UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

Algoritmos para la k-ésima estadística, la pareja de puntos más cercana y más lejana en el modelo de flujo de datos

T E S I S

QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A:

GERARDO CRUZ ZAGASTA

Director de tesis: José David Flores Peñaloza
Facultad de Ciencias, UNAM

Cotutor: Armando Castañeda Rojano
Instituto de Matemáticas, UNAM

CD. DE MÉXICO, 2017

Índice general

Índice de figuras

Capítulo 1

Introducción

La gran cantidad de datos que generan las aplicaciones actuales trajo consigo una serie de cambios en la forma en que se resuelven determinados problemas. Tradicionalmente se tenía un problema, una gran cantidad de memoria disponible y estructuras de datos que nos permitían contener totalmente a los elementos. Ante la inmensa cantidad de datos, hoy se tiene que pensar en técnicas para dar solución a estos problemas considerando una memoria limitada, y restricciones en cuantas veces se analizan todos los elementos. Como respuesta a esta problemática han surgido modelos de datos que expresan estas condiciones en las que se resuelven problemas de flujos de datos modernos. Uno de estos es el modelo de semiflujo, el cual impone restricciones sublineales de pasadas y de memoria respecto al tamaño del flujo de entrada y con el cual trabajamos a lo largo de este trabajo.

Analizamos primeramente un problema con gran relevancia en varias disciplinas: el problema de la K-ésima estadística. Presentamos una adaptación en este modelo del algoritmo tradicional de Blum et al. [4] que para un flujo de n elementos y una cantidad de memoria m soluciona el problema en tiempo $\mathcal{O}(n \log n)$, espacio $\mathcal{O}(\frac{n}{m} + m)$ y una cantidad de pasadas $\mathcal{O}(\log n)$.

Posteriormente analizamos dos problemas fundamentales de la geometría computacional: la pareja de puntos más cercana y más lejana. Haciendo uso de la estructura geométrica conocida como diagrama de Voronoi en combinación con algoritmos de localización de punto, presentamos un algoritmo esencialmente idéntico para resolver ambos problemas dado un flujo de n elementos y una cantidad de memoria m en tiempo $\mathcal{O}(\frac{n^2}{m} \log m)$, espacio $\mathcal{O}(m)$ y una cantidad de pasadas $\mathcal{O}(\frac{n}{m})$.

El presente trabajo se estructura como sigue:

En el capítulo 2 se encuentra un marco teórico de conceptos necesarios para dar sustento y poder presentar posteriormente a los algoritmos que dan solución a los problemas. En su mayoría son conceptos fundamentales de la geometría computacional.

En el capítulo 3 se presentan los problemas de la K-ésima estadística, la pareja de puntos más cercana y la pareja de puntos más lejana. Se analizan las soluciones clásicas a estos problemas bajo el modelo tradicional (RAM) para posteriormente presentar su solución en el modelo de semiflujo.

En el capítulo 4 se presenta la solución al problema de la K-ésima estadística en su versión de semiflujo. De igual manera se analiza y se discuten detalles relacionados al algoritmo propuesto.

En el capítulo 5 se presenta la solución al problema de la pareja de puntos más cercana en su versión de semiflujo. Se analiza el algoritmo y se discute como puede ser usado con pequeños cambios para dar solución también al problema de la pareja de puntos más lejana.

Finalmente en el capítulo 6 se da una conclusión general y se analiza el trabajo posterior de interés que puede surgir a raíz de lo propuesto en el presente trabajo.

Capítulo 2

Antecedentes

En este capítulo se revisan una serie de conceptos y algoritmos generales para ilustrar de mejorar manera las ideas involucradas en los algoritmos propuestos en capítulos posteriores. Se revisan en su mayoría conceptos importantes de geometría computacional y finalmente se da paso a presentar el modelo de flujo de datos que será el marco de trabajo general.

2.1. Geométricos

En esta sección se revisan algunos antecedentes geométricos. La mayoría de estos conceptos son fundamentales en el área de la geometría computacional, por lo que han sido sumamente estudiados y desarrollados hasta tener versiones bastante simplificadas y de gran aplicación.

2.1.1. Envolvente convexa

Skiena S. (2008) menciona en su libro [20] que "encontrar la envolvente convexa de un conjunto de puntos es el problema más elemental de la geometría computacional, justo como el problema del ordenamiento es a los algoritmos combinatorios". Se puede definir la envolvente convexa de la siguiente manera:

Definición 1 (Envolvente convexa)**.** *La envolvente convexa de un conjunto de puntos S es el polígono convexo más pequeño que contiene a todos los puntos de S.*

Por supuesto, al ser un problema clásico tan relevante existen una gran cantidad de algoritmos para solucionarlo (ver figura 2.1). En esta sección revisaremos el algoritmo de *Graham scan* por ser un procedimiento relativamente sencillo y con una cota de complejidad eficiente.

Algoritmos para calcular la envolvente convexa		
Nombre	**Complejidad temporal**	**Observaciones**
Gift wrapping	O(nh)	h es el número de puntos en la envolvente convexa.
Graham scan	O(n log n)	
Divide y vencerás (Preparata & Hong)	O(n log n)	
Quickhull	O(nh)	h es el número de puntos en la envolvente convexa. Basado en Quicksort.

Figura 2.1: Algoritmos para calcular la envolvente convexa.

En 1972 Graham publicó [13] un algoritmo para calcular la envolvente convexa que se conocería popularmente como *Graham scan*. Este algoritmo hace uso del concepto de giros. Hablar de giros es una abstracción empleada para referirse al ángulo exterior que se forma al conectar los últimos 2 puntos. El algoritmo se muestra a continuación:

1. Se busca al punto con el menor valor en la coordenada Y. En caso de que exista más de uno se elige al candidato con el menor valor en la coordenada X. Llamaremos a este punto P.

2. Se ordena en orden creciente el conjunto de puntos (exceptuando a P) respecto al ángulo que forman con P y el eje X.

3. Creamos una pila inicialmente con el punto P y empezamos a iterar sobre el conjunto de puntos ordenados. Sea P_c el punto actual en la iteración y P_b el punto en el tope de la pila, si para unir P_b con P_c se requiere hacer un giro hacia la derecha, entonces eliminaremos a P_b de la pila. Repetiremos este último paso hasta que se tenga un giro hacia la izquierda entre P_c y el punto que está en el tope de la pila, y finalmente añadiremos P_c a la pila.

4. El algoritmo termina cuando se completa la iteración sobre el conjunto de puntos y en ese momento la pila contiene a todos los puntos de la envolvente convexa.

Analicemos ahora cada uno de los pasos del algoritmo. El paso 1 tiene la función de encontrar a un punto de la envolvente convexa. Una manera sencilla de obtenerlo es irse a los extremos y obtener un punto con el mayor valor en alguna de sus coordenadas. Al

ubicarse en un extremo es fácil ver que el punto que se tiene forma parte de la envolvente convexa. Este paso toma tiempo $\mathcal{O}(n)$.

El paso 2 es un ordenamiento de los puntos. Se tiene un ordenamiento convencional por lo que cualquier algoritmo de ordenamiento de uso general sirve para cumplir con este paso. Asumiendo que se usa un algoritmo eficiente este paso toma tiempo $\mathcal{O}(n \log n)$.

El paso 3 define el verdadero funcionamiento del algoritmo. Al iterar sobre los puntos se menciona que es importante saber hacia donde se hace el giro para decidir que acción tomar. La razón de ello es que tener un giro a la derecha implica formar un ángulo exterior menor que 180° y por tanto con ello romper la convexidad (ver figura 2.2). Lo anterior significa que el último punto añadido a la pila está en el interior del polígono y con ello no forma parte de la envolvente convexa. Un giro a la izquierda implica que se forma un ángulo exterior mayor que 180° y por tanto se mantiene la convexidad. Es importante resaltar que verificar el valor del ángulo puede hacerse en tiempo constante por medio de sólo arimética.

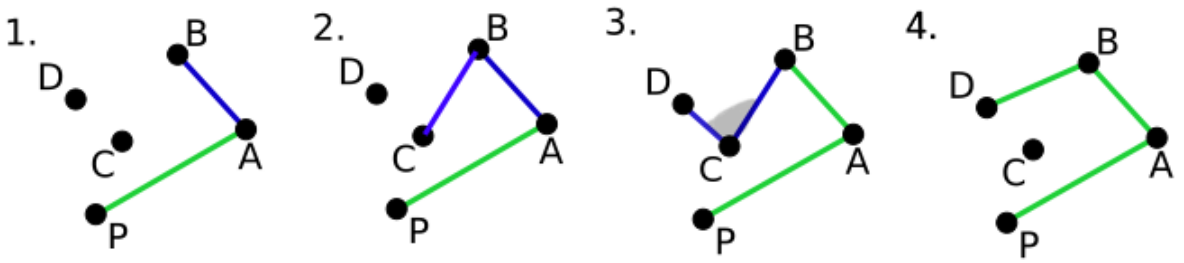

Figura 2.2: Algoritmo Graham scan. Del segundo al tercer paso se hace un giro hacia la derecha formando un ángulo exterior $\measuredangle DCB < 180°$ y con ello se rompe la convexidad. Por lo anterior en el paso 4 se elimina al punto C de la envolvente convexa.

En este paso se itera sobre todos los puntos y como se mencionó anteriormente en cada una de estas iteraciones se revisa potencialmente los puntos anteriores. Una vez que un punto ha sido descartado de la pila no vuelve a ser considerado para formar parte de la envolvente convexa (esto debido a que se encuentra al interior del polígono y eso no puede cambiar). Por tanto tenemos que cualquier punto P_i puede ser visto primero como parte de la iteración general, posteriormente durante la iteración del punto P_{i+1} donde se revisa el punto anterior para verificar que se mantenga la convexidad, y finalmente puede ser visto en una última ocasión si en una iteración avanzada se rompe la convexidad y se tiene que regresar hasta éste en la pila para verificarla. Entonces como una cota superior se tiene que se puede revisar cada punto en 3 ocasiones, por lo que se puede concluir que este paso toma tiempo $\mathcal{O}(n)$.

El paso 4 regresa los puntos que están en la pila y como potencialmente podrían ser todos este paso toma tiempo $\mathcal{O}(n)$. Por lo tanto como complejidad temporal total se tiene:

$$\mathcal{O}(n) + \mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n) \tag{2.1}$$

Es decir que la complejidad total del algoritmo queda dominada por el ordenamiento de los puntos.

2.1.2. Calibres rotatorios

Las líneas de soporte son aquellas líneas tangenciales a un vértice que dejan a todo el polígono a un lado de ésta. A la pareja de puntos que admite líneas paralelas de soporte se le llama *conjunto antipodal*.

En 1978 Shamos propuso una técnica para generar todos los conjuntos de puntos antipodales de un polígono convexo [19]. Esta técnica recibió el nombre de *calibres rotatorios*[1] como idea análoga a rotar un *calibre de Vernier* alrededor de un polígono convexo. La observación que permite permite obtener de manera eficiente todos los antipodales sin revisar de manera exhaustiva todas las parejas de puntos se describe a continuación.

Sea P un polígono convexo y p_i uno de sus vértices (los cuales se encuentran ordenados en sentido de las manecillas del reloj), denotaremos como $\overline{p_1 p_2}$ al arista del polígono delimitada por los puntos p_1 y p_2. Se empieza por recorrer los puntos a partir de p_i hasta encontrar el vértice más distante a $\overline{p_{i-1} p_i}$, llamaremos a este punto q_R. De la misma manera se realiza un recorrido en sentido contrario para encontrar al vértice más distante a $\overline{p_i p_{i+1}}$, llamaremos a este punto q_L. La sucesión de puntos que une a q_L y a q_R en sentido contrario a donde se ubica p_i forma el conjunto $C(p_i)$, el cual al emparejar elemento por elemento con p_i forma un antipodal. Para ilustrar lo anterior hay que revisar los ángulos exteriores de los puntos que son antipodales. Sea p_s un punto en el conjunto $C(p_i)$, α_1 el ángulo exterior formado por los segmentos $\overline{p_{i-1} p_i}$ y $\overline{p_i p_{i+1}}$, y α_s el ángulo exterior de los 2 segmentos adyacentes a p_s, la pareja es antipodal si existe una línea recta que cruza a la intersección de los 2 ángulos (ver figura 2.3). Lo que se hace entones al encontrar la sucesión de puntos que forma al conjunto $C(p_i)$ es garantizar que a partir de estos vértices se sigue manteniendo un solapamiento en la intersección de los ángulos y entonces pueden seguir siendo cruzados por una línea recta.

Con esta observación en mente se puede plantear un algoritmo de la siguiente manera:

Primero se encuentra una pareja cualquiera de puntos que sean antipodales, para ello sólo hay que fijar un punto p_i y recorrer los demás puntos hasta encontrar el más distante a éste, al cual llamaremos p_j. Con ello se forma la pareja antipodal (p_i, p_j). Imaginemos ahora las líneas paralelas de soporte de este conjunto antipodal. (ver figura 2.4).

[1] En inglés *Rotating calipers.*

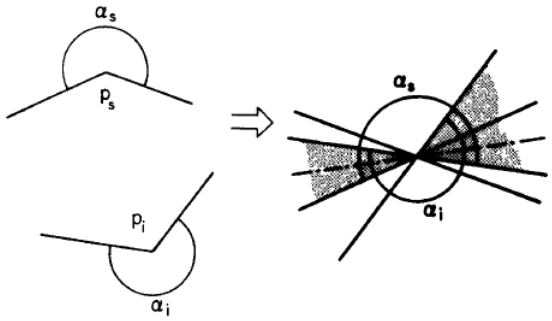

Figura 2.3: Caracterización de los antipodales.

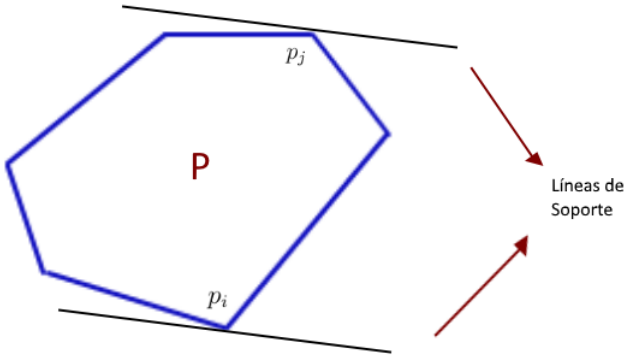

Figura 2.4: Líneas de soporte.

Empezamos a rotar las líneas de soporte en el mismo sentido, manteniendo en todo momento el paralelismo entre ellas. Se rotarán hasta que alguna línea de soporte impacten contra otro vértice. Estos dos vértices formarán un antipodal (ver figura 2.5).

