UNIVERSIDAD DEL BÍO-BÍO

FACULTAD DE CIENCIAS EMPRESARIALES
INGENIERÍA CIVIL EN INFORMÁTICA

Implementación en Java de algoritmos geométricos sobre grandes conjuntos de datos

Felipe José Lara Ramírez

Memoria para optar al título de Ingeniero Civil en Informática

Profesores Guía:
Gilberto Gutiérrez R.
María Antonieta Soto Ch.

Chillán, Octubre de 2014

Agradecimientos

Me gustaría agradecer a las personas que con su ayuda han colaborado en el desarrollo de este trabajo, al profesor Gilberto Gutiérrez, por la orientación, seguimiento y la supervisión del mismo, también por la motivación y el apoyo recibido a lo largo del desarrollo de este trabajo. También reconocer el apoyo dado por la profesora María Antonieta Soto Ch. en lo relacionado a la ayuda y el apoyo que me dío en el entendimiento y desarrollo del algoritmo junto a sus correcciones. Quisiera agradecer también a mis amigos de la carrera de Ingenieria Civil en Informática, por su amistad y colaboración durante todos los años que he estado en la Universidad. Agradecer a los profesores de la Universidad por su dedicación y ayuda al enseñar durante estos años. En especial agradecer a mis padres: Patricio y Soledad. Hermanos: Patricio, Rodrigo y Carolina. A toda mi familia, por ser mi soporte y darme ánimos en los momentos mas difíciles, darme su completo apoyo en todas mis decisiones y no permitirme rendirme nunca pese a cualquier situación. A Daniela, Reinaldo, Renato, Felipe, Fernando, Gustavo, Javier, René y David por su paciencia y el ánimo recibidos de ellos durante la mayor parte de mi vida.

A todos ellos, muchas gracias

Felipe José Lara Ramírez

Resumen

En este informe se describe la implementación en lenguaje JAVA del algoritmo *AREMAV* para resolver el problema de encontrar todos los rectángulos vacios maximales que existen en un conjunto de puntos S localizados en una región $R \subseteq R^2$. El algoritmo *AREMAV* se extendió para resolver una variante del problema que consiste en encontrar el rectángulo de mayor área en torno a un punto $q \notin S$. De acuerdo a la literatura, ambos problemas son de mucha utilidad práctica, en ámbitos como la minería de datos, sistemas de información geográfica, por nombrar algunos. En el informe se describen con mucho detalle ambos algoritmos y su implementación. Con el objeto de verificar la correctitud y eficiencia de los algoritmos se muestra una serie de experimentos considerando datos sintéticos. Los resultados de los experimentos fueron contrastados con resultados de experimentos similares reportados en el artículo original mostrando un comportamiento muy similar.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Introducción

La Geometría Computacional es un área de las matemáticas que se ocupa de estudiar y proponer soluciones algorítmicas a problemas geométricos. Es un área muy joven, pues sus primeros resultados se pueden apreciar en la década del 80. Sea S_1 y S_2 conjuntos de puntos ubicados en regiones $R_1 \subseteq R^d$ (típicamente $d = 2$) y $R_2 \subseteq R^d$ respectivamente, algunos de los problemas estudiados por la Geometría Computacional son: (i) encontrar la cerradura convexa de S_1, (ii) dado un punto $q \notin S_1$ y un parámetro $k > 0$, encontrar los k-puntos de S_1 más cercanos a q, (iii) dado un parámetro $k > 1$ encontrar los k pares de puntos (uno de S_1 y el otro de S_2) cuyas distancias sean las menores de entre todos los posibles pares que se puedan formar (iv) dado un punto $q \notin S_1$ encontrar el rectángulo vacío de mayor área incluido en R_1. La utilidad de contar con soluciones algorítmicas para estos problemas está muy bien establecida en la literatura.

Las soluciones a los problemas geométricos, desde la Geometría Computacional, suponen que es posible almacenar todos los objetos en la memoria de un computador. Sin embargo, con la aparición de grandes conjuntos de datos espaciales, se ha hecho necesario extender o crear soluciones que asuman que los datos se encuentran en estructuras de datos multidimensionales residentes en memoria secundaria (principalmente disco). En este contexto, las operaciones que predominan o determinan la eficiencia de un algoritmo corresponden a operaciones de entrada/salida o accesos a bloques de disco, las cuales son muy costosas en tiempo. Actualmente, existen soluciones para varios de los problemas indicados arriba. Por ejemplo, suponiendo que el

conjunto de puntos se encuentra almacenado en un R-tree, el cual es una estructura de datos tipo árbol similar al B-tree, el R-tree está diseñado para la indexación dinámica de un conjunto de objetos geométricos k-dimensional. [1], en [2] se propone un algoritmo que resuelve el problema (i), en [3] se describe un algortimo para el problema (ii), en [4, 5, 7] se aportan algoritmos para el problema (iii) y en [7, 8] se proponen soluciones para una variante del problema (iv).

En este informe se describe la implementación de un algortimo para una variante del problema (iv) propuesto en [6] y que llamaremos *MER* (Maximal Empty Rectangle). Informalmente, el problema *MER* consiste en dado un conjunto de puntos S localizados en una región $R \subseteq R^2$, encontrar todos los rectángulos vacios, maximales y que estén contenidos en R. El algoritmo considera el escenario en que los puntos se encuentran almacenados en un archivo sin formato (raw file). También se describe en este informe la implemantación de una variante *MER*, y que llamamos *qMER* (Query Maximal Empty Rectangle) y que consiste en dado un conjunto de puntos S localizados en una región $R \subseteq R^2$ y un punto $q \notin S$ encontrar el rectángulo maximal, de mayor área y que solo contine a q y que esté contenido en R. La utilidad de contar con soluciones para *MER* y *qMER* considerando grandes volúmenes de datos se pueden encontrar en [6] y [7, 8] respectivamente.

1.1.1. Objetivo General

El objetivo General de esta memoria de título es: "Implementar y Evaluar algoritmos geométricos para grandes conjuntos de datos espaciales".

1.1.2. Objetivos Específicos

- Implementar en lenguaje de programación JAVA algoritmo definido en [6] para el problema *MER* considerando grandes conjuntos de datos espaciales.
- Diseñar e implementar en lenguaje de programación JAVA algoritmo para resolver problema *qMER* considerando grandes volúmenes de datos espaciales inspirado en el algoritmo de [6].
- Evaluación experimental de ambos algortimos.

El resto de este informe está organizado de la siguiente forma. En el Capítulo 2, se describen formalmente los problemas *MER* y *qMER*. También

se realiza un revisión de la literatura (trabajos relacionados) describiendo los principales algoritmos disponibles para ambos problemas tanto desde el punto de vista de la Geometría Computacional, como desde las Bases de Datos Espaciales (grandes volúmenes de datos). El Capítulo 3 se ocupa de describir con mucho detalle el algoritmo propuesto [6] (que en adelante llamamos *AREMAV*). Se explican las ideas detrás del algoritmo, estructuras de datos utilizadas, ejemplos y aspectos de eficiencia. El Capítulo 4 muestra los detalles de la implementación (en lenguaje JAVA) del algoritmo *AREMAV* . También se expone el algoritmo *qAREMAV* para resolver el problema *qMER*. En el Capítulo 5 se exponen los resultados experimentales de los algoritmos *AREMAV* y $qAREMAV$. Finalmente en el Capítulo 6 se detallan las conclusiones y trabajos futuros.

Capítulo 2

Descripción del problema y trabajo relacionado

Tal como se comentó en los objetivos, este trabajo tiene como propósito principal implementar en JAVA el algoritmo *AREMAV* propuesto en [6] para el problema *MER* y el diseño de un algoritmo para *qMER* inspirado en AREMAV. En este capítulo definiremos de manera formal el problema *MER* y *qMER*, se presenta una serie de aplicaciones que sirven de motivación y se presentan trabajos relacionados.

2.1. Problemas *MER* y *qMER*

Dado un conjunto de puntos D ubicados en un subespacio $R \subseteq R^2$ definimos los siguientes problemas: (i) Encontrar todos los rectángulos vacíos maximales, de lados paralelos a los ejes que limitan a R y que se encuentran contenidos en R (*MER*)(ver Figura 2.1 (a), alli sólo se presenta uno de muchos rectángulos vacíos maximales) y (ii) dado un punto $q \notin D$ encontrar el rectángulo maximal de mayor área, de lados paralelos a R, que se encuentra contenido en R y que solo contiene a q (*qMER*), ver Figura 2.1 (b). Diremos que un rectángulo A es maximal si es vacío, contenido en R, con lados paralelos a los de R y no existe un rectángulo B, vacío, contenido en R de lados paralelos a R y que contenga a A.

Tal como como adelantamos, el objetivo principal de este trabajo es implementar en JAVA el algoritmo *AREMAV* propuesto en [6] para resolver el problema *MER*. El algoritmo supone que el conjunto D se encuentra en un archivo sin formato (raw file). Inspirados en el algoritmo *AREMAV*, un

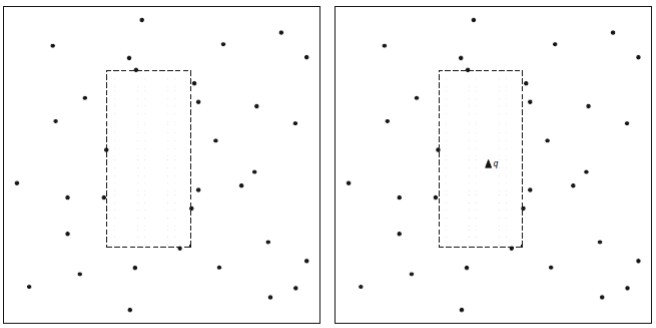


Figura 2.1: Problemas *MER* y *qMER*

segundo objetivo es proponer e implementar un algoritmo para el problema *qMER*.

2.2. Motivación

Supongamos que contamos con una lámina de acero en la cual existen pequeñas regiones con imperfecciones o fallas y que estamos interesados en obtener regiones de la lámina libre de fallas. Es claro que este problema se puede modelar como un problema *MER*. Otros ejemplos de aplicaciones los podemos encontrar en el contexto de los Sistemas de Información Geográfica (SIG). Por ejemplo, si se quiere construir un parque en una región y se tienen los hitos (edificios, casas, postes de alumbrado público, etc.) georeferenciados. Las siguientes consultas pueden ser interesante resolverlas eficientemente: (1)¿Cuál es la zona vacía de mayor área donde construir el parque? o (2) Encontrar el mayor espacio libre (con forma de rectángulo) en torno a un punto donde se desea construir el parque. Notar que el problema (1) se puede modelar como un problema *MER* y el problema (2) como *qMER*. Otra aplicación del problema *qMER* puede ser la siguiente. Supongamos que tenemos varios lugares donde podemos instalar una fábrica la cual genera cierta toxidad (por ejemplo, gases, ruido , etc.). Si contamos con información georreferenciada de poblaciones humanas, cultivos u otro elemento que no queremos contaminar con nuestra fábrica, nosotros podríamos elegir el punto adecuado donde instalarla.

En [6] se propone otro tipo de aplicaciones para *MER* dentro del contex-

to de la minería de datos. Por ejemplo, supongamos que una base de datos almacena las cantidades y las fechas de depósitos bancarios. Se considera un gráfico en donde se tiene el tiempo en el eje x y el monto en el eje y. Un espacio vacío, indicará que durante un período determinado de tiempo no hubo depósitos dentro de un cierto rango de cantidades. Por ejemplo, si nos encontramos con que durante los años 2007 y 2008, no existían depósitos de más de un millón de dólares, esto podría ser un síntoma de una nueva crisis económica que está surgiendo. Este escenario puede ser tratado como un problema *qMER* , si el punto de consulta q podría ser definido por un punto en el tiempo y una mínima cantidad de depósito. Otro ejemplo podría ser en una base de datos de un Sistema del Hospital. Teniendo en cuenta los datos sobre las operaciones de cirugía, es posible descubrir que no hay información sobre trasplantes de cara en la base de datos antes del año 2008. Este conocimiento indica que ese procedimiento no era posible antes de ese año, y puede ser introducido como una restricción de integridad de la base de datos, con el fin de realizar optimizaciones de consultas semánticas.

Actualmente, estos problemas han cobrado interés en el ámbito de las Bases de Datos Espaciales, donde se considera que los objetos residen en memoria secundaria y que tienen en cuenta las limitaciones de la memoria principal.

2.3. Trabajos relacionado

Inicialmente el problema *MER* fue establecido desde la geometría computacional, suponiendo que todos los puntos caben en la memoria principal. En este escenario el problema *MER* ha sido estudiado extensivamente. El primer trabajo conocido fue de Naamad et al.[9], donde se describen dos algoritmos los cuales consideran que los puntos están ubicados de forma aleatoria dentro del espacio, el primer algoritmo es de tiempo $O(n^2)$ y $O(n)$ de almacenamiento, el segundo es de tiempo $O(\log^2 n)$ y $O(n)$ de almacenamiento. Luego Chazelle et al.[11] presentan un algoritmo estilo divide y vencerás con un tiempo de $O(nlog^3 n)$ utilizando un almacenamiento de $O(n \log n)$. Un algoritmo con similar complejidad se discute en [12], este utiliza un orden de $O(n)$. Orlowski et al [10] presentan un algoritmo que utiliza un tiempo $O(s \log n)$, donde s es el número de rectángulos vacíos maximales. Además, el algoritmo tiene una complejidad de $O(n \log n)$. Otro trabajo más reciente (Nandy et al.) [13] el algoritmo utiliza un tiempo de $O(n \log^2 n + s)$ y

un espacio de almacenamiento de $O(\log n)$ utilizando un árbol de búsqueda prioritario (priority searchtree).

También hay trabajos enfocados a buscar los máximos espacios vacios en 3 dimensiones, en este caso el algoritmo computa cubos vacíos maximales [14],[15].

