UNIVERSIDAD DEL BÍO-BÍO

FACULTAD DE CIENCIAS EMPRESARIALES
MAGISTER EN CIENCIAS DE LA COMPUTACIÓN

Algoritmos geométricos para grandes conjuntos de datos espaciales[1]

Felipe José Lara Ramírez

Tesis para optar al grado de Magíster en Ciencias de la Computcación

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

Chillán, Julio de 2016

[1]Trabajo parcialmente financiado por el proyecto de Investigación interno DIUBB 144319 2/R y por el Grupo de Investigación "Bases de Datos"

Resumen

En esta tesis se describen el diseño, implementación y experimentación de diferentes algoritmos para resolver los problemas de: (i) encontrar todos los rectángulos vacíos maximales que existen en un conjunto de puntos S localizados en una región $R \subseteq \mathbb{R}^2$ y (ii) encontrar el rectángulo maximal vacío localizado en una región $R \subseteq R^2$ y que en su interior solo contenga un punto q dado como consulta. En ambos problemas se asume que no existe suficiente memoria para almacenar todos los objetos del conjunto S . El algoritmo *AREMAV*, el cual resuelve el problema (i) se extendió para resolver el problema (ii). 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. Como resultado, se logró diseñar e implementar dos nuevos algoritmos. El primero llamado algoritmo $qMER$, es un algoritmo que se compone de 3 etapas, donde elimina los puntos que son dominados por los puntos más cercanos a q y luego utiliza un algoritmo de geometría computacional para obtener el rectángulo maximal vacío que solo contiene a q. El segundo algoritmo se llamó algoritmo MER, este algoritmo separa el conjunto de puntos en 4 subregiones, donde se utiliza *AREMAV* en cada una de estas subregiones y se finaliza uniendo las soluciones. Estos algoritmos son más eficientes en cuanto al tiempo de ejecución comparados de forma experimental con otros algoritmos encontrados en la literatura. Con el objeto de verificar la eficiencia de los algoritmos se muestra los resultados de una serie de experimentos considerando datos sintéticos y reales. Los resultados de los experimentos fueron contrastados con resultados de experimentos similares reportados en la literatura, mostrando un mejor comportamiento.

Índice general

Lista de figuras **3**

Lista de tablas **5**

1. Introducción **6**
- 1.1. Introducción 6

2. Descripción del problema y trabajos relacionados **9**
- 2.1. Problemas *MER* y *qMER* 9
- 2.2. Motivación 10
- 2.3. Trabajos relacionados 11

3. Algoritmos *AREMAV* y *qAREMAV* **13**
- 3.1. Precondiciones del algoritmo *AREMAV* 13
- 3.2. Descripción de AREMAV 13
  - 3.2.1. Construyendo la escalera 16
  - 3.2.2. Generar rectángulos vacíos maximales 17
- 3.3. Descripción de qAREMAV 19

4. Algoritmos para resolver *qMER* **22**
- 4.1. Preliminares 22
- 4.2. Algoritmo *qAREMAVOpti* 23
- 4.3. Análisis de complejidad 24
- 4.4. Algoritmo qMER 24
- 4.5. Análisis de complejidad 27
- 4.6. Evaluación experimental de los algoritmos 27

5. Algoritmo para resolver el problema *MER* **30**
- 5.1. Preliminares 30
- 5.2. Algoritmo MER 31
- 5.3. Complejidad del algoritmo 35
- 5.4. Evaluación experimental del algoritmo MER 36

6. Conclusiones **43**

Índice de figuras

2.1. Problemas *MER* y *qMER* [8] 9

3.1. Conjunto de puntos D (a) y su representación como matriz M (b) 14
3.2. Funcionamiento de variables primera fila, segunda fila y actualización de los puntos altos. 15
3.3. *Escalera maximal* para (x, y) 16
3.4. Diferentes casos para constrir una escalera 16
3.5. Obteniendo los rectángulos vacíos maximales desde una escalera 17
3.6. Conjunto de puntos D (a) y su representación como matriz M (b) 20

4.1. Análisis de casos para algoritmo *qAREMAVOpti* 23
4.2. Zonas de dominancia 24
4.3. Algoritmo "k vecinos" con k=2 26
4.4. Influencia de distintos valores de k en el tiempo de filtrado de $qMER$ (etapas 1 y 2) y en el tamaño de S' 28
4.5. Comparación de algoritmos respecto al tiempo de ejecución. 29
4.6. Tiempo de ejecucion de $qMER$. 29

5.1. Conjunto de puntos original. 31
5.2. División del conjunto original en 4 cuadrantes. 31
5.3. Ejemplo de una partición considerando $d = 2$ y $b = 3$ 33
5.4. Algoritmo *AREMAV* aplicado al conjunto 1. 33
5.5. Cuadarante 3 después de haber sido analizado el cuadrante 1. 34
5.6. MER encontrado en el conjunto 3. 35
5.7. MER del conjunto inicial. 35
5.8. Distribución uniforme con densidad de 20 %. 36
5.9. Distribución Zipf con 125.000 puntos 36
5.10. Distribución cluster con 125.000 puntos 37
5.11. Conjunto de puntos reales sobre Norte América. 37
5.12. Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos de distintas densidades 38
5.13. Tiempo de ejecución(segundos) de algoritmos MER y *AREMAV* sobre conjuntos con datos reales. 39

5.14. Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución cluster. 40
5.15. Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución Zipf. 41
5.16. Tiempo de ejecución (segundos) algoritmos MER y *AREMAV* sobre conjuntos con distribución uniforme y con densidad 20 % 42

Índice de tablas

5.1. Tiempo de ejecución(segundos) de algoritmos MER y *AREMAV* sobre conjuntos con datos reales. 38
5.2. Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución cluster. 39
5.3. Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución Zipf. 40
5.4. Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución uniforme y densidad de 20 %. 41

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 relativamente reciente, pues sus primeros resultados se pueden apreciar en la década del 80. Sea S_1 y S_2 dos conjuntos de puntos ubicados en regiones $R_1 \subseteq \mathbb{R}^d$ (típicamente $d = 2$) y $R_2 \subseteq \mathbb{R}^d$ 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 que incluye a q de mayor área dentro de R_1, (v) dado un conjunto de puntos 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. Por ejemplo, si se tiene varios lugares dentro de una ciudad para instalar una cafeteria y se necesita instalar lo más alejada posible de otros locales similares, se puede modelar como problema (iv) siendo q el lugar donde se desea instalar y S_1 los demás establecimientos. Otro ejemplo es el localizar los mayores sectores vacíos dentro de una ciudad para construir un estacionamiento, este se modela como un problema (v).

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, y que está diseñado para la indexación dinámica de un conjunto de objetos geométricos d-dimensional [1]. En [2] se propone un algoritmo que resuelve el problema (i), en [3] se describe

un algoritmo para el problema (*ii*), en [4, 5, 7] se aportan algoritmos para el problema (*iii*) y en [7, 8] se proponen soluciones para el problema (*iv*), en [9, 10, 11] se describen algoritmos para solucionar el problema (*v*).

En este informe se describe la implementación de dos algortimos para resolver el problema (*iv*), el que llamaremos *qMER* (Query Maximal Empty Rectangle) y un algoritmo para resolver el problema (*v*) propuesto en [6], el que llamaremos *MER* (Maximal Empty Rectangle).

Definición de los problemas. El problema *qMER* se define como sigue: Sea S un conjunto de puntos sobre un rectángulo $R \subseteq \mathbb{R}^2$ cuyos lados son paralelos a los ejes del plano, y $q \notin S$ tal que $q \cap R \neq \emptyset$. Un rectángulo es llamado rectángulo vacío maximal que contiene solo a q (*qMER*) si: (a) sus lados son paralelos a los lados de R, (b) está contenido completamente en R, (c) no contiene otros puntos de S y (d) cada uno de sus lados o arcos contiene, al menos, un punto de S o está contenido en un lado de R y (e) contiene a q [8]. Nuestro problema (*qMER*) consiste en encontrar el rectángulo *QMER* de mayor área.

En cambio, el problema *MER* se define como: Sea S un conjunto de puntos sobre un rectángulo $R \subseteq \mathbb{R}^2$ cuyos lados son paralelos a los ejes del plano. Un rectángulo es llamado rectángulo vacío maximal si: (a) sus lados son paralelos a los lados de R, (b) está contenido completamente en R, (c) no contiene otros puntos de S y (d) cada uno de sus lados o arcos contiene, al menos, un punto de S o está contenido en un lado de R. El problema *MER* consiste en encontrar el rectángulo vacío maximal de mayor área.

La diferencia entre los problemas *MER* y *qMER* está dada en la letra (e) ya que en el problema *MER* no debe haber ningun punto dentro del rectángulo, en cambio *qMER* debe tener solo a q en su interior.

Hipótesis Es posible diseñar algoritmos que resuelvan los problemas *MER* y *qMER* para grandes conjuntos de datos espaciales (en memoria secundaria y que no caben en memoria principal) y que sean eficientes en términos de tiempo y almacenamiento.

Objetivos El objetivo de esta tesis es diseñar, implementar y evaluar experimentalmente algoritmos eficientes, que sean capaces de encontrar el rectángulo vacío maximal y que contenga un punto específico dentro del espacio en el problema *qMER*, y el de encontrar el rectángulo vacío maximal en el problema *MER*.

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 una revisión de la literatura 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 en detalle el algoritmo propuesto en [6] que en adelante llamamos *AREMAV* que resuelve el problema *MER* y su modificación llamada *qAREMAV* la que resuelve el problema *qMER*. Se explican las ideas detrás de ambos algoritmos y aspectos de eficiencia. El Capítulo 4 muestra en detalle el diseño e implementación (en lenguaje JAVA) de los algoritmos que resuelven los problemas *MER* y *qMER* junto a sus análisis de complejidades. En el Capítulo 5 se exponen los resultados experimentales de los algoritmos que resuelven ambos problemas. Finalmente, en el Capítulo 6 se detallan las conclusiones y trabajos futuros.

Capítulo 2

Descripción del problema y trabajos relacionados

En este capítulo definiremos de manera formal el problema *MER* y *qMER*, se presenta una serie de aplicaciones que sirven de motivación y se discuten los trabajos relacionados disponibles en la literatura.

2.1. Problemas *MER* y *qMER*

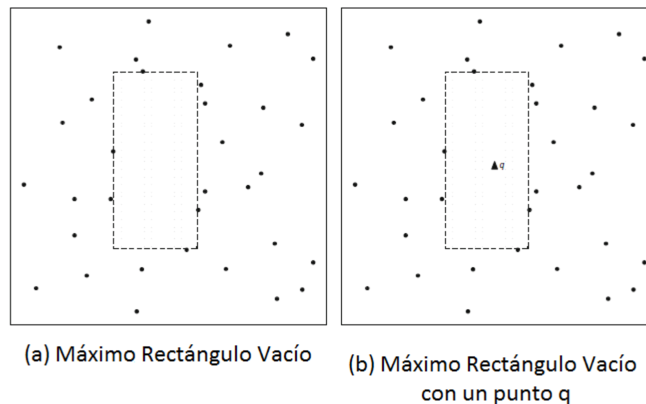

Figura 2.1: Problemas *MER* y *qMER* [8]

Dado un conjunto de puntos S ubicados en un subespacio $R \subseteq R^2$ definimos los siguientes problemas: (i) Encontrar el rectángulo vacío maximal, de lados paralelos a los ejes que limitan a R y que se encuentran contenidos en R o que sus lados contienen por lo menos un punto de S y no debe contener otros puntos de S (problema *MER*, ver Figura 2.1 (a)) y (ii) dado un punto $q \notin S$ encontrar el rectángulo maximal de mayor área, de lados paralelos a R, que se encuentra

contenido en R o que sus lados contienen por lo menos un punto de S y que solo contiene en su interior a q (problema *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 se adelantó, el objetivo principal de este trabajo es diseñar e implementar un algoritmo que resuelva el problema *MER* y sea más eficiente que el propuesto en [6]. Un segundo objetivo es proponer e implementar un algoritmo para resolver el problema *qMER* y que sea más eficiente que el algoritmo inspirado en *AREMAV* [19].