El proceso de rotado continúa hasta que las líneas alcancen a los puntos originales pero en sentido invertido, es decir la línea que empezó tocando al vértice p_i ahora estará sobre p_j, y de manera análoga la línea que inicialmente estaba sobre p_j ahora estará sobre p_i.

Dado que en cada momento o una u otra línea avanza sobre los vértices, se tiene entonces siempre un progreso. Si por simplicidad de análisis se supone que cada línea de soporte regresa a su vértice de inicio, entonces al finalizar el algoritmo se tendría una vuelta completa por cada una de ellas y con ello un total de $2N$ movimientos. Esto basta para observar que en el algoritmo presentado al momento de llegar al último par antipodal (que resulta ser el primero también) se toma un tiempo total de $\mathcal{O}(n)$.

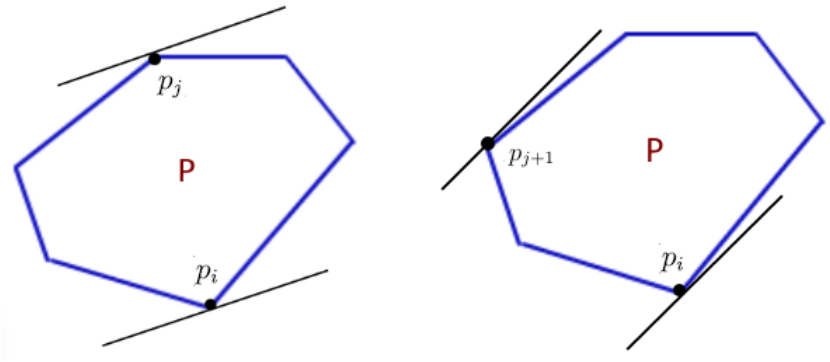

Figura 2.5: La rotación de las líneas de soporte genera primero el antipodal (p_i, p_j) y posteriormente al seguir rotando a (p_i, p_{j+1}).

2.1.3. Diagrama de Voronoi

El diagrama de Voronoi es una estructura geométrica muy importante en la geometría computacional con numerosas aplicaciones en las áreas de la física, astronomía, robótica, geografía y más [6]. Intuitivamente la idea general de esta estructura es tener una subdivisión del plano en regiones, donde cada una estas regiones contiene un punto que "domina" la región, es decir que tiene una distancia menor con todos aquellos puntos dentro de su región en comparación a los otros puntos dominantes. Estos puntos que dominan su región formalmente se llaman sitios. En general, el diagrama de Voronoi se define de la siguiente manera:

Definición 2 (Diagrama de Voronoi)**.** *Sea $P = \{p_1, p_2, ..., p_n\}$ un conjunto de n puntos distintos en el plano, estos puntos serán los sitios. Un diagrama de Voronoi es una subdivisión del plano en n regiones (una para cada sitio) llamadas celdas de Voronoi, con la propiedad de que un punto q está dentro de una celda con el sitio p_i, si y sólo si la distancia(q, p_i) < distancia(q, p_j) para todo $p_j \in P$.*

Al finalizar su construcción el diagrama de Voronoi tendrá un aspecto similar al mostrado en la figura 2.6. En esta sección revisaremos un algoritmo para construir el diagrama de Voronoi conocido como el algoritmo de *Fortune*.

En 1986 Steven Fortune publicó [11] un algoritmo para calcular el diagrama de Voronoi de un conjunto de puntos. El algoritmo se describe a continuación.

Se tiene una línea horizontal ℓ, la cual se mueve desde arriba hacia abajo barriendo el plano. Esta línea irá manteniendo a su paso la información de los sitios que no puede ser cambiada por los sitios que todavía no han aparecido y están debajo de ℓ (ver figura 2.7). ¿Qué información no puede ser cambiada? Denotaremos como ℓ^+ a la parte del plano

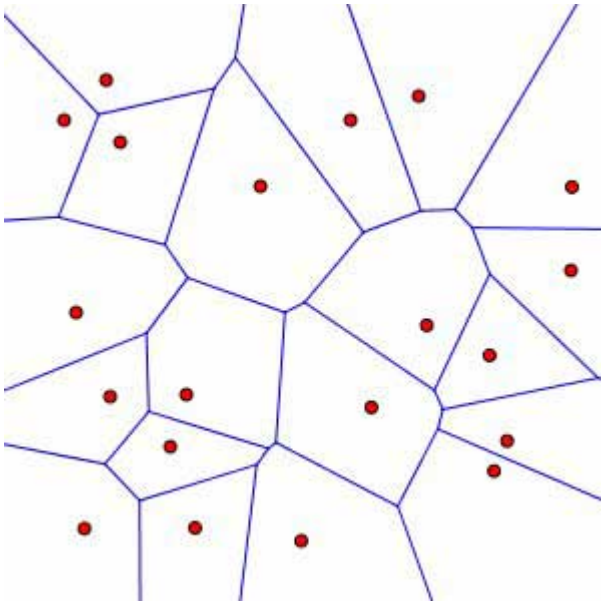

Figura 2.6: Diagrama de Voronoi. Los puntos son los sitios de cada celda y las líneas azules delimitan las celdas de Voronoi.

que está arriba de ℓ y con ℓ^- a la parte que está abajo. La distancia de un punto $q \in \ell^+$ a cualquier sitio debajo de ℓ es mayor que la distancia de q a ℓ, entonces si q está tan cerca de un sitio $p_i \in \ell^+$ como lo está de ℓ, eso significa que el sitio más cercano a q no puede estar debajo de ℓ. El espacio de puntos que están más cercanos a un sitio $p_i \in \ell^+$ que a ℓ esta delimitado por una parábola, por lo tanto el espacio de todos los puntos que están más cercanos a algún sitio $p_i \in \ell^+$ que a ℓ es un conjunto de arcos parabólicos. A esta secuencia de arcos se les llama *línea de playa* (ver figura 2.7). Hay que notar que en la justa intersección de 2 arcos parabólicos se tiene un punto que conserva la misma distancia respecto a 2 sitios, y por lo tanto eso significa que éste forma parte de un arista del diagrama de Voronoi que puede irse construyendo.

Se continua barriendo el plano mantieniendo en todo momento la línea de playa y conservando esencialmente la misma estructura hasta que sucede alguno de estos 2 eventos:

a) ℓ encuentra un nuevo sitio: En este caso hay que añadir un nuevo arco parabólico a la línea de playa correspondiente a este sitio. Cuando recién aparece el nuevo sitio forma una parábola degenerada a una semirecta y a medida que ℓ avanza ésta se va ampliando (ver figura 2.8).

b) Un arco desaparece: A medida que ℓ avanza su barrido algunos arcos se expanden y pueden provocar la desaparición de otro. La manera en que esto puede suceder es cuando

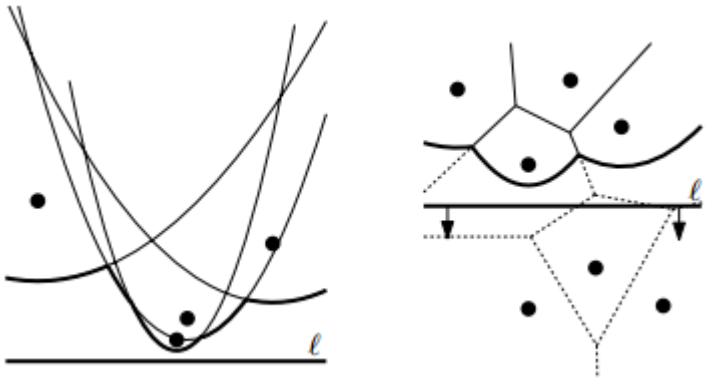

Figura 2.7: La línea de barrido va conservando la información que se sabe no va cambiar más. La línea de playa se construye por la intersección de las parábolas de cada sitio.

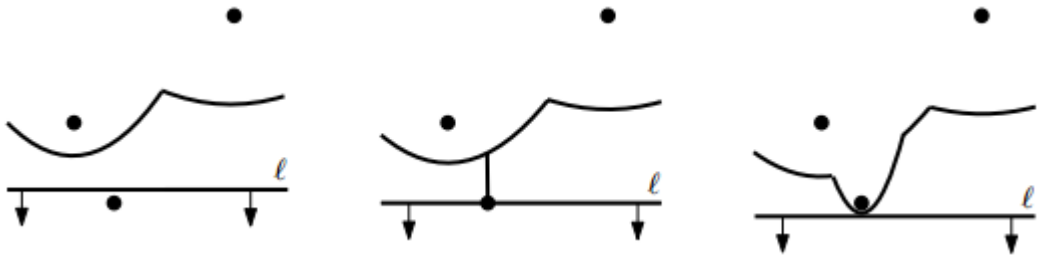

Figura 2.8: ℓ encuentra un nuevo sitio y forma una parábola que se expande a medida que el barrido continúa.

se tiene una secuencia de 3 arcos y los de los extremos se expanden hasta comprimir y desaparecer al arco central. En el último instante se tiene la intersección de 3 arcos y por tanto se tiene un punto que está a la misma distancia que 3 sitios, por lo que en el diagrama se debe construir como la unión de los límites de 3 celdas (ver figura 2.9).

Se continúa de esta manera hasta barrer el plano totalmente, cuidando en todo momento los eventos anteriormente mencionados. Al finalizar el barrido se tendrá el diagrama de Voronoi construido.

Se ha mostrado que con una correcta implementación [6] este algoritmo puede tener espacio $\mathcal{O}(n)$ y un complejidad temporal de $\mathcal{O}(n \log n)$. En general la implementación más común de este algoritmo almacena todos los aristas del diagrama en una lista doblemente enlazada y a los elementos en un árbol para hacer eficiente su acceso. Cada que sucede un evento se buscan en el árbol los puntos involucrados y dado que necesariamente cada uno de los n puntos está involurado en algún evento de los mencionados anteriormente se tiene

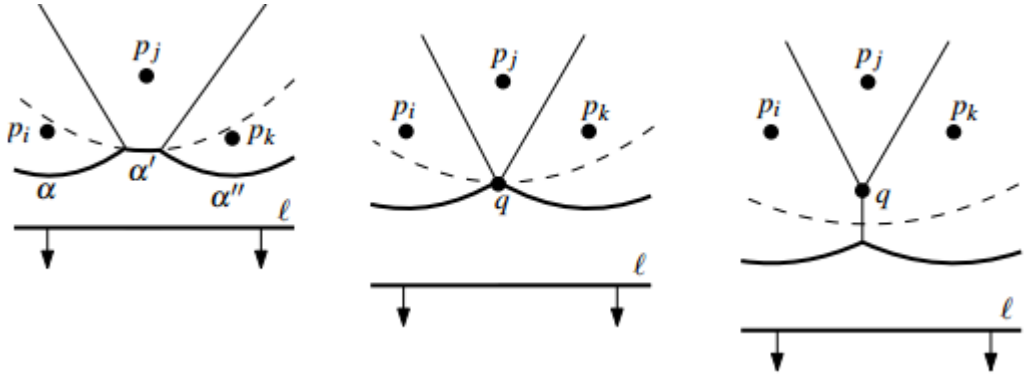

Figura 2.9: Dos arcos se expanden hasta comprimir y desaparecer a un tercero. En ese punto q se tiene la misma distancia a 3 sitios.

entonces una complejidad temporal de $n * \mathcal{O}(\log n) = \mathcal{O}(n \log n)$.

2.1.4. Diagrama de Voronoi de puntos más lejanos

Una variante del diagrama de Voronoi tradicional es la conocida como el *diagrama de Voronoi de puntos más lejanos* (de ahora en adelante diagrama VPL). Este diagrama divide al plano en regiones en las cuales los puntos dentro de éstas tienen en común a un mismo punto del conjunto de entrada como el más distante a ellos. Formalmente un diagrama VPL se define:

Definición 3 (Diagrama de Voronoi de puntos más lejanos)**.** *Sea* $P = \{p_1, p_2, ..., p_n\}$ *un conjunto de n puntos distintos en el plano, estos puntos seran los sitios. Un* diagrama VPL *es una subdivisión del plano en regiones llamadas celdas de Voronoi de puntos más lejanos, con la propiedad de que un punto q está dentro de la celda generada por el sitio* p_i *si y sólo si la distancia(q,* p_i*) > distancia(q,* p_j*) para todo* $p_j \in P$*.*

Sea P el conjunto de puntos. La primera observación que puede realizarse es que no todos los puntos de P van a tener una celda en el diagrama VPL. Como se demostró para el problema de la pareja de puntos más lejana (ver sección 3.3.1), para cualquier punto en el plano, el punto más distante del conjunto P es un punto de la envolvente convexa de P. Por lo tanto los únicos puntos que tendrán una celda en el diagrama VPL son aquellos que forman parte de la envolvente convexa de P. Un diagrama VPL tiene un aspecto como el mostrado en la figura 2.10.

Revisaremos ahora un algoritmo para construir el diagrama VPL. Primeramente, por la observación mencionada en el párrafo anterior solamente nos basta trabajar con los puntos

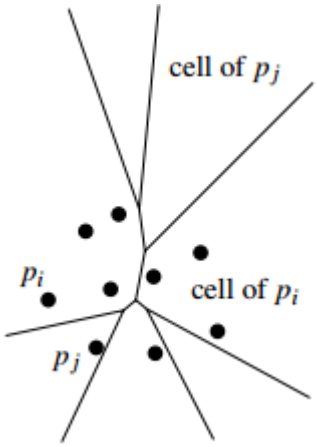

Figura 2.10: Diagrama de Voronoi de puntos más lejanos.

que se encuentran en la envolvente convexa. Con ello el primer paso es entones calcular la envolvente convexa del conjunto de puntos. En la sección 2.1.1 se revisó un algoritmo para realizar esta tarea.

Se toman los puntos de la envolvente convexa y se ponen en orden arbitrario. Sea este orden $p_1, p_2, ..., p_h$. Removemos los puntos $p_h, p_{h-1}, ..., p_4$, y justo en el momento que removemos a un punto p_i almacenamos a su punto vecino en sentido de las manecillas del reloj $cw(p_i)$ y también al punto que se encuentra en sentido contrario $ccw(p_i)$. Una vez que un punto ha sido removido no puede ser considerado para ser un vecino de otro punto posteriormente. Con los 3 puntos restantes después de la eliminación de puntos construimos un primer diagrama VPL que funcionará como el diagrama base para una construcción incremental.