En [16] y [17] se propone un algoritmo para resolver el problema *qMER*. Este algoritmo realiza un preprocesamiento de los datos, donde el espacio se divide en un conjunto de celdas de tal manera que todos los puntos que caen en la misma celda producen el mismo rectángulo vacío maximal que contiene al punto de consulta q. Estas celdas se almacenan en memoria principal, organizadas en una estructura de datos de dos dimensiones llamada árbol de rango (range tree). La etapa de preprocesamiento utiliza un almacenamiento de $O(n^2 \log n)$ y un tiempo de $O(n^2)$. Para extraer la respuesta se necesita un tiempo adicional de $O(\log n)$. Otra aproximación es la presentada por Kaplan et al. [18], esta corresponde a una mejora significante en términos de tiempo de pre procesamiento respecto a [16] y [17]. Específicamente, este algoritmo requiere un espacio de almacenamiento de $O(n\alpha(n) \log^3 n)$ esto para mantener la estructura de datos que se utiliza (SegmentTree) y un tiempo de $O(n\alpha(n) \log^4 n)$ para construir la estructura, donde el termino de $\alpha(n)$ es la inversa de la función de Ackermann. Estos algoritmos consideran que todos los objetos se encuentran en memoria principal. Más recientemente en [7] y [8] se proponen algoritmos que consideran limitaciones de memoria principal y asumen que los objetos residen en memoria secundaria en una estructura de datos multidimensional R-tree.

Sin embargo, existen muchos escenarios en donde los objetos considerados no caben en memoria principal y se encuentran almacenados en un archivo raw files. Edmonds et al.[6] proponen un algoritmo para obtener los rectángulos vacíos maximales en un área compuesta por grandes conjuntos de datos, este algoritmo requiere de un tiempo $O(|X| \times |Y|)$ y un almacenamiento de $O(|X|)$, con $|X|$ siendo el número de valores distintos encontrados en el eje x e $|Y|$ todos los valores distintos que se encuentren en el eje y.

Dado que en este trabajo el objetivo es solucionar los problemas *MER* y *qMER*, en el siguiente capítulo se explicará el funcionamiento del algoritmo *AREMAV* para luego adaptarlo a un algoritmo que resuelva $qMER$.

Capítulo 3

Algoritmo *AREMAV*

Tal como anticipamos en el capítulo anterior, el algoritmo *AREMAV* sirve para resolver el problema *MER*. En este capítulo se describe de manera muy detallada el algoritmo *AREMAV* propuesto en Edmons et al.[6]. Este algoritmo requiere que los datos entren en un orden específico, lo que permite contar con un algoritmo con tiempo de ejecución $O(|X| \times |Y|)$ (lineal con respecto a los datos de entrada para los $x \leq y$). Además, los requerimientos de memoria están en $O(|X|)$, el cual es un orden de magnitud menor que el tamaño $O(|X| \times |Y|)$ para todos los datos de entrada.

3.1. Precondiciones del algoritmo

El algoritmo *AREMAV* procesa los datos a partir de un archivo sin formato (raw file), el cual contiene las coordenadas ordenadas de mayor a menor por la coordenada y y luego (por cada y) por la coordenada x de mayor a menor, como se ve en la Figura 3.1. *AREMAV* requiere de una serie de pasos previos para que este pueda ser ejecutado, al igual que posee limitaciones a la hora de generar los rectángulos maximales vacíos. Los pasos previos y limitaciones son:

1. Un archivo raw files que contenga todos los puntos que se analizarán de forma ordenada como se explicó anteriormente (ordenar de forma descendente las y y dentro de ella ordenar por las x de forma descendente).

2. El programa debe leer primero la y, y luego la x.

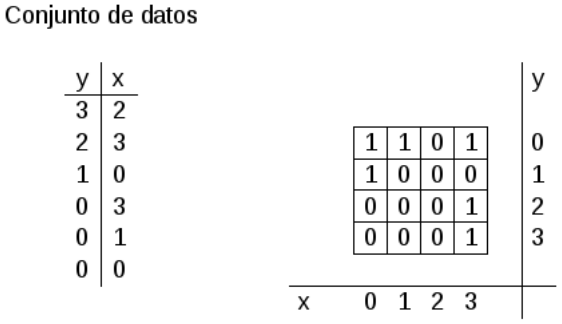


Figura 3.1: Conjunto de datos y su representación como matriz

3. Se debe saber cuáles son los límites del subespacio $R \subseteq R^2$ donde están contenidos los puntos.

4. Se debe conocer la cantidad de valores distintos de la coordenada de X que hay en el archivo y se asume que estas pueden ser almacenadas en memoria principal.

3.2. Descripción AREMAV

Para describir el algoritmo se hará de la misma forma en que se hace en el artículo [6]. El algoritmo $AREMAV$ se puede explicar más fácilmente diseñando una matriz M, la cual es una representación del conjunto de datos D almacenados en el archivo (ver Figura 3.1). Esta matriz nunca se construye en la implementación del algoritmo, solo se muestra para hacer más fácil la comprensión del mismo. De esta manera, el conjunto de datos puede ser transformado en una matriz donde solo contenga los puntos del conjunto de datos, mostrando con un 1 donde exista el punto y con un 0 donde no lo haya, así lo muestra la Figura 3.1.

El algoritmo recorre solo una vez el conjunto de datos y encuentra todos los rectángulos maximales dentro de los cuales solo hallan 0 (los 1 solo servirán para limitar la forma del rectángulo). Para que esto sea posible, solo se llevarán a memoria principal dos filas consecutivas de la matriz a la vez. Ambas filas se analizarán celda por celda de forma simultánea (la fila superior de la matriz será conocida como “primera fila” y la fila siguiente será conocida como “segunda fila”) como se puede ver en la Figura 3.2. El llevar a memoria principal ambas filas sirve para que durante el análisis de

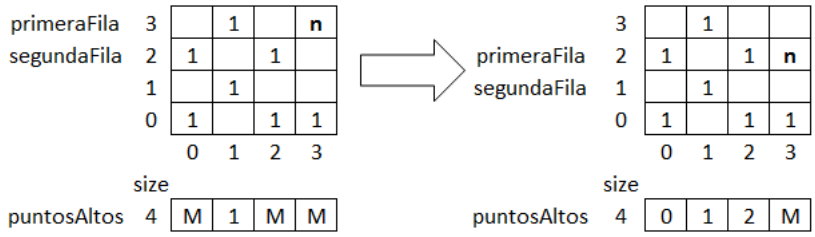


Figura 3.2: Funcionamiento de variables primera fila, segunda fila y actualización de los puntos altos.

la primera fila se encuentren los límites superiores de los rectángulos maximales vacíos, mientras que el análisis de la segunda fila servirá para detectar cuál es el límite inferior de los rectángulos maximales vacíos que puedan ser generados. Cuando se hayan recorrido todas las celdas de las filas, se descarta la primera fila y es reemplazada por la segunda fila. Por último se deberá traer desde memoria secundaria la fila inmediatamente inferior, la que se convertirá en la nueva segunda fila (ver Figura 3.2).

Para conocer cúales serán los límites superiores que tendrán los rectángulos maximales vacíos, se deberá crear, para cada columna, una variable $y_r(x, y)$, la cual es la coordenada del 0 más alto de la columna x que comienza en (x, y) y se extiende hacia arriba. (ver Figura 3.2)

En la Figura 3.2, se puede ver cómo cambian las variables de primera fila y segunda fila a medida que se ha ido recorriendo la matriz. La letra n representa la celda que se está analizando. La variable de puntos altos se actualiza a medida que se recorre la matriz como se ha explicado anteriormente, donde M es un valor muy grande, el cual indica que el límite superior está en el tope del espacio para esa columna.

Ahora que se tienen los límites inferiores y superiores para la generación de los rectángulos maximales, solo falta obtener el límite izquierdo y derecho, para esto se utiliza una estructura de *escalera* la cual se implementa mediante la estructura de datos Stack (pila). La escalera es solo una parte de una *escalera maximal*, la cual es la *escalera* más grande que se podrá formar en esa fila, ya que se extenderá lo más a la derecha posible y lo más alto posible (ver Figura 3.3), ella contendrá uno o muchos rectángulos maximales vacíos como se podrá ver más adelante cuando se explique cómo se

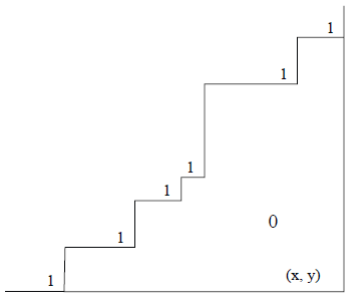


Figura 3.3: *Escalera maximal* para (x, y)

obtienen estos rectángulos a partir de la *escalera maximal*.

La *escalera* se iniciará vacía cada vez que se comience a utilizar una nueva fila. El comportamiento de la escalera irá variando con cada paso que se avance en la matriz, pudiendo ocurrir 2 casos cuando se analiza cada celda:

1. **La celda contiene un 1**: si esto sucede la pila se vaciará, ya que $y_r(x, y)$ no está bien definido pues el valor que contenía ya no es válido, el último cero ahora está bajo (x, y), si hay alguno, y si lo hay no se sabe la posición.

2. **La celda contiene un 0**: si esto sucede, se deberá a proceder con la etapa siguiente para ver si se empilará el dato perteneciente a $y_r(x, y)$ o no podrá ser empilado.

   Cuando el dato contenido en la celda de análisis es 0, pueden ocurrir 3 casos: (i) que la escalera aumente un peldaño (aumente su altura, como se ve en la Figura 3.4.a), (ii)que la escalera aumente el tamaño del peldaño actual (aumente en longitud, como se ve en la Figura 3.4.b) o (iii) que la escalera disminuya sus peldaños (reduzca su altura como se ve en la Figura 3.4.c).

   En cambio si el dato contenido en la celda de análisis es un 1, no se podrá construir la escalera por lo que esta deberá ser eliminada, si el dato de la celda siguiente es un 0, la escalera se comenzará a construir de nuevo.

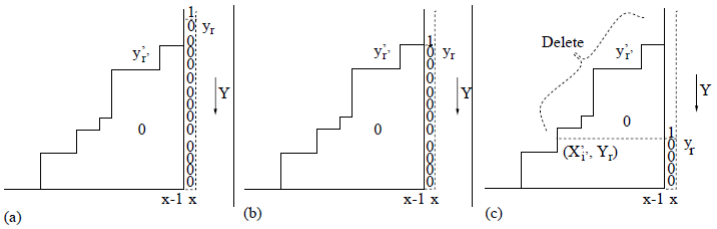


Figura 3.4: Los tres diferentes casos cuando se construye una escalera desde su paso anterior

3.2.1. Construyendo la escalera

Se debe tener presente que se está trabajando solo con 2 filas de la matriz, analizando solo una fila, celda por celda, y por cada celda la columna (x) se analizan las coordenadas de y_r. La forma en la que se construye la escalera es tomar la altura del punto que se está analizando y_r y el punto más alto anterior ($y'_{r'}$) que es la posición del último y más alto peldaño de la escalera. Se comparan estas dos alturas resultando 3 casos (los que se ven en la Figura 3.4).

- Caso 1) $y_r < y'_{r'}$: (Figura 3.4(a)). Si el nuevo peldaño superior es más alto que el anterior más alto, entonces, se procederá a empilar este punto, quedando como punto más alto del stack (la *escalera* crece en altura). Luego se sigue analizando la fila.
- Caso 2) $y_r = y'_{r'}$: (Figura 3.4(b)). Si el nuevo peldaño superior tiene la misma altura que el anterior más alto, se procederá a ampliar el peldaño hacia la derecha (aumenta su longitud), en este caso no se empilará nada, ya que el vertice superior izquierdo del peldaño seguirá siendo el mismo.
- Caso 3) $y_r > y'_{r'}$: (Figura 3.4(c)). Si el nuevo peldaño superior es más bajo que el anterior más alto, en este caso se desempilará (eliminarán peldaños de la escalera) tantas veces como sea necesario hasta que la coordenada $y'_{r'}$ del tope del stack cumpla que: $y_r \leq$y'$_{r'}$ (reducción de altura), de esta forma, dependiendo el valor que tome $y'_{r'}$, se aplicará uno de los casos antes vistos.

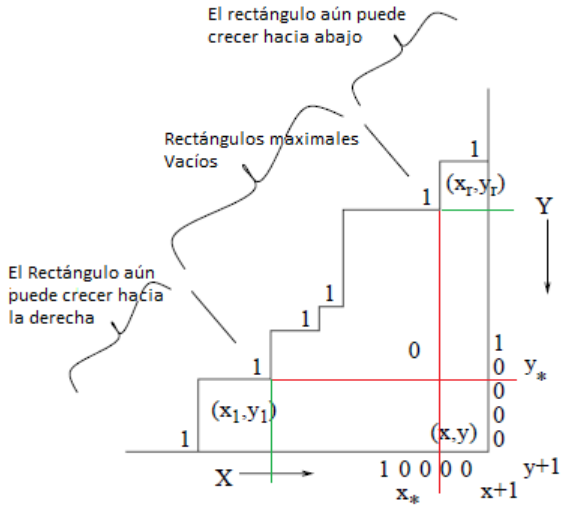


Figura 3.5: Obteniendo los rectángulos maximales vacíos desde una escalera

3.2.2. Mostrar rectángulos maximales vacíos

Ya que se tiene la escalera formada, solo falta cerrar el o los rectángulos maximales vacíos por debajo. Para determinar si esto es posible se debe revisar la fila siguiente a la que se está analizando (razón por la cual se llevó a memoria principal, como se mencionó anteriormente). Se debe considerar dos casos, (*i*) donde un rectángulo limitará por debajo con un 1, o (*ii*) donde el rectángulo no limita por debajo con un 1, por lo cual podrá seguir creciendo hacia abajo.

Los rectángulos que se formen como se dijo anteriormente, deben contener solamente 0 en su interior, y fuera de sus límites debe haber por lo menos un 1 en cada lado o estar en el límite del conjunto de datos. El rectángulo maximal vacío que se generará, tendrá su vértice inferior derecho en el lugar de la matriz que se está analizando y su vértice superior izquierdo, así sucesivamente hasta dejar la pila vacía.