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.) georreferenciados. Puede ser interesante resolver eficientemente las siguientes consultas: (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 toxicidad (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 contexto 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 se define mediante 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 quirúrgicas, es posible descubrir que no hay información sobre trasplantes de cara en la base de datos antes del año 2008, siendo q los transplantes de cara y S los tipos de operaciones realizadas por el hospital desde su creación. 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 [8].

2.3. Trabajos relacionados

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 extensamente. El primer trabajo conocido fue el 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 necesita de los puntos ordenados y compara los puntos entre sí, es de tiempo $O(n^2)$ y $O(n)$ de almacenamiento. El segundo es de tiempo esperado $O(n(\log^2 n))$ y $O(n)$ de almacenamiento, este algoritmo lee los puntos desordenados y los almacena en un heap semi dinámico (en adelante consideramos $\log n$ como $\log_2 n$).

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], donde utiliza un tiempo de $O(nlog^3 n)$ y almacenamiento $O(n)$. Orlowski [10] presenta un algoritmo que utiliza un tiempo $O(s \log n)$, donde s es el número de rectángulos vacíos maximales. Su algoritmo crea rectángulos utilizando dos puntos como vértices y luego los extiende hacia los lados hasta que se forma un *MER*. Este algoritmo tiene una complejidad de $O(n \log n + s)$ con s igual al número de rectángulos vacíos maximales. En otro trabajo más reciente Nandy et al.[13] propone un algoritmo que utiliza un tiempo de $O(n \log^2 n + s)$ y almacenamiento de $O(\log n)$ empleando un árbol de búsqueda prioritario (priority searchtree).

También hay trabajos enfocados en resolver el problema *MER* en 3 dimensiones. En este caso el algoritmo computa cubos vacíos maximales. En [14] y [15] se proponen algoritmos para resolver este problema.

En [16] y [17] se propone un algoritmo para resolver el problema *qMER*. Este algoritmo realiza un pre procesamiento 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 pre procesamiento utiliza un almacenamiento de $O(n \log^2 n)$ 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 significativa en términos de tiempo de pre procesamiento respecto a [16] y [17]. Específicamente, este algoritmo requiere un

espacio de almacenamiento de $O(n\alpha(n)\log^3 n)$ 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 término de $\alpha(n)$ es la inversa de la función de Ackermann [18]. Estos algoritmos consideran que todos los objetos se encuentran en memoria principal.

Más recientemente en [7] y [8] se proponen algoritmos para resolver el problema *qMER* y que consideran limitaciones de memoria principal y asumen que los objetos residen en memoria secundaria en una estructura de datos multidimensional R-tree. Estos algoritmos amplían las capacidades del R-tree. Es claro que estos algoritmos no son adecuados cuando los objetos no se encuentran en un R-tree, pues el proceso de construcción consume mucho tiempo.

Sin embargo, existen muchos escenarios en donde los objetos considerados no caben en memoria principal y se encuentran almacenados en un raw files. Edmonds et al.[6] proponen un algoritmo para obtener los rectángulos vacíos maximales (problema *MER*) 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* y su adaptación para que resuelva $qMER$ [19].

Capítulo 3

Algoritmos *AREMAV* y *qAREMAV*

Tal como anticipamos en el capítulo anterior, el algoritmo *AREMAV* propuesto en Edmons et al.[6] resuelve el problema *MER*. En este capítulo se describe de manera detallada los algoritmo *AREMAV* y *qAREMAV*. Estos algoritmos requieren que los datos entren en un orden específico, lo que permite contar con un algoritmo con tiempo de ejecución $O(|X| \times |Y|)$. 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 *AREMAV*

El algoritmo *AREMAV* procesa los datos a partir de un archivo sin formato (raw file), el cual contiene los puntos a procesar en la forma (x, y). Este algoritmo requiere de una serie de pasos previos para que pueda ser ejecutado, al igual que posee limitaciones a la hora de generar los rectángulos vacíos maximales. Los pasos previos y limitaciones son:

1. Un raw file que contenga todos los puntos que se analizarán ordenados de forma descendente respecto de y y, como segundo criterio, ordenados de forma descendente respecto de x.

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

3. 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 de AREMAV

Previo a esta investigación, se implementó un algoritmo [19], que resuelve el problema *MER* basado en [6], ya que es el único algoritmo existente en la literatura que resuelve el problema de la forma que necesitamos, denominado *AREMAV*. Inspirado en este algoritmo se diseñó uno para resolver el problema *qMER* llamado *qAREMAV*, el que se encuentra detallado en [19].

A continuación se describe el algoritmo *AREMAV* (ver Algoritmo 1).

El algoritmo *AREMAV* es un algoritmo ingenuo que será utilizado como baseline para los demás algoritmos que se presentan. Este algoritmo busca todos los rectángulos vacíos maximales y luego muestra el mayor de ellos. Este algoritmo utiliza 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.

El algoritmo se puede explicar teniendo el conjunto de puntos D almacenado en un archivo sin estructura (raw file) como se puede observar en la figura 3.1(a), para luego considerar una matriz M, la cual es una representación del conjunto de puntos D, de esta manera, el conjunto de puntos puede ser visualizado como una matriz que contiene 1s donde existan puntos y 0s donde no los hay, como lo muestra la Figura 3.1(b). Esta matriz nunca se construye en la implementación del algoritmo, solo se muestra para facilitar la comprensión del mismo.

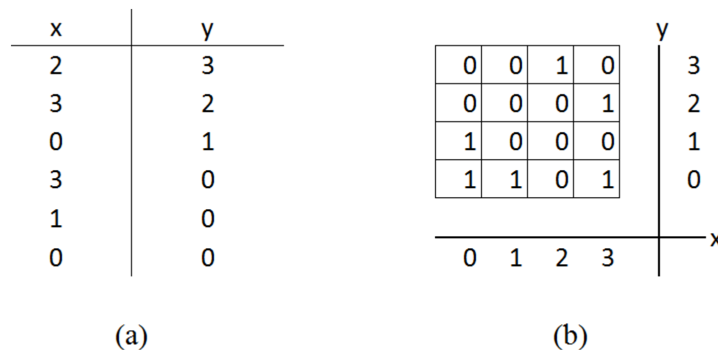

Figura 3.1: Conjunto de puntos D (a) y su representación como matriz M (b)

El algoritmo recorre solo una vez el conjunto de puntos y encuentra todos los rectángulos maximales dentro de los cuales solo existan celdas con valor 0 (los 1 solo servirán para limitar la forma del rectángulo). Para que esto sea posible, solo se llevan a memoria principal dos filas consecutivas de la matriz a la vez. Ambas filas se analizan 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. Llevar ambas filas a memoria principal sirve para que, durante el análisis de la primera fila, se encuentren los límites superiores de los rectángulos vacíos maximales, mientras que el análisis de la segunda fila sirve para detectar cuál es el límite inferior de los rectángulos vacíos maximales que puedan ser generados.

Cuando se hayan recorrido todas las celdas de la primera filas, se descarta esta fila y se reemplaza por la segunda fila. Por último, se trae desde memoria secundaria la fila inmediatamente inferior, la que se convierte en la nueva segunda fila (ver Figura 3.2).

Para conocer cuáles son los límites superiores que tienen los rectángulos vacíos maximales, se debe crear, para cada columna, una variable $y_r(x, y)$ (Figura 3.4, Algoritmo 1 línea 8, página 19), la cual es la coordenada anterior al 0 más alto de la columna x que comienza en (x, y) y se extiende hacia arriba. (ver Figura 3.4)

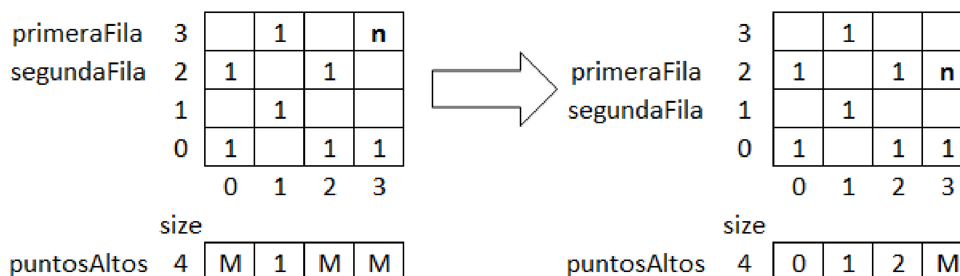

Figura 3.2: Funcionamiento de variables primera fila, segunda fila y actualización de los puntos altos.

En la Figura 3.2, se puede ver cómo cambian los valores de la primera fila y segunda fila a medida que se ha ido recorriendo la matriz. La variable 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 una estructura de datos Stack (pila) (ver Algoritmo 1 línea 3, página 19). La escalera es solo una parte de una *escalera maximal*, la cual es la *escalera* más grande que se puede formar en esa fila, ya que se extiende lo más a la derecha posible y lo más alto posible(ver Figura 3.3), ella contiene uno o muchos rectángulos vacíos maximales como se podrá ver más adelante cuando se explique cómo se obtienen estos rectángulos a partir de la *escalera maximal*.

La *escalera* se inicia vacía cada vez que se comience a analizar una nueva fila. La forma 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** (x, y) **contiene un 1**: Lo anterior significa que existe un punto en (x, y). Si esto sucede se eliminan todas las posiciones almacenadas en la pila quedando esta vacía, puesto que no es posible formar rectángulos vacíos maximales usando (x, y) como esquina inferior derecha interna de los rectángulos con bordes $x + 1$ e $y + 1$. Es más, la ordenada y de (x, y) constituirá el borde superior de algún rectángulo maximal que se pueda generar al examinar nuevas filas. Dado lo anterior, y_r debe actuaizarse para la columna x, siendo su nuevo valor y. Tómese como referencia la Figura 3.4(c).

