El par de vecinos más cercanos cuando sólo uno de los conjuntos cuenta con un índice espacial

Gilberto Gutiérrez R. y Pablo Sáez G.

Universidad del Bío-Bío
Departamento de Ciencias de la Computación
y Tecnologías de la Información
Chillán, Chile
{ggutierr, psaezg}@ubiobio.cl

Resumen En este trabajo se presenta un algoritmo para resolver el problema de encontrar el par de vecinos más cercanos entre dos conjuntos de puntos del plano almacenados en memoria secundaria en el caso en que sólo uno de los dos cuenta con un índice espacial. Este problema surge naturalmente en escenarios en donde el conjunto no indexado proviene, por ejemplo, del resultado de una consulta espacial. La forma ingenua de resolver este problema consiste en construir un índice espacial para el conjunto no indexado y aplicar algoritmos estándar en este escenario. Nuestro algoritmo se basa en particionar el espacio del conjunto que no tiene índice y en las propiedades de un conjunto de métricas definidas entre dos MBRs. Por medio de una serie de experimentos, que consideraron datos sintéticos, se evaluó el rendimiento de nuestro algoritmo comparado con el del algoritmo ingenuo. Los resultados obtenidos muestran que nuestro algoritmo requiere entre un 0.4% y un 35% de operaciones de entrada/salida de las efectuadas por el algoritmo ingenuo. Además, el algoritmo hace uso de una regla de filtrado de puntos que alcanzó experimentalmente un 70% de efectividad.

1 Introducción

Las Bases de Datos Espaciales (BDE) reciben permanente atención dado el amplio rango de aplicaciones informáticas que tienen, destacando entre ellas las que se relacionan con los Sistemas de Información Geográfica (SIG). Muchos de los fabricantes de Sistemas de Administración de Bases de Datos, tales como Oracle, Postgres, entre otros, han incorporado en sus productos capacidades para manejar tipos de datos espaciales (definición, métodos de acceso, lenguaje de consulta, etc.) con el propósito de facilitar la implementación de aplicaciones espaciales.

Para las BDEs existen varios tipos de consultas tales como WQ (Window Query), EMQ (Exact Match Query), IQ (Intersection Query), entre otras [6, 21]. Una consulta espacial relevante, denominada *k-CPQ(P, Q)*, es la que determina el par o pares de vecinos más cercanos de dos conjuntos P y Q de objetos. Por ejemplo, uno podría querer encontrar el hotel más cercano a alguna estación

del metro, o encontrar en una imagen el tumor más cercano a un organo vital. Existen soluciones apropiadas [20] para este problema en el área de la geometría computacional, donde se asume que existe una capacidad suficiente de memoria principal para almacenar los conjuntos P y Q. Sin embargo el problema también es importante y se ha estudiado en el contexto de BDEs [3, 4, 10], en donde una consideración importante es que los dos conjuntos de objetos no siempre van a caber en memoria principal. Se conocen varios algoritmos para resolver *k-CPQ(P, Q)* en este contexto. Algunos, como los propuestos en [3, 4, 10], consideran que ambos conjuntos se encuentran almacenados en un índice espacial (típicamente un R-tree). Otros, como el propuesto en [17], resuelven el problema suponiendo que ninguno de los dos conjuntos se encuentra indexado. En [7] se aborda el caso en que sólo uno de los conjuntos cuenta con un índice. Estos dos últimos escenarios surgen típicamente en los siguientes casos (entre otras posibilidades): (a) cuando uno de los dos o ambos conjuntos de objetos espaciales provienen de resultados de consultas a Bases de Datos, (b) cuando los datos provienen de archivos secuenciales generados por dispositivos de captura de datos o (c) cuando los datos son la salida de un programa computacional de simulación de algún fenómeno espacial. Es decir que se trata de situaciones frecuentes.

El caso que se considera en este artículo es el mismo abordado en [7], es decir, cuando sólo uno de los dos conjuntos cuenta con índice espacial. Sin embargo, nuestro algoritmo difiere del propuesto en ese trabajo en que genera una partición del espacio del conjunto sin índice que depende de la distribución espacial de los objetos y en que, además, se apoya en una regla de filtrado basada en las propiedades de las métricas MINMINDIST y MINMAXDIST definidas entre MBRs y propuestas originalmente en [3, 4].