Posteriormente se empiezan a insertar los puntos que faltan $p_4, p_5, ..., p_h$. Para poder agregar un punto p_i se requiere tener el diagrama VPL de los puntos $p_1, ..., p_{i-1}$. La observación clave por hacer es que la celda del punto p_i tiene que estar en medio de las celda $cw(p_i)$ y la celda $ccw(p_i)$, y por tanto esto delimita el espacio. Al insertar a p_i se tomará su bisector con el punto $ccw(pi)$ y se extenderá imaginariamente hasta cruzar todo el diagrama; de igual manera se realiza el mismo procedimiento con el punto en el otro sentido $cw(pi)$. Se unen los puntos donde los bisectores impactan por primera vez contra un arista para formar la parte superior de la celda de p_i. Se siguen extendiendo estos bisectores para formar en el otro extremo del diagrama los límites que tendrá la celda p_i. Para terminar de formar la celda hay que eliminar todas aquellas aristas que queden dentro de estos segmentos que acabamos de delimitar, y con ello habremos construido la celda para el punto p_i (ver figura 2.11).

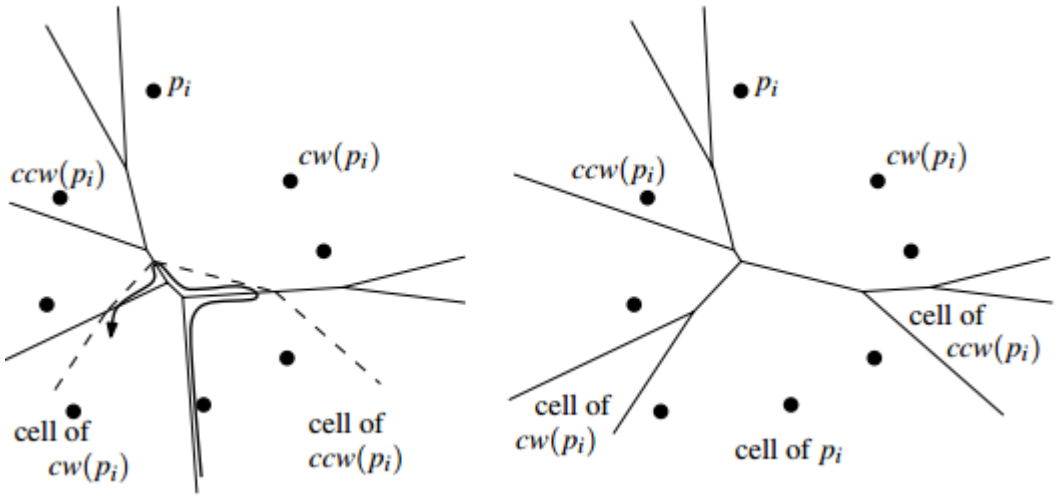

Figura 2.11: Construcción incremental del diagrama VPL.

Y así se continúa el proceso de manera incremental hasta construir el diagrama VPL final. Se ha mostrado [6] que con una correcta implementación este algoritmo puede construir el diagrama VPL en tiempo promedio $\mathcal{O}(n \log n)$ y un espacio $\mathcal{O}(h)$, donde h es el número de puntos involucrados en la envolvente convexa. En un peor caso el algoritmo toma tiempo $\mathcal{O}(n^2)$ y de hecho es la mejor cota que se conoce [1].

2.1.5. Localización de punto

Con la gran relevancia que ha tomado la tecnología móvil en la última década, cada vez se ha hecho más común hacer uso de aplicaciones o servicios que hacen uso de la localización. Si bien hoy existen gran cantidad de dispositivos y satélites que se encargan de resolver este problema en segundos, determinar la posición en su manera más básica es un problema que sigue siendo muy interesante en la geometría computacional.

En esta sección hablaremos del problema de *localización de punto*. Se enuncia de la siguiente manera:

Problema 1. *Dada una subdivisión planar en regiones y un punto q, encontrar en que región se encuentra q.*

Una subdivisión planar es una división del plano que está construida sólo con segmentos de línea recta y sus aristas se intersectan sólo en los extremos. [18]. Estas intersecciones forman los vértices de la subdivisión.

Es importante mencionar que por la naturaleza del problema, la mayoría de los algoritmos existentes que resuelven este problema no sólo se encargan de dar una respuesta, sino además construyen alguna clase de estructura para que las consultas posteriores se realicen de manera más eficiente. En 1976 Dobkin & Lipton plantearon [7] la idea fundamental de construir nuevos objetos geométricos sobre la división del plano, de tal manera que se pudiera realizar la búsqueda binaria. Esta es la idea clave en la mayoría de los algoritmos de localización de punto.

El primer algoritmo que presentaron es conocido como *el método de losas*. Sea S una subdivisión planar con n vértices, empezaremos por dibujar líneas verticales que atraviesen cada uno de los vértices existentes, con lo que al finalizar este paso tendremos a S dividida en losas (ver figura 2.12). Si almacenamos los vértices ordenados respecto a la coordenada X, entonces para localizar cual es la losa que contiene a un punto q sólo nos bastaría realizar una búsqueda binaria en ellas.

Debido a la manera en que se construyeron las losas, dentro de cada una de ellas no puede existir ningún vértice y sólo pueden contener segmentos de recta que las cruzan totalmente. Lo anterior es debido a que si sucediera lo contrario esto provocaría la construcción de otra losa (ver figura 2.13). Esta observación nos permite ver que también podemos ordenar respecto a la coordenada Y a los segmentos de recta que cruzan totalmente a la losa. Por lo tanto para ubicar a un punto dentro de una losa también podemos realizar búsqueda binaria. El algoritmo consta por tanto de dos etapas.

Primeramente se pasa por una fase de pre-procesamiento para preparar la estructura para responder a consultas de manera eficiente de la siguiente manera:

1. Se ordenan los vértices respecto a la coordenada X. Se almacena esto en una lista.

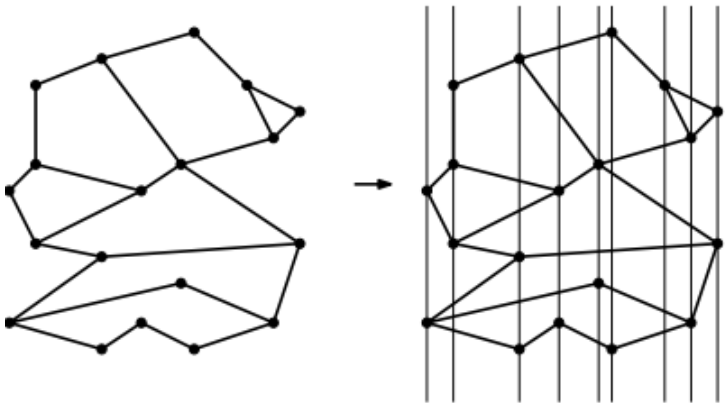

Figura 2.12: Al extender líneas verticales sobre cada vértice se forman las losas, las cuales son las que le dan el nombre al algoritmo.

Figura 2.13: Estructura de una losa.

2. Se crean las losas extendiendo líneas verticales en cada vértice.

3. Se ordenan los segmentos de recta dentro de cada losa respecto a la coordenada Y. Se almacena esto como una lista, de tal manera que la estructura completa esta formada por una lista de listas.

Posteriormente al pre-procesamiento el algoritmo funciona de la siguiente manera, dado un punto q:

1. Se realiza una búsqueda binaria en los vértices. En cada paso se determina si el punto q que buscamos está hacia la derecha o izquierda de un determinado vértice.

2. Se realiza una búsqueda binaria dentro de la losa. En cada paso se determina si q se ubica arriba o abajo de un segmento de recta.

3. Devolvemos la región donde se encuentra q.

No es muy difícil observar que la complejidad temporal del algoritmo está dominada totalmente por el ordenamiento de los vértices, es decir $\mathcal{O}(n \log n)$, sin embargo como se mencionó al principio de esta sección, el interés de resolver estos problemas es responder una gran cantidad de consultas de manera eficiente, y por tanto estamos más interesados en el tiempo de consulta que nos ofrece el algoritmo. Hay que notar que el pre-procesamiento sólo es necesario realizarlo una vez y a partir de ahí las consultas siguientes pueden aprovechar el procesamiento ya realizado.

Una vez que se ha realizado el pre-proesamiento una nueva consulta sólo necesita realizar 2 búsquedas binarias para tener respuesta, y con ello se tiene un tiempo de consulta $\mathcal{O}(\log n)$.

En algunos modelos de datos (ver sección 2.2) existen grandes restricciones respecto al espacio de memoria que pueden utilizar, por lo que también resulta conveniente realizar un análisis de espacio al algoritmo presentado.

Primeramente se requiere almacenar al conjunto de vértices en memoria y generar las losas, como cada 2 vértices adyacentes se genera una losa, se puede deducir que este paso toma espacio $\mathcal{O}(n)$. Adicionalmente cada losa contiene segmentos de recta que la cruzan y también requieren ser almacenados. En un peor caso se puede diseñar un escenario donde cada losa es intersectada por prácticamente todos los aristas, formando así una especie de rejilla. Para ilustrar un ejemplo con este escenario se puede revisar la figura 2.14.

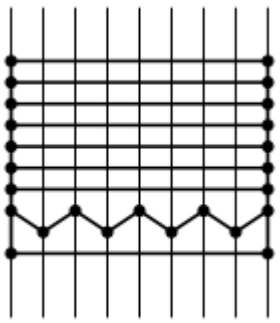

Figura 2.14: Ejemplo de como puede suceder el peor caso para el método de losas. Cada losa se intersecta con prácticamente cada arista formando así una rejilla.

Al tener $\mathcal{O}(n)$ losas con $\mathcal{O}(n)$ intersecciones, se tiene entonces que para el peor caso el espacio puede ser $\mathcal{O}(n^2)$.

Como se mencionó al principio de esta sección, existen numerososo algoritmos para resolver este problema con distintas complejidades y características (ver figura 2.15).

ALGORITMOS PARA LOCALIZACIÓN DE PUNTO			
NOMBRE	**TIEMPO DE CONSULTA**	**COMPLEJIDAD ESPACIAL**	**OBSERVACIONES**
Método de losas	$O(\log n)$	$O(n^2)$	Requiere como pre-procesamiento un ordenamiento de los vértices.
Refinamiento de tríangulos	$O(\log n)$	$O(n)$	Internamente hace uso de algún algoritmo para triángular un polígono.
División trapezoidal	$O(\log n)$	$O(n^2)$	El espacio esperado es O(n).
Cadena de divisiones monotonas	$O(log^2 n)$	$O(n)$	La implementación es compleja.

Figura 2.15: Algoritmos para resolver el problema de localización de punto.

Por supuesto la elección de alguno de ellos dependerá específicamente del tipo de aplicación que se está trabajando.

2.2. Modelo de flujo de datos

En las aplicaciones modernas se generan millones de datos de un segundo a otro, como ejemplo se tiene a las redes sociales o aplicaciones que hacen uso de sensores y reciben cantidades masivas de datos como respuesta. Con esta gran cantidad de información, el modelo de datos tradicional (RAM) donde contamos con todos los elementos almacenados en alguna estructura de datos y gran cantidad de memoria auxiliar, empieza a mostrar algunas limitaciones al no reflejar las condiciones reales que enfrentamos en los problemas actuales. Hoy, tenemos que enfrentar a cantidades de datos tan grandes que no pueden ser almacenadas totalmente en memoria y por tanto exigen pensar de manera diferente la forma en que se resuelven estos problemas.

El *modelo de flujo de datos*[2] es un modelo que nace para representar con más precisión las condiciones en las que se enfrenta a los problemas que involucran cantidades de datos masivas, y donde se espera este comportamiento donde la información sigue fluyendo. Podemos definir un flujo de la siguiente manera:

Definición 4 (Flujo de datos)**.** *Un flujo de datos es una secuencia ordenada de elementos que sólo pueden ser leídos de manera secuencial.*

Con ello podemos definir al modelo de flujo de datos de la siguiente manera:

Definición 5 (Modelo de flujo de datos)**.** *Sea P un problema con un flujo de n elementos como entrada y S un algoritmo para solucionar P, entonces S es una solución en el modelo de flujo si y sólo si hace uso de como máximo una cantidad de espacio $polylog(n)$ y sólo realiza 1 pasada al flujo de elementos.*

Como se puede observar, aunque es un modelo atractivo computacionalmente, también presenta condiciones que resultan muy restrictivas para solucionar un problema. Estas restricciones tal y como están impiden que algoritmos que consiguen hacer uso de muy poco espacio pero consumiendo un poco más de pasadas queden totalmente fuera del modelo de flujo. Por situaciones como la anterior se relajaron un poco las condiciones del modelo para dar paso al modelo de semiflujo:

Definición 6 (Modelo de semiflujo de datos)**.** *Sea P un problema con un flujo de n elementos como entrada y S un algoritmo para solucionar P, entonces S es una solución en el modelo de semiflujo si y sólo si hace uso de como máximo una cantidad de espacio $o(n)$ y sólo realiza un número sublineal de pasadas al flujo de elementos, es decir $o(n)$ pasadas.*

[2]En inglés *streaming*.

Y con lo anterior se tiene un modelo ligeramente más flexible que el anterior, sin perder el atractivo de dar solución al problema con una pequeña cantidad de memoria.

Al principio de esta sección se mencionó que el modelo es atractivo para muchas aplicaciones modernas, con lo que podría pensarse que es un modelo de reciente aparición. El concepto de algoritmo de semiflujo fue enunciado por primera vez en 2005 en una publicación [9] de Feingenbaum et al., pero con el modelo de flujo de datos la historia es diferente. Si bien, la definición formal de los algoritmos de streaming se popularizó en 1996 con una publicación [2] de Noga Alon, Yossi Matias y Mario Szegedy[3], se tienen registros de modelos bastantes similares desde 1982 con Philippe Flajolet and G. Nigel Martin [10] y más atrás en 1980 con Munro y Paterson [16].

Munro y Paterson exponen en su trabajo un modelo donde tiene "una secuencia de n de elementos distintos almacenados en una cinta de sólo lectura y una cantidad de memoria severamente restringida. Al llegar al final de la cinta se rebobina automáticamente para empezar la lectura desde el principio". Con conceptos de su época, en su trabajo estos autores presentaron un modelo que es prácticamente equivalente al modelo actual de flujo de datos y por tanto sus resultados merecen ser comparados contra los algoritmos de flujo modernos.

[3]Estos autores recibieron el premio Gödel en 2005 por su "contribución a la fundación de los algoritmos de flujo".

Capítulo 3

Problemas de estudio

En el presente capítulo se presentan tres problemas de estudio. Primero se revisa el problema de la K-ésima estadística, un problema clásico en varias áreas, y posteriormente se revisan dos problemas de geometría computacional estrechamente relacionados: la búsqueda de la pareja de puntos más cercana y la pareja de puntos más lejana. Para cada uno de ellos se presenta su solución tradicional en el modelo convencional (el modelo RAM) y luego se da paso a definir su versión en el modelo de semiflujo.

Con las grandes cantidades de información que se generan en los sistemas actuales, este modelo cobra gran importancia al representar de manera más adecuada las condiciones con las que enfrentamos problemas modernos. Una gran motivación para la elección particular de éstos problemas fue el hecho de que son fundamentales, y por tanto resulta ser bastante interesante su análisis bajo este modelo.

