UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias

Autor: Adrián Aguilera Moreno

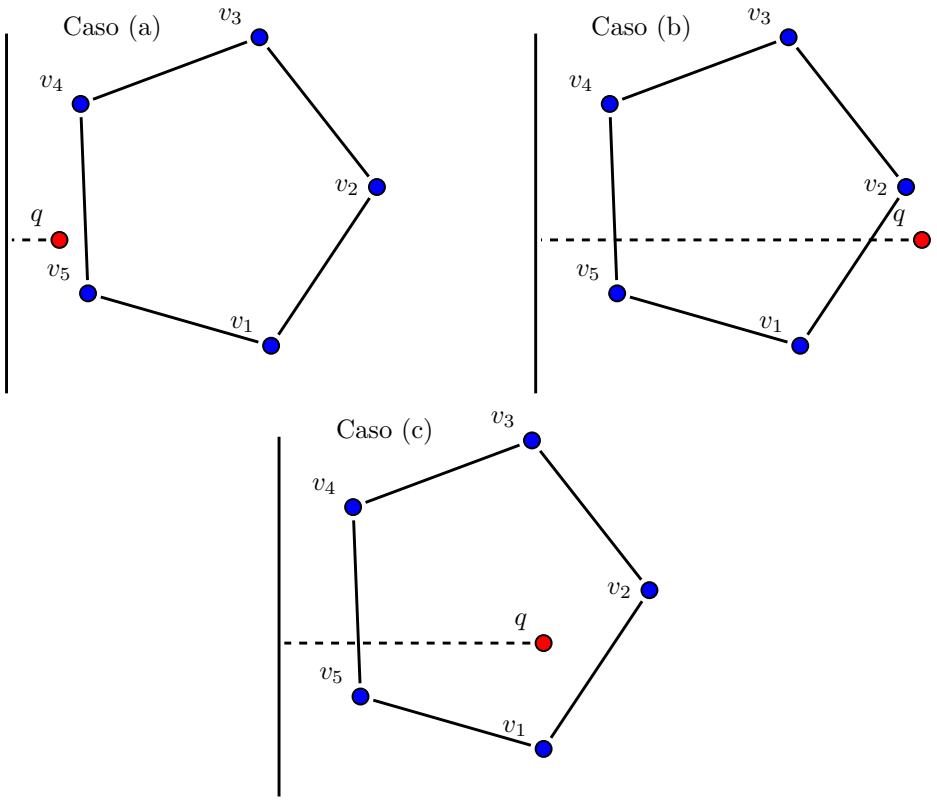
Geometría Computacional

Tarea 02

1. Sea P un polígono convexo de n vértices. Suponga que los vértices de P son dados en un arreglo ordenado. Muestra que, dado unpunto q, podemos detectar si $q \in P$ en tiempo $\mathcal{O}(\log n)$.

Demostración: Exhibamos un algoritmo en tiempo $\mathcal{O}(\log n)$ para encontrar que $q \in P$. Esto lo podemos hacer de varias maneras, en partícular podemos usar una de las estructuras de datos vistas en clase o podemos usar una técnica basa en búsqueda binaria. Por simplicidad usaré la técnica basada en búsqueda binaria, a continuación se muestra el algoritmo deseado:

1. Encontremos el extremo "más alto" (digamos a) y "más bajo" (digamos b) en P en $\mathcal{O}(1)$, pues tenemos los puntos ordenados en un arreglo.
2. Verificamor que $b.y \leq q.y \leq a.y$ en $\mathcal{O}(1)$. Si la condición anterior se cumple, entonces seguimos con el algoritmo. Si no se cumple la condición anterior, entonces concluimos que q no se encuentra destro de P.
3. Ahora tracemos una recta ℓ a partir de q tal que intersecte al polígono P respecto a una proyección, digamos la proyección hacia la izquierda. Ahora, tendremos tres posibles opciones:
   (a) ℓ intersecta en 0 puntos (Caso análogo a 2). Entonces $q \notin P$.
   (b) ℓ intersecta en 2 puntos (Caso análogo a 1). Entonces $q \notin P$.
   (c) ℓ intersecta en 1 punto, entonces $q \in P$.



Obs. Encontrar q respecto a los puntos de P se realiza en $\mathcal{O}(\log n)$. Producir las recta y encontrar las intersecciones lo realizamos en $\mathcal{O}(1)$.