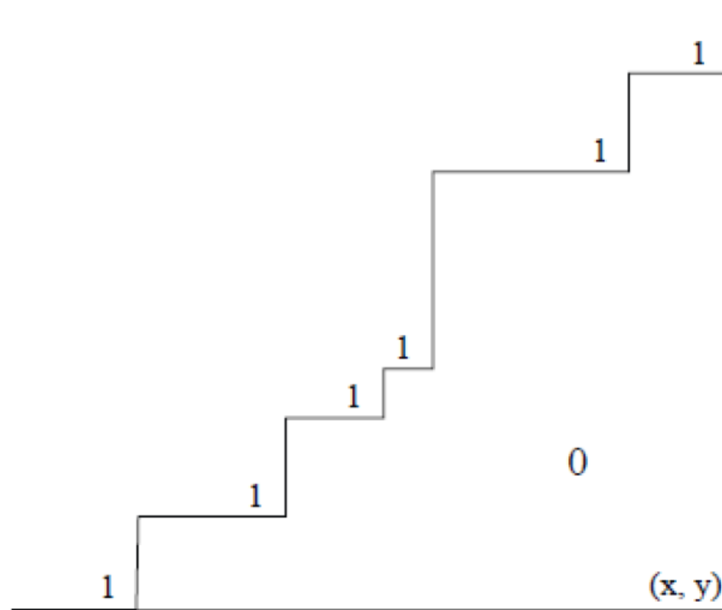

Figura 3.3: *Escalera maximal* para (x, y)

2. **La celda** (x, y) **contiene un 0**: si esto sucede, se debe seguir con la etapa siguiente para actualizar la escalera. Posteriormente, se forman los rectángulos vacíos maximales que sean posibles con bordes $(x+1)$ e $(y+1)$. El procedimiento a seguir en ambas etapas se describe más adelante.

3.2.1. Construyendo la escalera

Se debe recordar que se trabaja con 2 filas seguidas de la matriz, donde se analiza cada celda de la primera fila y por cada x se analiza su y_r correspondiente. Para construir la escalera es necesario utilizar el y_r y el peldaño más alto de la escalera $(y'_{r'})$, donde pueden ocurrir los siguientes 3 casos:

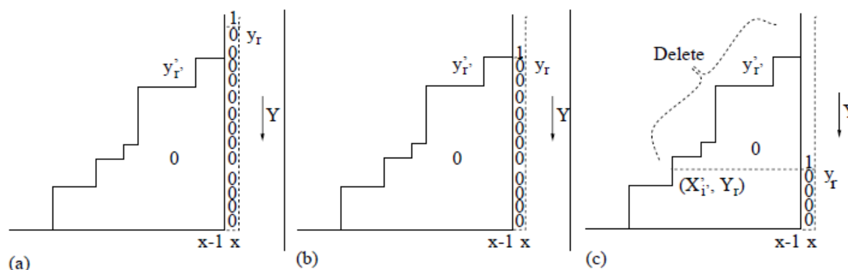

Figura 3.4: Diferentes casos para constrir una escalera

- Caso 1) $y_r < y'_{r'}$: (Figura 3.4(a)). Dado que y_r se ubica sobre el peldaño más alto de la escalera, este formará un nuevo peldaño de la misma. Para lo anterior, se procede a empilar el punto conformado por (x, y_r), quedando como el punto más alto del Stack (la *escalera* crece en altura).

- Caso 2) $y_r = y'_{r'}$: (Figura 3.4(b)). Dado que y_r tiene la misma altura que el peldaño más alto de la escalera, se procede a ampliar dicho peldaño (aumenta su longitud). En este caso no se empila nada, ya que el vertice superior izquierdo del peldaño seguirá siendo el mismo (el cual ya se encuentra empilado).

- Caso 3) $y_r > y'_{r'}$: (Figura 3.4(c)). Dado que y_r se ubica bajo el peldaño más alto de la escalera, se debe proceder a desempilar (eliminar peldaños de la escalera, reduciendo su altura) tantas veces como sea necesario hasta que la coordenada $y'_{r'}$ de la escalera cumpla que: $y_r \leq y'_{r'}$, de esta forma, dependiendo del valor que tome $y'_{r'}$, se aplica uno de los casos antes vistos.

Los casos antes descritos, se implementan en Algoritmo 1 líneas 10, 13 y 16 (página 19).

3.2.2. Generar rectángulos vacíos maximales

Ya que se tiene la escalera formada, solo falta cerrar el o los rectángulos vacíos maximales por debajo (Algoritmo 3 línea 23, página 19). Para determinar si esto es posible se debe revisar la fila siguiente a la que se está analizando (razón por la cual se mantiene en memoria principal, como se mencionó anteriormente). Se debe considerar dos casos, (i) donde un rectángulo limita por debajo con un 1, o (ii) donde el rectángulo no limita por debajo con un 1, por lo cual puede seguir creciendo hacia abajo.

Los rectángulos que se forman, 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 dentro de los límites de R (en el caso del algoritmo *AREMAV*) y en el caso del algoritmo *qAREMAV* debe contener en su interior solamente 0 y el punto q. El rectángulo vacío maximal que se genera, tendrá su vértice inferior derecho en el lugar de la matriz que se está analizando (puntos (x, y)) y su vértice superior izquierdo se encuentra en el peldaño más alto de la escalera ($y'_{r'}$) (Figura

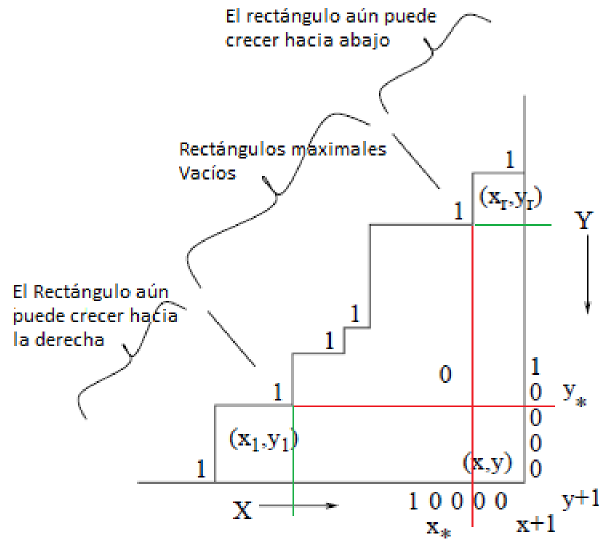

Figura 3.5: Obteniendo los rectángulos vacíos maximales desde una escalera

3.4), luego, se debe ir analizando hacia abajo los peldaños de la escalera de forma sucesiva 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 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) extendiéndose entre x_i y x, y sus lados izquierdo y derecho (paralelos al eje y) entre y_i e y, es maximal sí y solo sí $x_i < x_*$ 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.

Algoritmo 1 Algoritmo AREMAV

```
1: AREMAV(R, S){R son los puntos límite del conjunto de puntos, S el conjunto de puntos}
2: ordS= ExternalMergeSort(S) {ordS es el conjunto de puntos ordenado de forma descendente por las y}
3: Stack() escalera {escalera es la pila donde se tendran los peldaños}
4: Rectangulo rectMaximal {Variable que almacena un rectángulo vacío maximal}
5: Rectangulo rectMER {Variable que almacena el rectángulo vacío maximal}
6: for desde y = 0 hasta n do
7:    for desde x hasta m do
8:       y'_{r'} ← escalera.verUltimoPunto.y
9:       y_r ← y
10:      if y'_{r'} < y_r then
11:         escalera.empilar(x, y_r) {Si el punto anterior es mas bajo que el nuevo punto se empila el punto}
12:      end if
13:      if y'_{r'} = y_r then
14:         escalera.verUltimoPunto.SetX(x) {Si el punto anterior es igual de alto que el nuevo punto, el peldaño
            crece}
15:      end if
16:      if y'_{r'} > y_r then
17:         while y'_{r'} > y_r do
18:            escalera.desempilar {Si el punto anterior es mas alto que el nuevo punto se desempilará hasta que
               el nuevo punto sea menor o de la misma altura que el anterior}
19:         end while
20:      end if
21:      while escalera ≠ ∅ do
22:         rectMaximal ← obtenerMER(escalera.pop(), x, y)
23:         if rectMaximal.superficie ≤ rectMER.superficie then
24:            rectMER ← rectMaximal
25:         end if
26:      end while
27:   end for
28: end for
```

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

Al terminar el algoritmo entrega el mayor rectángulo vacío maximal. El Algoritmo 1, muestra el algoritmo *AREMAV*.

3.3. Descripción de qAREMAV

El algoritmo *qAREMAV* se basa en el algoritmo *AREMAV*, por lo que tiene las mismas precondiciones, además se debe agregar un punto q como consulta, el cual no forma parte del conjunto D como se puede observar en la Figura 3.6(b).

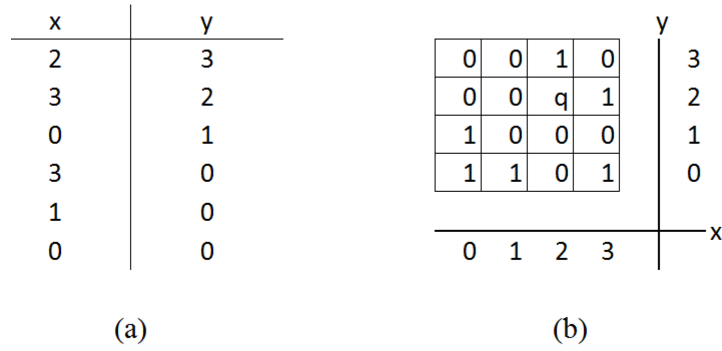

Figura 3.6: Conjunto de puntos D (a) y su representación como matriz M (b)

La mayor diferencia entre el algoritmo *qAREMAV* y *AREMAV* se halla en las lineas 23 y 24 de ambos algoritmos (1 y 2), ya que el algoritmo *AREMAV* obtiene solo el rectángulo vacío de mayor superficie, mientras que *qAREMAV* solo obtiene el rectángulo vacío de mayor superficie y que contiene solo a q en su interior.

Algoritmo 2 Algoritmo qAREMAV

1: *qAREMAV*(R, q, S){R son los puntos límite del conjunto de puntos, q punto de consulta, S el conjunto de puntos}
2: *ordS= ExternalMergeSort(S) {ordS es el conjunto de puntos ordenado de forma descendente por las y}*
3: *Stack() escalera {escalera es la pila donde se tendran los peldaños}*
4: *Rectangulo* $rectMaximal$ *{Variable que almacena un rectángulo vacío maximal}*
5: *Rectangulo* $rectMaximalq$ *{Variable que almacena un rectángulo maximal que sólo contiene a q en su interior}*
6: ***for*** *desde* $y = 0$ *hasta* n ***do***
7:     ***for*** *desde* x *hasta* m ***do***
8:         $y'_{r'} \leftarrow escalera.verUltimoPunto.y$
9:         $y_r \leftarrow y$
10:         ***if*** $y'_{r'} < y_r$ ***then***
11:             $escalera.empilar(x, y_r)$ *{Si el punto anterior es mas bajo que el nuevo punto se empila el punto}*
12:         ***end if***
13:         ***if*** $y'_{r'} = y_r$ ***then***
14:             $escalera.verUltimoPunto.SetX(x)$ *{Si el punto anterior es igual de alto que el nuevo punto, el peldaño crece}*
15:         ***end if***
16:         ***if*** $y'_{r'} > y_r$ ***then***
17:             ***while*** $y'_{r'} > y_r$ ***do***
18:                 *escalera.desempilar {Si el punto anterior es mas alto que el nuevo punto se desempilará hasta que el nuevo punto sea menor o de la misma altura que el anterior}*
19:             ***end while***
20:         ***end if***
21:         ***while*** $escalera \neq \emptyset$ ***do***
22:             $rectMaximal \leftarrow obtenerMER(escalera.pop(), x, y)$
23:             ***if*** $rectMaximal.contieneQ(q)$ & $rectMaximalq.superficie \leq rectMaximal.superficie$ ***then***
24:                 $rectMaximalq \leftarrow rectMaximal$
25:             ***end if***
26:         ***end while***
27:     ***end for***
28: ***end for***

La implementación de los algoritmos *AREMAV* y *qAREMAV* se explica más detalladamente en [19].