3.1. La K-ésima estadística

El problema de la K-ésima estadística, también conocido como el problema de selección, es un problema de planteamiento simple pero con un gran impacto no sólo en el área de algoritmos, sino también en otras áreas como la estadística [3]. Se trata de una primitiva muy importante y como tal fue estudiada por mucho tiempo hasta encontrar un algoritmo eficiente. El problema se enuncia de la siguiente manera:

Problema 2. *Dado un conjunto S de n elementos distintos y un entero $1 \leq K \leq n$, encontrar el elemento $x \in S$ tal que x es mayor que exactamente $K-1$ elementos de S.*

Una manera trivial de resolver este problema es simplemente ordenar el conjunto de elementos y acceder directamente a la K-ésima posición. De esta manera el problema se

resuelve en tiempo $\mathcal{O}(n \log n)$. Existe un algoritmo más eficiente para resolver este problema y será presentado a continuación.

3.1.1. Modelo convencional (RAM)

El algoritmo trivial tiene el ordenamiento como cuello de botella, por tanto para hacer más eficiente el proceso hay que evitar hacer uso de éste. En 1973 Blum et al. [4] propusieron un ingenioso algoritmo para resolver este problema. Este algoritmo tiene como procedimiento clave al cálculo de la mediana.

Definición 7 (Mediana)**.** *La mediana de un conjunto de n elementos ordenados es el valor del elemento central. Si n es un número impar la mediana es el valor del elemento que ocupa la posición n/2. Si n es par la mediana es la media arimética de los elementos centrales, es decir, sea x_i el valor del elemento en la posición i, la mediana es $(x_{\frac{n}{2}} + x_{\frac{n}{2}+1})/2$.*

El algoritmo se enuncia a continuación:

1. Se empieza por partir el conjunto de entrada de n elementos en $n/5$ grupos de 5 elementos cada uno, y un último grupo de $n \, mod \, 5$ elementos.

2. Posteriormente se calcula la mediana de cada uno de estos grupos. Dado que se tienen grupos de tamaño constante se puede usar cualquier procedimiento exhaustivo para obtener las medianas.

3. Con todas las medianas obtenidas se forma un nuevo grupo al cual de igual manera se le calcula su mediana. Llamaremos a esta mediana x.

4. Haremos una pasada más al conjunto de entrada para contar que tantos elementos son menores que x. Este procedimiento retornará el índice de x como si el conjunto estuviera ordenado. Llamamos a este elemento *ÍndiceX*.

5. Si $K = IndiceX$ entonces se retorna x. En otro caso, si $K > IndiceX$ entonces se resuelve recursivamente el problema descartando a todos los elementos que están del lado izquierdo de x. De manera análoga si $K < IndiceX$ se resuelve de manera recursiva descartando a todos los elementos que están del lado derecho de x.

La observación clave en este algoritmo es el índice de x, que para ilustrar mejor la idea puede ser comparado a la función que tiene el pivote en el algoritmo de ordenamiento *Quicksort*, en donde la correcta elección de este elemento define la eficiencia total del algoritmo [5]. El índice x contiene algunas propiedades que de igual manera permiten que el algoritmo funcione de manera eficiente. Si se pudiera observar gráficamente todos

los grupos ordenados con sus respectivas medianas (ver figura 3.1) se observaría que el elemento x es cuando menos mayor o igual que la mitad de medianas, y dado que se tienen grupos de 5 elementos, entonces se puede concluir que por transitividad x es mayor que al menos 3 elementos de cada uno de esos grupos. Descontamos 2 grupos que no aportan 3 elementos, uno de ellos es el último grupo donde n tal vez no se dividió de manera exacta, y el otro grupo es el mismo donde se encuentra x. Considerando estos casos se tiene que x es mayor que al menos:

$$3\left(\frac{1}{2}\left(\frac{n}{5}\right)-2\right) \geq \frac{3n}{10}-6 \tag{3.1}$$

De manera análoga existe esta misma cota de elementos que son menores o iguales que x. Por tanto en el peor caso al llamar recursivamente al algoritmo, se habrá reducido al conjunto en al menos esta fracción de elementos, es decir que la recursión se llevaría a cabo con $7n/10+6$ elementos. La observación clave es que se está reduciendo el conjunto de entrada en una fracción constante por cada nivel de la recursión.

Se puede entonces analizar la recursión. En cada nivel el problema se parte en $n/5$ grupos y la recursión como se mencionó anteriormente se lleva a cabo en a lo más $7n/10+6$ elementos. Los procesos de dividir en grupos y calcular la mediana de grupos pequeños toman tiempo $\mathcal{O}(n)$. Por tanto se tiene una recursión de la siguiente manera:

$$T(n) \leq T(n/5)+T(7n/10+6)+\mathcal{O}(n) \tag{3.2}$$

La solución a esta recursión es $\mathcal{O}(n)$ con lo que se tiene un algoritmo lineal y por tanto más eficiente que el algoritmo trivial presentado al inicio de esta sección.

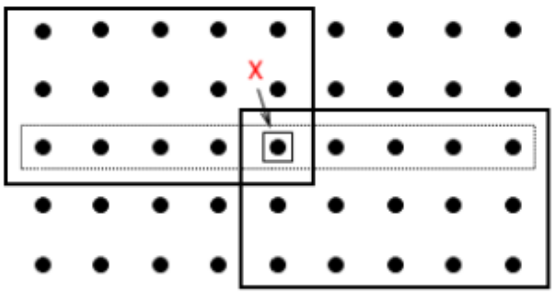

Figura 3.1: La mediana de medianas x tiene la propiedad de ser mayor o igual que la mitad de las medianas, y a su vez mayor que 3 de los elementos de esos grupos. Esta observación permite descartar una fracción constante de elementos en cada nivel de la recursión.

3.1.2. Modelo de semiflujo

El problema de la K-ésima estadística en el modelo de semiflujo se enuncia de la siguiente manera:

Problema 3. *Dado un flujo S de n elementos distintos, un entero $1 \leq K \leq n$, y una cantidad de memoria $m = o(n)$, encontrar el elemento x del flujo tal que x es mayor que exactamente $K - 1$ elementos de S, usando sólo la memoria disponible, y una cantidad sublineal de pasadas al flujo.*

El principal reto que presenta este problema en su versión de semiflujo es enfrentar la restricción de memoria. En el modelo convencional (ver sección 3.1.1) el algoritmo trabaja de manera recursiva, y como tal para adaptarlo a este modelo hay que tener gran cuidado en la cantidad de datos que se almacenan en cada nivel de la recursión. Particularmente en este algoritmo existe un cálculo de medianas que se hace en cada nivel de la recursión del que se requiere tomar en cuenta para cumplir con las restricciones de memoria en este modelo. Adicionalmente el algoritmo convencional tiene entre sus pasos hacer una pasada completa al conjunto de elementos, evento que de manera estricta no puede realizarse en el modelo de semiflujo, y por tanto requiere una adaptación o alternativa.

3.2. La pareja de puntos más cercana

Uno de los problemas fundamentales que se plantearon en la geometría computacional es el de dada una nube de puntos en el plano devolver cuales son los dos puntos más cercanos entre sí [18]. No resulta complicado pensar la gran cantidad de aplicaciones que tiene este problema no sólo como subrutina para problemas teóricos más complejos, sino también para aplicaciones en telecomunicaciones, como por ejemplo localizar la radiobase más cercana a un teléfono móvil; en aviación para saber cuál es el aeropuerto de emergencia más cercano a otro, entre muchas otras aplicaciones de ingeniería. El problema se enuncia de la siguiente manera:

Problema 4. *Dado un conjunto de $n \geq 2$ puntos en el plano en posición general, retornar una pareja de puntos cuya distancia euclideana mutua es menor o igual que cualquier otra pareja de puntos en el conjunto.*

Por supuesto una solución trivial al problema sería revisar todas las parejas de puntos exhaustivamente y almacenar en memoria cual es la mejor que se ha visto visto. No es muy díficil ver que este proceso lleva a una complejidad de $\Theta(n^2)$. Como podría esperarse existe un algoritmo para resolver eficientemente este problema y será descrito a continuación.

3.2.1. Modelo convencional (RAM)

Un algoritmo que resuelve eficientemente este problema hace uso de la técnica divide y vencerás. Esta técnica divide al problema en un conjunto de subproblemas de menor tamaño y los resuelve recursivamente. Al final se combinan las soluciones locales para generar la solución global al problema. Se puede revisar [5] para más detalles sobre esta técnica.

Como pre-procesamiento sobre el conjunto de puntos $P = \{P_1, P_2, ...P_n\}$ se crean dos arreglos; uno de ellos con todos los puntos ordenados respecto a la coordenada X, y el otro con los puntos ordenados respecto a la coordenada Y. El algoritmo empieza por dividir a P con una línea imaginaria paralela al eje Y. Esta línea dividirá por la mitad al conjunto en dos subconjuntos, por un lado P_L con todos los puntos que están del lado izquierdo de la línea y por otro lado P_R con todos los puntos del lado derecho de la misma (ver figura 3.2). Llamaremos a ésta línea divisora.

Posteriormente en cada mitad aplicamos el mismo proceso de manera recursiva dividiendo el problema cada vez en instancias más pequeñas y pasando a su vez los subarreglos ordenados respecto a cada coordenada. Cuando el problema sea suficientemente pequeño, es decir 3 puntos o menos, se resuelve el problema de manera directa con el procedimiento de fuerza bruta mencionado al principio de esta sección.

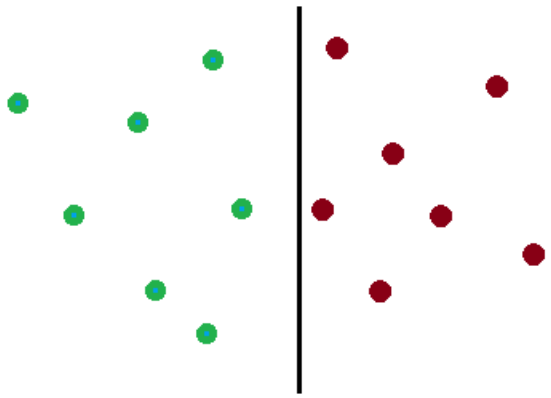

Figura 3.2: Los puntos son divididos por la mitad de acuerdo a su coordenada X. Los puntos color verde pertenecen a P_L y los puntos rojos pertenecen a P_R.

En cada nivel del problema se hacen entonces 2 llamadas recursivas, una para cada subconjunto de puntos. Como resultado a éstas llamadas se retorna a δ_L y a δ_R que denotan la distancia más pequeña calculada para el subconjunto P_L y P_R respectivamente, y los puntos involucrados. Nos quedamos con la menor distancia de ambas llamadas $\delta = min(\delta_L, \delta_R)$. Hay que notar que la pareja de puntos más cercana podría involucrar un punto en el lado izquierdo de la línea divisora y el otro en el lado derecho, y por lo tanto todavía se tiene que analizar como combinar las soluciones parciales para considerar este caso.

Dado que ya se tiene la distancia mínima calculada en cada subconjunto, sólo resta considerar las parejas de puntos que son candidatas a tener una distancia menor a δ donde un punto pertenece a P_L y el otro a P_R. Lo anterior nos delimita una zona donde se deberían encontrar los puntos en caso de que existiesen. Se puede representar esta zona de manera gráfica (ver figura 3.3) como una banda que tiene a la línea divisora en la mitad y se expande hacia ambos lados de ésta una distancia δ. Llamaremos a ésta banda de solución. Aún con esta observación en el peor de los casos todos los puntos podrían permanecer en la banda de solución, y se tendría que comparar exhaustivamente las distancias entre todos los puntos. Aunque pareciera que no se ha ganado nada con la banda de solución, aún queda una observación clave por hacer que será la que permita hacer un algoritmo eficiente.

Se empieza a iterar por cada punto en la banda de solución. Se puede observar que dado un punto no es necesario calcular la distancia contra todos los demás puntos en la banda porque con algunos se supera claramente la distancia δ. De hecho, si hay una pareja de puntos cuya distancia es menor que δ, ésta debe estar contenida en un rectángulo de $2\delta * 2\delta$. Una manera sencilla de ver lo anterior es que si se fija un punto, solamente se

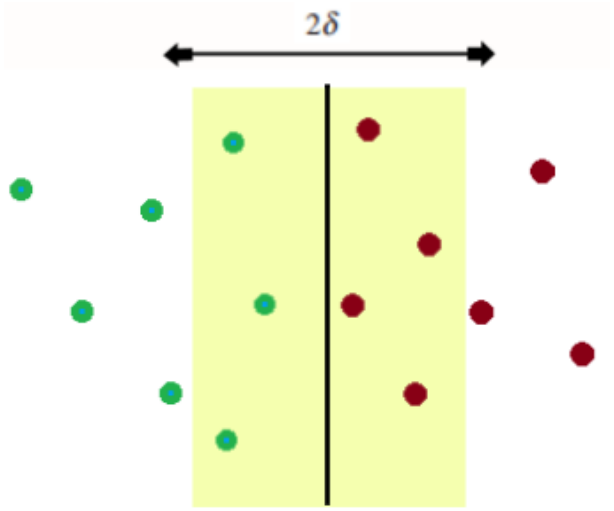

Figura 3.3: La banda de solución (en color amarillo) delimita la zona de búsqueda donde en caso de existir debiera encontrarse una mejor solución al problema.

tiene buscar δ unidades hacia arriba, abajo, derecha e izquierda de él, y con ello se genera este rectángulo. Si se subdivide este rectángulo en 16 cajas de $\delta/2 * \delta/2$ (ver figura 3.4), se observa que solamente puede existir un punto por caja (de lo contrario δ no sería la menor distancia vista en los conjuntos P_L y P_R). Esta observación hace notar que dado un punto en la banda de solución basta solamente con verificar una cantidad constante de puntos en el arreglo ordenado por la coordenada Y para saber si existe una distancia menor a δ.

Es importante resaltar que en este caso se tienen que revisar los 15 puntos siguientes (sobre la coordenada Y) al punto que se esta analizando, sin embargo esta cantidad puede ser reducida [5]. El resultado importante es que se tiene una cantidad de trabajo constante.

