	142930067	142827476
186	18194936	18182780	311	144778693	144691459
187	18604056	18581488	312	146651337	146574597
188	18994427	18986583	313	148567768	148475568
189	19408599	19397947	314	150520766	150394070
190	19827308	19816368	315	152416948	152330718
191	20258559	20241309	316	154357760	154287831
192	20683408	20672890	317	156383661	156262551
193	21126452	21111674	318	158345141	158256229
194	21567385	21557236	319	160420961	160269863
195	22024364	22009572	320	162432125	162301705
196	22490055	22469585	321	164522670	164352762
197	22946687	22936685	322	166543800	166424029
198	23424080	23410041	323	168628548	168513696
199	23907057	23892097	324	170754601	170624536
200	24398216	24380697	325	172903377	172755111
201	24900734	24876732	326	175012691	174903987
202	25401556	25380947	327	177179575	177075643
203	25904104	25892326	328	179414864	179266673
204	26423669	26410985	329	181618714	181475612
205	26951389	26938261	330	183844053	183710064
206	27496349	27472907	331	186073866	185962814
207	28031897	28015191	332	188330228	188234510

Tabla 4.4: Conjuntos encontrados con número de cruce pequeño de tamaño a lo más 350.

n	Cota anterior	Cota nueva	n	Cota anterior	Cota nueva
208	28584211	32046822	339	204957646	204736720
215	32670021	32656321	340	207549963	207181406
216	33298262	33273214	341	210042769	209643205
217	33926386	33899968	342	212361044	212127350
218	34544238	34535013	343	215124267	214635817
219	35200148	35178722	344	217561663	217163991
220	35845814	35832059	345	220242248	219718290
221	36511281	36493557	346	222633976	222290782
222	37189035	37165213	347	226033934	224889338
223	37872007	37845543	348	227910492	227511014
224	38569221	38535171	349	230670737	230152547
225	39240526	39233760	350	233058661	232821165

Tabla 4.5: Conjuntos encontrados con número de cruce pequeño de tamaño a lo más 350.

No se obtuvo una nueva cota asintótica a pesar de que se logró mejorar la mayoría de los conjuntos. Sin embargo, el haber mejorado estos conjuntos motiva a buscar nuevas heurísticas que, haciendo uso del explorador de puntos, puedan encontrar una cota nueva.

Bibliografía

[ÁCFM+10] B. M. Ábrego, M. Cetina, S. Fernández-Merchant, J. Leaños, and G. Salazar. 3-symmetric and 3-decomposable geometric drawings of K_n. *Discrete Appl. Math.*, 158(12):1240–1458, 2010.

[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.

[AH74] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[AIL+14] Greg Aloupis, John Iacono, Stefan Langerman, Özgür Özkan, and Stefanie Wuhrer. The complexity of order type isomorphism. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 405–415. SIAM, 2014.

[AK01] Oswin Aichholzer and Hannes Krasser. The point set order type data base: A collection of applications and results. In *Proceedings of the 13th Canadian Conference on Computational Geometry, University of Waterloo, Ontario, Canada, August 13-15, 2001*, pages 17–20, 2001.

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*.

Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[BDMH14] Luis Barba, Frank Duque, Ruy Fabila Monroy, and Carlos Hidalgo-Toscano. Drawing the horton set in an integer grid of minimun size. In *Proceedings of the 26th Canadian Conference on Computational Geometry, CCCG 2014, Halifax, Nova Scotia, Canada, 2014*. Carleton University, Ottawa, Canada, 2014.

[BF87] Imre Bárány and Zoltán Füredi. Empty simplices in the euclidean space. *Canad. Math. Bull.*, (30):436–445, 1987.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[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.

[Duq14] Frank Duque. Algoritmo amortizado para el número de cruces rectilíneos sobre gráficas completas. Master's thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2014.

[EG73] P. Erdős and R. K. Guy. Crossing number problems. *Amer. Math. Monthly*, 80:52–58, 1973.

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

[FL14] Ruy Fabila-Monroy and Jorge López. Computational search of small point sets with small rectilinear crossing number. *Journal of Graph Algorithms and Applications*, 18(3):393–399, 2014.

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

[GP80] Jacob E. Goodman and Richard Pollack. On the combinatorial classification of nondegenerate configurations in the plane. *Journal of Combinatorial Theory, Series A*, 29(2):220 – 235, 1980.

[GP83] Jacob E. Goodman and Richard Pollack. Multidimensional sorting. *SIAM J. Comput.*, 12(3):484–507, 1983.

[GP84] Jacob E. Goodman and Richard Pollack. Semispaces of configurations, cell complexes of arrangements. *J. Combin. Theory Ser. A*, 37(3):257–293, 1984.

[GP86] Jacob E. Goodman and Richard Pollack. Upper bounds for configurations and polytopes in $\mathbb{R}^d$. *Discrete & Computational Geometry*, 1(1):219–227, 1986.

[GPS89] J. E. Goodman, R. Pollack, and B. Sturmfels. Coordinate representation of order types requires exponential storage. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 405–410, New York, NY, USA, 1989. ACM.

[GPS90] Jacob E Goodman, Richard Pollack, and Bernd Sturmfels. The intrinsic spread of a configuration in r d. *Journal of the American Mathematical Society*, pages 639–651, 1990.

[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.

[HT13] Carlos Hidalgo-Toscano. Un algoritmo para contar polígonos convexos vacíos en conjuntos de puntos en el plano. Tesis de Licenciatura, Universidad Nacional Autónoma de México, 2013.

[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.

[MP78] D.E. Muller and F.P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217 – 236, 1978.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

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

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

[OvL81] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.

[Pre79] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Commun. ACM*, 22(7):402–405, July 1979.

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

[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.

[SA96] R. Seidel and C.R. Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996.

[Sha78] M.I. Shamos. *Computational Geometry*. PhD thesis, Yale University, 1978.

[Tou83] Godfried Toussaint. Solving geometric problems with the rotating calipers. *In Proc. IEEE MELECON '83*, pages 10—02, 1983.