Con esto se dice que cada peldaño es un extremo de un rectángulo maximal vacío, siempre y cuando se cumpla la siguiente condición:

- Para saber si un rectángulo es maximal, se deben utilizar dos valores llamados x_* e y_*. Donde x_* es la posición del 0 ubicado en el extremo

```
loop y=1...n
 loop x=1...m
        (I) Construir escalera(x,y)
        (II) Mostrar todos los rectangulos maximos vacios
```

Figura 3.6: Algoritmo AREMAV visto de forma simple

izquierdo de la sucesión de ceros que se inicia en $(x, y+1)$ y es antecedido por un 1, y_* es el 0 más alto que se encuentra en la columna $(x+1, y)$ y este debe ser precedido por un 1.(Figura 3.5).

- Se debe considerar además un peldaño de la escalera (x, y) cuya esquina superior izquierda es (x_i, y_i), el rectángulo (x_i, x, y_i, y), que tiene sus lados superior e inferior (paralelos al eje x) extendíendose entre x_i y x, y sus lados izquierdo y derecho (paralelos al eje y) entre y_i e y, es maximal si y solo si $x_i < x_*$ e $y_i < y_*$

- Si $x_i \geq x$, implica que el rectángulo es bastante angosto como para extenderse hacia abajo hasta el bloque de ceros de la fila $(y+1)$. Por ejemplo, el peldaño más alto en la Figura 3.5 corresponde al caso indicado. Por otra parte, si $x_i < x_*$, entonces el rectángulo que se forma con (x, y) no se puede extender hacia abajo porque es bloqueado por el 1 ubicado inmediatamente a la izquierda de x_*, tal es el caso del primer peldaño de la escalera (el más bajo) en la Figura 3.5. De manera similar, si $y_i \geq y_*$ entonces el rectángulo es suficientemente bajo como para extenderse hacia la derecha hasta el bloque de ceros de la columna $(x+1)$. Esto se puede observar en el primer peldaño de la escalera en la Figura 3.5.

Una vez finalizada la obtención de los rectángulos maximales vacíos para (x, y), se debe continuar con el análisis de la celda a la derecha de esta en la misma fila, esto es, $(x+1, y)$.

La Figura 3.6, muestra el algoritmo que se acaba de explicar, escrito de forma muy resumida.

Capítulo 4

Implementación

El algoritmo *AREMAV* se utilizó como base para la definición e implementación del algoritmo *qAREMAV* el cual se describirá a continuación. Este algortimo se implementó en el lenguaje de programación Java utilizando tres clases.

- **Clase** *qAREMAV*: es la clase principal del programa, en esta clase se encuentra toda la lógica y el código que se encarga de hacer una llamada a la clase ExternalMergeSort para que ordene el conjunto de datos de entrada, de forma que la clase *qAREMAV* pueda leerlos y analizarlos.
- **Clase ExternalMergeSort**: es la clase encargada de tomar el raw file y ordenarlo de forma que la clase *qAREMAV* pueda leerlo sin problemas como se ha explicado anteriormente.
- **Clase Punto**: es la clase que contiene la definición de un punto e implementa las operaciones sobre este, permitiendo procesar el conjunto de datos de un modo mas fácil.

4.1. Antecedentes

4.1.1. Diferencias entre algoritmos

Si bien el algoritmo *AREMAV* de Edmonds et al.[6] obtiene todos los rectángulos maximales vacíos de un conjunto de puntos, no resuelve el problema específico que se aborda en este trabajo. Por lo tanto, se precisa modificarlo para obtener el rectángulo maximal vacío de mayor área que contenga el punto q.

```
    Rectangulo qAREMAV(File f, Punto q)
1   begin
2       // Suponemos que f contiene puntos, cada uno almacenado como y, x, id
3       // Suponemos además que q no se encuentra almacenado en f
4       ordF ← externalMerge(f)
5       fila1 ← leeFila(ordF) // Una fila es un arreglo de puntos con igual y, ordenados en forma ascendente según x
6       fila2 ← leeFila(ordF) // Suponemos que es posible leer una segunda fila
7       ptsAltos ← fila1
8       yBordeSup ← fila1[0].getY()
9       escalera ← new Stack();
10      rqMayArea ← null   // rqMayArea es Rectangulo, y se retornará como el rectángulo maximal de mayor área
                           // que contiene a q
11      while fila2 <> null and  do
12          ptoBajo ← null
13          for i=0, i<fila1.size()-1, i++ do   // Analiza hasta el penúltimo elemento o columna de fila1
14              p(x,y) ← fila1[i]  // p(x,y) representa la posición x,y que puede contener un punto, si es así p(x,y) es Punto
15              if p(x,y).esPunto() && i>0 then  // Se actualiza ptsAltos de x si no es el borde Izq. del conjunto de puntos
16                  ptsAltos[i] ← p(x,y)
17              end If
18              ptoAlto ← ptsAltos[i]       // El punto entre yBordeSup e y de la columna x, nuevo peldaño de la escalera
19              generaEscalera(escalera, ptoAlto, yBordeSup)
20              if fila2[i].esPunto() then   // fila2[i] contiene elemento ubicado en (x,y-1), puede ser Punto
21                  ptoBajo ← fila2[i]
22              end If
23              ptoAltoDer ← ptsAltos[i+1]      //  Recupera punto ubicado en columna a la derecha entre yBordeSup e y
24              esqInfDer ← fila2[i+1]     // Recupera la posición que está abajo y a la derecha de p(x,y) como Punto
25              rqMax ← obtieneRectMayArea(escalera, esqInfDer, ptoBajo, ptoAltoDer, q)
26              if rqMax<>null and rqMayArea<>null and rqMay.area()>rqMayArea.area() or rqMayArea=null then
27                  rqMayArea = rqMax
28              end if
29          end for
30          escalera.clear()
31          fila1 ← fila2
32          fila2 ← leeFila(ordF)
33      end while
34      return rqMayArea
35  end
```

Figura 4.1: Código que representa al algoritmo *qAREMAV*

```
    generaEscalera(Stack escalera, Punto ptoAlto, double yBordeSup)
1   begin
2       if escalera.isEmpty() then
3           escalera.push(ptoAlto)
4       else
5           if escalera.top().getY() < ptoAlto.getY() then
6               // Caso 1: escalera crecerá hacia arriba un peldaño
7               escalera.push(ptoAlto)
8           else
9               if escalera.top().getY() = ptoAlto.getY() then
10                  // Caso 2: Aumenta longitud del último peldaño de la escalera, la que no se modifica
11              else
12                  if escalera.top().getY() > ptoAlto.getY() then
13                      // Caso 3: escalera elimina peldaños
14                      while ¬escalera.empty() and escalera.top().getY() >= ptoAlto.getY() do
15                          ultPto ← escalera.pop();
16                      end while
17                      // Caso 1, pero ultPto y ptoAlto no tienen x adyacente, no son adyacentes en la matriz
18                      escalera.push(new Punto(ultPto.getX(), ptoAlto.getY()))
19                      escalera.push(new Punto(ptoAlto.getX(),yBordeSup))
20                  end if
21              end if
22          end if
23      end if
24  end
```

Figura 4.2: Método del algoritmo que genera la escalera

```
    obtieneRectMayArea(Stack escalera, Punto esqInfDer, Punto ptoBajo, Punto ptoAltoDer, Punto q)
1   if estaEnBordeInferiorDererechoDelConjPts(esqInfDer) then
2       rectMax ← obtieneRectMayAreaBordeInfDer(escalera, esqInfDer, q)
3   else
4       if estaEnBordeDerechoDelConjPts (esqInfDer) and ptoBajo<>null then
5           rectMax ← obtieneRectMayAreaBordeDer(escalera, esqInfDer, ptoBajo, q)
6       else
7           if estaEnBordeInferiorDelConjPts (esqInfDer) then
8               rectMax ← obtieneRectMayAreaBordeInf(escalera, esqInfDer, ptoAltoDer, q)
9           else
10              if ptoBajo<>null then
11                  rectMax ← obtieneRectMayAreaInterior(escalera, esqInfDer, ptoAltoDer, ptoBajo, q)
12              end if
13          end if
14      end if
15  end if
```

Figura 4.3: Método del algoritmo que genera los rectángulos

Otro aspecto a considerar es que los lados de los rectángulos que se desean generar deben contener al menos un punto, en [6] se obtienen rectángulos que no incluyen puntos.

4.1.2. Estructuras de datos a utilizar

Para desarrollar el algoritmo, se implementó el algoritmo expuesto en [6], salvo ciertas modificaciones, las cuales se exponen detalladamente más adelante, este algoritmo se llamará *qAREMAV* de aquí en adelante. Dado lo anterior se hizo uso de las estructuras de datos señaladas en [6].

- **Arreglos unidimensionales**: Estos arreglos, son listas que poseen un indice y un valor determinado. En este caso sirven para contener mayormente los puntos más altos por los que se han pasado y para almacenar los valores de todas las x.

- **Stack**: Un stack o pila es una estructura de datos del tipo de lista ordenada que utiliza un modo de acceso a los datos del tipo LIFO (último en entrar, primero en salir). En este caso la pila servirá principalmente para construir la escalera y generar los rectángulos maximales vacíos. De esta estructura de datos se utilizarán las siguientes operaciones:

  - **push()**: Sirve para empilar un dato y dejarlo en la parte superior de la pila

  - **pop()**: Sirve para desempilar un dato del stack y sacarlo, dejando el dato siguiente en el tope

  - **peek()**: Operación que sirve para ver que hay en el tope del stack.

  - **empty()**: Operación nos dice si la pila esta vacía o no.

  - **size()**: Operación que sirve para saber el tamaño del stack.

4.2. Implementación del algoritmo

Para comenzar la implementación del algoritmo *qAREMAV* se debieron definir ciertas condiciones que se deben dar para que el algoritmo funcione como se ha planteado en este proyecto. Estas condiciones son las mismas planteadas en el capítulo anterior, salvo que en esta implementación los rectángulos maximales contendrán espacios vacíos, pero en cada uno de los bordes que los definen exisitirá al menos un punto 1, excepto si se trata de

un borde que coincida con los del conjunto de datos.

Otro de los cambios realizados, fue en cuanto a la lectura de los datos, ya que en esta implementación se hace con y decreciente y con x creciente, al contrario de lo que se plantea en [6].

El algoritmo se separa en dos etapas, en la primera se preparan los datos para que sean analizados, se buscan los límites del conjunto de datos, y se ordenan los datos de entrada (Figura 4.1 líneas desde 1 hasta 10). La segunda etapa es donde se buscan los rectángulos maximales vacíos que contienen el punto q, utilizando lo realizado en la etapa anterior (Figura 4.1 líneas desde 11 hasta 34). En la secciones que siguen se detallan cada una de estas etapas.

4.2.1. Preparación de los datos

La primera etapa de preparación de datos contempla varios pasos que se describen a continuación.

El primer paso consiste en ordenar los datos almacenados en el raw file, dado que estos no siempre se encuentran ordenados del modo requerido por el algoritmo. Cada línea del archivo almacena un punto cuyo formato es (y, x, id), pudiendo contener una cantidad tal de puntos que impida almacenarlos todos en memoria principal, por lo que se debe aplicar un algoritmo de ordenamiento que opere sobre memoria secundaria tal como MergeSort (Figura 4.1 línea 4). En la API de Java existe una clase llamada $ExternalMergeSort$ que implementa dicho algoritmo de ordenamiento, el cual tiene complejidad $O(n \log n)$.

El siguiente paso tiene como finalidad conocer la cantidad de valores distintos de x que existen en el conjunto de datos, por lo que este se recorre y se almacenan todas las x en una HashTable (se sabe que todas las x caben en memoria principal). La HashTable solo almacena valores únicos por lo que no habrán x repetidas. A continuación la HashTable se transformará en un arreglo unidimensional. Este será ordenado utilizando el método Sort el cual devuelve el arreglo de las x ordenadas de menor a mayor, este arreglo se llamará *distintasX* . Por lo que obteniendo el primer y último dato del arreglo se tendán los límites izquierdo y derecho del conjunto de datos (Esto aparece más detallado en Apendice A).

Al mismo tiempo que se recorre todo el conjunto de datos para obtener

las x también se obtendrán las y, de las y se quiere conocer el valor mayor y menor. El valor mayor sirve para poder conocer cuál es el tope superior del conjunto de datos, mientras que el valor menor, sirve para conocer el tope inferior del conjunto de datos. Para obtener el mayor valor de y solo basta con obtener el primer dato desde el conjunto de datos ya ordenado, mientras que para obtener el y menor, es necesario haber recorrido todo el conjunto de datos hasta llegar a la última fila (Figura 4.1 línea 8) .

Luego se deberán crear los arreglos *fila1* y *fila2* (Figura 4.1, líneas 5 y 6), estos se crearán y llenarán de la siguiente forma:

- Se deben inicializar con tamaño igual al arreglo *distintasX* que se creó en el segundo paso. Estos arreglos deberán ser llenados con las combinaciones de los y obtenidos del conjunto de datos y con las x almacenadas en el arreglo *distintasX*. Para hacer la combinación se leerá el archivo que contiene el conjunto de datos ordenado.

- Luego se obtiene el dato que será el punto inicial de la fila que se este leyendo. Cuando se tenga el primer dato se debe ver cúal es el valor de su x e y.

- Teniendo este dato, se recorre el arreglo *distintasX* y se comparan las x, donde las x no coincidan se agregará un elemento tipo Punto con los datos $(x, y, false)$ en el arreglo *fila1*, esto nos dice que se crea un punto pero no es un 1, sino un 0 , esto es, no existe un punto para esa posición en el conjunto de datos. A dicho punto se le llamará punto falso $(x, y, false)$.

- En cambio si el valor de x coincide con su posición en el arreglo de las X, se creará en el arreglo *fila1* un elemento Punto con datos $(x, y, true)$, esto quiere decir que aquí hay un 1.

- Después de poner el 1 correspondiente al primer elemento de la fila, se analiza el siguiente punto del conjunto de datos. Si aquí el valor de y cambia, se está en presencia de un cambio de fila, por lo que se vuelve a hacer lo mismo que antes, solo que esta vez se rellenará el arreglo *fila2* que representa la segunda fila de la matriz.

En el tercer paso, cada celda del arreglo de los puntos más altos, llamado *ptsAltos* se iniciará con los puntos del arreglo de *fila1*(Figura4.1 línea 7).