¿Por qué será cierto que lo anterior es suficiente? R./ Sabemos que P es convexo, entonces toda línea trazada puede cortar a P en a lo más 2 puntos. Además, como la proyección siempre es a la izquierda podemos tomar como referencia los puntos de corte respecto al polígono[1]. En otro caso, podemos considerar una orientación distinta para la línea en la que proyectaremos ℓ.

Análisis de Complejidad. La complejidad del algoritmo esta contenida en

$$\mathcal{O}(1) + \mathcal{O}(\log n) \in \mathcal{O}(\log n)$$

□

2. Muestra que si un polígono tiene $\mathcal{O}(1)$ vértices no regulares, entonces el algoritmo de triangulación visto en clase se puede adaptar para usar tiempo $\mathcal{O}(n)$.

▷ **Solución:** Recordemos que el algoritmo visto en clase toma $\mathcal{O}(n + k \cdot \log n)$, dónde n es lo que nos cuesta realizar el barrido punto a punto (estaciones por eventos) y $k \cdot \log n$ son los k vértices une y divide (vértices no regulares), y la búsqueda que se debe hacer en el estado de la línea (cada vértice une y divide) es exactamente una búsqueda binaria en $\log n$.

Ahora, observemos que $k \in \mathcal{O}(1)$, lo que implicaría que

$$\begin{aligned} k \cdot \log n \approx \log n \quad &\Rightarrow \quad \mathcal{O}(n + k \cdot \log n) \approx \mathcal{O}(n + \log n) \\ &\Rightarrow \quad \mathcal{O}(n + \log n) = \mathcal{O}(n). \end{aligned}$$

Obs. [Acerca del orden] Se considera que tenemos un orden y que nuestro polígono no es regular. Sin embargo, se me hace curioso saber ¿Cómo obtenemos un orden en tiempo $\mathcal{O}(n)$? Sabemos que no podemos ordenar por comparaciones en menos que $\mathcal{O}(n \log n)$, sin embargo, en un barrido de línea no necesitamos ordenar con prioridades respecto a (x, y) y de hecho basta con tener un orden en x o en y. Así, podemos ordenar basándonos en una proyección, digamos que proyectamos cada punto al eje X, esto nos tomará $O(n)$, luego basta recorrer el eje X, en $\mathcal{O}(n)$, y tenemos un orden (No angular), este orden es suficiente para nuestro barrido de línea. La construcción de la estructura para el estado de la línea se hará en tiempo $\mathcal{O}(n)$, pues son n puntos y nuestras consultas son $k \in \mathcal{O}(1)$ sobre el estado de la línea que sabemos tardan $\log n$.

De esta manera, nuestra complejidad queda contenida en $\mathcal{O}(n)$. ◁

[1] Es simple plantear un pequeño sistemas de ecuación que me indique en cuántos puntos se corta respecto a ℓ, todo esto en $\mathcal{O}(1)$.

3. Demuestra que cualquier polígono admite una triangulación, incluso si el polígono tiene hoyos.

Demostración: Para este ejercicio basta exhibir un algoritmo que genere la triangulación del polígno con hoyos. La idea será separar nuestro polígono en polígonos sin hoyos y triangularlos por separado. A continuación se exhibe el algoritmo deseado:

1. Ordenemos nuestros puntos. Esto lo realizamos en $\mathcal{O}(n \log n)$.

2. Realizamos un barrido de línea para encontrar los hoyos. Este barrido nos tomará $\mathcal{O}(n)$, y detectaremos hoyos cuando el estado de la línea nos indique que hay al menos 4 aristas en su posición y conforme el barrido avance esto se sigue cumpliendo hasta que eventualmente llegamos a que 2 de los (al menos) 4 segmentos que siempre estuvieron en el estado de la línea (no necesariamente los mismos segementos), siempre fueron "internos" de los otros segmentos (siempre estuvieron "entre ellos").

   Una manera de hacer esto sería coloreando las aristas, si en un punto el estado de la línea me indica que las trayectorias formadas hasta el momento (con el avance de la línea) y coloreadas de distinto color se unen, entonces cambio el color a la trayectoria más pequeña. Si una trayectoria núnca se unió y además forma un ciclo interno al polígono, entonces este es un hoyo. **Obs.** Tomar esta idea sin mayor cuidado nos puede llevar a una complejidad cuadrada (aquí podemos hacer uso de la estrategia usada en el agoritmo de Borůvka-Kruskal para encontrar ciclos y agrupar).