El resto del artículo está organizado de la siguiente manera. En la Sección 2 se define formalmente el problema, las métricas utilizadas por el algoritmo y se revisan los trabajos relacionados. En la Sección 3 se presenta nuestra propuesta detallando las ideas detrás del algoritmo. En la Sección 4 se muestran una serie de experimentos cuyos resultados permiten mostrar las ventajas de nuestra propuesta. Finalmente, en la Sección 5 se entregan algunas conclusiones.

2 Definición del Problema, Métricas utilizadas y Trabajos relacionados

Formalmente el problema de encontrar el par de objetos más cercanos se puede definir de la siguiente manera [3, 4]. Sean $P = \{p_1, p_2, \ldots, p_n\}$ y $Q = \{q_1, q_2, \ldots, q_m\}$ dos conjuntos de puntos del plano euclideano. El par de vecinos más cercano de los dos conjuntos, 1-$CPQ(P, Q)$, corresponde al par

$$(p_z, q_l), p_z \in P \wedge q_l \in Q$$

tal que

$$dist(p_i, q_j) \geq dist(p_z, q_l), \forall p_i \in P \wedge \forall q_j \in Q,$$

donde *dist* representa la distancia usual en el plano euclideano.

En otras palabras, el par de vecinos más cercanos de P y Q es el par que tiene la menor distancia entre todos los pares de objetos que pueden ser formados eligiendo un objeto de P y otro de Q. El problema se puede extender a encontrar los k-pares de vecinos más cercanos con $k > 1$, es decir, k-$CPQ(P, Q)$.

Nuestro algoritmo utiliza varias de las métricas descritas en [3, 5] definidas entre dos MBRs (del inglés Minimum Bounding Rectangle). El MBR para un conjunto finito C de puntos se define como el menor rectángulo paralelo a los ejes de coordenadas que incluye a todos los puntos de C. Nótese que, dada esta definición, cada lado del MBR incluye siempre al menos un punto de C. Las métricas se pueden visualizar gráficamente en la Fig. 1. Estas métricas son utilizadas por una serie de algoritmos propuestos en [3, 5].

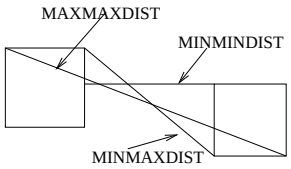

Fig. 1. Dos MBRs y las métricas definidas entre ellos

Las definiciones de estas métricas son las siguientes: sean N_P y N_Q dos conjuntos finitos de puntos y M_P y M_Q los MBRs de N_P y N_Q respectivamente. Sea r_1, r_2, r_3 y r_4 los cuatro lados de M_P y s_1, s_2, s_3 y s_4 los cuatros lados de M_Q. Se define MINDIST(r_i, s_i) como la distancia mínima entre dos puntos que caen sobre r_i y s_i. De la misma manera se define MAXDIST(r_i, s_i) como la máxima distancia entre dos puntos que caen sobre r_i y s_i. Basándose en MINDIST y MAXDIST las métricas de la Fig. 1 se definen de la siguiente manera:

$$MINMINDIST(M_P, M_Q) = \min_{i,j}\{MINDIST(r_i, s_j)\}.$$

En el caso de que los MBRs se intersecten, MINMINDIST(M_P, M_q) se define como cero. Las siguientes dos métricas, sin embargo, se pueden definir tanto si los MBRs se intersectan como si no.

$$MINMAXDIST(M_P, M_Q) = \min_{i,j}\{MAXDIST(r_i, s_j)\}.$$

y

$$MAXMAXDIST(M_P, M_Q) = \max_{i,j}\{MAXDIST(r_i, s_j)\}.$$

Para cada par de puntos (p_i, q_j), $p_i \in M_P$ y $q_j \in M_Q$, se cumple la siguiente relación:

$$\begin{aligned} &MINMINDIST(M_P, M_Q) \leq dist(p_i, q_j) \\ &\leq MAXMAXDIST(M_P, M_Q) \end{aligned} \quad (1)$$