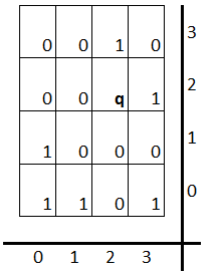


Figura 4.4: Matriz que representa el conjunto de datos que se quiere analizar

Para finalizar la primera etapa, se consulta al usuario cúal es el punto q que debe contener el rectángulo maximal vacío (parametro de entrada en Figura 4.1).

La segunda etapa del algoritmo *qAREMAV* se explicará en la sección siguiente.

4.2.2. Análisis de datos y construcción de la escalera

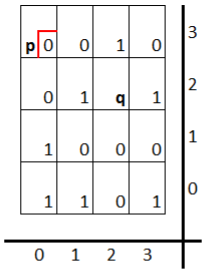
Teniendo todos los datos necesarios, ahora se explicará cómo funciona la segunda etapa del algoritmo, en la que se leen dos filas y construye la escalera mientras se analizan los datos y se utilizarán las siguientes variables, ademas de las ya vistas.

- *ptoBajo*: es el punto más hacia la derecha que se encuentre en la *fila2*
- *escalera*: es el stack donde se almacena la pila

Para una mejor comprensión de esta etapa del algoritmo, este se explicará resolviendo el problema ejemplo presentado en la Figura 4.4. El punto q que se da para el análisis será el punto $q = (2, 2)$ (Fig 4.4). La letra p corresponde a la ubicación actual que se está analizando (x, y). La escalera se presentará de color rojo para así visualizar mejor su comportamiento.

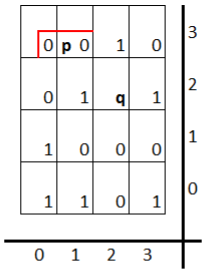
Esta etapa, al igual que la anterior, está compuesta por varios pasos, los que se explican a continuación:

El primer paso: es asegurar que la *fila2* contenga puntos, de esa forma se podrá analizar el conjunto de datos. Luego se debe recorrer en su totalidad la fila *fila1* (Figura 4.1 línea 11). Cuando se termina de utilizar el arreglo *fila1*, el arreglo *fila2* pasará a ser el arreglo *fila1* y el arreglo *fila2* se rellenará



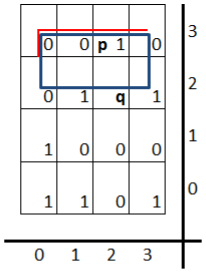
fila1 = [(0,3,*false*)],[(1,3,*false*)],[(2,3,*true*)],[(3,3,*false*)]
fila2 = [(0,2,*false*)],[(1,2,*true*)],[(2,2,*false*)],[(3,2,*true*)]
ptsAltos = [(0,3,*false*)],[(1,3,false)],[(2,3,*true*],[(3,3,*false*)]
ptoBajo = *null*
escalera = [(0,3,true)]

Figura 4.5: Matriz y escalera que representa el análisis de primera celda



fila1 = [(0,3,*false*)],[(1,3,*false*)],[(2,3,*true*)],[(3,3,*false*)]
fila2 = [(0,2,*false*)],[(1,2,*true*)],[(2,2,*false*)],[(3,2,*true*)]
ptsAltos = [(0,3,*false*)],[(1,3,false)],[(2,3,*true*],[(3,3,*false*)]
ptoBajo =(1,2,*true*)
escalera = [(0,3,true)]

Figura 4.6: Matriz y escalera que representa el análisis de la segunda celda



fila1 = [(0,3,*false*)],[(1,3,*false*)],[(2,3,*true*)],[(3,3,*false*)]
fila2 =[(0,2,*false*)],[(1,2,*true*)],[(2,2,*false*)],[(3,2,*true*)]
ptsAltos = [(0,3,*false*)],[(1,2,*true*)],[(2,3,*true*],[(3,3,*false*)]
ptoBajo = (1,2,*true*)
escalera = [(0,2,true)],[(1,3,true)]

Figura 4.7: Matriz y escalera que representa el análisis de la tercera celda

de la forma que ya se explicó anteriormente (Figura 4.1 líneas 31 y 32).

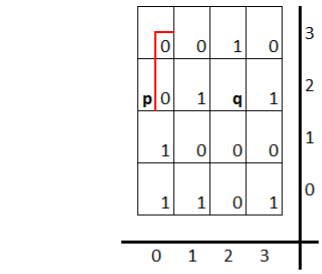
Con esto se hallarán leidas las dos filas que deben ser procesadas por el algoritmo. Estas filas no cambiarán a menos que ya se haya terminado de analizar la primera fila. La última columna solo sirve para ubicar el borde y no se analiza, al igual que la última fila de la matriz.

Al ser recorrida la *fila1* se deberá actualizar el arreglo de los puntos altos (*ptsAltos*) y el stack para la escalera (*escalera*). Esto es explicado en el paso que sigue.

En el segundo paso: se analiza la posición que se está analizando en el arreglo *fila1*, si es un 1 y no es un borde, se almacenará el valor de sus coordenadas en la posición correspondiente en el arreglo *ptsAltos* (Figura 4.1 líneas desde 15 hasta 17).

En el tercer paso: se analizará la posición abajo de (x, y), esto es, la posición $(x, y-1)$ que se encuentra en la *fila2*, si esa celda contiene un 1 se almacenará ese punto en una variable llamada *ptoBajo* (Figura 4.1 línea 18).

En el cuarto paso: se actualizará la escalera, la cual se genera de forma similar a lo planteado en [6] en el capítulo anterior (Figura 4.1 línea desde 19). A continuación se describen los casos que se pueden presentar (Figura 4.2):



fila1 = [(0,2,false)],[(1,2,true)],[(2,2,false)],[(3,2,true)]
fila2 = [(0,1,*true*)],[(1,1,false)],[(2,1,*false*)],[(3,1,*false*)]
ptsAltos = [(0,3,*false*)],[(1,3,false)],[(2,3,*true*],[(3,3,*false*)]
ptoBajo = (0,1,*true*)
escalera = [(0,3,true)]

Figura 4.8: Matriz y escalera que representa el análisis de la cuarta celda

- Caso 1) Si $ptsAltos(x) > escalera.peek$: Se deberá almacenar en el tope del stack las coordenadas correspondiente a la combinación de x obtenido de la posición que se está analizando del arreglo *ptsAltos* y la coordenada y obtenida del punto que se encuentra en el tope del stack (Figura 4.2 líneas desde 40 hasta 42).
- Caso 2) Si $ptsAltos(x) = escalera.peek$: No se generarán nuevos peldaños por lo que no se empilará nada (Figura 4.2 líneas desde 44 hasta 45).
- Caso 3) Si $ptsAltos(x) < escalera.peek$: Se deberá desempilar el stack tantas veces como sea necesario hasta que se cumpla que $ptsAltos(x) \geq escalera.peek$, luego se debe proceder como se indica en los casos 1, o 2 , según corresponda (Figura 4.2 líneas desde 47 hasta 54).

El quinto paso: ocurre despues de actualizada la escalera, será necesario ver si hasta ese punto se pueden generar rectángulos maximales vacíos que contengan a q. Para esto es necesario analizar la posición siguiente en el arreglo *ptsAltos*, junto al punto almacenado en *ptoBajo* $(x, y-1)$ y determinar si cumple con los requisitos planteados en el capitulo anterior para la generacion de rectángulos maximales vacíos. Junto con verificar que cada rectángulo generado sea maximal vacío, se deberá verificar que también contiene al punto q (Figura 4.1 líneas 25, se explica mejor en 4.3). Los rectángulos de mayor área podrán ser generados de la siguiente forma:

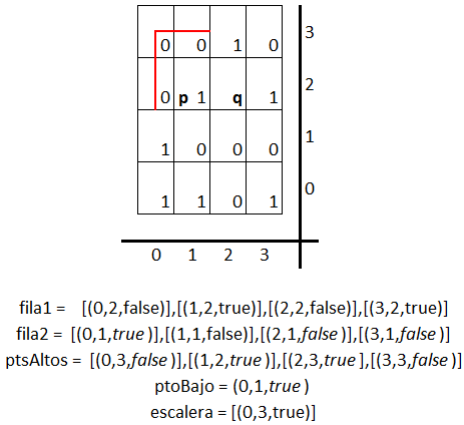


Figura 4.9: Matriz y escalera que representa el análisis de la quinta celda

- El rectángulo se genera estando (x, y) en la esquina inferior derecha del conjunto de datos: Aquí el punto inferior derecho con el que se genera el rectángulo no sera (x, y), sino $(x + 1, y - 1)$, *fila2* será el borde inferior, y la coordenada $x + 1$ estará en la columna derecha del conjunto de datos (Figura 4.3 línea 61 y 62).

- El rectángulo se genera estando (x, y) en el extremo derecho del conjunto de datos: Aquí el punto inferior derecho se encuentra en cualquier posición de la última columna de la derecha del conjunto de datos, por lo que se deberá generar utilizando la variable de *ptoBajo* (Figura 4.3 línea 64 y 65).

- El rectángulo se genera estando (x, y) en el extremo inferior del conjunto de datos: Aquí el punto inferior derecho se encuentra en cualquier posición sobre la última fila del conjunto de datos, aquí no será necesario utilizar la variable *ptoBajo* (Figura 4.3 línea 67 y 68).

- El rectángulo se genera estando (x, y) en cualquier posición dentro del conjunto de datos: Aquí el punto inferior derecho se encuentra en cualquier posición que no sea alguno de los casos anteriores. Para este caso se debera utilizar las variables *ptoBajo* y *ptosAltos[i+1]* (Figura 4.3 línea 70 y 71).

En el conjunto de datos que se muestra como ejemplo (Figura 4.4), durante el análisis de la primera fila segunda celda (Fig 4.6), durante el paso 3 ocurre que la variable de *ptoBajo* se actualiza. El peldaño de la escalera

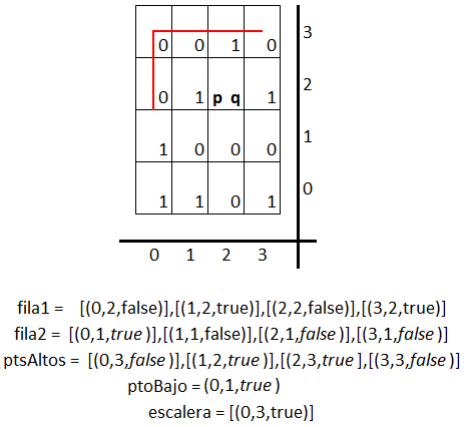


Figura 4.10: Matriz y escalera que representa el análisis de la sexta celda

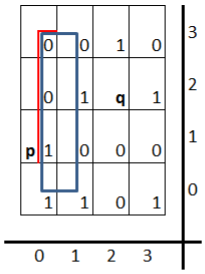
aumenta su longitud (paso 4).

En el análisis de la tercera celda de la *fila1*, se genera un rectángulo maximal vacío, el cual se ve en la Figura 4.7 cuyos vertices son $(0, 3), (3, 2)$, ya que cuando se llega al paso 5, se detecta que se llegó al borde derecho y por debajo limita con el *ptoBjo* $(1, 2)$. Este rectángulo maximal vacío contiene al punto q, por lo que se almacenan sus coordenadas de vértice inferior derecho y vértice superior izquierdo para así obtener su área. Luego si se encuentra otro rectángulo maximal vacío que contenga el punto q, se comparará su área contra la que ya se tiene almacenada y, de ser mayor, se almacenarán los nuevos puntos, de no ser así, se descartará el último rectángulo que se haya formado.

Continuando con el ejemplo (Figura 4.4) y terminada de analizar la primera fila, se debe proceder a analizar la siguiente fila, por lo que la variabe *fila2* pasará a ser *fila1* y la *fila2* será llenada como se explicó en el paso 1 de esta segunda etapa del algoritmo.

Analizando la primera celda de esta nueva *fila1*, (Figura 4.9, en el paso 3, se deberá cambiar la variable de *ptoBajo* por la coordenada $(0, 1)$

En el análisis de la segunda celda de la *fila1* (Fig 4.9) se cambiará la coordenada perteneciente al arreglo de *ptsAltos* correspondiente a $x = 1$ (paso 2), aquí el valor sera de $(1, 2, true)$. Con esto, más adelante se cambiará



fila1 = [(0,1,*true*)],[(1,1,*false*)],[(2,1,*false*)],[(3,1,*false*)]
fila2 = **borde** [(0,0,*true*)],[(1,0,*true*)],[(2,0,*false*)],[(3,0,*true*)]
ptsAltos = [(0,3,*false*)],[(1,2,*true*)],[(2,3,*true*],[(3,3,*false*)]
ptoBajo = **borde**
escalera = [(0,3,*true*)]

Figura 4.11: Matriz y escalera que representa el análisis de la septima celda

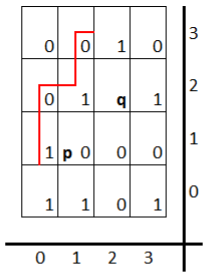
la forma de la escalera.

Al terminar el análisis de la fila (tercera celda Fig 4.10), en el paso 5 se generará otro rectángulo maximal vacío que contiene a q, solo que no podrá ser mostrado ya que el mismo punto (x, y) bajo análisis el cúal genera el rectangulo es el punto q.

Ya terminada de analizar la segunda fila de la matriz, se deberá hacer el cambio de filas en los arreglos, donde como se explicó anteriormente (paso 1), *fila2* pasará a ser *fila1* y *fila2*, en este caso, la última fila sera el límite inferior del conjunto de datos, por lo que *fila2* estará en un caso espacial que se llamará **caso borde**.

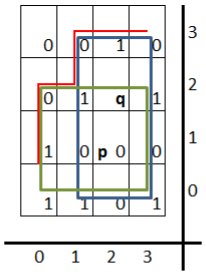
Analizando esta última fila, se comienza con la celda ubicada en la Figura 4.11, donde la celda tiene un valor de 1 y se deberá actualizar el valor correspondiente en el arreglo de *ptsAltos* (paso 2). Durante el análisis de esta fila no será necesario realizar el paso 3, ya que la *fila2* será el límite inferior del conjunto. La escalera comenzará en el punto correspondiente al $(0, 3)$ (paso 4). En el paso 5 se generará un rectangulo maximal vacío, con las coordenadas de $(0, 3), (1, 0)$. Este rectángulo generado no contiene al punto q por lo que es descartado.