Analicemos ahora la complejidad total del algoritmo. Antes de empezar el proceso de división del problema en subproblemas más pequeños, se requiere como pre-procesamiento ordenar los puntos respecto a cada coordenada; esto tomará $\mathcal{O}(n \log n)$. Ahora queda analizar la recursión. En cada nivel se divide el problema en dos subproblemas que reciben como entrada la mitad del conjunto de puntos, adicionalmente se requiere enviar a a cada uno de estos subproblemas el subarreglo de sus puntos ordenados respecto a cada coordenada, esta división puede realizarse en $\mathcal{O}(n)$. Por tanto se tiene una recursión de la siguiente manera:

$$T(n) = \begin{cases} 2T(n/2) + \mathcal{O}(n), \ si \ n > 3 \\ \mathcal{O}(1), \ si \ n \leq 3 \end{cases} \tag{3.3}$$

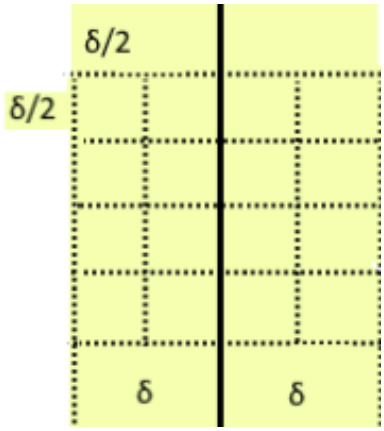

Figura 3.4: Si una pareja de puntos tiene distancia menor a δ entonces debe residir en un rectángulo de $2\delta * 2\delta$.

La solución a esta recursion es $T(n) = \mathcal{O}(n \log n)$. Si a ello se le añade el costo de pre-procesamiento se tiene entones una complejidad algorítmica total de $\mathcal{O}(n \log n)$, lo cual es claramente mejor que el procedimiento exhaustivo de fuerza bruta.

3.2.2. Modelo de semiflujo

El problema de la pareja de puntos más cercana en el modelo de semiflujo se enuncia de la siguiente manera:

Problema 5. *Dado un flujo de $n \geq 2$ puntos en el plano en posición general y una cantidad de memoria $m = o(n)$, encontrar la pareja de puntos cuya distancia euclideana mutua es menor o igual que cualquier otra pareja de puntos en el flujo, usando sólo la cantidad de memoria disponible y una cantidad sublineal de pasadas al flujo.*

El reto principal que enfrenta este problema en su versión de semiflujo es el hecho de que el algoritmo tradicional queda totalmente descartado. La razón de lo anterior es que tan solo el pre-procesamiento del algoritmo convencional (ver sección 3.2) requiere ordenar al conjunto de entrada, situación que rompe las restricciones de memoria presentes en el modelo de semiflujo. Pero eso no es todo, si se supone que por alguna razón el conjunto de entrada esta ordenado, el algoritmo tradicional usa la técnica divide y vencerás, la cual por si misma divide al problema en mitades recursivamente, y éstas tan sólo en la primera iteración nuevamente romperían las restricciones de memoria del modelo de semiflujo.

Con ello, el reto de solucionar el problema de la pareja de puntos más cercana en este modelo exige una manera de pensar totalmente distinta a la convencional, teniendo

la necesidad de cambiar totalmente la técnica empleada, y con ello un gran atractivo computacional.

3.3. La pareja de puntos más lejana

Así como el problema de la pareja de puntos más cercana es uno de los problemas elementales de la geometría computacional, de manera natural se planteó el problema análogo: la búsqueda de la pareja de puntos más lejana. Contrario a lo que podría parecer, la solución eficiente a este problema no tiene relación alguna con la del problema análogo. Preparata & Shamos (1985) mencionan en su libro [18] que "La complejidad de este tipo de problemas son cuestiones básicas en la geometría computacional, las cuales tenemos el deber de responder aún en ausencia de aplicaciones prácticas". El problema se enuncia se la siguiente manera:

Problema 6. *Dado un conjunto de $n \geq 2$ puntos en el plano en posición general, encontrar una pareja de puntos cuya distancia euclideana mutua es mayor o igual que cualquier otra pareja de puntos en el conjunto.*

A este problema también se le conoce en la literatura como el problema del diámetro de un conjunto de puntos.

3.3.1. Modelo convencional (RAM)

La primera observación que se puede realizar sobre el problema es que la pareja de puntos que se busca tiene relación directa con la envolvente convexa:

Lema 1. *La pareja de puntos más lejana de un conjunto de puntos forma parte de los vértices de su envolvente convexa.*

Demostración. Por contradicción, supongamos que la pareja de puntos más lejana P y Q no forma parte de la envolvente convexa. Ahora sin pérdida de generalidad supongamos que Q no es un vértice de la envolvente convexa. Si se imagina que estos dos puntos están unidos por una línea recta, entonces se podría observar que dado que Q no es parte de la envolvente convexa, entonces la línea proveniente desde P podría ser prolongada aún más hasta impactar con la envolvente convexa. Tenemos entonces 2 casos:

a) La línea impacta contra un vértice de la envolvente convexa: En este caso se tiene que la línea se ha estirado y ha tocado contra otro vértice, por tanto dado que se esta agregando un segmento más a una recta, significa que la nueva pareja de puntos tiene una distancia mayor que la pareja P y Q. Lo anterior no puede ser posible por que la hipótesis es que éstos son la pareja más distante y con ello se tiene una contradicción.

b) La línea impacta contra un arista de la envolvente convexa: La línea prolongada impacta contra un arista de la envolvente convexa formando un ángulo (ver figura 3.5), llamaremos X a este punto de impacto. Si el ángulo no es recto, entonces el arista tiene una inclinación que al seguirla en dirección hacia donde se apertura el ángulo mayor debe concluir en un vértice R de la envolvente convexa. Se puede dibujar entonces un triángulo con vértices en P, R y X, donde P y R forman la hipotenusa y por tanto ésta pareja de puntos tiene una distancia mayor que la pareja P y Q, con lo que se tiene una contradicción. Si el ángulo es recto entonces se camina desde X sobre el arista de impacto en cualquier sentido hasta encontrar un vértice R, y se tiene que se puede formar un triángulo rectángulo, y bajo el mismo argumento mencionado anteriormente se contradice nuestra hipótesis inicial. □

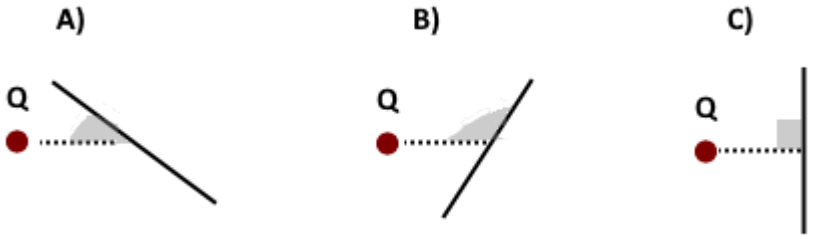

Figura 3.5: Tipos de arista de impacto.

En 1961 Yaglom-Boltyanskii [21] enunciaron un teorema relacionado:

Teorema 1. *El diámetro de una figura convexa es la mayor distancia entre sus líneas paralelas de soporte.*

Las líneas de soporte son aquellas líneas tangenciales a un vértice que dejan a todo el polígono a un lado de ésta. No cualquier pareja de puntos puede tener estas líneas (ver figura 3.6). A la pareja de puntos que admite líneas paralelas de soporte se le llama *antipodales* [18].

Por tanto, el problema se reduce a generar de manera eficiente todas las parejas de puntos antipodales (ver sección 2.1.2). Dado que todas las parejas de puntos antipodales pueden encontrarse en tiempo lineal, y que calcular la distancia de cada antipodal puede hacerse en tiempo constante, entonces se puede encontrar la pareja de puntos más lejana de un polígono convexo en $\mathcal{O}(n)$. Por supuesto en el caso general no siempre se tiene un polígono convexo por lo que se requiere hacer un pre-procesamiento para calcular la envolvente convexa. Este pre-procesamiento toma $\mathcal{O}(n \log n)$ (ver sección 2.1.1) y domina totalmente la complejidad del algoritmo.

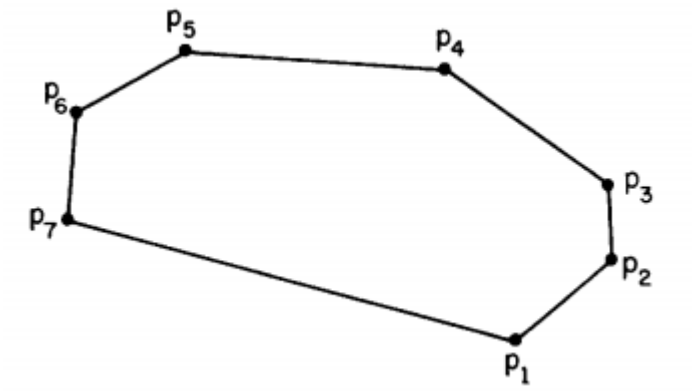

Figura 3.6: El vértice p_4 y el vértice p_6 no son antipodales debido a que no existe un par de líneas paralelas de soporte que puedan pasar por ambos puntos.

3.3.2. Modelo de semiflujo

El problema de la pareja de puntos más lejana en el modelo de semiflujo se enuncia de la siguiente manera:

Problema 7. *Dado un flujo de $n \geq 2$ puntos en el plano en posición general y una cantidad de memoria $m = o(n)$, encontrar la pareja de puntos cuya distancia euclideana mutua es mayor o igual que cualquier otra pareja de puntos en el conjunto, usando sólo la cantidad de memoria disponible y una cantidad sublineal de pasadas al flujo.*

De manera similar al problema de la pareja de puntos más cercana, adaptar este problema al modelo de semiflujo tiene como reto principal el hecho de que su solución convencional no puede ser usada de ninguna manera. En el algoritmo convencional (ver sección 3.3.1) se requiere calcular la envolvente convexa del conjunto de puntos. Calcular la envolvente convexa es un pre-procesamiento que implica ordenar el conjunto de entrada o emplear la técnica de divide y vencerás; en cualquier caso se romperían totalmente las restricciones de memoria que impone el modelo de semiflujo.

Por lo anterior, la solución a este problema en su versión de semiflujo tiene que replantear la manera en que se ataca este problema tradicionalmente, probablemente prescindiendo de un pre-procesamiento sobre el flujo y usando estructuras que sean eficientes en espacio. Todo esto hace de este problema un reto algorítmico interesante.

Capítulo 4

La K-ésima estadística en semiflujo

En el presente capítulo propondremos una solución para el problema de la pareja de puntos más cercana en el modelo de semiflujo. El algoritmo propuesto es una adaptación del algoritmo clásico presentado en la sección 3.1.

4.1. Algoritmo

El algoritmo de Blum et al. [4] emplea la técnica de divide y vencerás para resolver el problema. Esta técnica consiste en ir dividiendo el conjunto de entrada en grupos más pequeños, y resolver estos subproblemas para combinar posteriormente las soluciones parciales en una solución global. Cuidando pequeños detalles de memoria y adaptando los pasos que requieren recorrer todo el flujo de elementos, encontramos que es posible adaptar el algoritmo al modelo de semiflujo. Sea m un tamaño de memoria y un valor K. El algoritmo es el siguiente:

1. Se empieza por leer los primeros m elementos del flujo y se forma un grupo con ellos. Se calcula y se almacena la mediana de este grupo. Para calcular la mediana de estos grupos se puede usar el algoritmo convencional presentado en la sección 3.1. Se repite este paso con los siguientes m elementos del flujo y así sucesivamente hasta terminar de haber procesado cada elemento del flujo.

2. Con las medianas almacenadas en el paso anterior se forma un nuevo grupo al que nuevamente se le calcula su mediana, llamaremos a este elemento P.

3. Por medio de una rutina llamada PARTICIÓN (más adelante se explica su implementación) se encuentra el índice de P como si el flujo estuviera ordenado. Llamamos a este valor *IndiceP*.

4. Si $IndiceP = K$ entonces se retorna P. En otro caso, si $K > IndiceP$ entonces se resuelve recursivamente el problema descartando a todos los elementos que son menores que P. De manera análoga si $K < IndiceP$ se resuelve de manera recursiva descartando a todos los elementos que son mayores a P. Si la fracción de elementos que se eliminó es la que tiene valores mayores a P, el valor de K se conserva igual. Si se eliminó la fracción de elementos con valores menores a P, sea E la cantidad de elementos eliminados, se debe actualizar K con el nuevo valor de $K = K - E$.

La rutina PARTICIÓN tiene como objetivo encontrar el índice del elemento P en un ordenamiento ascendente del flujo. Para realizar esta tarea lo que se hace es simplente procesar todo el flujo y contar qué tantos elementos son menores a P. La rutina PARTICIÓN se describe a continuación:

1. Se empieza por leer un elemento E del flujo.

2. Definimos un contador llamado $ValoresMenores$ el cual almacenará la cantidad de elementos del flujo que son menores a P. Se compara si $E < P$ y en caso de resultar verdadero se incrementa la variable $ValoresMenores$. Se mantiene esta variable con su valor en iteraciones posteriores.

3. Se vuelve al paso 1 eligiendo ahora otro elemento del flujo. Si ya no hay elementos restantes por procesar en el flujo entonces se retorna $ValoresMenores + 1$.

4.2. Análisis

Dado que el algoritmo es una adaptación del algoritmo tradicional, su validez se sigue esencialmente del análisis realizado en la sección 3.1, y por lo tanto en este análisis sólo se limitará a hacer un estudio de las complejidades.

4.2.1. Complejidad temporal

Al igual que el algoritmo clásico, la clave para que el algoritmo funcione eficientemente es reducir en cada iteración una fracción constante de la cantidad de elementos que se está trabajando. Antes de hacer el cálculo de complejidad temporal hay que analizar como es que el algoritmo realiza esta tarea.

Para lograr lo anterior existe una observación por hacer respecto al elemento P (ver paso 3 del algoritmo). Imaginemos que se tienen cada uno de los grupos que se formaron ordenados de manera ascendente respecto al valor de su mediana y de igual manera cada grupo tiene sus elementos ordenados, de tal manera que dentro de un grupo los elementos

menores o iguales al valor de la mediana permanecerán del mismo lado, y de igual manera los elementos mayores permanecerán en otro. El grupo del centro tendrá a la mediana de medianas (el elemento P). Al observar al elemento P, nos daremos cuenta que es al menos mayor o igual que la mitad de las medianas y dado que estas medianas a su vez son cuando menos mayores o iguales que la mitad de los elementos del grupo al que pertenecen, entonces por transitividad P es mayor también que estos elementos (ver figura 4.1).

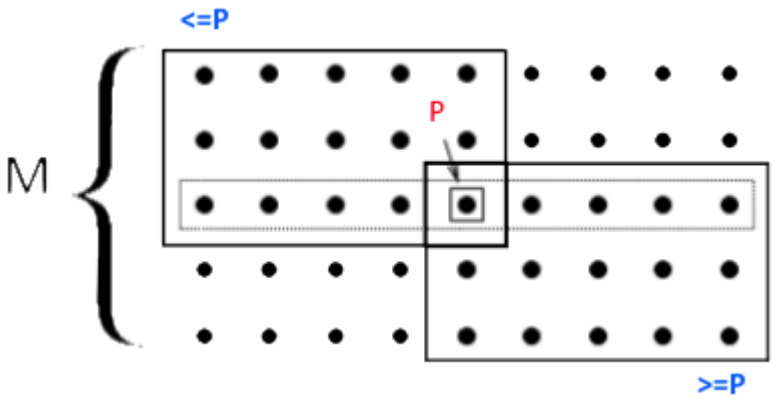

Figura 4.1: El elemento P tiene algunas propiedades que permiten descartar a una fracción constante de elementos en cada nivel de la recursión.