Además siempre existe al menos un par de puntos (p_i, q_j), con $p_i \in N_P$ y $q_j \in N_Q$ tal que

$$dist(p_i, q_j) \leq MINMAXDIST(M_P, M_Q) \quad (2)$$

El problema $k\text{-}CPN(P, Q)$ en el contexto de las BDEs se puede entender como un tipo especial de join espacial, operador para el cual se han propuesto variados algoritmos [2, 8, 11, 13–16, 19]. Una revisión exhaustiva de las diferentes técnicas para procesar la operación de join espacial se puede encontrar en [12]. Para el caso particular de $k\text{-}CPN(P, Q)$ también se han propuesto varios algoritmos [3, 4, 10]. Dichos algoritmos consideran que ambos conjuntos se encuentran indexados mediante una estructura de datos espacial R-tree [9]. Un algoritmo reciente [17] aborda el caso en que ninguno de los dos conjuntos cuenta con índice espacial. El algoritmo en una primera etapa particiona el espacio total de cada conjunto asignando a cada celda o bucket de la partición a) un buffer de memoria, b) un MBR que encierra todos los objetos contenidos en la celda y c) una referencia a una lista de bloque de disco (o archivo) en donde se almacenan los objetos de la celda o bucket. En una etapa posterior, procesa las listas de objetos apoyándose en las métricas definidas entre MBRs. En [7] se aborda el mismo escenario considerado en este trabajo. Para resolverlo se propone un algoritmo que mantiene en memoria una partición regular y que no toma en cuenta la distribución de los objetos en el espacio. Este algoritmo tampoco aprovecha las propiedades de las métricas definidas entre dos MBRs para mejorar su rendimiento. Existen otros algoritmos que resuelven variantes del problema. Por ejemplo, encontrar los k pares más cercanos en un rango determinado [18] o cuando ambos conjuntos son iguales [20].

3 Algoritmo propuesto

En esta sección describimos *VA-File/Rtree Bucket Closest Pair* (VR-BCP), un nuevo algoritmo para encontrar el par de vecinos más cercanos sobre conjuntos en los cuales sólo uno de ellos cuenta con un índice espacial R-tree. En la Sección 3.1 se exponen las ideas generales del algoritmo incluyendo las estructura de datos utilizadas y algunos lemas necesarios. Posteriormente, en la Sección 3.2 se describe de manera detallada el algoritmo.

3.1 Ideas generales del algoritmo

VR-BCP se inspira en las ideas descritas en [17] para particionar un espacio y en las expuestas en [3, 5] para encontrar el par de vecinos más cercanos a partir de dos conjuntos indexados por un R-tree.

Partición del espacio del conjunto sin índice. Para la partición del espacio ocupado por el conjunto no indexado se usó la estrategia definida en [1] para la estructura VA-File. La idea es particionar el espacio (d-dimensional en general,

2-dimensional para efectos del presente trabajo) en 2^b particiones, de modo que cada partición 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 $i = 1...d$. Para efectos de nuestro trabajo consideramos $b = b_1 + b_2$, dado que estamos en 2 dimensiones. 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:

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

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. 2 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 (objetos), para lo cual se pueden tomar en cuenta todos los objetos del conjunto o bien, de manera estocástica, se puede utilizar una muestra de ese conjunto [1]. Para efectos de nuestro algoritmo consideraremos esta última posibilidad, con una cantidad de puntos igual a *sample*, un parámetro del algoritmo, Alg. 1.