En la segunda celda de esta fila (Figura 4.12), la variable *escalera* crecerá hacia arriba y a la derecha por el punto 1 que se encuentra en la coordenada $(1, 2)$ (paso 4).



fila1 = [(0,1,*true*)],[(1,1,*false*)],[(2,1,*false*)],[(3,1,*false*)]
fila2 = **borde** [(0,0,*true*)],[(1,0,*true*)],[(2,0,*false*)],[(3,0,*true*)]
ptsAltos = [(0,3,*false*)],[(1,2,*true*)],[(2,3,*true*],[(3,3,*false*)]
ptoBajo = **borde**
escalera = [(0,2,*true*)],[(1,3,*true*)]

Figura 4.12: Matriz y escalera que representa el análisis de la octava celda



fila1 = [(0,1,*true*)],[(1,1,*false*)],[(2,1,*false*)],[(3,1,*false*)]
fila2 = **borde** [(0,0,*true*)],[(1,0,*true*)],[(2,0,*false*)],[(3,0,*true*)]
ptsAltos = [(0,3,*false*)],[(1,2,*true*)],[(2,3,*true*],[(3,3,*false*)]
ptoBajo = **borde**
escalera = [(0,2,*true*)],[(1,3,*true*)]

Figura 4.13: Matriz y escalera que representa el análisis de la novena celda

Para finalizar la resolución del problema que se propuso al inicio de esta sección (problema ejemplo), en la tercera celda de la fila (Figura 4.13) se analiza la útima casilla que no es borde. Con esta celda se pueden generar dos rectángulos en el paso 5, los cuales deben ser construidos como se mencionó anteriormente. El primero será construido con el valor que se encuentra mas arriba de la pila, el cúal es (1,3) y el valor inferior derecho de donde se está analizando (3,0). Luego de que ese valor se desempila, queda en el tope de la pila el valor (0,2) y (3,3). En ambos casos los rectángulos maximales vacíos contienen al punto q y su área es mayor al primer rectángulo obtenido.

Ya terminado de resolver el ejemplo, cabe notar que es muy probable que los puntos del conjunto de datos no tendrán x e y consecutivas, esto es, el eje X e Y de la matriz almacenará valores no uniformemente distribuidos. Además, se debe especificar que no se analizan todas las celdas, ya que la última columna y última fila de la matriz sirve de límites derecho e inferior respectivamente, para el conjunto de datos.

El código completo de *qAREMAV* se encuentra en el Apendice A.

4.3. Modelo de Clases

4.3.1. Clase qAREMAV

La clases *qAREMAV* (Figura 4.14) es la clase principal, aquí se encuentra todo el código necesario para resolver el problema de hallar todos los rectángulos mvacíos, y el máximo rectángulo vacío que contiene el punto q.

El método para obtener todos los rectángulos maximales vacíos y el método que selecciona el rectángulo maximal vacío que contenga al punto q, es un método iterativo, el cual necesita de otros métodos de apoyo para cumplir esta tarea.

Se trabajó con varios métodos ya que ello facilita la reutilización de código, así como entender su funcionamiento.

4.3.2. Implementación de ExternalMergeSort

La clase ExternalMergeSort (Figura 4.15) sirve para ordenar el conjunto de datos de entrada, este conjunto de datos está compuesto por datos

Figura 4.14: clases implementadas

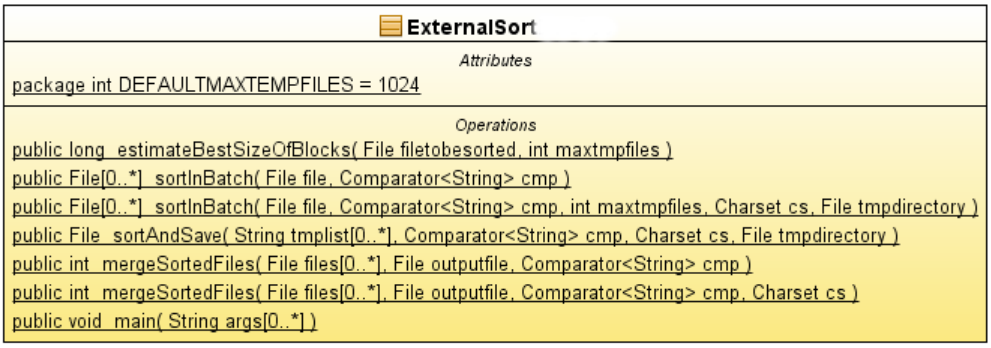


Figura 4.15: Diagrama de clase ExternalMergeSort

desordenados de la forma y, x, id.

4.4. Ejecución del algoritmo

El modo de uso es simple, se debe ejecutar por consola ya que no dispone de una interfaz con la que interaccionar.
En el apéndice C se encuentran las imagenes paso por paso de la ejecución del algoritmo, junto aun gráfico del rectángulo maximal vacío dentro del conjunto de puntos que se ingresa al programa.

1. El programa pedirá ingresar el nombre del archivo donde se encuentra el set de datos desordenado.
2. El usuario deberá ingresar el nombre y luego hacer clic en la tecla ENTER.
3. El programa pedirá las coordenadas x e y donde ubica el punto q de consulta.
4. El usuario deberá ingresar los números en forma de decimales o enteros.
5. Al hacer clic en ENTER el programa procesará los datos y entregará el mayor rectángulo vacío que contega a q.

Se debe recordar, que el punto q no debe pertenecer al conjunto de datos inicial.

Capítulo 5

Validación y experimentación

La validación de este programa se desarrolló en base a la experimentación y que forman parte de los objetivos que se propusieron en este proyecto. Estos experimentos son en su mayoría comparar los resultados del programa que se implementó, contra los resultados obtenidos en en el artículo original [6].

Antes de comenzar la experimentación se compararon las máquinas donde se harían los experimentos, esto debido a que los experimentos realizados en [6] fueron en el año 2001, entonces las máquinas eran más lentas, por lo cual si se realizan estos mismos experimentos en las nuevas máquinas, los resultados cambiarian tanto en la velocidad como en la capacidad de almacenamiento en memoria principal.

La máquina que se utilizó en [6] fue: IBM RISC System/6000, 256 MB RAM, multiuser 42MHz.

Para estos experimentos se utilizó un espacio en el servidor Parra, disponible en la Universidad del Bío-Bío, el cual tiene fines de investigación para alumnos que desarrollan su actividad de titulación. La máquina tiene las siguientes características.

Servidor Debian, 8GB RAM, 4 Núcleos de 3.092 MHz.

Como se puede ver, a simple vista la máquina con la que se trabaja actualmente es mucho más poderosa que la se utilizó hace varios años atrás por los autores [6], por lo que se podrá ver esto reflejado en la velocidad de

procesamiento, pero no deberá afectar la forma de los gráficos que se verán en la sección siguiente.

Para entender como se hace la validación es necesario definir algunos términos que se aplicaron a la hora de ejecutar los experimentos:

Tupla (T): Una tupla corresponde a un punto p existente en la matriz. Esta tupla corresponde a un punto dado en el conjunto de datos inicial.

Tamaño de matriz (`S`): Es el tamaño de la matriz, este tamaño está dado por los distintos valores de x (`N`), multiplicados por los distintos valores de y (`M`).

Densidad (`D`): La densidad está dada por la ecuación $D = \frac{T}{NM}$, esta ecuación define cuál es la densidad de separación de las tuplas dentro de la matriz. Así una densidad menor, serán menos tuplas dentro de la matriz, y una densidad mayor serán tuplas más cercanas entre si dentro de la matriz.

Antes de proceder a la experimentación se deben estudiar los resultados obtenidos en [6]. Sus experimentos se basaron en datos obtenidos al azar con una densidad específica del 20 % y junto a eso, un número fijo de `N`=1000. De esta manera, con esos antecedentes se crearon los conjuntos de datos que serían necesarios para esta experimentación. (El programa se encuentra en el Apéndice B)

Los experimentos que se realizaron son en relación a (*i*) cómo influye la densidad en la velocidad de procesamiento, (*ii*) cómo influye la densidad en la cantidad de máximos rectángulos vacíos, (*iii*) la velocidad de distintos tamaños de la matriz y (*iv*) la cantidad de máximos rectángulos vacíos respecto al tamaño de la matriz.

Los primeros dos experimentos se realizaron con datos generados al azar de 500.000 tuplas, 1.000.000 de tuplas, 1.500.000 de tuplas y 2.000.000 de tuplas; cada una con densidades de 10 %, 20 %, 30 %, 40 %, 50 % y 60 %.

Los últimos dos experimentos se realizaron con datos generados al azar de 50.000.000 de tuplas, 100.000.000 de tuplas, 150.000.000 de tuplas y 200.000.000 de tuplas con una densidad del 20 %. Esto debido a que los experimentos en el artículo original fueron desarrollados con esta densidad ya que era la que más aproximaba a datos reales [6].

Puntos / Densidad	500k	1m	1,5m	2m
10 %	47126[ms]	85405[ms]	122085[ms]	179702[ms]
20 %	28324[ms]	54608[ms]	86502[ms]	116811[ms]
30 %	18676[ms]	38388[ms]	56243[ms]	69362[ms]
40 %	16244[ms]	26847[ms]	41056[ms]	51557[ms]
50 %	11294[ms]	19284[ms]	29681[ms]	38110[ms]
60 %	7792[ms]	13765[ms]	24529[ms]	29088[ms]

Tabla 5.1: Resultados Densidades y velocidades

Ahora se detallarán los experimentos realizados, contra los experimentos realizados en [6]. Con estos resultados se podrán comparar las semejanzas y diferencias que se obtuvieron en la realización de estos experimentos. Posteriormente se discuten algunas conclusiones.

1. **Influencia de las distintas densidades en la velocidad de procesamiento.**
   En este primer experimento, se quiere probar como afectan las distintas densidades aplicadas a distintos conjuntos de datos en relación a la velocidad. Para obtener estos resultados se crearon 10 set de datos distintos para todas las tuplas (500.000, 1.000.000, 1.500.000 y 2.000.000) en sus distintas densidades desde 10 % hasta el 60 %. Después de obtenidos los resultados, se calcularon los promedios para luego ser tabulados en el Cuadro 5.1 y graficados en la Figura 5.1. Los resultados obtenidos por Edmonds et al.[6] se encuentran en la Figura 5.2

   Como se puede apreciar en la Figura 5.1 y Figura 5.2 ambas implementaciones tienen comportamientos muy parecidos, lo que permite afirmar que nuestra implementación está dentro de la complejidad del algoritmo *AREMAV* .

2. **Influencia de las distintas densidades en la generación de rectángulos maximales vacíos.**
   Este experimento se enfocó principalmente en saber cuántos rectángulos maximales vacíos se generaban al tener más números de tuplas a distintas densidades. Se utilizaron los mismos conjuntos de datos

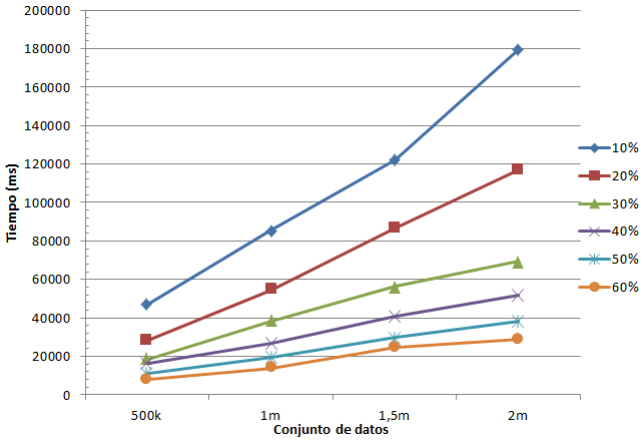


Figura 5.1: Gráfico Resultados Densidades y velocidades

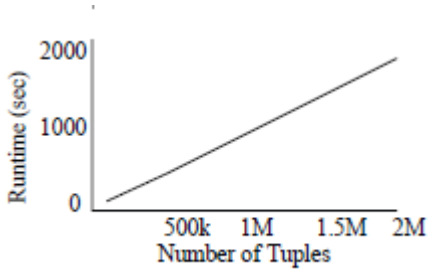


Figura 5.2: Resultados de velocidad del algoritmo obtenido por los autores, con una densidad del 20 % [6]

Puntos / Densidad	500k	1m	1,5m	2m
10 %	1.017.818	2.035.203	3.051.799	4.068.361
20 %	629.414	1.135.060	1.888.129	2.516.943
30 %	443.473	886.613	1.329.708	1.772.420
40 %	328.624	656.932	984.776	1.312.701
50 %	247.820	495.459	742.667	990.034
60 %	186.090	372.042	557.830	743.162

Tabla 5.2: Resultados Densidades y cantidad de rectángulos generados

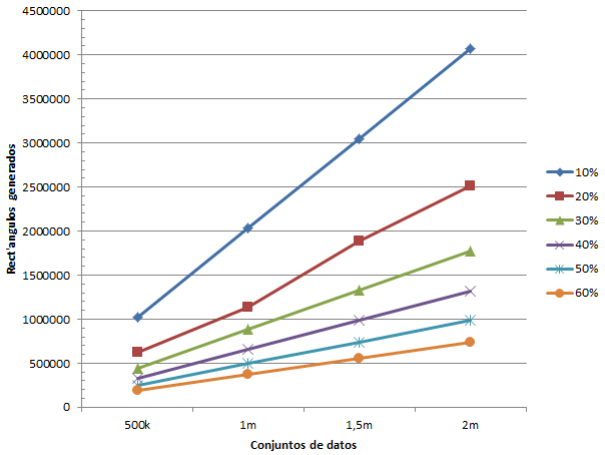


Figura 5.3: Resultados Densidades y Rectángulos

del experimento anterior, esta vez se enfocó en obtener la cantidad de rectángulos maximales vacíos en vez de la velocidad. El Cuadro 5.2 muestra el promedio de los rectángulos maximales vacíos obtenidos al analizar los 10 conjuntos de datos con las densidades desde 10 % hasta 60 % y están graficados en la Figura 5.3. Luego son comparados con la Figura 5.4 que obtuvieron Edmonds et al. en sus experimentos con una densidad específica del 20 % [6].