3. Ahora que hemos encontrado los ciclos, podemos con gran facilidad segmentar nuestro polígono en nuevos polígonos sin hoyos. Para esto;

   (a) tomamos una arista del hoyo O_i, digamos e, entonces unimos los extremos de e con los vértices más cercanos en el polígono y verificamos si estas nuevas aristas son "estrictamente internas al polígono" o no. En caso de que sí, entonces serán parte de nuestra subdivisión, en otro caso la descartamos. Esto lo podemos hacer con ayuda de un nuevo barrido y nos toma $\mathcal{O}(n)$ barrer nuestros puntos y $\mathcal{O}(n)$ realizar las comparaciones para encontrar las nuevas aristas para nuestra subdivisión. Si el número de aristas de O_i es lineal respecto al número de vértices en el polígono, entonces tendremos que hacer $\mathcal{O}(n^2)$ comparaciones. De lo anterior tenemos una subdivisión.

4. Ahora aplicamos el algoritmo de triangulación visto en clase a cada parte de la subdivisón encontrada.

Análisis de complejidad. La complejidad de este algoritmo esta contenida en

$$\mathcal{O}(n + k \cdot \log n) + \mathcal{O}(n) + \mathcal{O}(n \log n) + \mathcal{O}(n^2) = \mathcal{O}(n^2).$$

Como el algoritmo anterior nos regresa una triangulación de nuestro polígno con hoyos, entonces hemos mostrado lo requerido. □

4. Un árbol de rango en un conjunto de n puntos en el plano requiere $\mathcal{O}(n \log n)$ de almacenamiento. Uno podría reducir los requisitos de almacenamiento almacenando estructuras asociadas solo con un subconjunto de los nodos en el árbol principal.

- Supongamos que sólo los nodos con profundidad $0, 2, 4, 6, \cdots$ tienen una estructura asociada. Muestre cómo se puede adaptar el algoritmo de consulta para responder consultas correctamente.
- Analice los requisitos de almacenamiento y el tiempo de consulta de dicha estructura de datos.

▷ **Solución:** Para este problema dividamos la solución en dos posibles opciones:

1. ¿Cómo adaptamos nuestro árbol de rangos para que cumpla lo requerido en el primer punto? Como sabemos que un árbol, al menos, tendrá una raíz y las estructuras estarán "colgadas" de niveles pares. Entonces, la información respecto de Y de los niveles impares estarán contenidas en el árbol colgado en su nivel par, inmediato, anterior.

   Así, las consultas para los niveles pares se quedan exactamente iguales. Para los niveles impares consultamos respecto de X y hacemos "bracktraking" al nivel par, inmediato, anterior para terminar la consulta en la estructura colgada en el nodo de ese nivel.

2. *Análisis de almacenamiento.* El almacenamiento, aunque a primera vista se reduce en la cantidad de niveles pares, realmente las estructuras que estaban colgadas en estos niveles ahora formarán parte de las estructuras en niveles pares. El ahorro de almacenamiento es $\mathcal{O}(1)$, pues solo nos ahorramos las raíces de los niveles impares (que son, normalmente, hojas de los niveles pares).

   Análisis de tiempo de consulta. El tiempo de consulta para un elemento en un nivel par es el mismo, para un elemento en un nivel impar es el mismo en términos globales, pues el "backtraking" lo realizamos en tiempo $\mathcal{O}(1)$ y la consulta en la estructura "colgada" sigue siendo $\mathcal{O}(\log n) + k$. Como la consulta en árbol respecto a X (árbol grande) es $\mathcal{O}(\log n)$, concluimos una complejidad de consulta en $\mathcal{O}(\log^2 n)$

Obs. Para este problema estoy trabajando con un árbol de rango rectangulares. ◁

5. Se pueden utilizar las estructuras de búsqueda de rangos ortogonales para determinar si un punto particular (a, b) está en un conjunto dado, haciendo una consulta al rango $[a : a] \times [b : b]$.

a) Prueba qué hacer una consulta así en un árbol Kd toma tiempo $\mathcal{O}(\log n)$.

b) ¿Cuál es la complejidad para una consulta así en un árbol de rangos?