Luego de la implementación del algoritmo *qAREMAV* se creó una optimización en cuanto al tiempo de ejecución del algoritmo, dicha implementación se llamó *qAREMAVOpti* el cual es basado en *qAREMAV* y será explicado a continuación junto a otras soluciones que resuelven el problema *qMER*.

Capítulo 4

Algoritmos para resolver *qMER*

En este capítulo se describen dos algoritmos que resuelven el problema *qMER*. El primero de ellos (*qAREMAVOpti*) está inspirado en el algoritmo *qAREMAV* explicado en el capítulo anterior.

El segundo algoritmo llamado $qMER$, considera dos etapas principales. En la primera se reduce el tamaño del conjunto inicial descartando todos aquellos puntos que no tienen posibilidad de formar un rectángulo maximal conteniendo sólo a q. Para ello se calculan los k (un parámetro del algoritmo) puntos más cercanos en cada cuadrante generados por las coordenadas de q (ver Figura 4.2). Estos k puntos generan zonas de dominancia que se utilizan como zonas de descarte, es decir, los puntos que intersectan estas zonas son descartados. Posteriormente, en la segunda etapa, mediante el algoritmo de Orlowski [10], el cual permite obtener el recángulo vacío de mayor área de un conjunto de puntos almacenados en memoria principal, se resuelve el problema considerando como entrada todos los puntos que no fueron descartados.

4.1. Preliminares

Una definición precisa del problema *qMER* dada en [8] es la siguiente. Sea S un conjunto de puntos sobre un rectángulo $R \subseteq \mathbb{R}^2$ cuyos lados son paralelos a los ejes del plano, y $q \notin S$ tal que $q \cap R \neq \emptyset$. Un rectángulo es llamado rectángulo vacío maximal que contiene solo a q (*QMER*) si: (a) contiene a q, (b) sus lados son paralelos a los lados de R, (c) está contenido completamente en R, (d) no contiene otros puntos de S y (e) cada uno de sus lados o arcos contiene, al menos, un punto de S o está contenido en un lado de R. Nuestro problema (*qMER*) consiste en encontrar el rectángulo *QMER* de mayor área.

Dado que los lados de un *QMER* son paralelos a los ejes de R, este queda completamente definido por medio de dos puntos: (i) el punto inferior izquierdo y (ii) el punto superior derecho. De acuerdo a la definición, un *QMER* es un rectángulo que no se puede ampliar o extender en ninguna dirección, pues de permitirlo puede violar las requisitos (c) o (d), es decir, extenderse más allá de R o incluir un punto de S [8].

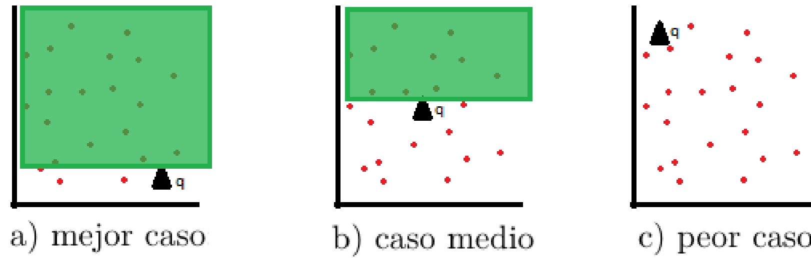

Figura 4.1: Análisis de casos para algoritmo *qAREMAVOpti*

4.2. Algoritmo *qAREMAVOpti*

Algoritmo 3 Algoritmo qAREMAVOpti

1: *qAREMAVOpti*$(R, q, S)\{R \subseteq R^2$ donde se encuentra el conjunto, q punto de consulta, S el conjunto de puntos$\}$
2: *ordS= ExternalMergeSort(S) {ordS es el conjunto de puntos ordenado de forma descendente por las y}*
3: *Stack() escalera {escalera es la pila donde se tendran los peldaños}*
4: *Rectangulo* $rectMaximal$ *{Variable que almacena un rectángulo vacío maximal}*
5: *Rectangulo* $rectMaximalq$ *{Variable que almacena un rectángulo maximal que sólo contiene a q en su interior}*
6: ***for*** *desde* $y = 0$ *hasta* n ***do***
7:     ***for*** *desde* x *hasta* m ***do***
8:         ***if*** $y \leq q.y$ ***then***
9:             $y'_{r'} \leftarrow escalera.verUltimoPunto.y$
10:             $y_r \leftarrow y$
11:             $casosEscalera(y'_{r'}, escalera, x, y)$
12:             ***while*** $escalera \neq \emptyset$ ***do***
13:                 $rectMaximal \leftarrow obtenerMER(escalera.pop(), x, y)$
14:                 ***if*** $rectMaximal.contieneQ(q)$ & $rectMaximalq.superficie \leq rectMaximal.superficie$ ***then***
15:                     $rectMaximalq \leftarrow rectMaximal$
16:                 ***end if***
17:             ***end while***
18:         ***end if***
19:     ***end for***
20: ***end for***

El algoritmo *qAREMAVOpti* (Algoritmo 3) es una optimización del algoritmo *qAREMAV*. La diferencia radica en que al momento de generar la escalera para cada caso, se compara la posición en que se encuentra x, y contra la posición del punto q (Algoritmo 3, línea 8) y si el punto q está más abajo (y menor que la del punto de análisis) no se generará una escalera. Con esta condición al no generar una escalera se está ahorrando tiempo de ejecución el cual varía con las pruebas realizadas a distintos conjuntos de puntos y con el punto q en distintas posiciones.

4.3. Análisis de complejidad

La complejidad del algoritmo *qAREMAVOpti* está dada por la cantidad de puntos que se leen y no se ve afectada por la cantidad de escaleras que se generan al igual que en el algoritmo *AREMAV*.

Como se puede ver en la Figura 4.1 dependiendo donde se ubique el punto q la cantidad de escaleras disminuirá (El rectángulo por sobre el punto q son los puntos que no generarán escaleras) por lo que en el peor caso el tiempo de ejecución puede ser igual al de *qAREMAV* y en el mejor caso ese tiempo podrá disminuir a más de la mitad como se verá más adelante. Sin embargo, su complejidad, tanto temporal como espacial sigue siendo la misma respecto a *qAREMAV*.

A continuación, se presenta el algoritmo $qMER$ el cual es más eficiente en cuanto al tiempo de ejecución que el algoritmo *qAREMAVOpti*, ya que utiliza algoritmos y estructuras más adecuadas que las usadas en *AREMAV*.

4.4. Algoritmo qMER

La idea detrás del algoritmo *qMER* es reducir el tamaño del conjunto inicial de puntos, para posteriormente resolver el problema con un conjunto de mucho menor tamaño. Para reducir el tamaño del conjunto inicial, se usan las zonas de dominacia generadas por los puntos del conjunto más cercanos respecto al punto q.

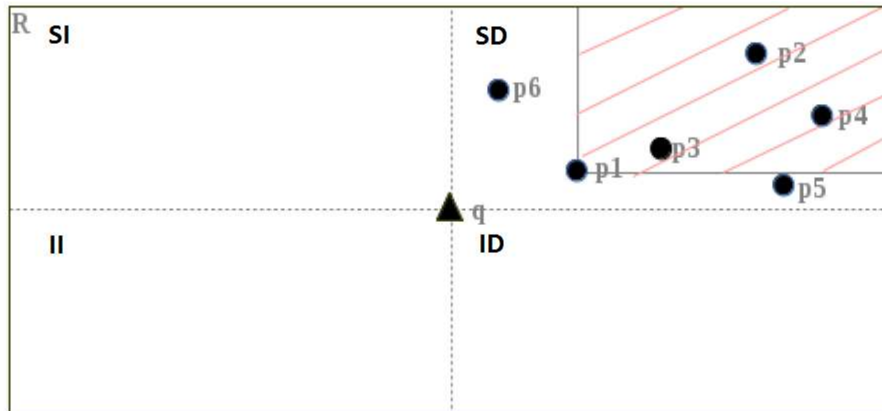

Figura 4.2: Zonas de dominancia

En la Figura 4.2 podemos ver que el punto q divide el rectángulo R en cuatro cuadrantes: Superior Izquierdo (SI), Superior Derecho (SD), Inferior Izquierdo (II) e Inferior Derecho (ID). Nosotros definimos la zona de dominancia de un punto p respecto de un punto q como aquella zona formada por el rectángulo definido por p y el punto de la esquina de R opuesta a p dentro del cuadrante. Por ejemplo, en la Figura 4.2, y considerando el cuadrante SD, la zona de dominancia

de p_1 (zona achurada) está dada por el rectángulo definido por el punto p_1 y el punto extremo superior derecho de R. De manera análoga se definen las zonas de dominancia para los restantes cuadrantes.

Algoritmo 4 Algoritmo *qMER*

1: *qMER*(R, q, S, k){q Punto de consulta, S el conjunto de puntos y k indica la cantidad de puntos que se utilizarán para calcular las zonas de dominancia }
2: *QUAD=quadrant(R,q) {QUAD almacena la definición de cada cuadrante (SI, SD, II, ID)}*
3: *Sea* $kNN_i[k]$ *un arreglo de puntos de tamaño* k *para almacenar los* k*-vecinos más cercanos a* q *en el cuadrante* i
4: *Sea* S' *conjunto de puntos que no fue posible filtrar*
5: $kNN[k] \leftarrow$ *k-vecinosMasCercanos*(R, S, k, q) *{Función que obtiene los* k *vecinos más cercanos al punto* q *para todos los cuadrantes en QUAD}*
6: zi = *dominance*$(kNN, QUAD)$ *{calcula las zonas de dominancia de cada cuadrante}*
7: ***while*** $S \neq \emptyset$ ***do***
8: $p \leftarrow getNext(S)$
9: $i \leftarrow getQuadrant(QUAD, p)$
10: ***if*** $p \cap z \neq \emptyset$ *para alguna zona de dominancia* $z \in zi_i$ ***then***
11: $S' \leftarrow S' \cup p$
12: ***end if***
13: ***end while***
14: $ans \leftarrow Orlowski(R, q, S' \cup kNN_i)$
15: ***return*** ans

El algoritmo *qMER* usa estas zonas de dominancia como zonas de descarte con el objeto de obtener un conjunto $S' \subseteq S$ de tamaño menor que el de S (se supone que S' es suficientemente pequeño para residir en memoria principal) y resolver el problema *qMER* mediante un algoritmo de Geometría Computacional, considerando como entrada el conjunto S'. En la Figura 4.2, los puntos p_2, p_3 y p_4 son descartados, pues se ubican dentro de la zona de dominancia de p_1. Los puntos que caen dentro de zonas de dominancia se pueden descartar, pues si uno de los lados de un *QMER* Q pasa por p y se intenta expandir hacia un punto p_i dentro de su zona de dominancia, el nuevo rectángulo Q' contiene a p y, por lo tanto, no es un *QMER*. Esto se puede ver en la Figura 4.2 donde cualquier *QMER* Q cuyo lado derecho o superior tiene a p_1 y si se intenta extender para que dicho lado pase por p_3 necesariamente dejará a p_1 en su interior. Notar que algo similar ocurre si se considera a p_2 o p_4 en lugar de p_3.