Entonces se tiene que dado existen $\lceil \frac{n}{m} \rceil$ grupos y P es cuando menos mayor o igual que la mitad de medianas de cada uno de estos grupos entonces:

$$P \geq \frac{1}{2}\left(\lceil \frac{n}{m} \rceil \right) \tag{4.1}$$

A su vez por transitividad P es mayor o igual que al menos la mitad de los elementos de cada uno de estos grupos, por lo tanto:

$$P \geq \left(\frac{1}{2}\right)\left(\lceil \frac{n}{m} \rceil \right)\left(\frac{m}{2}\right) \tag{4.2}$$

Finalmente, hay que notar que no todos los grupos aportan la mitad de sus elementos, particularmente el grupo donde se encuentra P no lo hace, y en caso de que $n \, mod \, m \neq 0$ se tiene un grupo más. Por lo tanto:

$$\begin{aligned} P &\geq \frac{1}{2}\left[\left(\left\lceil \frac{n}{m} \right\rceil\right)\left(\frac{m}{2}\right) - 2\right] \\ P &\geq \frac{1}{2}\left[\frac{n}{2} - 2\right] \\ P &\geq \frac{n}{4} - 1 \end{aligned} \tag{4.3}$$

Con ello se tiene que P es al menos mayor o igual que aproximadamente $\frac{n}{4}$ de los elementos, y de manera simétrica es menor que al menos la misma fracción. Con esto en mente en cada paso se puede eliminar $\frac{1}{4}$ de los elementos y resolver el problema de manera recursiva con $\frac{3}{4}n$. Al eliminar una fracción constante en cada nivel de la recursión se tiene que la profundidad llegará a ser en un peor caso una cantidad logarítmica respecto a n.

Aún queda un detalle por analizar y es como avanzar de un nivel de la recursión a otro. En el modelo tradicional sin mucho problema se puede sencillamente eliminar $\frac{1}{4}$ de los elementos y copiar los $\frac{3}{4}$ restantes al siguiente nivel de la recursión. En el modelo de semiflujo no se puede optar por esta estrategia por que rompería las restricciones de memoria fácilmente. Para solucionar este problema lo que se hará es apoyarse en un bit de memoria adicional por cada nivel. Dado que lo que interesa en cada nivel de la recursión es si $K > \mathit{IndiceP}$ o $K < \mathit{IndiceP}$, podemos almacenar esto en un bit de memoria indicando simplemente con un 0 si para el siguiente nivel de la recursión se mantienen activos los elementos menores a $\mathit{IndiceP}$, o un 1 en el caso contrario. Para que esta estrategia funcione no sólo se necesita un bit de memoria por cada nivel de la recursión, sino también almacenar el elemento P de cada nivel, de esta manera en un determinado nivel de la recursión se puede acotar debidamente el intervalo de elementos que se está trabajando. A esta combinación del elemento P de cada nivel y su respectivo bit le llamaremos *predicado*.

El funcionamiento es como sigue, en un determinado nivel de la recursión h se tienen $h-1$ predicados. Se empieza por leer cada elemento del flujo y se le aplica uno a uno el conjunto de predicados para saber si el elemento es parte de la secuencia de entrada del nivel recursivo h. La observación importante que hay que realizar es que después de filtrar los elementos hay que actualizar el valor de K debido al recorrimiento de elementos que puede existir durante este proceso. Dicho de otra manera, el valor de K que se busca en una determinada iteración h puede ser diferente en la iteración $h+1$ y por lo tanto puede requerirse actualizar su valor para conservar el objetivo inicial del algoritmo, tal y como se explica en el paso 4. El pseudocódigo completo del algoritmo se encuentra en Algoritmo 1.

Después de haber realizado esta observación ahora si podemos juntar todo para determinar la complejidad temporal del algoritmo. En el paso 1 se procesa el flujo completo por lotes de tamaño m para obtener una mediana por cada grupo. Para obtener la mediana de

un grupo se usa el algoritmo tradicional (ver sección 3.1) que toma tiempo $\mathcal{O}(m)$. Dado que se tienen un total de $\lceil \frac{n}{m} \rceil$ grupos, entonces el primer paso toma tiempo:

$$\begin{aligned} &Complejidad\,temporal\,primer\,paso = \frac{n}{m} * \mathcal{O}(m) \\ &Complejidad\,temporal\,primer\,paso = \mathcal{O}(n) \end{aligned} \tag{4.4}$$

En el paso 2 con las medianas almacenadas en el paso anterior se obtiene la mediana de medianas (el elemento P). Esto puede realizarse en $\mathcal{O}(\frac{n}{m})$.

En el paso 3 se hace uso de la rutina PARTICIÓN la cual esencialmente sólo requiere hacer un recorrido lineal de los elementos y por tanto consume tiempo $\mathcal{O}(n)$.

En el paso 4 se compara el *IndiceP* con el valor de K para decidir si el algoritmo continúa de manera recursiva. En caso de que se requiera actualizar el valor de K puede ser necesaria realizar una pasada más al flujo, por tanto este paso está dominado por tiempo $\mathcal{O}(n)$.

Con ello se tiene que la complejidad temporal de una iteración del algoritmo está dominada por $\mathcal{O}(n)$. Dado que el algoritmo se ejecuta recursivamente un número logarítmico de veces en un peor caso, entonces la complejidad temporal total del algoritmo es:

$$\begin{aligned} &Complejidad\,temporal\,total = \mathcal{O}(n) * \mathcal{O}(\log n) \\ &Complejidad\,temporal\,total = \mathcal{O}(n \log n) \end{aligned} \tag{4.5}$$

Si realizamos un análisis amortizado de la complejidad temporal total respecto al número de elementos en el flujo, se tiene que el tiempo consumido por elemento es:

$$\begin{aligned} &Complejidad\,Temporal\,Amortizada\,Por\,Elemento = \mathcal{O}\left(\frac{n \log n}{n}\right) \\ &Complejidad\,Temporal\,Amortizada\,Por\,Elemento = \mathcal{O}(\log n) \end{aligned} \tag{4.6}$$

4.2.2. Complejidad espacial

En cuanto a la memoria hay que realizar algunas observaciones interesantes. Primeramente para realizar el procesamiento por lotes de tamaño m es claro que se necesita espacio $\mathcal{O}(m)$. En la primera iteración el algoritmo almacena $\lceil \frac{n}{m} \rceil$ valores en memoria (ver figura 4.2). Con estos valores se calcula el elemento P y una vez que se tiene este valor es posible liberar la memoria usada por los $\lceil \frac{n}{m} \rceil$ valores y reusarla. Para calcular el elemento P se hace uso del algoritmo convencional que consume espacio lineal respecto a la entrada, es decir $\mathcal{O}(\frac{n}{m})$. En la siguiente iteración se repetirá esta secuencia de eventos y dado que de una iteración a otra se elimina siempre una fracción constante de elementos,

Algorithm 1 CalcularKesimoElemento(K, flujo, M, predicados)

Require: K, flujo, M, predicados

```
1: var listaDeMedianas
2: //PASO 1:
3: while aunHayElementosEnElFlujo() do
4:    var elemento = leerUnElementoDelFlujo(flujo);
5:    if elementoActivoDespuesDeAplicarPredicados(elemento , predicados) then
6:       grupo.agregarElemento(elemento);
7:    end if
8:    if grupo.contarElementos() = M OR (aunHayElementosEnElFlujo() = false) then
9:       listaDeMedianas.agregar( calcularMediana(grupo) )
10:      grupo.borrar()
11:   end if
12: end while
13: //PASO 2:
14: var P = calcularMediana(listaDeMedianas)
15: //PASO 3:
16: var indiceP = particion(P, predicados)
17: //PASO 4:
18: if indiceP = K then
19:    return P
20: end if
21: if indiceP > K then
22:    predicados.agregar(1, P)
23:    return calcularKesimoElemento(flujo, K, predicados)
24: end if
25: if indiceP < K then
26:    predicados.agregar(0, P)
27:    var cantidadDeElementosEliminados = contarElementosEliminados(flujo, predicados)
28:    var KActualizada = K − cantidadDeElementosEliminados
29:    return calcularKesimoElemento(flujo, KActualizada, predicados)
30: end if
```

Algorithm 2 particion(P, flujo, predicados)

Require: P, flujo, predicados

```
1: var valoresMenores = 0;
2: while aunHayElementosEnElFlujo() do
3:    var elemento = leerUnElementoDelFlujo(flujo);
4:    if elementoActivoDespuesDeAplicarPredicados(elemento, predicados) = true then
5:       if elemento < P then
6:          valoresMenores = valoresMenores + 1
7:       end if
8:    end if
9: end while
10: return valoresMenores + 1
```

Algorithm 3 elementoActivoDespuesDeAplicarPredicados(elemento , predicados)

Require: elemento, predicados

```
1: while aunHayPredicadosPorAplicar() do
2:    var predicado = tomarPredicado(predicados);
3:    if elementoEsDescartadoPorPredicado(elemento, predicado) = true then
4:       return false;
5:    end if
6: end while
7: return true
```

Algorithm 4 contarElementosEliminados(flujo , predicados)

```
Require: flujo, predicados
1: var elementosEliminadosEnElUltimoPredicado = 0
2: while aunHayElementosEnElFlujo() do
3:   var elemento = leerUnElementoDelFlujo(flujo);
4:   //Aplicamos todos los predicados excepto el último
5:   if elementoActivoDespuesDeAplicarPredicados(elemento, predicados - predicados.obtenerUltimoPredicado()) = true then
6:     //Aplicamos el último predicado y contamos si el elemento fue eliminado por él
7:     if elementoActivoDespuesDeAplicarPredicados(elemento, predicados.obtenerUltimoPredicado() ) then
8:       elementosEliminadosEnElUltimoPredicado++;
9:     end if
10:  end if
11: end while
12: return elementosEliminadosEnElUltimoPredicado
```

entonces la cantidad de valores almacenados en un determinado momento en la segunda iteración es menor que la cantidad de valores almacenados en un determinado momento en la primera iteración. Este comportamiento se extiende a través de la profundidad de la recursividad. Con lo anterior se tiene que asintóticamente la primera iteración es la que domina el espacio en memoria del algoritmo, es decir $\mathcal{O}(\frac{n}{m} + m)$. Dado que por cada nivel se almacena un predicado (elemento P acompañado de 1 bit) y dado que en un peor caso se tienen $\mathcal{O}(\log n)$ iteraciones, entonces al sumar todo se tiene que la cantidad de memoria usada por el algoritmo es:

$$\begin{aligned} Complejidad\ Espacial &= \mathcal{O}(m) + \mathcal{O}(\frac{n}{m}) + \mathcal{O}(\log n) \\ Complejidad\ Espacial &= \mathcal{O}(m + \frac{n}{m}) \end{aligned} \tag{4.7}$$

Hay que notar que descartamos el factor logarítmico de la suma final debido a que este siempre queda totalmente dominado por alguno de los otros dos sumandos. Si m tiene un valor medianamente grande entonces el primer término lo domina totalmente, y en el caso de que m tenga un valor pequeño es el segundo término el que lo domina. Por lo anterior en la suma final lo hemos despreciado.

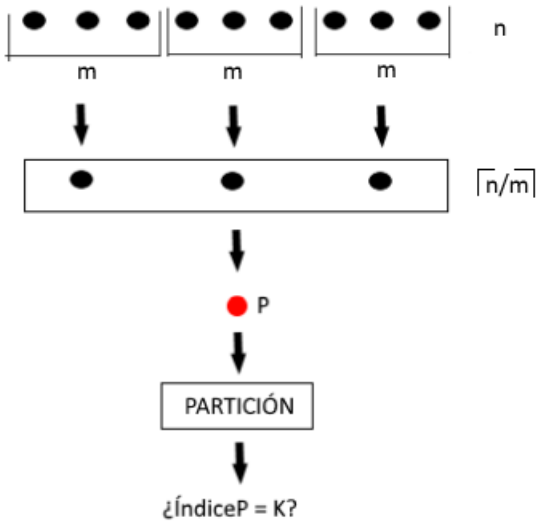

Figura 4.2: Estructura general de la primera iteración del algoritmo para encontrar el K-ésimo elemento en semiflujo. Si $K \neq P$ el algoritmo continúa recursivamente eliminando una fracción de los elementos.

4.2.3. Total de pasadas

Analicemos ahora el total de pasadas. El algoritmo consume 1 pasada en el paso 1 (filtrado y procesamiento de todo el flujo en lotes), otra pasada en el paso 3 (encontrar el índice de P por medio de la rutina PARTICIÓN) y dependiendo de la implementación puede usarse una pasada más para actualizar el valor de K en caso de ser necesario. Con ello se tienen un total de 3 pasadas por iteración en un peor caso. Anteriormente se mencionó que la profundidad de la recursividad al eliminar una fracción constante de elementos en cada nivel puede llegar a ser tanto como una cantidad logarítmica respecto a n, por tanto se tiene:

$$Total\ de\ pasadas = \mathcal{O}(1) * \mathcal{O}(\log n) = \mathcal{O}(\log n) \tag{4.8}$$

Entonces resumiendo las propiedades del algoritmo, se tiene:

$$\begin{aligned} &Complejidad\ espacial = \mathcal{O}(\frac{n}{m} + m) \\ &Total\ de\ pasadas = \mathcal{O}(\log n) \\ &Complejidad\ temporal\ total = \mathcal{O}(n \log n) \\ &Complejidad\ temporal\ amortizada\ por\ elemento = \mathcal{O}(\log n) \end{aligned} \tag{4.9}$$

4.2.4. Valores de m

A continuación vamos a analizar para que valores de m el algoritmo está en el modelo de semiflujo. Dado que la cantidad de pasadas del algoritmo no depende del valor de m, entonces solamente tenemos que enfocarnos en la cantidad de memoria que usa el algoritmo. Se desea que el valor de m (recordando que es el tamaño del lote de procesamiento) esté en función del tamaño del flujo n. Si recordamos la restricción de memoria para que el modelo de semiflujo es que se use como máximo una cantidad sublineal de memoria.