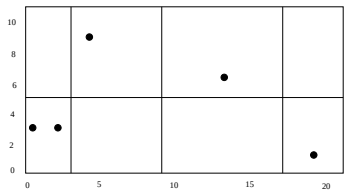

Fig. 2. Ejemplo de una partición considerando $d = 2$ y $b = 3$

Estructura de datos de los buckets. Nuestro algoritmo hace uso, para cada celda de la partición o bucket, de la siguiente estructura de datos: $\langle idBucket, MBR, pBuffer, pFile \rangle$, donde $idBucket$ corresponde al identificador de b bits de la partición o bucket, MBR es el rectángulo mínimo que encierra todos los puntos incluídos en la partición, $pBuffer$ es una referencia a un buffer en memoria principal y $pFile$ es una referencia a un archivo o a un bloque de disco, según si se opta por mantener en archivos separados los objetos de cada

bucket o si se decide tener un sólo archivo compartiendo bloques de diferentes buckets. Nótese que *MBR* incluye tanto los objetos almacenados en el buffer como los almacenados en disco. El propósito del buffer es almacenar de manera temporal los objetos antes de ser grabados definitivamente en disco. Esto tiene la ventaja de evitar que por cada objeto incluido en un bucket se deba realizar un acceso a disco.

Lema utilizado. Haremos uso del lema indicado más abajo para efectos de demostrar la correctitud de nuestro algoritmo para 1-$CPQ(P,Q)$, donde P y Q son dos conjuntos finitos de puntos del plano. Suponemos que tanto P como Q se encuentran particionados en subconjuntos $\pi_i, i = 1...m$ y $\rho_j, j = 1...n$ respectivamente. Es decir que $\pi_i \cap \pi_j = \emptyset$ para $i \neq j$, y $\rho_i \cap \rho_j = \emptyset$ para $i \neq j$, $\pi_1 \cup \pi_2 \cup ... \cup \pi_m = P$, y $\rho_1 \cup \rho_2 \cup ... \cup \rho_n = Q$. Llamaremos R_i al MBR de π_i, para $i = 1...m$ y S_i al MBR de ρ_j, para $j = 1...n$; y llamaremos P_{MBR} al conjunto de los R_i y Q_{MBR} al conjunto de los S_j.

Considérense las métricas definidas en la Sección 2 y notemos que si $d =$ *MINMAXDIST*(M_P, M_Q) para algún $M_P \in P_{MBR}, M_Q \in Q_{MBR}$ entonces d es una cota superior para 1-$CPQ(P,Q)$. En efecto asumiendo $d =$ *MAXDIST*(r_i, s_j) tenemos, por definición de MBR, al menos un punto $p_k \in P$ en r_i y al menos un punto $q_l \in Q$ en s_j, luego $d \geq dist(p_k, q_l)$ (por definición de *MAXDIST*). Esta observación nos permite establecer el siguiente lema:

Lema 1 *Si para algún $M_P \in P_{MBR}$ existen dos MBRs $M'_P \in P_{MBR}, M'_Q \in Q_{MBR}$ tales que para todo $M_Q \in Q_{MBR}$*

$$MINMINDIST(M_P, M_Q) > \textit{MINMAXDIST}(M_{P'}, M_{Q'}),$$

y si 1-$CPQ(P,Q) = dist(p_i, q_j)$, entonces $p_i \notin M_P$. Es decir, podemos descartar los puntos de M_P para efectos del cálculo de 1-$CPQ(P,Q)$.

Demostración: Claramente si para algún x tenemos que $(\forall M_Q \in Q_{MBR})$*MINMINDIST*$(M_P, M_Q) > x$, entonces para todo $p \in M_P, q \in Q$, $dist(p,q) > x$. Pero si tenemos que $x =$ *MINMAXDIST*$(M_{P'}, M_{Q'})$, como x es una cota superior para 1-$CPQ(P,Q)$, un punto $p \in P$ tal que $d(p,q) =$ 1-$CPQ(P,Q)$ no puede pertenecer a M_P.

A partir de los resultados del Lema 1 se establece una regla de filtrado la cual se detalla en la Sección 3.2. Dicha regla permite descartar objetos en la etapa 1 (Alg. 1) y por lo tanto disminuir el tamaño del conjunto no indexado en la etapa 2.

3.2 Descripción del algoritmo

VR-BCP considera dos etapas (ver Alg. 1). La primera contempla la partición del espacio del conjunto sin índice en un conjunto de buckets según lo explicado en la Sección 3.1. La segunda consiste en obtener el par de vecinos más cercanos considerando el conjunto de buckets y el R-tree del conjunto indexado de puntos.