La idea detrás del Algoritmo 4 es conseguir, en una primera etapa, zonas de dominancia que cubran un área lo más cercana posible al área de R con el propósito de lograr, en una segunda etapa, reducir el tamaño de S. Para ello, por cada cuadrante se obtienen los k-vecinos más cercanos a q y con estos se definen las zonas de dominancia por cada cuadrante (ver Figura 4.3).

Como se comentó anteriormente, en la primera etapa obtenemos los k-vecinos más cercanos a q en cada cuadrante (línea 5 algoritmo 4). Dichos vecinos se obtienen con el Algoritmo 5 el cual utiliza un heap de tamaño k por cada cuadrante. En los heaps se almacenan puntos y se encuentran organizados de mayor a menor distancia a q. El Algoritmo 5 realiza una pasada por

Algoritmo 5 Algoritmo obtenerVecinosMasCercanos

1: k-$vecinosMasCercanos(R, S, k, q)$
2: Sea H un arreglo de cuatro heap, cada uno de tamaño k y que almacenan los k puntos más cercanos a q en cada cuadrante
3: **while** $S \neq \emptyset$ **do**
4: $p \leftarrow getNext(S)$
5: $i \leftarrow getQuadrant(QUAD, p)$
6: **if** $size(H_i) < k$ **then**
7: $insert(H_i, p)$
8: **else**
9: $d \leftarrow distance(p, q)$
10: $p_1 \leftarrow max(H_i)$
11: **if** $d < distance(p_1, q)$ **then**
12: $extractMax(H_i)$
13: $insert(H_i, p)$
14: **end if**
15: **end if**
16: **end while**
17: **return** H

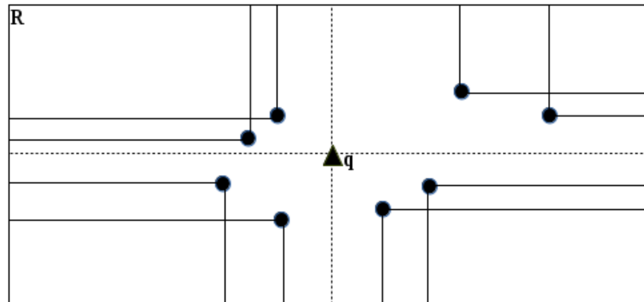

Figura 4.3: Algoritmo "k vecinos" con k=2

todos los puntos de S. Por cada punto verifica si la distancia a q es mejor (más cercana) que los puntos mantenidos en el heap de su cuadrante (línea 11). Notar que si el heap no está lleno los puntos se insertan directamente en el heap (línea 6).

Una vez obtenidos los k-vecinos más cercanos, el Algoritmo 4 obtiene las zonas de dominancia para cada cuadrante (línea 6). Posteriormente (segunda etapa), se realiza una segunda pasada por todos los puntos de S. Por cada punto se verifica si el punto intersecta alguna zona de dominancia de su correspondiente cuadrante. De ser así el punto es descartado, en caso contrario se agrega al conjunto S'. Finalmente (etapa 3), en la línea 14 del Algoritmo 4 se procesa el conjunto S' con el Algoritmo de Orlowski propuesto en [10].

4.5. Análisis de complejidad

Como anteriormente se ha mostrado, el algoritmo $qMER$ se puede separar en 4 fases, siendo la primera (i) encontrar los k puntos más cercanos a q por cada cuadrante, (ii) recorrer nuevamente los puntos y descartar todos los puntos que son dominados por los k más cercanos en cada cuadrante y (iii) resolver el problema *qMER* con el algoritmo de Orlowski dando como conjunto de entrada todos los puntos que no hayan sido dominados.
Teniendo esto en cuenta la complejidad está dada por la suma de las complejidades de las 3 fases antes mencionadas. Así.
$(i) : O(n \log k)$ $\quad$ $(ii) : O(n\dot{k})$ $\quad$ $(iii) : O(n \log n + s)$

Dado que el algoritmo toma k como un valor constante la ecuación en la segunda fase será constante. Luego se tienen dos complejidades iguales las cuales serán simplificadas a una, resultando esta ecuación.
$(i) : O(n)$ $\quad$ $(ii) : O(n)$ $\quad$ $(iii) : O(n \log n + s)$

Donde la complejidad del algoritmo $qMER$ está dada por la complejidad mayor dentro de las anteriores. De este modo, la complejidad temporal del algoritmo $qMER$ es igual a la del algoritmo Orlowski ($O(n \log n + s)$).

4.6. Evaluación experimental de los algoritmos

Por medio de una serie de experimentos se compararon nuestros algoritmos con el propuesto en [6] y que fue modificado (*qAREMAV*). Los algoritmos *qAREMAV*, *qAREMAVOpti* y $qMER$ fueron implementados en lenguaje de programación JAVA.

Los experimentos fueron realizados en una máquina con un procesador de 4 núcleos con 3.092 MHz y 8GB RAM.

Se consideraron conjuntos entre 10.000 y 50.000.000 de puntos con distribución uniforme en espacio $[0, 1] \times [0, 1]$. Se estudiaron varios valores para el parámetro k (1, 5, 10, 15, 20, 50, 100) para el caso de $qMER$.

En los experimentos se midió el tiempo de ejecución de los algoritmos. Para el caso de $qMER$, se midió además el porcentaje de filtrado, que corresponde al porcentaje de puntos que no fueron descartados.

En un primer experimento se estudió el efecto de k sobre el tamaño del conjunto S' y sobre el tiempo de filtrado en $qMER$. Los diferentes valores de k se estudiaron sobre un conjunto de 10 millones de puntos. Los resultados se pueden ver en la Figura 4.4. Es posible apreciar que en la medida que k aumenta, el tiempo de ejecución tiende a mantenerse relativamente constante en torno a los 22 segundos. Nosotros usaremos $k = 10$ para los restantes experimentos, ya que como muestra la Figura 4.4, mientras mayor sea k, el tiempo de filtrado aumenta muy poco y tampoco S' disminuye considerablemente. La razón del rendimiento de $qMER$ (etapas 1 y 2) en función de k es que en la medida que aumenta k el costo en tiempo invertido en calcular los

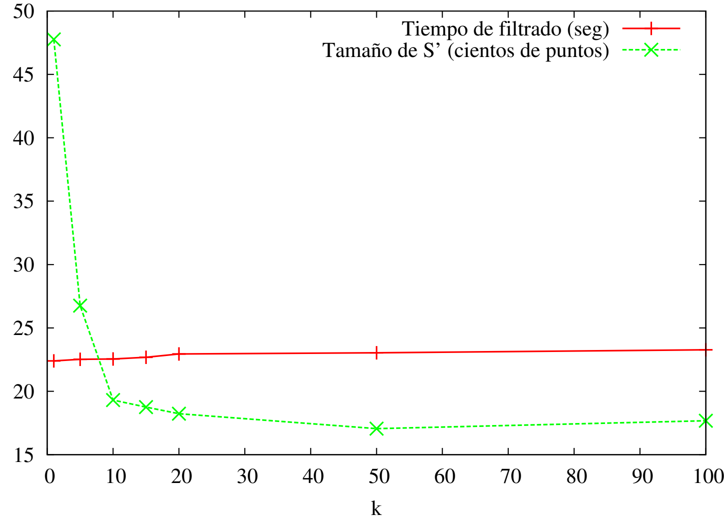

Figura 4.4: Influencia de distintos valores de k en el tiempo de filtrado de $qMER$ (etapas 1 y 2) y en el tamaño de S'

k-vecinos no es conveniente comparado con el beneficio obtenido, es decir, con la reducción del tamaño de S'. El dejar k constante, ayuda a que siempre se logre una reducción significativa del conjunto original sin aumentar demasiado el tiempo de ejecución.

Ya habiendo definido un k adecuado para $qMER$, este se comparó con los algoritmos $qAREMAV$ y $qAREMAVOpti$. Los resultados se muestran en la Figura 4.5. En dicha figura los algoritmos $qAREMAV$ y $qAREMAVOpti$ son muy similares, ya que como se mencionó anteriormente, $qAREMAVOpti$ genera menor número de escaleras en comparación con $qAREMAV$. Por otro lado $qMER$ supera por alrededor de 2 a 3 órdenes de magnitud a los demás algoritmos. Esta diferencia a favor de $qMER$ se logra debido a la reducción del tamaño del conjunto original en las etapas 1 y 2.

Finalmente, se estudió el comportamiento de $qMER$ en función del tamaño del conjunto de puntos. Los tamaños considerados fueron 10, 20, 30, 40 y 50 millones de puntos. En la Figura 4.6 el tiempo corresponde al tiempo total de ejecución de $qMER$ (etapas 1, 2 y 3). Se puede observar que, para los conjuntos estudiados, el tiempo de ejecución presenta un comportamiento aproximadamente lineal.

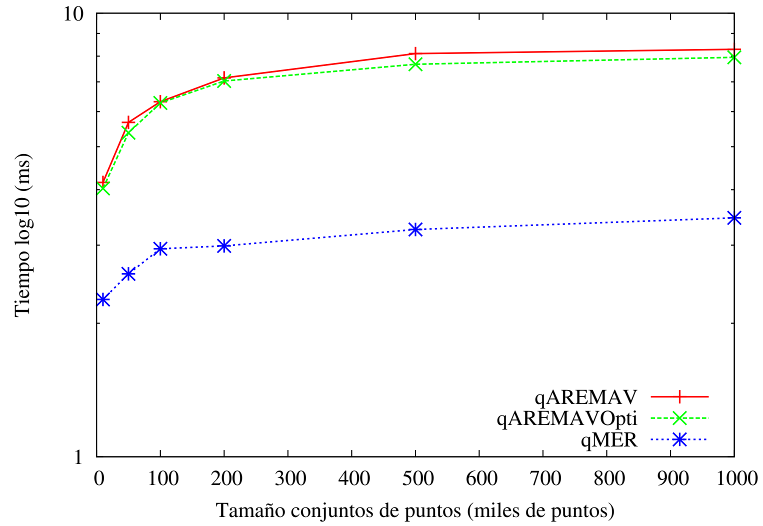

Figura 4.5: Comparación de algoritmos respecto al tiempo de ejecución.

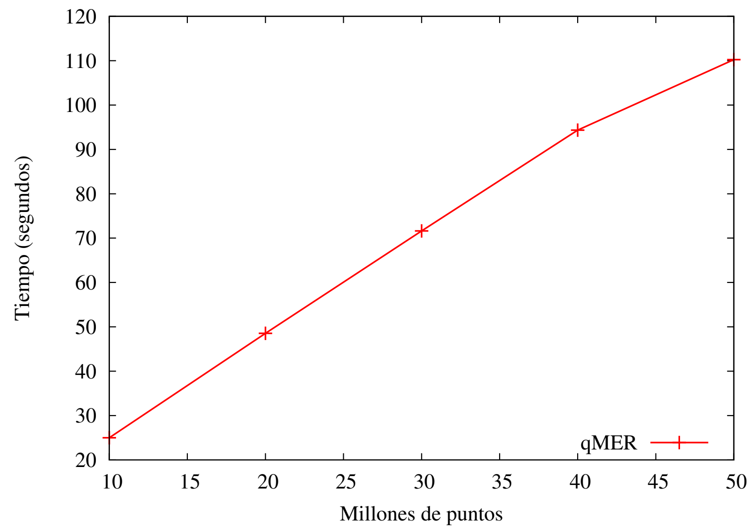

Figura 4.6: Tiempo de ejecucion de $qMER$.