Naturalmente se puede pensar como candidato al valor de m en una función logarítmica respecto a n por ser una función que se sabe matemáticamente crece sus valores muy lento (por supuesto posteriormente habría que verificar que se mantiene dentro del modelo). Sin embargo aún no se ha observado un detalle y es el hecho de que la memoria usada por el algoritmo es $\mathcal{O}(\frac{n}{m} + m)$, con lo que el valor de m es el denominador de un cociente y como tal al ir decreciendo aumenta la memoria total, por lo que lo ideal es buscar un balance entre el valor de m y la memoria total. Si $m = \log n$ entonces la cantidad de memoria usada por el algoritmo es $\mathcal{O}(\frac{n}{\log n} + \log n)$.

Una alternativa interesante a la función logarítmica es $\mathcal{O}(\sqrt{n})$. Esta función tiene un crecimiento lento y de igual manera es sublineal en n. Al crecer de manera ligeramente más rapida que la función logarítmica, el cociente discutido anteriormente tendrá un mejor balance al aumentar ligeramente la cantidad de elementos que se pueden almacenar por lotes (valor de m) y disminuir la memoria. Si $m = \sqrt{n}$ entonces la cantidad de memoria usada por el algoritmo es $\mathcal{O}(\frac{n}{\sqrt{n}} + \sqrt{n}) = \mathcal{O}(\sqrt{n} + \sqrt{n})$.

Veamos un ejemplo para ver para que valor de m de estas opciones se obtienen mejores valores. Sea $n = 1,073,741,823$ y $m = \log n$, es decir $m = 29$, entonces el número de pasadas es de igual manera $\log n = 29$, y la memoria usada por el algoritmo es:

$$Memoria = \frac{1,073,741,823}{29} + 29 = 37,025,609 \tag{4.10}$$

Sea $n = 1,073,741,823$ y $m = \sqrt{n} = 32,768$ entonces la cantidad de pasadas es $m = 29$ y la memoria total usada por el algoritmo es:

$$Memoria = 32,768 + 32,768 = 65,536 \tag{4.11}$$

Con ello se muestra una gran diferencia de espacio entre los distintos valores de m. Con lo anterior podemos proponer que $m = \mathcal{O}(\sqrt{n})$ es un buen valor para nuestro algoritmo. En este punto podría surgir la pregunta de cual es el rango de valores que puede tener m para seguir manteniéndose en el modelo de semiflujo. Para ello hay que observar la complejidad espacial del algoritmo:

$$Memoria\,total = \mathcal{O}(\frac{n}{m} + m) \tag{4.12}$$

Dado que las restricciones del modelo permiten hasta una cantidad de memoria sublineal respecto a n, el valor de m tiene que estar acotado entre ciertos valores. Por ejemplo m no puede ser tan pequeña como un número constante, por que entonces se tendría una complejidad espacial lineal en n, ni tampoco puede ser tan grande como $\frac{n}{2}$ por la misma razón anterior. Dado que el segundo término m cumple de manera trivial la restricción de memoria, solamente enfocaremos el análisis al primer término. Encontramos que un rango de valores de m para los cuales el algoritmo se mantiene dentro del modelo de semiflujo es cuando $m = \omega(1)$ y $m = o(n)$.

Lema 2. *$\frac{n}{m} = o(n)$, si $m = \omega(1)$ y $m = o(n)$.*

Demostración. Primeramente por hipótesis asumiremos que $m = \omega(1)$. Esto significa que $\forall c$, $\exists n_0$ tal que $\forall n \geq n_0$ se cumple que:

$$m \geq c \cdot 1 \tag{4.13}$$

Fijaremos n_0 para usarlo más adelante. Entonces por demostrar que $\forall c$, $\exists n_1$ tal que $\forall n \geq n_1$ se cumple que:

$$\frac{n}{m} \leq c \cdot n \tag{4.14}$$

Reescribiremos la constante como $c_0 = \frac{1}{c}$:

$$\frac{n}{m} \leq \frac{n}{c_0} \tag{4.15}$$

Ahora tomaremos $n_1 = n_0$ y hay que notar la siguiente observación. Dado que anteriormente se había constatado que cuando se tiene n_0 se cumple la desigualdad $m \geq c$, entonces al usar $n_1 = n_0$ la desigualdad:

$$\frac{n}{m} \leq \frac{n}{c_0} \tag{4.16}$$

Se sigue conservando ya que al tener $m \geq c_0$, entonces el lado izquierdo divide de igual manera o en mayores partes a n, en comparación al lado derecho. □

Por tanto si $m = \omega(1)$ y $m = o(n)$ el algoritmo está en el modelo de semiflujo

4.3. Trabajo relacionado

Existen antecedentes de trabajos que resuelven este problema en condiciones similares a las que impone el modelo de semiflujo. El trabajo más antiguo del que se tiene registro es el de Munro & Paterson en 1980 [16]. En este trabajo los autores presentan un algoritmo que con un valor de $m \leq \mathcal{O}((\log n)^2)$ puede resolver el problema con $\mathcal{O}((\log n)^3/m)$ pasadas, haciendo uso de un modelo esencialmente equivalente. En comparación al algoritmo presentado en este trabajo se tienen algunas observaciones interesantes. Primeramente la cantidad de pasadas que realiza nuestro algoritmo es $\mathcal{O}(\log n)$ la cual es esencialmente la misma cota que la presentada por Munro & Paterson (considerando que usan la máxima m posible). En cuanto a la cantidad de memoria si nuestro algoritmo hace uso de $m = \mathcal{O}((\log n)^2)$ consume espacio $\mathcal{O}(\frac{n}{(\log n)^2} + (\log n)^2)$, lo cual es evidentemente una cantidad superior a la usada por el algoritmo de estos autores, con lo que este aspecto favorece a su algoritmo aunque involucra una cantidad mayor de detalles que hay que tener en cuenta a la hora de utilizarlo.

Pero eso no es todo, en el mismo trabajo se muestra un algoritmo que resuelve el problema con p pasadas y espacio $\mathcal{O}(n^{1/p}(\log n)^{2-2/p})$. Como se puede observar, el detalle interesante en esta ocasión es que los autores parametrizaron el algoritmo en términos del número de pasadas, a diferencia del enfoque usado en este trabajo donde se parametrizó el algoritmo propuesto en términos de una cantidad de memoria.

Otro antecedente interesante es el trabajo publicado por Frederickson en 1987 [12]. En este trabajo se resuelve el problema con complejidad temporal $\mathcal{O}(n \log m + (n \log n)/\log m))$, una cantidad de memoria m y una cantidad de pasadas constante. Lo relevante de este trabajo no es sólo que mejoran las cotas de Munro & Paterson (y por consiguiente las nuestras), sino que imponen un estilo para resolver el problema de la K-ésima estadística bajo el modelo de semiflujo (o similares). En este trabajo los autores hacen uso de *filtros* en cada nivel para ir eliminando elementos y de alguna manera proponen una variación del algoritmo tradicional de Blum et al. [4]. Esta estrategia ha sido usada por distintos trabajos [8] [17] posteriormente para ir mejorando las cotas. Es interesante notar que en el presente trabajo usamos una técnica similar apoyados en el concepto de *predicados*.

De ahí en adelante la mayoría de los trabajos [15] [14] que han abordado este problema con restricciones similares a las del modelo de semiflujo se han enfocado en proponer algoritmos probabilísticos, cuyo análisis queda fuera del alcance de este trabajo.

Capítulo 5

La pareja de puntos más cercana y más lejana en semiflujo

En el presente capítulo propondremos una solución para el problema de la pareja de puntos más cercana y más lejana bajo el modelo de semiflujo.

5.1. Algoritmo

El modelo de semiflujo impone una restricción sobre la memoria usada por el algoritmo. Denotaremos como m a un parámetro de memoria, destacando el hecho de que en el contexto de un flujo de n elementos, m representa una determinada cantidad de esos elementos. Describimos el algoritmo a continuación:

1. Tomamos m puntos del flujo de n elementos.

2. Se construye un diagrama de Voronoi V (ver sección 2.1.3) con los m puntos como entrada. Estos m puntos serán los sitios del diagrama de Voronoi resultante.

3. Se calcula la distancia entre los sitios que son adyacentes en el diagrama V. Se registra como R la pareja con menor distancia vista.

4. Se resuelve el problema de localización de punto (ver sección 2.1.5) teniendo como entrada a V como subdivisión del plano y a cada uno de los puntos del conjunto $\{n-m\}$ como punto de consulta. Al realizar esta combinación de estructuras lo que estamos haciendo es que cada celda del diagrama de Voronoi se convierta en una región de búsqueda para el problema de localización de punto. Por tanto al resolver este último problema estaremos ubicando al punto en una determinada región del

diagrama. Por las propiedades del diagrama de Voronoi se sabe que si un punto q se ubica en una determinada celda entonces el punto más cercano a q es el sitio de la celda. Por lo anterior cada vez que se ubique a un punto en la celda de Voronoi correspondiente se calcula la distancia respecto al sitio y se actualiza R en caso de tener una pareja más cercana.

5. Se vuelve al primer paso eligiendo m elementos que no hayan sido seleccionados anteriormente o en el caso de la última iteración los elementos restantes si $n mod m \neq 0$.

6. Se regresa la pareja de puntos más cercana R.

Se puede observar el pseudocódigo del algoritmo completo en Algoritmo 5.

5.2. Análisis

El algoritmo trivial para resolver este problema calcula las $\binom{n}{2}$ parejas posibles y se queda con la mejor distancia vista. Una observación es que para cada punto q existe un punto r que de todo el conjunto es el punto más cercano a él. Si supieramos desde un principio cual es este punto r nos podríamos ahorrar todos los demás cálculos contra los otros puntos. Precisamente este es el principio bajo el cual opera nuestro algoritmo.

En el paso 2 se construye un diagrama de Voronoi con m puntos. En el paso 3 se calcula la distancia entre los sitios adyacentes del diagrama. Hacer esto es necesario por que los sitios adyacentes son las primeras parejas candidatas a ser la pareja de puntos más cercana. Solamente basta con verificar con los sitios adyacentes debido a que si no lo son significa que existe un sitio que se interpone en medio de los dos puntos y por tanto la distancia a este sería menor.

En el paso 4 se combinan las estructuras del diagrama de Voronoi y el problema de localización de punto. Como se explicó en la descripción del algoritmo, se resuelve el problema de localización de punto para cada uno de los puntos del conjunto $\{n - m\}$. Lo que se hace con esto, es por medio del diagrama de Voronoi encontrar para cada punto q cual es el punto r más cercano a él. Posteriormente sólo basta con calcular la distancia entre q y r y comparar con el registro de la mejor pareja vista. Si continuamos iterando el algoritmo de esta manera seleccionando m puntos diferentes en cada iteración, entonces después de $\frac{n}{m}$ iteraciones habremos terminado y necesariamente habríamos visto la pareja más cercana en algún momento.

Supongamos que el algoritmo no entregó la pareja R de puntos más cercana al finalizar su ejecución. Sean p_1 y p_2 los puntos de esta pareja. Dado que el algoritmo itera sobre

todos los puntos, entonces podemos reducir a 2 casos:

A) Los 2 puntos fueron seleccionados en el mismo lote m: En este caso dado que ambos puntos forman la pareja más cercana necesariamente son adyacentes en el diagrama de Voronoi. Por tanto, en el paso 3 donde se calcula la distancia entre los sitios adyacentes del diagrama estos puntos habrían sido procesados como la pareja más cercana.

B) Sólo un punto está en m: Sea p_1 sin pérdida de generalidad el punto que está en m. En el paso 2 se construye un diagrama de Voronoi en el cual p_1 será el sitio de una celda y posteriormente en el paso 4 se resuelve localización de punto para el conjunto de puntos $\{n - m\}$. En el momento en que esta iteración procese al punto p_2 para resolver el problema de localización de punto, se tiene que dado que el punto más cercano es p_1 entonces por las propiedades del diagrama de Voronoi la estructura necesariamente ubicará al punto p_2 en la celda que tiene como sitio al punto p_1. Posteriormente se calculará la distancia entre p_1 y p_2 y con ello estos puntos serán procesados como la pareja más cercana.

Con lo anterior se concluye que el algoritmo siempre devuelve la pareja de puntos más cercana y por lo tanto es correcto.

5.2.1. Complejidad temporal

Empecemos por analizar la complejidad temporal del algoritmo. El paso 1 consiste en sencillamente tomar una cantidad de puntos m. Se toman los puntos en el orden en que vayan apareciendo en el flujo. Este paso toma tiempo $\mathcal{O}(m)$.

En el paso 2 se construye un diagrama de Voronoi con m puntos como entrada. En la sección 2.1.3 se discutió un algoritmo para realizar esta tarea. Construir un diagrama de Voronoi de m puntos toma tiempo $\mathcal{O}(m \log m)$.

En el paso 3 se calcula la distancia entre los sitios adyacentes del diagrama de Voronoi. Dependiendo de la implementación de la estructura de datos que almacene al diagrama esta tarea puede variar su complejidad. Si al momento que se van construyendo las celdas del diagrama de Voronoi se cuida almacenar también referencias a las celdas adyacentes a cada una de estas, entonces este paso se reduce a recorrer los m sitios y realizar cálculos de tiempo constante, por lo tanto este paso puede realizarse en tiempo $\mathcal{O}(m)$.

En el paso 4 se resuelve el problema de localización de punto para cada punto del conjunto $\{n - m\}$. En la sección 2.1.5 se revisó un algoritmo para ello y se mostró que resolver una consulta toma tiempo $\mathcal{O}(\log m)$, con lo que este paso toma un tiempo total de $\mathcal{O}((n - m) \log m)$.

En el paso 5 se menciona que hay que iterar el algoritmo ahora seleccionando m puntos

que no hayan sido seleccionado antes y en caso de existir se vuelve al paso 1. Este paso toma tiempo $\mathcal{O}(m)$. Dado que se toman m puntos en cada iteración el algoritmo iterará un total de $\lceil \frac{n}{m} \rceil$ veces para poder procesar a todos los puntos del flujo.

En el paso 6 finalmente se regresa la pareja de puntos más cercana en tiempo $\mathcal{O}(1)$.

Entonces en total el algoritmo tiene como complejidad temporal:

$$\begin{aligned} Complejidad\,temporal &= \mathcal{O}\left(\frac{n}{m}\big(m + m\log m + m + (n-m)\log m + m\big)\right) \\ Complejidad\,temporal &= \mathcal{O}\left(\frac{n}{m}\big(n\log m + m + m\log m - m\log m\big)\right) \\ Complejidad\,temporal &= \mathcal{O}\left(\frac{n^2}{m}\log m + n\right) \\ Complejidad\,temporal &= \mathcal{O}\left(\frac{n^2}{m}\log m\right) \end{aligned} \tag{5.1}$$

En la transición del penúltimo al último paso dado que el primer término domina en complejidad al segundo podemos quedarnos solamente con el primer término en el resultado final. Si hacemos un análisis amortizado de esta complejidad respecto al número de elementos en el flujo tenemos:

$$\begin{aligned} Complejidad\,Temporal\ Amortizada\ Por\ Elemento &= \mathcal{O}\left(\frac{\frac{n^2}{m}\log m}{n}\right) \\ Complejidad\,Temporal\ Amortizada\ Por\ Elemento &= \mathcal{O}\left(\frac{n}{m}\log m\right) \end{aligned} \tag{5.2}$$

5.2.2. Complejidad espacial

Ahora es turno de analizar la complejidad espacial. En el paso 1 se toman m elementos del flujo, por lo que este paso toma espacio $\mathcal{O}(m)$.