De manera similar que para el caso del tiempo de jecución, la cantidad de rectángulos maximales vacíos de ambas implementaciones del algoritmo, evidencian el mismo comportamiento (ver Figura 5.3 y 5.4).

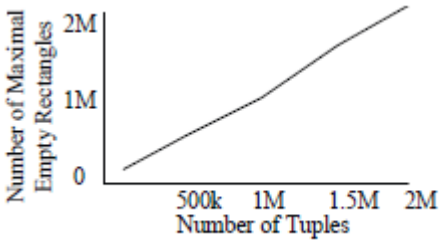


Figura 5.4: Rectángulos obtenido por Edmonds en distintos conjuntos de datos a densidad del 20 %

Tamaño	Tiempo (ms)
50m	43130[ms]
100m	57014[ms]
150m	63588[ms]
200m	66519[ms]

Tabla 5.3: Tamaño de matriz y velocidad

3. **Influencia de el tamaño de la matriz en velocidad.**
   Este experimento tiene como finalidad ver la incidencia de los distintos tamaños de matriz con un número constante de `N` y el tamaño de la matriz variable (50.000.000, 100.000.000, 150.000.000, 200.000.000), todo con una densidad de el 20 % para los distintos conjuntos de datos contra la velocidad de procesamiento. Los resultados están tabulados en el Cuadro de resultados 5.3 y gráficados en la Figura 5.5. Luego se comparan los datos obtenidos en [6] en la Figura 5.6.

   Como se puede ver en las Figuras 5.5 y 5.6 las implementaciones difieren en cierto grado, esto producto de que las velocidades no son las mismas debido a las distintas máquinas utilizadas. Las similitudes que tienenen ambas Figuras (5.5 y 5.6) es que ambas son funciones crecientes, por lo que sí se puede afirmar que nuestra implementación está dentro de la complejidad del algoritmo *AREMAV* .

4. **Influencia del tamaño de la matriz en la generación de Rectángulos maximales vacíos.**
   El ultimo experimento que se realizó no fue realizado por Edmonds,

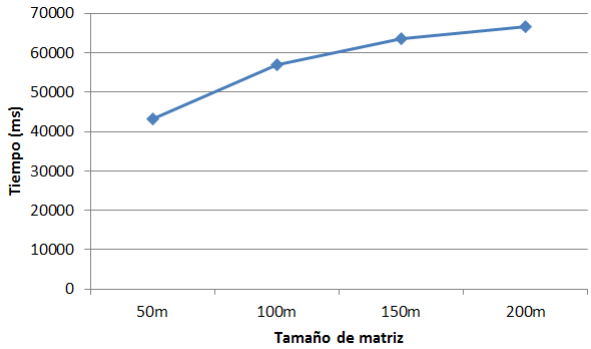


Figura 5.5: Gráfico Resultados Tamaño de Matriz y Velocidad

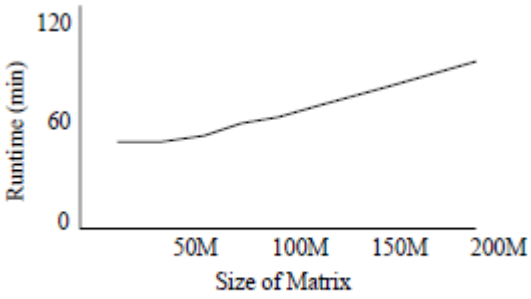


Figura 5.6: Gráfico Resultados Tamaño de Matriz y Velocidad obtenido por Edmonds con una densidad del 20 %

Tamaño	Rectángulos
50m	888.356
100m	1.007.313
150m	1.061.112
200m	1.090.694

Tabla 5.4: Tamaño de matriz y velocidad

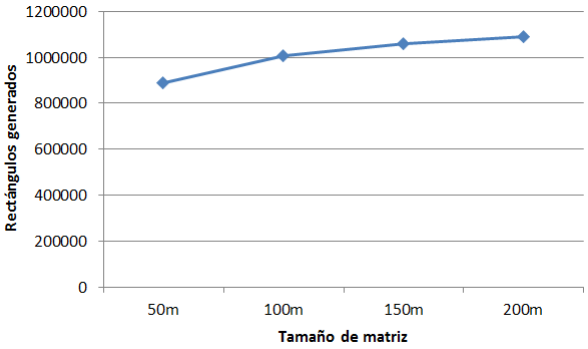


Figura 5.7: Resultados Tamaño de Matriz y de Rectángulos generados con densidad del 20 %

en este experimento se analizan los anteriores conjuntos de datos para saber cuántos rectángulos maximales vacíos se podrían obtener con grandes tamaños de matrices y con una densidad constante del 20 %. Los datos están tabulados en el Cuadro de resultados 5.4 y graficados en la Figura 5.7.

Este experimento sirvió para demostrar la cantidad de rectángulos maximales vacíos generados con conjuntos de datos cuya densidad es del 20 %. La Figura 5.7 muestra la consistencia del algoritmo *AREMAV* con grandes conjuntos de datos con una densidad del 20 %, donde se ha mostrado que el algoritmo es muy lento en comparación a las otras densidades.

Capítulo 6

Conclusiones

El trabajo muestra la implementación del algoritmo *AREMAV* para encontrar todos los rectángulos vacíos maximales posibles de obtener en un conjunto de puntos. La implementación fue realizada en JAVA. Con el objeto de validar la implementación del algoritmo se replicaron una serie de experimentos realizados en el trabajo original. Los resultados experimentales permiten concluir que nuestra implementación se ajusta a la complejidad del algoritmo. La diferencia en la cantidad de rectángulos maximales vacíos está dado por que en [6] no se utiliza los límites del conjunto de datos, en cambio en el algoritmo desarrollado en este trabajo, sí se considera la generación de rectángulos maximales vacíos con los límites del conjunto de datos, por lo que se generarán más rectángulos maximales vacíos en este programa.

Otra conclusión que se puede obtener de los experimentos realizados, fue acerca de la influencia de las distintas densidades en los conjuntos de datos. Donde se puede ver que a mayor densidad el algoritmo funciona mejor, ya que existen menos espacios vacíos para ser analizados y una mayor cantidad de tuplas, por lo que la escalera no crecerá mucho y se irá vaciando continuamente. Al contrario, teniendo una menor densidad el algoritmo se demora más en ejecutar ya que se debe utilizar mucho la escalera, tanto en empilar datos como en desempilarlos. También existen una cantidad mayor de espacios vacíos por lo que el algoritmo deberá detenerse en muchas celdas para analizarlas. Con respecto al tamaño de la matriz hay algunas diferencias mayores. Sin embargo, solo en lo relativo al tiempo de ejecución, ya que utilizando una matriz tan grande y con una densidad del 20 % los resultados experimentales deberían ser muy similares a los valores obtenidos con los conjuntos de datos a densidades del 20 % y así resultó, por lo que

complementa las afirmaciones realizadas anteriormente.

Para finalizar se puede decir que el desarrollo de este trabajo permitió conocer la importancia de lograr encontrar los espacios vacíos de forma rápida y efectiva ya que son de mucha utilidad para la minería de datos y otras áreas de estudio. Como una alternativa de trabajo futuro, es la de lograr optimizar la forma en que el algoritmo *AREMAV* encuentra el rectángulo maximal vacío que contenga al punto q. Ya sea optimizando su velocidad descartando celdas que no sean útiles u optimizándolo o utilizando algún método para compactar el conjunto de datos, para que luego al revisarlo sea menor el tiempo de análisis.

Bibliografía

[1] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In ACM SIGMOD Conference on Management of Data, pages 47–57. ACM, 1984

[2] Christian Böhm and Hans-Peter Kriegel. Determining the convex hull in large multidimensional databases. In Proceedings of the Third International Conference on Data Warehousing and Knowledge Dis- covery, DaWaK '01, pages 294–306, London, UK, 2001. Springer- Verlag.

[3] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In Proceeding of SIGMOD '95, pages 71–79, San Jose, CA, USA, 1995.

[4] Antonio Corral, Yannis Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos.Algorithms for processing k closest-pair queries in spatial databases. Data Knowl. Eng., 49(1):67–104, 2004.

[5] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Cost models for distance joins queries using r-trees. Data Knowl. Eng., 57:1-36, April 2006.

[6] Edmonds J, Gryz J, Liang D, Miller RJ (2003) Mining for empty spaces in large data sets. Theor. Comput Sci 296:435-452

[7] Gilberto Gutierrez and Jose R. Parama. Finding the largest empty rectangle containing only a query point in large multidimensional databases. In Anastasia Ailamaki and Shawn Bowers, editors, Scientific and Statistical Database Management, volume 7338 of Lecture Notes in Computer Science, pages 316–333. Springer Berlin Heidelberg, 2012.

[8] Gutiérrez G, Paramá J, Brisaboa N, Corral A. (2013) The largest empty rectangle containing only a query object in Spatial Databases

[9] Naamad A, Lee DT, HsuW-L (1984) On the maximum empty rectangle problem. Discrete Appl Math 8:267-277

[10] Orlowski M(1990) A new algorithm for the largest empty rectangle problem. Algorithmica 5:65-73

[11] Chazelle B, Drysdalet RL, Lee DT (1986) Computing the largest empty rectangle. SIAM J Comput 15:300-315

[12] Aggarwal A, Suri S (1987) Fast algorithms for computing the largest empty rectangle. In: Proceedings of SCG ?7. ACM, pp 278-290

[13] De M, Nandy SC (2011) Inplace algorithm for priority search tree and its use in computing largest empty axis-parallel rectangle. CoRR abs/1104.3076

[14] Minati D, Nandy S (2011) Space-efficient algorithms for empty space recognition among a point set in 2d and 3d. In: Proceedings of the 23rd annual Canadian conference on computational geometry, pp 34753

[15] Nandy S, Bhattacharya B (1998) Maximal empty cuboids among points and blocks. Comput Math Appl 36(3):11-20

[16] Augustine J, Das S, Maheshwari A, Nandy SC, Roy S, Sarvattomananda S (2010) Recognizing the largest empty circle and axis-parallel rectangle in a desired location. CoRR abs/1004.0558

[17] Augustine J, Das S, Maheshwari A, Nandy SC, Roy S, Sarvattomananda S (2010) Querying for the largest empty geometric object in a desired location. CoRR abs/1004.0558v2

[18] Kaplan H, Mozes S, Nussbaum Y, Sharir M (2012) Submatrix maximum queries in monge matrices and monge partial matrices, and their applications. In: Proceedings of SODA 2012, SIAM, pp 338-355

Apéndice A

Codigo qAREMAV

qAREMAV.java

```java
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Collection;
import java.util.Hashtable;
import java.util.StringTokenizer;
import java.util.Stack;
import java.util.Arrays;
import java.util.ArrayList;

/**
 *
 * @author Felipe Lara R.
 */
public class qAREMAV {
  //Los distintos valores de X en el conjunto de puntos,
  //ordenados de menor a mayor
  private static double[] equis;
  // Arreglo que almacena todos los puntos mas proximos a la fila de analisis,
  //incluye esta (Yr)
  private static Punto [] yr ;
  // Variables que almacenan los valores extremos de Y en el conjunto de puntos
  private static double yMayor, yMenor=0;
  //Almacena el punto mas a la derecha en la fila ubicada bajo la que se analiza
  private static Punto ptoBajoIzq;
  //El valor "" indica que no se ha iniciado la lectura para generar rectangulos
  public static String lineaNoProcesada="";
  // Variables que almacenan los valores de x e y que se quieren buscar
  public static Punto puntoQuery;
  // Variable que almacena la superficie del rectangulo obtenido
  public static double area = 0;


  public static void main(String[] arg) {
    //Despliega el contenido de las variables relevantes del algoritmo
    //para verificar su funcionamiento

    // Variable que cuenta el numero de rectangulos que se van generando
    long cont = 0;
    int nroPuntosAProcesar;
    double xMaxDer;
    Punto[] primeraFila = null;
    Punto[] segundaFila = null;