Capítulo 5

Algoritmo para resolver el problema *MER*

5.1. Preliminares

La definición del problema *MER* es muy similar a la definición dada para el problema *qMER*, es decir, sea S un conjunto de puntos sobre un rectángulo $R \subseteq \mathbb{R}^2$ cuyos lados son paralelos a los ejes del plano. Un rectángulo es llamado rectángulo vacío maximal si: (a) sus lados son paralelos a los lados de R, (b) está contenido completamente en R, (c) no contiene otros puntos de S y (d) cada uno de sus lados o arcos contiene, al menos, un punto de S o está contenido en un lado de R. Nuestro problema (*MER*) consiste en encontrar el rectángulo *MER* de mayor área.

Dado que los lados de un *MER* son paralelos a los ejes de R, este queda completamente definido por medio de dos puntos: (i) el punto inferior izquierdo y (ii) el punto superior derecho. De acuerdo a la definición, un *MER* es un rectángulo que no se puede ampliar o extender en ninguna dirección, pues de permitirlo puede violar las requisitos (c) o (d), es decir, extenderse más allá de R o incluir un punto de S [8].

Esta propuesta se basa en que el algoritmo *AREMAV*, visto anteriormente, tal como fue reportado en [6], presentó un pobre rendimiento cuando el conjunto de puntos S, sobre el cual se aplica, tiene una baja densidad $D = \frac{|T|}{|X| \times |Y|}$, con $|T|$ el número de puntos, $|X|$ e $|Y|$ la cantidad de valores distintos de la coordenada de x e y, respectivamente.

Nuestro algoritmo MER intenta complementar al algoritmo *AREMAV*, mejorando el tiempo de ejecución en conjuntos de puntos de menores densidades, los cuales son muy frecuentes en amplio rango de aplicaciones. De esta forma, MER pretende ser un complemento de *AREMAV* en lugar de un algoritmo que lo reemplace. La decisión de cuál algoritmo usar se puede establecer sobre la base de la densidad del conjunto de puntos y cuál se puede obtener en tiempo lineal considerando que el conjunto se encuentra ordenado.

La idea detrás del algoritmo MER es dividir el conjunto de puntos original en subconjuntos de puntos de tamaño aproximadamente iguales. Posteriormente, cada uno de estos subconjuntos se resuelve con *AREMAV* y, finalmente, se combinan las soluciones para obtener el resultado.

5.2. Algoritmo MER

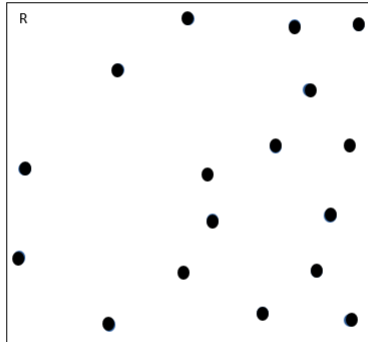

Figura 5.1: Conjunto de puntos original.

El algoritmo MER (Algoritmo 6) se caracteriza por la división del espacio R original en 4 subconjuntos (SI,SD,II,ID como se vío anteriormente en el algoritmo $qMER$) (Ver Figuras 5.1 y 5.2) (Algoritmo 6, línea 2), esta separación se realiza con el fin de disminuir la cantidad de puntos a analizar, por lo que algoritmo *AREMAV* se demora menor cantidad de tiempo. Se busca también que en cada subconjunto exista una cantidad similar de puntos, ya que de esta forma el algoritmo será más rápido.

La división del conjunto de puntos se realizó con una estructura de datos presentada en [20] llamada VA-File (Vector aproximation files). La idea detras de VA-File es particionar el espacio (d-dimensional en general, 2-dimensional para este algoritmo) en 2^b particiones o celdas,

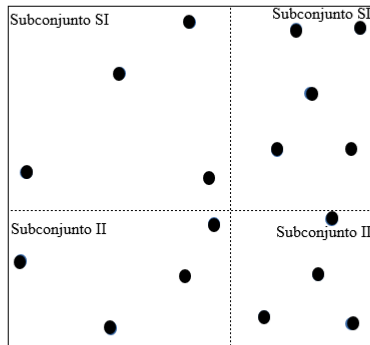

Figura 5.2: División del conjunto original en 4 cuadrantes.

Algoritmo 6 Algoritmo MER

1: $MER(R, S)\{R \subseteq \mathbb{R}^2$ donde se encuentra el conjunto, S el conjunto de puntos$\}$
2: $CUAD_i \leftarrow VA(S, muestra)$ $\{i$ es el número del cuadrante, VA es la estructura usada para obtener la defición de los 4 cuadrantes utilizando una muestra del conjunto $S\}$
3: $Conjunto_i \leftarrow crearConjuntos(CUAD, S)$\{Se crean 4 conjuntos, uno por cada cuadrante conteniendo los puntos que los intersectan\}
4: *Rectangulo* MER *\{Se crea un rectángulo vacío que almacenará el rectángulo vacío de mayor área\}*
5: ***for*** *i = 1* ***to*** *4* ***do***
6: $AREMAV(CUAD_i, Conjunto_i)$
7: $rectangulosCreadosConBordeDerecho \leftarrow AREMAV.obtenerRectBordeDer()$ *\{El algoritmo AREMAV almacena los rectángulos vacíos maximales y se buscan los que fueron creados con soporte en el borde del cuadrante\}*
8: $rectangulosCreadosConBordeInferior \leftarrow AREMAV.obtenerRectBordeInf()$
9: ***if*** *i=1* ***then***
10: $escribir(Conjunto_2, rectangulosCreadosConBordeDerecho)$ *\{Se graban en el conjunto de puntos los rectángulos que se crearon en los bordes\}*
11: $escribir(Conjunto_3, rectangulosCreadosConBordeInferior)$
12: ***end if***
13: ***if*** *i=2* ***then***
14: $escribir(Conjunto_4, rectangulosCreadosConBordeInferior)$
15: ***end if***
16: ***if*** *i=3* ***then***
17: $escribir(Conjunto_4, rectangulosCreadosConBordeDerecho)$
18: ***end if***
19: ***if*** $MER.superficie < AREMAV.obtenerMER().superficie()$ ***then***
20: $MER \leftarrow AREMAV.obtenerMer()$
21: ***end if***
22: ***end for***
23: ***return*** *MER*

de modo que cada celda tenga asociado un identificador de b bits. Los b bits se reparten en $b_1 + b_2 + ... + b_d = b$ bits, donde b_i bits se asocian a la i-ésima dimensión, para $i = 1...d$. Dado que estamos trabjando en 2 dimensiones consideramos $b = b_1 + b_2$,. El número de bits b_i para establecer las particiones en cada dimensión i se determina en función del número total de bits (b) y del número de dimensiones d como sigue:

$$b_i = \left\lfloor \frac{b}{d} \right\rfloor + \begin{cases} 1 & i \leq (b \bmod d) \\ 0 & \text{en caso contrario} \end{cases}$$

El número de bits en cada dimensión se utiliza para determinar los puntos de partición y consecuentemente las regiones dentro de cada dimensión. En particular existen 2^{b_i} regiones dentro de la dimensión i, definidas por $2^{b_i} + 1$ puntos de partición. En la Fig. 5.3 se muestra un ejemplo en el cual $b = 3$, $d = 2$, $b_1 = 2$ y $b_2 = 1$, lo que significa que existen dos bits para la dimensión x y un bit para la dimensión y. Por lo tanto considerando la dimensión x existen cuatro regiones (cinco puntos de partición) y de la misma manera dos particiones del espacio desde el punto de vista de la dimensión y (3 puntos de partición). En cada dimensión i, los 2^{b_i} puntos de partición $pc_i[0], pc_i[1], \ldots, pc_i[2^{b_i}]$ se establecen de tal manera que cada una de las regiones tenga aproximadamente la misma cantidad de puntos, para lo cual se pueden tomar en cuenta

todos los puntos del conjunto o bien, de manera aleatoria, se puede utilizar una muestra de ese conjunto [20]. En este caso, se eligió aleatoriamente una pequeña muestra del conjunto S para que se utilice VA-File, dando como resultado los puntos de partición de cada división.

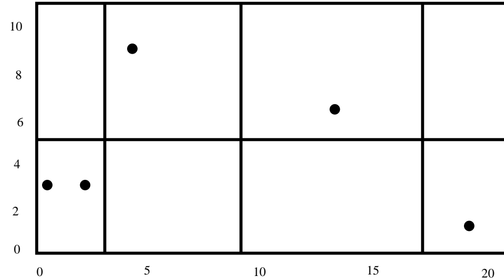

Figura 5.3: Ejemplo de una partición considerando $d = 2$ y $b = 3$

Al obtener las particiones de R, es necesario leer todos los puntos del conjunto original y determinar a que partición intersectan (Algoritmo 6, línea 3). De esta forma se crean los 4 subconjuntos de puntos (Figura 5.2).

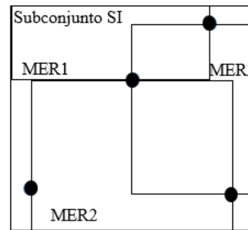

Figura 5.4: Algoritmo *AREMAV* aplicado al conjunto 1.

Teniendo los puntos pertenecientes a cada subconjunto ya definidos, se procede a utilizar el algoritmo *AREMAV* para cada subconjunto (Algoritmo 6, líneas 5 y 6). Los rectángulos que genera *AREMAV* (Figura 5.4) pueden ser de distinto tipo, agrupándolos como sigue:

- **Rectángulos que se generan dentro del subconjunto:** Estos rectángulos son los que *AREMAV* genera con todos sus lados contenidos dentro de la subregión $MER1$ (ver Figura 5.4).
- **Rectángulos que se generan con el borde inferior:** Estos rectángulos son los que *AREMAV* genera teniendo como borde inferior, el mismo borde de la subregión donde están contenidos (ver Figura 5.4, $MER2$).
- **Rectángulos que se generan con el borde derecho:** Estos rectángulos son los que *AREMAV* genera teniendo como borde derecho, el mismo borde de la subregión donde están contenidos (ver Figura 5.4, $MER3$).

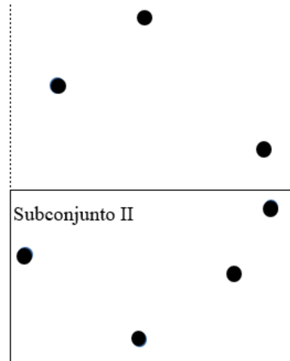

Figura 5.5: Cuadarante 3 después de haber sido analizado el cuadrante 1.

El saber qué tipo de rectángulo se genera, es útil al momento de analizar el conjunto siguiente, ya que, si sabemos que el conjunto 1 (Figura 5.4) generó un rectángulo con el borde inferior del subconjunto SI, este rectángulo seguirá creciendo hacia abajo (de igual forma un rectángulo que se generá con el borde derecho, podrá seguirá creciendo hacia el lado derecho). De esta forma, el algoritmo *AREMAV* puede ir generando rectángulos más grandes sin la necesidad de volver a leer subconjuntos ya procesados.