▷ **Solución:** Para este problema dividamos la solución en dos posibles casos:

a) Sabemos que hacer una consulta, en un árbol Kd, para un rango en general nos toma $\mathcal{O}(\sqrt{n}+k)$. Sin embargo, consultar un punto en un árbol Kd sería equivalente a particionar la nuve de puntos por la mitad y preguntarnos en qué parte queda nuestro punto distinguido, llamemosle q. Así, descartamos aproximadamente la mitad de puntos en dónde no se encuentra q. Luego, subdividimos nuevamente el conjunto de puntos restantes en 2 y nos preguntamos en qué parte se encuentra q y podemos descartar la parte en la que no se encuentre. De esta manera y recursivamente nuestro espacio de búsqueda se reduce a la mitad cada vez, esto equivale hacer un recorrido de la raíz de nuestro árbol kd hacia las hojas en búsca del punto q. Como cada vez descontamos la mitad del conjunto de puntos de búsqueda, tenemos la recurrencia $T(n) = 2T(n-1)$ que sabemos que nos genera un orden logarítmico en base 2. Cómo el número de consultas es igual a 1, entonces $k = 1 \in \mathcal{O}(1)$. Por tanto, la complejidad de esta consulta es $\mathcal{O}(\log n)$.

b) En este caso, tenemos una complejidad general de $\mathcal{O}(\log^2 n + k)$, cómo solo estamos consultando un punto y no un rango, entonces $k = 1 \in \mathcal{O}(1)$. En la consulta debemos bajar por el árbol hasta encontrar $q.X$ y bajar su árbol "colgado" o su árbol asociado en y, hacer esto es igual a $\log m$ dónde m es la altura del árbol asociado con $m \neq n$, pues cada nivel y nodo tiene un árbol compacto de tamaño constante. Por tanto, la complejidad esta contenida en $\mathcal{O}(\log m \cdot \log n) = \mathcal{O}(\log n)$ con m constante respecto de n.

◁

6. En algunas aplicaciones solo nos interesa el número de puntos que caen dentro de un rango y no reportar cada uno de ellos. En este caso nos gustaría evitar el término $\mathcal{O}(k)$ en el tiempo de consulta.

a) Describe cómo un árbol de rangos de una dimensión puede adaptarse para que una consulta así se pueda realizar en tiempo $\mathcal{O}(\log n)$.

b) Usando la solución al problema para una dimensión, describe cómo se pueden responder consultas de conteo en rangos de d dimensiones en tiempo $\mathcal{O}(\log^d n)$.

c) Describe cómo se puede usar la técnica de cascada para mejorar el tiempo de consulta en un factor $\mathcal{O}(\log n)$ para dos y más dimensiones.

▷ **Solución:** A continuación se da solución a cada uno de los incisos:

a) La versión para una dimensión sería bastante simple, y de hecho sería como un árbol binario de búsqueda balanceado, pues basta con preservar el árbol general sin los árboles "colgados" que tiene el árbol de búsuqeda ortogonal.

La manera de ahorrar $\mathcal{O}(k)$ durante la búsqueda de rango, sería no recorrer las hojas del rango encontrado. Pues, basta con realizar dos recorridos hacia abajo desde la raíz, uno para encontrar la cota inferior del rango y otro para encontrar la cota superior. Así, y por la propiedad de tener mayores a la derecha y menores a la izquierda concluimos que las hojas entre las seleccionadas forman parte del rango.

b) De la misma manera, que en el caso de una dimensión, nos ahorramos $\mathcal{O}(k)$ al no recorrer las hojas que quedan dentro del rango. Para generalizar a d dimensiones basta reproducir la idea de los árboles "colgados", pues cada nodo que no sea hoja tendrá, inicialmente una referencia al árbol con coordenadas en Y. Luego los árboles con coordenadas en Y tendrán colgados árboles con coordenadas en Z, y así de manera recursiva. Cuándo se requiera obtener el rango bastará con bajar por el árbol principal y recorrer cada árbol asociado de manera recursiva para inspeccionar que efectivamente el punto quede dentro del rango requerido. Cómo recorremos $\log n$ el árbol principal y por cada nodo distinto a una hoja recorremos su árbol asociado y a su vez los asociados recursivamente, entonces tenemos una complejidad de $k \cdot \log n$ dónde $k = \log m \cdot \log m' \cdots \log m^{\cdots} \approx \log^{d-1} n$ y concluimos que la complejidad en general por recorrido para encontrar un punto en el rango es de $\mathcal{O}(\log^d n)$.