```

```
46      Punto pInfDer;
47      Stack<Punto> escalera = new Stack<Punto>();
48
49      try {
50        // Obtiene los datos que requiere el algoritmo MER
51        nroPuntosAProcesar = obtieneDatosIniciales(arg);
52        xMaxDer = equis[equis.length-1];
53
54        // Despliega valores extremos de los ejes X (maximo) e Y (minimo)
55        System.out.println("\nNro de puntos a procesar: " + nroPuntosAProcesar);
56        System.out.println("xMaxDer=" + (int)xMaxDer + "; yMenor=" + (int)yMenor);
57
58        // Implementacion algoritmo MER para obtener rectangulos maximales
59        BufferedReader br = new BufferedReader(new FileReader("DatosOrdena.txt"));
60
61        int nrosProcesados = 0;
62        primeraFila = leeFila(br, equis);
63        if (primeraFila.length > 0) {
64          segundaFila = leeFila(br, equis);
65        }
66
67        while (segundaFila != null && segundaFila.length > 0) {
68          ptoBajoIzq = null;
69          //Existe una fila a procesar (primeraFila) y
70          //la siguiente a ella (segundaFila)
71          System.out.print("\nPrimera fila = ");
72          despliegaFila(primeraFila);
73          System.out.print("\nSegunda fila = ");
74          despliegaFila(segundaFila);
75
76          // Inicializa variables para procesar primeraFila
77          actualizaYr(yr, primeraFila);
78          System.out.print("\n\nPuntos Altos = ");
79          despliegaFila(yr);
80          escalera.clear();
81
82          // Procesa cada punto de primeraFila, excepto el mas derecho
83          for(int i=0; i<primeraFila.length-1; i++) {
84            pInfDer = primeraFila[i];        // Corresponde a (x,y)
85            generaEscalera(i, pInfDer, yr, escalera);
86            System.out.println("\nPILA = " + escalera);
87            cont += obtieneRectangulos(i, primeraFila, segundaFila, xMaxDer,
88                    yr, escalera);
89          }
90
91          nrosProcesados += obtieneNroPuntos(primeraFila);
```

```
92          System.out.println("\nNro puntos procesados hasta el momento... "
93                  + nrosProcesados);
94          primeraFila = segundaFila;
95          segundaFila = leeFila(br, equis);
96        }
97
98        System.out.println("\nNmero total de rectangulos generados: " + cont);
99        br.close();
100     }
101     catch (Exception e) {
102       System.err.println(e);
103     }
104   }
105
106   public static int obtieneDatosIniciales(String[] arg) throws
107           FileNotFoundException, IOException {
108     // Variables para medir tiempos
109     long inicio, termino, total;
110     // Ordenar archivo de texto de coordenadas
111     inicio = System.currentTimeMillis();
112     ExternalSort.main(arg);
113     termino = System.currentTimeMillis();
114     total = termino-inicio;
115     System.out.println("tiempo en ordenar ExternalMerge :"+ total);
116
117     // Variables a usar
118
119     // Crea el file
120     BufferedReader br = new BufferedReader(new FileReader("File.txt"));
121     StringTokenizer st = null;
122     String xst,ys=null,linea=null; // String para cada coordenada y flags
123     //hashtable para almacenar X sin que se repitan
124     Hashtable<String,Double> ht = new Hashtable<String,Double>();
125     Object[] xO;// Arreglo de Objetos para almacenar desde una collection
126     // Crear una collection para almacenar todos los X sin que se repitan
127
128     // Almacena los puntos para inicializar yr
129     ArrayList<Punto> ptosPrimeraFila = new ArrayList<Punto>();
130     int nroPuntosAProcesar=0;          // Para saber cuantos puntos hay en el archivo,
131     // Obtiene variables iniciales a partir del archivo ordenado
132     inicio = System.currentTimeMillis();
133     try{
134       linea=br.readLine();
135       while(linea!=null){
136         st = new StringTokenizer(linea);
137         ys=st.nextToken();
```

```
138         if (nroPuntosAProcesar == 0) {
139           yMayor=Double.valueOf(ys);
140         }
141         xst=st.nextToken();
142         // Se lee el id, pero no se procesa por no ser relevante en este caso
143         st.nextToken();
144         ht.put(xst, Double.parseDouble(xst));
145         // Almacena los puntos de la primera fila para inicializar en yr
146 
147          // Los puntos estan ordenados descendentes en y,x
148         if (yMayor==Double.valueOf(ys)) {
149           ptosPrimeraFila.add(0,new Punto(Double.parseDouble(xst),yMayor,true));
150         }
151         nroPuntosAProcesar++;         // Almacena la cantidad de puntos leidos
152         linea = br.readLine();
153       }
154     }catch(Exception e){
155       System.out.println(e);
156     }
157     if (ys!=null) {
158       yMenor = Double.valueOf(ys);
159     }
160     Collection<Double> v = ht.values();
161     x0 = v.toArray();
162     equis = new double[x0.length];
163     for (int i = 0; i < equis.length; i++) {
164       equis[i]=Double.parseDouble(x0[i].toString());
165     }
166     // Nueva linea
167     Arrays.sort(equis);
168     termino = System.currentTimeMillis();
169     total = termino-inicio;
170     System.out.println("Tiempo utilizado en ordenar arreglo de las Xs: "+total);
171     // Rellenar lista de puntos altos con valores disponibles
172     yr = new Punto[equis.length]; //almacenar todos los puntos altos para cada x
173     for(int i=0; i<equis.length; i++){
174       if(!ptosPrimeraFila.isEmpty() && ptosPrimeraFila.get(0).getX()==equis[i]){
175         yr[i] = ptosPrimeraFila.get(0);
176         ptosPrimeraFila.remove(0);
177       }else {
178         yr[i]=(new Punto(equis[i],yMayor,false));
179       }
180     }
181     br.close();
182     return nroPuntosAProcesar;
183   }
```

```
184
185   private static Punto[] leeFila(BufferedReader br, double[] equis) throws
186           FileNotFoundException, IOException {
187     String linea;
188     StringTokenizer stk;
189     double y,x,yAntiguo;
190     int posActual=equis.length;
191     Punto[] fila = new Punto[equis.length];
192     if (lineaNoProcesada == null) {  // El archivo ya fue leido en su totalidad
193       return new Punto[0];
194     }
195     else {
196       if (lineaNoProcesada.equals("")) { // El archivo recien se comienza a leer
197         linea = br.readLine();
198         if (linea == null) {        // El archivo esta vacio
199           return new Punto[0];
200         }
201       // El archivo se esta leyendo y queda una linea sin procesar en la lectura
202       //de la fila anterior (correspondiente a una nueva fila)
203       }else {
204         linea = lineaNoProcesada;
205       }
206     }
207     stk = new StringTokenizer(linea);
208     y = Double.parseDouble(stk.nextToken());
209     x = Double.parseDouble(stk.nextToken());
210     // Se lee el id, pero no se procesa por no ser relevante en este caso
211     Integer.parseInt(stk.nextToken());
212     yAntiguo = y;
213     while (yAntiguo == y && linea != null) {
214       // Agrega punto leido a la fila
215       posActual = agregaPunto(fila, x, y, posActual-1, equis);
216       linea = br.readLine();
217       if (linea != null) {
218         stk = new StringTokenizer(linea);
219         y = Double.parseDouble(stk.nextToken());
220         x = Double.parseDouble(stk.nextToken());
221         Integer.parseInt(stk.nextToken())   // Se lee el id, pero no se procesa
222       }
223     }
224     rellenaFila(fila, yAntiguo, posActual-1, equis);
225     lineaNoProcesada = linea;
226     return fila;
227   }
228
229   private static int agregaPunto(Punto[] fila, double x, double y, int inicio,
```

```
230           double[] equis){
231     // Precondicion: inicio>=0
232     // Agrega el punto en la posicion correspondiente desde inicio
233     int i=inicio;
234     while (i>=0 && equis[i] != x){ //se crea cada elemento de la fila
235       fila[i] = new Punto(equis[i],y,false);
236       i--;
237     }
238     if (i >= 0) {
239       fila[i] = new Punto(x,y,true);
240       return i;         // Retorna ubicacion donde se agrego el punto
241     }
242     // Indica error, pues x del punto a agregar no corresponde a los
243     //puntos hallados antes en el mismo archivo
244     return -1;
245   }
246
247   private static void rellenaFila(Punto[] fila, double y, int inicio,
248           double[] equis) {
249     // Rellena con puntos ficticios hasta el final (primer elemento) de la fila
250     for (int i=inicio; i>=0; i--) {
251       fila[i] = new Punto(equis[i],y,false);
252     }
253   }
254
255   private static int obtieneNroPuntos(Punto[] fila) {
256     int nroPuntos=0;
257     for (Punto posiblePunto : fila) {
258       if (posiblePunto.esPunto()) {
259         nroPuntos++;
260       }
261     }
262     return nroPuntos;
263   }
264
265   private static void despliegaFila(Punto[] fila) {
266     for (Punto posiblePunto : fila) {
267       System.out.print(posiblePunto + "-" + posiblePunto.esPuntoStr() + "; ");
268     }
269   }
270
271   private static void actualizaYr(Punto[] yr, Punto[] fila) {
272     // Actualiza puntos altos
273     for (int i = 0; i < fila.length; i++) {
274       if(fila[i].esPunto() && fila[i].getX() > equis[0]){
275         /* La condicion fila[i].getX() > equis[0] deja fuera los puntos
```

```
276             presentes en el extremo izquierdo de los datos de este modo no son
277             considerados puntos altos a fin de no distorsionar la escalera,
278           dado que ese extremo es parte de rectangulos maximales
279         */
280         yr[i] = fila[i];
281       }
282     }
283   }
284
285   public static void generaEscalera(int iDequis, Punto pInfDer, Punto[] yr,
286           Stack<Punto> escalera) {
287     // Genera escalera hasta el punto eQuis[i] de la primera fila
288     Punto ultimoPuntoPop = null;
289     if(escalera.isEmpty() /*&& yr[iDequis].getY() >= pInfDer.getY()*/) {
290       // Caso 0, inicia la escalera
291       /* La segunda condicion no se observa cuando pueda presentarse al comenzar
292         a generar la escalera, pues siempre se empila como inicio del primer
293         peldao el extremo superior izquierdo del area de puntos (0,yMayor)
294       */
295       escalera.push(yr[iDequis]);
296     }else{
297       // Caso 1, y del ultimo punto (tope de la pila) < y del nuevo punto
298       if(escalera.peek().getY() < yr[iDequis].getY()) {
299         if (yr[iDequis].getY() >= pInfDer.getY()) {
300           escalera.push(yr[iDequis]);
301         }
302       }else{
303         // Caso 2, y del ltimo punto (tope de la pila) > y del nuevo punto
304         if(escalera.peek().getY() > yr[iDequis].getY()) {
305           while(!escalera.empty()&& escalera.peek().getY()>=yr[iDequis].getY()){
306             /* La condicion: escalera.peek().getY() >= yr[iDequis].getY()
307               elimina la posibilidad que dos puntos con igual coordenada y
308               sean parte de la escalera
309             */
310             ultimoPuntoPop = escalera.pop();
311           }
312           if (yr[iDequis].getY() >= pInfDer.getY()) {
313             // Sube desde el ultimo peldao de la escalera hasta y de yr
314             escalera.push(new Punto(ultimoPuntoPop.getX(), yr[iDequis].getY(),
315                     false));
316             /* Hace que la escalera llegue hasta el tope superior del area de
317             datos, el que constituye un posible lado de rectangulos maximales
318             */
319             escalera.push(new Punto(yr[iDequis].getX(),yMayor,false));
320           }
321         }else {
```

```
322           // Caso 3, escalera.tope.y = yr[iDequis].y
323           // Se conserva el punto de la pila y se desecha yr
324         }
325       }
326     }
327   }
328
329   private static int obtieneRectangulos(int iDequis, Punto[] primeraFila,
330      Punto[] segundaFila, double xMaxDer, Punto[] yr, Stack<Punto> pilaAltura) {
331     // Precondicion: 0 <= iDequis <= equis.length-2, pila no vacia
332     int nroRectObtenidos = 0;
333     Stack<Punto> stack = new Stack<Punto>();
334     stack.addAll(pilaAltura);
335     Punto pInfDer = primeraFila[iDequis];
336     Punto pInfDerDelBorde = segundaFila[iDequis+1];
337     // y* es la coordenada y del punto del borde derecho mas bajo desde la
338     //fila 1 hasta la fila pInfDerDelBorde
339     Punto yrDer=yr[iDequis+1]; // Corresponde a y*
340     // x* es la coordenada x del borde inferior mas a la derecha desde la
341     //columna 1 hasta la columna iDequis
342     if (segundaFila[iDequis].esPunto()) {
343       ptoBajoIzq = segundaFila[iDequis];        // Corresponde a x*
344     }
345     System.out.println("Punto inferior derecho (bajo analisis): " + pInfDer);
346     System.out.println("Punto inferior derecho del borde: " + pInfDerDelBorde);
347     if (ptoBajoIzq != null) {
348       System.out.println("x*="+(int)ptoBajoIzq.getX()+";y*="+(int)yrDer.getY());
349     }else {
350       System.out.println("No existe un x*" + "; y*=" + (int)yrDer.getY());
351     }
352     // Genera rectangulos, si es posible
353     if (pInfDerDelBorde.getX() == xMaxDer && pInfDerDelBorde.getY() == yMenor) {
354       // Ultimo punto de la "matriz (inferior derecho)"
355       System.out.print("Rectangulo(s) con el punto del extremo inferior derecho"
356               + " de la matriz ( _| ) ==>");
357       nroRectObtenidos += construirRectangulosConTodaLaEscalera(stack,
358               pInfDerDelBorde);
359       System.out.println();
360     } else {
361       if (pInfDerDelBorde.getX() == xMaxDer && ptoBajoIzq!=null) {
362         /* Penultimo punto de la fila bajo anlisis (punto en el
363           [extremo derecho-1] de la primera fila)
364           Luego no es necesario verificar: yMax >= pInfDer.getY()
365         */
366         System.out.print("Rectangulo(s) con el punto del extremo derecho "
367                 + "de la fila ( -| ) ==>");
```

```
368         nroRectObtenidos += construirRectangulosConBordeExtremoDerecho(stack,
369                 pInfDer, pInfDerDelBorde, ptoBajoIzq.getX());
370         System.out.println();
371       } else {
372         /* Se elimino la verificacipon que en la posicion de pInfDer se almacene
373           un punto, pues evita la formacion de rectangulos maximales que lo
374           tengan como lado superior
375         */
376         if (pInfDerDelBorde.getY() == yMenor){
377         //El punto de la fila bajo analisis se ubica en el [extremo inferior-1],
378         //no se debe verificar con ptoBajoIzq
379           if (pInfDer.getY() <= yrDer.getY()){
380             System.out.print("Rectangulos cuyo punto inferior derecho se ubica "
381                     + "en el extremo inferior de la matriz ( __ ) ==>");
382             nroRectObtenidos += construirRectangulosConBordeExtremoInferior (
383                     stack, pInfDer, pInfDerDelBorde, yrDer.getY());
384             System.out.println();
385           }
386         }else{
387           /* Si no es un caso limite, se debe cumplir que debe existir un punto
388             en la matriz entre:
389            * la coordenada Y del punto tope de la pila y la coordenada Y del
390                  punto de la primera fila bajo analisis,
391            * la coordenada X del punto tope de la pila y la coordinada X del
392                  punto de la primera fila bajo analisis.
393            */
394           System.out.print("Rectangulo(s) cuyo punto inferior derecho se "
395                   + "encuentra al interior de la matriz (no es borde) ==>");
396           if (ptoBajoIzq != null && yrDer.getY() >= pInfDer.getY() &&
397                   ptoBajoIzq.getX() <= pInfDer.getX()) {
398             nroRectObtenidos += construirRectangulosEnInteriorDeLaRegion(
399               stack, pInfDer, pInfDerDelBorde, ptoBajoIzq.getX(), yrDer.getY());
400           }
401           System.out.println();
402         }
403       }
404     }
405     return nroRectObtenidos;
406   }
407
408   private static int construirRectangulosConTodaLaEscalera(
409           Stack<Punto> escalera, Punto pInfDerDelBorde) {
410     Punto puntoPila;
411     int nroRectObtenidos=0;
412     while (escalera.size() >= 1 ) {
413       puntoPila = escalera.pop();
```

```
414       if (generarRectangulo(puntoPila, pInfDerDelBorde))
415         nroRectObtenidos++;
416     }
417       return nroRectObtenidos;
418   }
419
420   private static int construirRectangulosConBordeExtremoDerecho(
421      Stack<Punto> escalera, Punto pInfDer, Punto pInfDerDelBorde, double xMax){
422     Punto puntoPila;
423     int nroRectObtenidos=0;
424     while (escalera.size() >= 1) {
425       puntoPila = escalera.pop();
426       if (puntoPila.getX() < xMax && pInfDer.getX() >= xMax) {
427         if (generarRectangulo(puntoPila, pInfDerDelBorde))
428           nroRectObtenidos++;
429       }
430     }
431     return nroRectObtenidos;
432   }
433
434   public static int construirRectangulosConBordeExtremoInferior (
435      Stack<Punto> escalera, Punto pInfDer, Punto pInfDerDelBorde, double yMax) {
436     // pInfDerDelBorde.getY() <= yMax
437     Punto puntoPila;
438     int nroRectObtenidos=0;
439     while (escalera.size() >= 1) {
440       puntoPila = escalera.pop();
441       if (puntoPila.getY() > yMax && pInfDer.getY() <= yMax) {
442         if (generarRectangulo(puntoPila, pInfDerDelBorde))
443           nroRectObtenidos++;
444       }
445     }
446     return nroRectObtenidos;
447   }
448
449   private static int construirRectangulosEnInteriorDeLaRegion(
450         Stack<Punto> escalera, Punto pInfDer, Punto pInfDerDelBorde,
451           double xMax, double yMax) {
452     int nroRectObtenidos=0;
453     Punto puntoPila;
454     while (escalera.size() >= 1) {
455       puntoPila = escalera.pop();
456       if (puntoPila.getX() < xMax && pInfDer.getX() >= xMax && puntoPila.getY()
457               > yMax && pInfDer.getY() <= yMax) {
458         if (generarRectangulo(puntoPila, pInfDerDelBorde))
459           nroRectObtenidos++;
```

```
460       }
461     }
462     return nroRectObtenidos;
463   }
464
465   private static boolean generarRectangulo(Punto pSupIzq, Punto pInfDer) {
466     if (pSupIzq.getX() !=  pInfDer.getX() &&  pSupIzq.getY() != pInfDer.getY()){
467     // Verifica que no sea punto ni linea.
468     // se debe verificar que el rectangulo contenga al punto q
469       if(contienePuntoq(pSupIzq, pInfDer)){
470         System.out.print("{(" + (int)pSupIzq.getX() + ", " + (int)pSupIzq.getY()
471             + ") - " + "(" +(int)pInfDer.getX() + ", " + (int)pInfDer.getY() +
472                 ")} *** ");
473         return true;
474       }else{
475         return false;
476       }
477     } else {
478       return false;
479     }
480   }
481
482   private static boolean contienePuntoq(Punto pSupIzq, Punto pInfDer){
483     double nuevaArea ;
484     if(pSupIzq.getX()< puntoQuery.getX() && pSupIzq.getY() > puntoQuery.getY()){
485       if(pInfDer.getX()>puntoQuery.getX() && pInfDer.getY() <puntoQuery.getY()){
486         //se debe verificar que el area sea mayor a otra obtenida anteriormente
487         nuevaArea=(pInfDer.getX() - pSupIzq.getX()) *
488                 (pSupIzq.getY() - pInfDer.getY());
489         if(nuevaArea > area){
490           area = nuevaArea;
491           return true;
492         }else
493           return false;
494       }else{
495         return false;
496       }
497     }else{
498       return false;
499     }
500   }
501 }
```