Para que estos rectángulos sigan creciendo, es necesario traer los puntos que los conforman a los subconjuntos donde pueden seguir creciendo (Algoritmo 6 líneas 9, 13 y 16). Entonces, en el caso de estar analizando el subconjunto SI los rectángulos que tienen su borde en el lado derecho son llevados al subconjunto SD y los rectángulos que tienen su borde inferior en el subconjunto SI, son llevados al subconjunto II (Algoritmo 6, línea 9). Si se está analizando el subconjunto SD, en este caso solo se toman los rectángulos que se generan en el borde inferior del subconjunto SD y son llevados al subconjunto ID(Algoritmo 6, línea 13). En el caso de analizarse el subconjunto II, solo se toman los rectángulos generados en el borde derecho y estos son llevados al subconjunto ID (Algoritmo 6, línea 16). Cuando se analiza el conjunto 4 no es necesario analizar los tipos de rectángulos ya que sin importar su tipo, no pueden ser llevados a otro conjunto.

En la Figura 5.5 se muestra el subconjunto II, después de haber obtenido los rectángulos del subconjunto SI. Al haber encontrado un rectángulo que tiene su borde inferior en el subconjunto SI, los puntos que conforman el rectángulo son llevados al subconjunto II para ser analizados y ver hasta dónde crecerá, como se muestra en la Figura 5.6.

A medida que se analizan los subconjuntos, se obtienen los rectángulos vacíos maximales y se va dejando el de mayor área (Algoritmo 6, línea 19).

Para finalizar, se analiza el subconjunto ID, en este subconjunto se generan los rectángulos vacíos maximales propios del subconjunto junto a los últimos rectángulos vacíos maximales que no se han podido “cerrar” en los subconjuntos anteriores, por lo que todos los rectángulos que

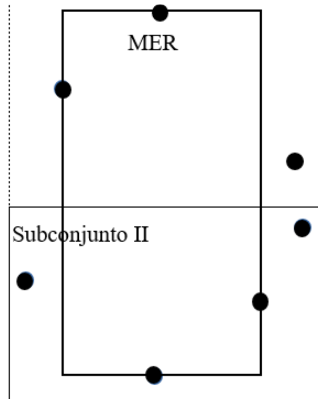

Figura 5.6: MER encontrado en el conjunto 3.

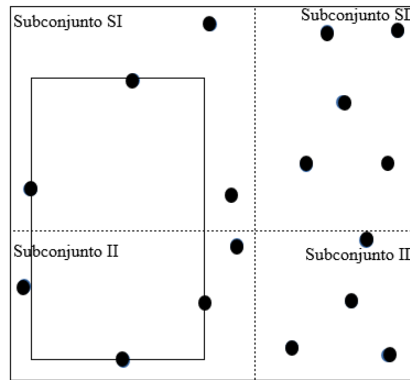

Figura 5.7: MER del conjunto inicial.

se obtienen se comparan con el anterior más grande. El rectángulo más grande es el MER. En el conjunto de datos de la Figura 5.1, el MER se obtiene en el subconjunto II, como se muestra en la Figura 5.7.

5.3. Complejidad del algoritmo

El algoritmo MER se compone de diferentes pasos para su ejecución. Estos pasos son: (*i*) tomar una muestra del conjunto original de puntos y dividir S en 4 subconjuntos con una cantidad similar de puntos y (*ii*) utilizar *AREMAV* en todos los subconjuntos. De estos pasos el que presenta una complejidad temporal mayor es el paso (*ii*) de $O(|X| \times |Y|)$.

5.4. Evaluación experimental del algoritmo MER

En esta sección se muestra una serie de experimentos que comparan el algoritmo MER con el algoritmo *AREMAV*.

Para estos experimentos se utilizó la misma máquina que se usó para los experimentos del algoritmo *qMER* en el capítulo anterior.

Como se dijo anteriormente, los algoritmos MER y *AREMAV* son sensibles a la densidad de los conjuntos de puntos.

Se realizaron experimentos con conjuntos de puntos reales y sintéticos. Dentro de los conjuntos de puntos sintéticos se utilizaron distintas distribuciones (uniforme, zipf y cluster).

Las distribuciones que se generaron son las siguientes:

- Distribución **Uniforme** (Figura 5.8). Conjuntos cuyo tamaño varía entre 500.000 y 10.000.000 de puntos y densidades de 10 % y 20 %. Estos conjuntos de puntos fueron construidos como se menciona en [6], donde los conjuntos de puntos tienen un $|X| = 1000$.

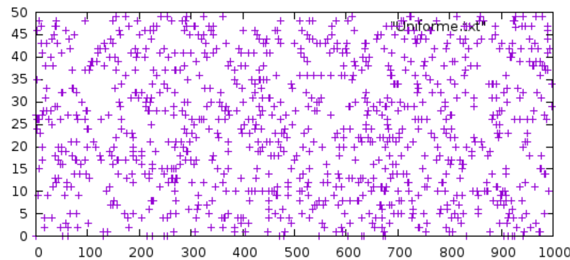

Figura 5.8: Distribución uniforme con densidad de 20 %.

- Distribución **Zipf**(Figura 5.9). Conjuntos con 125.000 y 250.000 tuplas y densidad de 5 %.

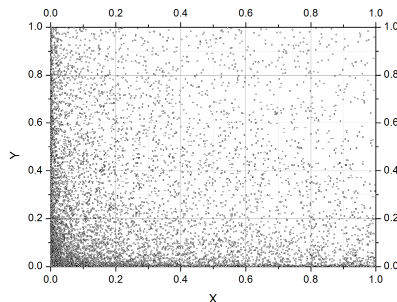

Figura 5.9: Distribución Zipf con 125.000 puntos

- Distribución **Cluster**(Figura 5.10). Conjuntos con 125.000 y 250.000 tuplas y densidad de 1 %.

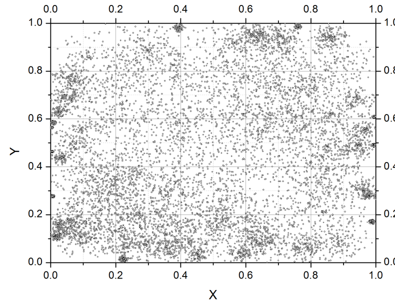

Figura 5.10: Distribución cluster con 125.000 puntos

- Los datos reales (Figura 5.11) corresponden a puntos en Norte América [2]. Estos conjuntos tienen una densidad de $\approx 0\,\%$, ya que según la fórmula que se vió con anterioridad, no puede existir un conjunto con densidad del 0 %, pero si con una densidad que se aproxime al 0 %.

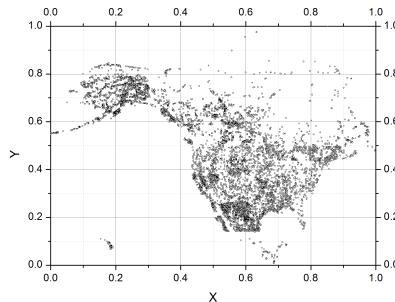

Figura 5.11: Conjunto de puntos reales sobre Norte América.

Los experimentos que se realizaron son en relación a cómo influyen los distintos tipos de distribuciones y densidades en el tiempo de ejecución de los algoritmos *AREMAV* y MER.

Densidad en distintos algoritmos. Este primer experimento se realizó con el fin de mostrar el efecto que tiene la densidad en los dos algoritmos, utilizando un conjunto de 200.000 puntos y densidades de $\approx$0 %, 1 %, 5 %, 10 %, 15 % y 20 %. De esta forma, podemos demostrar experimentalmente que, para conjuntos de menor densidad es mejor utilizar el algoritmo MER y para conjuntos de mayor densidad es mejor utilizar el algoritmo *AREMAV*. Los resultados se pueden ver en la Figura 5.12.

[2] http://spatialhadoop.cs.umn.edu/datasets.html

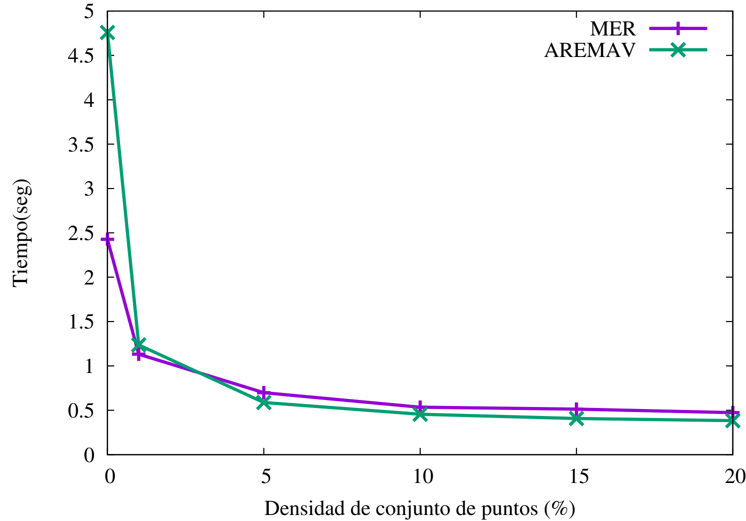

Figura 5.12: Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos de distintas densidades

Como se puede ver en la Figura 5.12, el algoritmo MER es más eficiente, en cuanto a tiempo de ejecución, que el algoritmo *AREMAV* mientras la densidad sea menor a 7 %, ya que aumentando esta densidad el algoritmo *AREMAV* utiliza un menor tiempo de ejecución que el algorimto MER.

Conjunto de puntos reales

Este experimento muestra el comportamiento de ambos algoritmos respecto a los conjuntos de puntos reales (Figura 5.11). Los resultados se pueden apreciar en la Tabla 5.1, y Figura 5.13.

Algoritmo \ Puntos	9.200	24.500	191.640	383.280	1.138.240
AREMAV (segundos)	30	142	19.260	107.378	983.962
MER (segundos)	6	33	7.219	35.482	273.007

Tabla 5.1: Tiempo de ejecución(segundos) de algoritmos MER y *AREMAV* sobre conjuntos con datos reales.

Este experimento muestra que para datos reales el algoritmo MER es más eficiente en cuanto a velocidad de ejecución que el algoritmo *AREMAV*. MER supera a *AREMAV* pues requiere de un 28 %, en promedio, del tiempo requerido por *AREMAV* (Figura 5.13). Este mayor rendimiento se debe a que el algoritmo MER, al separar el conjunto de puntos en subconjuntos,

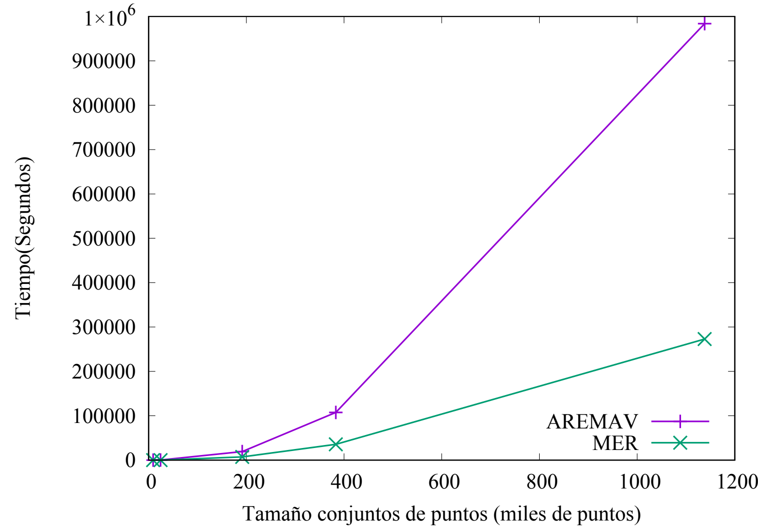

Figura 5.13: Tiempo de ejecución(segundos) de algoritmos MER y *AREMAV* sobre conjuntos con datos reales.

reduce significativamente el tamaño de $|X|$ e $|Y|$, por lo que la generación de escaleras se reduce, disminuyendo también el tiempo de ejecución.