Como esto se tiene que hacer para el punto "menor" en el rango y para el punto "mayor" en el rango, entonces tenemos una complejidad contenida en

$$2 \cdot \mathcal{O}(\log^d n) \in \mathcal{O}(\log^d n).$$

c)

◁

7. Da un algoritmo que calcule en tiempo $O(n \log n)$ una diagonal que divida un polígono simple con n vértices en dos polígonos simples cada uno con a lo más $\lfloor \frac{2n}{3} \rfloor + 2$ vártices. **Hint**: Usa la gráfica dual de la triangulación.

▷ **Solución:** A continuación se exhibe un algoritmos que resuelve este problema

1. Obntener la triangulación de nuestro polígono. Esto lo realizamos en $\mathcal{O}(n \log n)$ con el algoritmo visto en clase.

2. Sabemos que toda triangulación tiene $n - 3$ aristas y por tanto nuestra gráfica dual que se formará a partir de cada arista interna será de un orden $\mathcal{O}(n)$. Entonces, la construcción de la gráfica dual nos toma $\mathcal{O}(n)$. Además, la gráfica dual es un árbol.

3. Encontramos los puntos más lejanos en una gráfica. Esto, por algoritmos I sabemos que lo podemos hacer en $\mathcal{O}(n)$ para un árbol. La técnica se basa en "colgar" el árbol por medio de un recorrido a profundidad desde la raíz hasta la hoja de mayor profundidad y tomar esa hoja como la nueva raíz y volver a recorrer hasta la hoja más profunda.

4. Recorrer la gráfica dual iniciando desde cada uno de los puntos más lejanos. Eligimos una orientación y siempre seguimos esta orientación (La orientación, la podemos tener con DCEL) para el reccorido. Así evitaremos dividir el polígono en más de 2 polígonos nuevos. Esto lo hacemos en $\mathcal{O}(n)$, pues el orden de las aristas de nuestra gráfica dual es lineal.

5. Durante cada recorrido colorearemos los vértices del triángulo en el que estemos, todos del mismo color. En el momento que hayamos coloreado, exactamente $\lfloor \frac{2n}{3} \rfloor + 2$ vértices. Entonces regresamos la arista e_T de la triangulación por la que fue creada la arista siguiente a la última hasta haber coloreado los $\lfloor \frac{2n}{3} \rfloor + 2$ vértices. Esto lo hacemos en $3 \cdot \left(\lfloor \frac{2n}{3} \rfloor + 2 \right)$ pasos que obedece a un orden líneal.

   ¿Con cuál recorrido nos quedamos? Con el que llegue primero a colorear los $\lfloor \frac{2n}{3} \rfloor + 2$ vértices y cumpla que la arista divide al polígono en las características deseadas.

   Obs. Podríamos hacer una versión con $\lfloor \frac{n}{3} \rfloor - 2$ tomandolo por cota.

Al final, e_T es la diagonal búscada.

Análisis de la complejidad. La complejidad del algoritmo esta contenida en

$$\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n).$$

◁

8. Sea S un conjunto de n puntos en el plano y A una subdivisión plana con $\mathcal{O}(n)$ regiones:

- Muestra que toma $\mathcal{O}(n \log n)$ localizar los puntos de S en A.
- Sea T una triangulación de S, ¿cómo podría ayudar esta información para localizar los puntos de S en A? Muestra que a pesar de tener la triangulación de S, la cota sigue siendo logarítmica.

Demostración: Para probar los incisos anteriores basta exhibir un algoritmo que encuentre los puntos requeridos en $\mathcal{O}(n \log n)$. A continuación se solucionan los incisos anteriores, esto es

- Usando el algoritmo de David Kirkpatrik visto en clase, podemos calcular la localización de cada uno de los puntos. Localizar un punto por medio de este algoritmo nos toma $\mathcal{O}(\log n)$, cómo buscamos localizar n puntos, entonces tenemos que aplicar este algoritmo n veces. Así, localizar la región de A en la que esta contenido cada punto nos cuesta $\mathcal{O}(n \log n)$.
- Tener una triangulación en los puntos nos ayudaría a ahorrar algunas subdivisiones, sin embargo, habría que verificar que nuestra subdivisión A más T siga siendo plana, por lo que no necesariamente ahorramos complejidad. Incluso, si ahorraramos complejidad esta sería pequeña, de hecho sería de tamaño constante y nuestro algoritmo seguiría necesitando de $\mathcal{O}(n \log n)$ para localizar todos los puntos, pues los demás pasos se siguen teniendo (ir quitando los de vértices de menor grado < 8 e ir retriangulando, luego regresarnos por medio de nuestro árbol).

□