En el paso 2 se construye un diagrama de Voronoi con m puntos. Construir este diagrama toma espacio $\mathcal{O}(m)$.

En el paso 3 se calcula la distancia entre los sitios adyacentes del diagrama de Voronoi. Dado que cada sitio tiene una referencia directa a sus sitios adyacentes y sólamente ocupamos realizar cálculos, entonces este paso puede realizarse en espacio $\mathcal{O}(1)$.

En el paso 4 se resuelve el problema de localización de punto. Los algoritmos más eficientes (ver sección 2.1.5) resuelven este problema para una subdivisión del plano con m vértices en espacio $\mathcal{O}(m)$. Por lo tanto asumiendo que se hace uso de alguno de estos algoritmos este paso toma espacio $\mathcal{O}(m)$.

En el paso 5 se verifica que aún queden elementos por procesar en el flujo y en caso de ser cierto se vuelve al paso 1. Dado que no existe una necesidad de acumular memoria de una iteración a otra (solamente se conserva el mejor valor visto de la pareja más cercana en memoria) entonces la memoria usada por cada paso puede ser reutilizada perfectamente. Este paso toma espacio $\mathcal{O}(1)$.

Finalmente en el paso 6 se devuelve un resultado por lo que este paso toma espacio $\mathcal{O}(1)$.

Por lo tanto la complejidad espacial total del algoritmo es la siguiente:

$$\begin{aligned} &Complejidad\,espacial = \mathcal{O}(m) + \mathcal{O}(m) + \mathcal{O}(1) + \mathcal{O}(m) + \mathcal{O}(1) + \mathcal{O}(1) \\ &Complejidad\,espacial = \mathcal{O}(m) \end{aligned} \tag{5.3}$$

5.2.3. Total de pasadas

Finalmente hay que revisar el número de pasadas que ocupa el algoritmo. Solamente el paso 1 y el paso 5 interactúan con el flujo directamente y en una sola ocasión para realizar su función, por tanto una iteración del algoritmo usa un número constante de pasadas, es decir $\mathcal{O}(1)$. Dado que en una de iteración del algoritmo se procesan m elementos y necesariamente se ocupan ver todos los elementos del flujo para encontrar la pareja más cercana, entonces es claro que con $\left\lceil \frac{n}{m} \right\rceil$ iteraciones basta para procesar a todos los puntos del flujo. Por lo tanto el número de pasadas es:

$$\begin{aligned} &Total\,de\,pasadas = \mathcal{O}(\frac{n}{m}) * \mathcal{O}(1) \\ &Total\,de\,pasadas = \mathcal{O}(\frac{n}{m}) \end{aligned} \tag{5.4}$$

Entonces resumiendo las propiedades del algoritmo se tiene:

$$\begin{aligned} &Complejidad\,espacial = \mathcal{O}(m) \\ &Total\,de\,pasadas = \mathcal{O}(\frac{n}{m}) \\ &Complejidad\,temporal = \mathcal{O}\left(\frac{n^2}{m}\log m\right) \\ &Complejidad\,temporal\,amortizada\,por\,elemento = \mathcal{O}\left(\frac{n}{m}\log m\right) \end{aligned} \tag{5.5}$$

Algorithm 5 ParejaDePuntosMasCercana(flujo, mejorParejaVista)

Require: flujo, mejorParejaVista

```
1: //PASO 1:
2: while aunHayElementosEnElFlujo() do
3:    var elemento = leerUnElementoDelFlujo(flujo);
4:    grupo.agregarElemento(elemento);
5:    if grupo.contarElementos() = M OR aunHayElementosEnElFlujo() = false then
6:       //PASO 2:
7:       var V = calcularDiagramaDeVoronoi(grupo)
8:       //PASO 3:
9:       var mejorParejaEntreSitios = calcularMejorParejaEntreSitios(V)
10:      var mejorDistanciaEntreSitios = mejorParejaEntreSitios.obtenerDistancia()
11:      if mejorDistanciaEntreSitios < mejorParejaVista.obtenerDistancia() then
12:         mejorParejaVista = mejorParejaEntreSitios
13:      end if
14:      //PASO 4:
15:      for punto in (flujo − grupo) do
16:         var region = calcularLocalizacionDePunto(punto)
17:         var sitio = obtenerSitioDeRegion(region)
18:         var distanciaAlSitio = obtenerDistanciaEntrePuntos(punto, sitio)
19:         if distanciaAlSitio < mejorParejaVista.obtenerDistancia() then
20:            mejorParejaVista = pareja(punto, sitio)
21:         end if
22:      end for
23:   end if
24: end while
25: return mejorParejaVista
```

5.2.4. Valores de m

Como se describió anteriormente el algoritmo propuesto hace uso de un parámetro de memoria m del cual dependen todas las complejidades mostradas. Una discusión interesante es determinar para qué valores de m el algoritmo se mantiene dentro del modelo de semiflujo.

Si recordamos las restricciones del modelo de semiflujo nos dicen que el algoritmo debe usar a lo más una cantidad sublineal de pasadas y de memoria respecto a n. Dado que la complejidad espacial del algoritmo es $\mathcal{O}(m)$, cumplir con esta restricción es trivial y solamente bastaría elegir un valor de $m = o(n)$. Con lo anterior solamente tenemos que verificar que el número de pasadas se mantenga dentro del modelo.

El número de pasadas del algoritmo es $\lceil \frac{n}{m} \rceil$ y se desea que se mantenga como una función sublineal respecto a n. m no puede tener un valor tan pequeño como una constante, por que sino claramente el número de pasadas se vuelve lineal, ni tampoco puede tener un valor tan grande como un número lineal respecto a n por la misma razón anterior.

Usando el lema 2 presentado en el capítulo anterior, se tiene que si $m = \omega(1)$ y $m = o(n)$ el algoritmo está en el modelo de semiflujo. Observemos un ejemplo para ver para qué valor de m se obtienen buenos valores. Propondremos 2 valores para m, uno de ellos $m = \log n$ que representa un valor más cercano al límite inferior del intervalo, y el otro valor será $m = \sqrt{n}$ que se encuentra un poco más centrado. Por supuesto ambos valores son claramente sublineales respecto a n.

Sea $n = 1,073,741,823$ y $m = \log n$, es decir $m = 29$, entonces la memoria es de igual manera $\log n = 29$, y el número de pasadas necesarias para el algoritmo son:

$$\begin{aligned} Cantidad\,de\,pasadas &= \frac{n}{m} \\ Cantidad\,de\,pasadas &= \frac{1,073,741,823}{29} = 37,025,580 \end{aligned} \tag{5.6}$$

Sea $n = 1,073,741,823$ y $m = \sqrt{n} = 32,768$ entonces la memoria usada es de igual manera $\sqrt{n} = 32,768$ y la cantidad de pasadas necesarias para el algoritmo son:

$$\begin{aligned} Cantidad\,de\,pasadas &= \frac{n}{m} \\ Cantidad\,de\,pasadas &= \frac{1,073,741,823}{32,768} = 32,768 \end{aligned} \tag{5.7}$$

Con lo anterior es claro que $m = \sqrt{n}$ resulta una mejor elección, aunque por supuesto en la práctica esto variará dependiendo del tamaño del flujo y el objetivo particular de la aplicación.

5.3. Trabajo relacionado

No se encontraron registros de que el problema de la pareja de puntos más cercana haya sido trabajado antes en el modelo de flujo de datos. Aunque el algoritmo propuesto está sustentado en estructuras geométricas que han sido muy estudiadas y pueden definirse perfectamente en más de 2 dimensiones, desafortunadamente al extender el número de dimensiones algunas complejidades de estas estructuras se elevan hasta hacerlas inutilizables. Particularmente se ha mostrado [18] que la complejidad espacial de construir un diagrama de Voronoi para ciertos conjuntos de puntos puede ser $\mathcal{O}(n^{d/2})$ donde d es la dimensión, con lo que se tiene una cantidad muy elevada de memoria y esta situación en el modelo de flujo no está permitida. En caso de que el conjunto de puntos tenga una estructura favorable, es decir, pocos puntos en su envolvente convexa, es posible que la complejidad espacial de las estructuras de las que hace uso el algoritmo no se eleven demasiado y por tanto el algoritmo podría seguir siendo aplicado.

5.4. La pareja de puntos más lejana en semiflujo

Para resolver el problema de la pareja de puntos más lejana se puede usar prácticamente el mismo algoritmo propuesto en la sección anterior. La modificación importante se tiene que hacer en el paso 2 del algoritmo donde se construye el diagrama de Voronoi. Como se presentó en la sección 2.1.4 el diagrama de Voronoi de puntos más lejanos es una estructura que divide al plano en celdas que tienen en común a un punto del conjunto como su punto más distante. Por tanto, reemplazando del algoritmo el diagrama de Voronoi por el diagrama de Voronoi de puntos más lejanos, y por supuesto invirtiendo las comparaciones, se obtiene un algoritmo que funciona para obtener la pareja de puntos más lejana en el modelo de semiflujo con exactamente las mismas complejidades.

Bajo este esquema podría pensarse que se tiene un algoritmo general para encontrar la K-ésima pareja, es decir si todas las parejas posibles estuvieran ordenadas, regresar la pareja en la K-ésima posición. Particularmente la pareja de puntos más cercana y la más lejana resultan ser las distancias extremas, y por tanto en este arreglo ordenado aparecerían en la primera y ultima posición respectivamente. Esta característica de ser distancias extremas es precisamente la que permite poder ir construyendo una solución parcial a medida que se van procesando los lotes de elementos. Si se tiene la solución en 2 lotes de elementos distintos sólo basta con comparar estas soluciones para ver cual es la mayor o menor (dependiendo de lo que se desee) y con ello se tendrá una solución a ambos lotes. En el caso de la K-ésima pareja no es posible combinar las soluciones entre lotes como se describió anteriormente por que hace falta información para tomar una decisión.

Por ejemplo, sea A la tercera pareja de puntos más cercana de un lote y B la de otro lote, si $A < B$ no se puede concluir la tercera pareja de puntos más cercana de ambos lotes, por que en este caso se descarta a B pero no se sabe como son los elementos que están debajo de B. Estos elementos pueden ser tan variados como sea posible y por tanto no permiten llegar a una conclusión respecto a A, con lo que se concluye que el algoritmo propuesto no funciona para solucionar el problema de la K-ésima pareja.

Capítulo 6

Conclusiones y trabajo futuro

Ante la presente era de la información no cabe duda que los modelos de flujo de datos cobrarán cada vez más importancia en el estudio de las ciencias de la computación. Los problemas con cantidades masivas de datos se vuelven cada vez más comunes y la necesidad de diseñar nuevos algoritmos crece. Con el presente trabajo mostramos soluciones a problemas fundamentales e ilustramos que algunos algoritmos tradicionales no tienen que ser replanteados totalmente, sino que cuidando detalles pueden ser adaptados al modelo de flujo de datos o equivalentes, como es el caso del algoritmo para calcular la K-ésima estadística.

La adaptación de este algoritmo que presentamos es la versión más fiel que pudimos desarrollar, entendiendo esto como el hecho de que prácticamente se convirtió paso por paso el algoritmo original de Blum et al. [4] a la versión mostrada en este trabajo. Es claro al revisar el estado del arte que nuestro algoritmo no ofrece las mejores complejidades, sin embargo a nuestro juicio si ofrece una de las versiones más simplificadas que existen en el modelo de semiflujo; aspecto que cobra relevancia a la hora de la implementación.

En cuanto al problema de la pareja de puntos más cercana y más lejana se propuso una solución elegante sustentada en estructuras geométricas y generalizable a dimensiones superiores. Durante el desarrollo de estos algoritmos pensamos como trabajo a futuro en la posibilidad de diseñar un algoritmo bajo las restricciones del modelo, que funcione no sólo para la pareja de puntos más cercana y más lejana, sino que también permita obtener la K-ésima pareja de puntos más cercana. No se encontró un camino claro que pudiera seguirse para resolver tal problema, ni se consiguió algún argumento contundente que mostrara que no es posible lograrlo, sin embargo el problema surgió de manera natural y parece ser bastante interesante como para dedicarle tiempo a su análisis en un trabajo posterior.

Por supuesto una manera natural de extender este trabajo es plantear algoritmos que solucionen más problemas fundamentales a los presentados en este documento. Básicamente

cualquier problema fundamental tiene potencial de tener su versión en el modelo de flujo de datos y seguro más de uno representa un desafío computacional interesante.

Bibliografía

[1] P. K. Agarwal, M. De Berg, J. Matousek, and O. Schwarzkopf. Constructing levels in arrangements and higher order voronoi diagrams. *SIAM journal on computing*, 27(3):654–667, 1998.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[3] B. C. Arnold, N. Balakrishnan, and H. N. Nagaraja. *A first course in order statistics*. SIAM, 2008.

[4] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of computer and system sciences*, 7(4):448–461, 1973.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms second edition*. The MIT Press, 2001.

[6] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[7] D. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM Journal on Computing*, 5(2):181–186, 1976.

[8] A. Elmasry, D. D. Juhl, J. Katajainen, and S. R. Satti. Selection from read-only memory with limited workspace. *Theoretical Computer Science*, 554:64–73, 2014.

[9] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[10] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[11] S. Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the second annual symposium on Computational geometry*, pages 313–322. ACM, 1986.

[12] G. N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *Journal of Computer and System Sciences*, 34(1):19–26, 1987.

[13] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information processing letters*, 1(4):132–133, 1972.

[14] S. Guha and A. McGregor. Approximate quantiles and the order of the stream. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 273–279. ACM, 2006.

[15] S. Guha and A. McGregor. Stream order and order statistics: Quantile estimation in random-order streams. *SIAM Journal on Computing*, 38(5):2044–2059, 2009.

[16] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[17] J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theoretical Computer Science*, 165(2):311–323, 1996.

[18] F. P. Preparata and M. Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 1985.

[19] M. I. Shamos. Computational geometry. 1978.

[20] S. S. Skiena. *The algorithm design manual*, volume 1. Springer Science & Business Media, 2008.

[21] I. M. Yaglom and V. Boltyanskii. Convex figures. 1961.