Distribución Cluster con densidad 1 %

Para este experimento se utilizó la distribución cluster (Figura 5.10), ya que en este tipo de distribución los puntos se agrupan en distintos clusters en el plano, donde hay mayor densidad en el centro de los clusters y menor densidad a medida que se alejan del centro. Los resultados se pueden apreciar en la Tabla 5.2, y Figura 5.14.

Puntos(K) / Algoritmo	125	250
AREMAV (segundos)	3.873	16.888
MER(segundos)	2.083	8.498

Tabla 5.2: Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución cluster.

Como se puede ver en la Figura 5.14 el algoritmo MER posee un mejor tiempo de ejecución comparado con el algoritmo *AREMAV*. Por otro lado, se puede apreciar que el rendimiento del algoritmo *AREMAV* ha mejorado bastante en comparación al caso anterior.

Distribución Zipf con densidad 5 %

En este experimento se disminuyó la densidad y se cambió el tipo de distribución, utilizando la Zipf (Figura 5.9), donde los puntos se agrupan cerca del origen y se van dispersando a medida

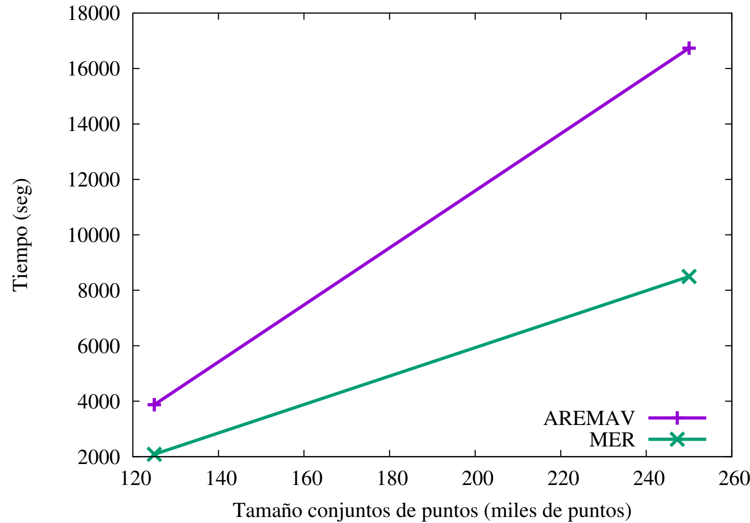

Figura 5.14: Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución cluster.

que estos se alejan. Los resultados se pueden apreciar en la Tabla 5.3, y Figura 5.15.

Puntos(K) / Algoritmo	125	250
AREMAV (segundos)	29	204
MER(segundos)	22	159

Tabla 5.3: Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución Zipf.

Como se puede ver en la Figura 5.15 el algoritmo MER es un poco mejor que el algoritmo *AREMAV*, pero este último alcanza tiempos de ejecución muy parecidos a MER, ya que al haber alta densidad, las escaleras de *AREMAV* no alcanzan a crecer, por lo que el tiempo utilizado en hacer crecer las escaleras es muy poco.

Distribución Uniforme densidad 20 %

En este último experimento se comparan ambos algoritmos en el escenario donde el algoritmo *AREMAV* toma más ventaja como se exlica en [6] y se demuestra en [19], ya que mientras mayor sea la densidad entre los puntos, mejor es su rendimiento. Esto sucede porque las escaleras no alcanzan tamaños muy grandes y utilizarlas no requiere un tiempo significativo. Los datos se

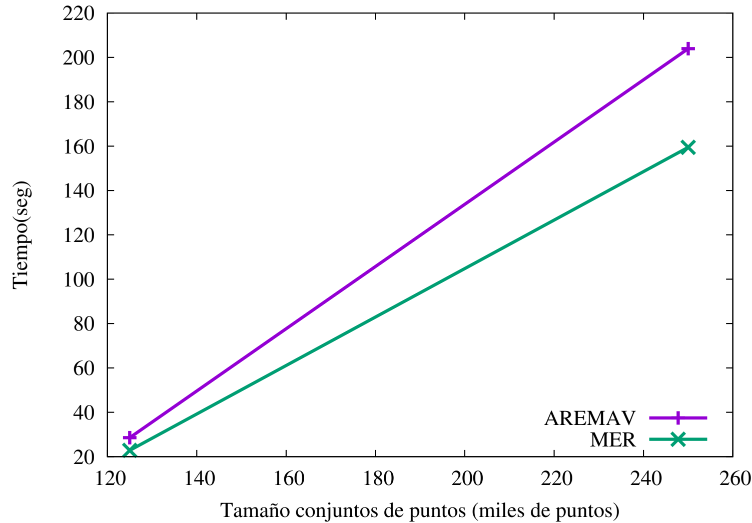

Figura 5.15: Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución Zipf.

pueden ver en la Tabla 5.4 y se hallan graficados en la Figura 5.16.

Algoritmo \ Puntos(K)	500	1000	1500	2000	5000	10000
AREMAV (segundos)	4	7	10	13	38	85
MER(segundos)	6	14	22	33	107	308

Tabla 5.4: Tiempo de ejecución algoritmos MER y *AREMAV* sobre conjuntos con distribución uniforme y densidad de 20 %.

Como se puede apreciar en la Figura 5.16, ambos algoritmos tienen comportamientos similares, pero el algoritmo MER no es más eficiente en cuanto a tiempo de ejecución que el algoritmo *AREMAV*. Si bien el algoritmo MER utiliza el algoritmo *AREMAV* para obtener los rectángulos vacíos maximales, también utiliza muchos accesos a disco, por lo que aumenta el tiempo de ejecución.

Por lo tanto, de acuerdo a los resultados experimentales obtenidos en los diferentes escenarios, podemos decir que MER es un algoritmo complementario a *AREMAV*, pues permite resolver problemas en escenarios muy desfavorables para *AREMAV* (conjuntos con baja densidad). De esta forma, y mediante una heurística basada en la densidad del conjunto, se puede decidir usar uno

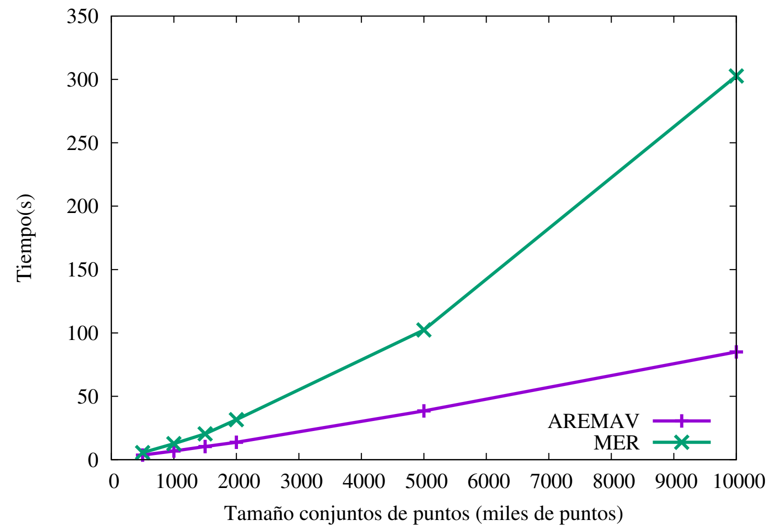

Figura 5.16: Tiempo de ejecución (segundos) algoritmos MER y *AREMAV* sobre conjuntos con distribución uniforme y con densidad 20 %

u otro algoritmo.

Capítulo 6

Conclusiones

En esta tesis se muestran dos algoritmos que resuelven el problema *qMER* y un algoritmo complementario a *AREMAV* para resolver el problema *MER*. En varios escenarios los algoritmos muestran mejoras respecto de los algoritmos conocidos en la literatura. Con el objeto de evaluar el rendimiento de los algoritmos se realizaron una serie de experimentos con conjuntos de puntos sintéticos y reales.

Para el problema *qMER* se propusieron dos algoritmos. El primero *qAREMAVOpti*, el cual es una variante optimizada del algoritmo *qAREMAV* implementado en [19] y el algoritmo *qMER* el cual considera tres etapas. En las dos primeras reduce el tamaño del conjunto de puntos original que reside en memoria secundaria y en la tercera se aplica un algoritmo de la Geometría Computacional al conjunto reducido.

Los resultados muestran que *qMER* supera a los algoritmos $qAREMAV$ y $qAREMAVOpti$ en varios órdenes de magnitud. Esta diferencia se explica por la reducción del tamaño del conjunto original que se logra en las etapas 1 y 2, el que se redujo, en promedio, a un 0,02 %.

Al comparar nuestro algoritmo con $qAREMAV$ y $qAREMAVOpti$, podemos concluir que es ampliamente ventajoso tanto en requerimientos de memoria como en tiempo de ejecución. Esto permite que con *qMER* sea posible resolver problemas considerando grandes conjuntos de puntos y que no es posible almacenarlos en memoria principal.

El algoritmo que resuelve el problema *MER* se basa en el algoritmo *AREMAV* donde la diferencia radica en que el algoritmo MER pre procesa los puntos y los divide en 4 subconjuntos para luego resolver cada subconjunto con el algoritmo *AREMAV* y mezclar las soluciones.

Los resultados experimentales muestran la influencia de las distintas densidades de los conjuntos de puntos en el caso del algoritmo MER, donde se puede observar que a menor densidad su rendimiento es mejor que el de *AREMAV*, ya que MER utiliza un 57 % menos de tiempo de ejecución. Por el contrario, se puede observar que a mayor densidad el algoritmo *AREMAV* presenta un mejor rendimiento que MER, siendo más veloz en un 50 % aproximadamente. Estos

comportamientos muestran que ambos algoritmos se complementan.

En la actualidad hay muchas aplicaciones que trabajan con conjuntos de puntos reales con baja densidad, por lo que utilizar el algoritmo MER podría ser mucho más ventajoso que otros.

Como trabajo futuro, se podría optimizar la forma en que el algoritmo MER encuentra el rectángulo vacío maximal. Ya sea creando más cuadrantes de forma recursiva hasta que los puntos contenidos se puedan almacenar en memoria principal y de esta forma, utilizar en estos sub conjuntos un algoritmo que funcione en memoria principal para luego unir las soluciones.

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 Gutiérrez and José R. Paramá. 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. (2014) The largest empty rectangle containing only a query object in Spatial Databases, In GeoInformatica, volume 18, pages 193-228.

[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

[19] Lara F. (2014) Implementación en Java de algoritmos geométricos sobre grandes conjuntos de datos. In: Memoria de Título, Ingeniería Civil en Informática, Universidad del Bío-Bío

[20] Weber, R., Schek, H. J., Blott, S. (1998) A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces, Proc. VLDB Conf. 194-205