Apéndice B

Codigo CrearSetDatos

CreadorPuntos.java

```java
package creador.puntos;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.Random;
/**
 *
 * @author Felipe
 */
public class CreadorPuntos {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws IOException {
        System.out.println("Ingresar cant Y");
        BufferedReader brQuery = new BufferedReader(
                new InputStreamReader(System.in));
        String my = brQuery.readLine();
        int mY = Integer.parseInt(my);
        String nombre = "100p2";
        int mx = 100;
        for(int u=0;u<1;u++){
        boolean[][] matriz = new boolean[mx][mY];
        //rellanar matriz
        Random ram = new Random();
        int x = (int)(ram.nextDouble()*mx);
        int y = (int)(ram.nextDouble()*mY);
        int i = 0;
        int total =0;

            while(i<100){
                if(matriz[x][y] == false){
                    matriz[x][y] = true;
                    i++;
                }
                x = pedirNumeroX(mx);
                y = pedirNumeroY(mY);
            }

```

```
46          for(int j=0;j<mx;j++){
47              for(int k=0;k<mY;k++){
48                  if(matriz[j][k]==true){
49                      total++;
50              }
51          }
52
53
54      }
55          System.out.println("matriz "+ total);
56          total = 0;
57          imprimirMatriz(matriz, mY, u, nombre, mx);
58          imprimirMatrizOR(matriz, mY, u, nombre, mx);
59          }
60      }
61      public static int pedirNumeroX(int mx){
62          Random ram = new Random();
63          int x = (int)(ram.nextDouble()*mx);
64          return x;
65      }
66      public static int pedirNumeroY(int mY){
67        Random ram = new Random();
68        int y = (int)(ram.nextDouble()*mY);
69        return y;
70      }
71      public static void imprimirMatriz(boolean matriz[][], int mY, int u,
72              String nombre, int mx){
73          File f;
74          f = new File(nombre+u+".txt");
75          //Escritura
76          try{
77              FileWriter w = new FileWriter(f);
78              BufferedWriter bw = new BufferedWriter(w);
79              PrintWriter wr = new PrintWriter(bw);
80              for(int i=0;i<mx;i++){
81                  for(int j=0;j<mY;j++){
82                      if(matriz[i][j]==true){
83                          wr.write(i+" "+j);
84                          bw.newLine();
85                      }
86                  }
87              }
88              wr.close();
89              bw.close();
90          }catch(IOException e){
91              System.out.println(e);
```

```
92          }
93      }
94      public static void imprimirMatrizOR(boolean matriz[][], int mY,int u,
95              String nombre, int mx){
96          File f;
97          f = new File(nombre+"ORD"+u+".txt");
98          long id = 0;
99          //Escritura
100         try{
101             FileWriter w = new FileWriter(f);
102             BufferedWriter bw = new BufferedWriter(w);
103             PrintWriter wr = new PrintWriter(bw);
104             for(int i=0;i<mx;i++){
105                 for(int j=0;j<mY;j++){
106                     if(matriz[i][j]==true){
107                         wr.write(j+" "+i+" "+id);
108                         bw.newLine();
109                        id++;
110
111                     }
112                 }
113             }
114             wr.close();
115             bw.close();
116         }catch(IOException e){
117             System.out.println(e);
118         }
119     }
120 }
```

Apéndice C

Ejecución y resultados

En este apéndice se mostrarán las imagenes relacionadas a la ejecución del algoritmo y el resultado que se obtiene al graficar el rectángulo maximal vacío que entrega el programa, sobre el conjunto de datos.

```
[flara@localhost pruebas de AREMAV]$ javac qAREMAV.java
[flara@localhost pruebas de AREMAV]$ java qAREMAV
```

Figura C.1: Paso 1: compilar y ejecutar el algoritmo.

```
[flara@localhost pruebas de AREMAV]$ javac qAREMAV.java
[flara@localhost pruebas de AREMAV]$ java qAREMAV
Ingresar nombre del archivo:
1000t
Ingresar numero de archivo
1
```

Figura C.2: Paso 2: ingresar archivo de puntos.

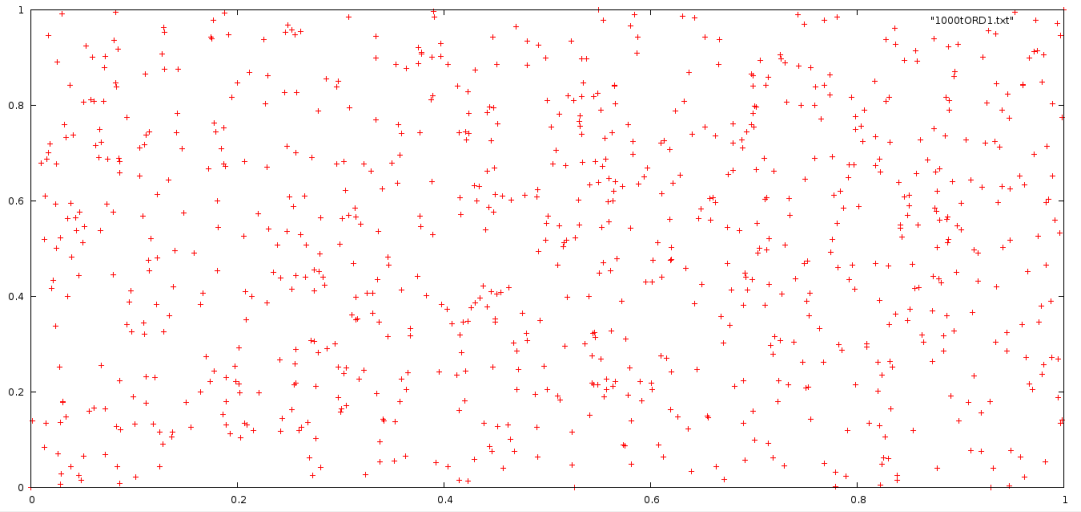


Figura C.3: Archivo de puntos mostrado de forma gráfica

```
[flara@localhost pruebas de AREMAV]$ javac qAREMAV.java
[flara@localhost pruebas de AREMAV]$ java qAREMAV
Ingresar nombre del archivo:
1000t
Ingresar numero de archivo
1
tiempo en ordenar ExternalMerge :4784 [ns]
Tiempo utilizado en ordenar arreglo de las Xs: 18 [ns]

Nro de puntos a procesar: 1002
xMaxDer=1; yMenor=0
```

Figura C.4: Paso 3: se muestran los tiempos y datos que lee el algortimo

```
[flara@localhost pruebas de AREMAV]$ javac qAREMAV.java
[flara@localhost pruebas de AREMAV]$ java qAREMAV
Ingresar nombre del archivo:
1000t
Ingresar numero de archivo
1
tiempo en ordenar ExternalMerge :4784 [ns]
Tiempo utilizado en ordenar arreglo de las Xs: 18 [ns]

Nro de puntos a procesar: 1002
xMaxDer=1; yMenor=0
X del punto a analizar
0.45
y del punto a analizar
0.45
```

Figura C.5: Paso 4: Se debe ingresar x e y del punto q

```
[flara@localhost pruebas de AREMAV]$ javac qAREMAV.java
[flara@localhost pruebas de AREMAV]$ java qAREMAV
Ingresar nombre del archivo:
1000t
Ingresar numero de archivo
1
tiempo en ordenar ExternalMerge :4509 [ns]
Tiempo utilizado en ordenar arreglo de las Xs: 18 [ns]

Nro de puntos a procesar: 1002
xMaxDer=1; yMenor=0
X del punto a analizar
0.45
y del punto a analizar
0.45

Rectangulos que se generan :
{(0.27868623602976095, 0.46523347966131223) - (0.5611894403915315, 0.44913532768332565)}, Superficie =  0.0045477795180839855
{(0.3463248616921768, 0.5015790030567988) - (0.4911212458694044, 0.4414625256213214)}, Superficie =  0.008704648562129026
{(0.38895355883030147, 0.5395731333892236) - (0.4911212458694044, 0.4228477413908135)}, Superficie =  0.01192556331921018
{(0.37481293710407726, 0.5288146263055341) - (0.4911212458694044, 0.4228477413908135)}, Superficie =  0.012324829169561215

Numero total de rectangulos generados: 4
Tiempo total en ejecucion de qAREMAV : 411 [ns]
[flara@localhost pruebas de AREMAV]$
```

Figura C.6: Paso 5: se muestran los rectángulos maximales vacíos que contienen al punto q

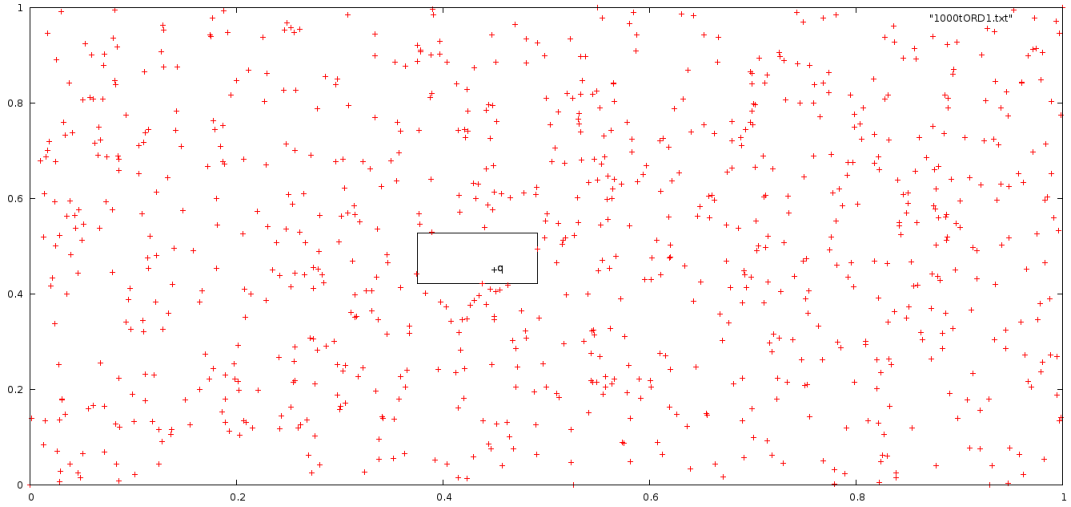


Figura C.7: Rectángulo maximal vacío que contiene a q de forma gráfica