Alg. 1 Algoritmo VR-BCP

1: **VR-BCP(Rtree** r**, File** f**, int** b**, int** *sample*) {r es el índice del primer conjunto, f el archivo con el segundo conjunto, b es la cantidad de bits que tendrán los identificadores de los buckets, según lo explicado en la sección 3.1, y *sample* es la cantidad de puntos de la muestra que se utilizarán para generar las particiones}
2: Bucket bucket[] = ParticionaryFiltrar(r, f, b, *sample*) {**etapa 1**}
3: Punto p = oneIndexCP(r, *bucket*[]) {**etapa 2**}
4: **return** p

Alg. 2 Etapa 1: Partición y filtrado

1: **ParticionaryFiltrar(Rtree** r**, File** f**, int** b**, int** *sample*)
2: Sea n el nodo raíz de r
3: Sea p un arreglo que contiene *sample* puntos de f
4: Sea $d = 2$ {d es el número de dimensiones del espacio}
5: Sea pc=Particionar(b, p, d) {Generar los puntos de partición en cada dimensión}
6: Sea Bucket *buck*[] =CreateBuckets(pc, b, p) {Se crean los buckets vacios en base a los puntos de partición establecidos para cada una de las coordenadas}
7: **for** cada punto $q \in p$ **do**
8: j = BucketNumber(pc[][], q){Se obtiene en función de las coordenadas de q el número (identificador) de b bits del bucket asociado según lo explicado en 3.1}
9: insertar q en el bucket *buck*[j]
10: **end for**
11: **for** cada punto q en f y que no esté en p **do**
12: j = BucketNumber(pc[][], q)
13: **if** *buck*[j].$pBuffer$ está lleno **then**
14: **if** Filtrar(*buck*[j], n, j, b) **then**
15: redefinir *buck*[j].MBR en función del nuevo punto q, si se da el caso
16: **else**
17: grabar en disco los puntos almacenados en el buffer *buck*[j].$pBuffer$
18: **end if**
19: Limpiar *buck*[j].$pBuffer$
20: **end if**
21: Insertar q en bucket *buck*[j]
22: **end for**
23: **return** *buck*

La etapa 1 se materializa en Alg. 2. En primer lugar se crean las particiones (ver Alg. 3) usando *sample* puntos del conjunto no indexado almacenado en archivo f (líneas 5 y 6 de Alg. 2). Una vez que los buckets han sido creados, los puntos se van insertando en los correspondientes buckets de acuerdo a los valores de sus coordenadas según lo explicado en la sección 3.1. Previo a la inserción de un punto en el buffer de un bucket l, el algoritmo aplica una regla de filtrado cada vez que el buffer de l se llena, que permite decidir si los objetos pertenecientes a l (almacenados tanto en el buffer $pBuffer$ como en el archivo $pFile$) pueden ser descartados. La regla toma en cuenta dos conjuntos de MBRs. El primero, R_1, corresponde a los MBRs definidos en las entradas del nodo raíz del R-tree del conjunto indexado y el segundo, R_2, a los MBRs de los buckets del conjunto no indexado. Sea l el bucket cuyo buffer se encuentra lleno. En primer lugar se obtiene el menor valor de *MINMINDIST* entre el MBR del bucket l y cada uno de los MBRs de R_1; sea m este valor. Posteriormente se verifica si existe algún par (r_1, r_2), con $r_1 \in R_1$, $r_2 \in R_2$ y r_2 distinto del MBR del bucket l tal que *MINMAXDIST*$(r1, r2)$ sea inferior a m. Si esta última condición se cumple, entonces todos los objetos que se encuentran almacenados en el bucket pueden ser descartados. El lema 1 hace ver que esta regla es correcta; ésta se detalla en Alg. 4.

Alg. 3 Algoritmo para particionar

1: **Particionar(int** b**, Point** $p[]$**, int** d**)**
2: Para $1 \leq i \leq d, b_i = \lfloor \frac{b}{d} \rfloor + \begin{cases} 1 & i \leq b \mod d \\ 0 & \text{en caso contrario} \end{cases}$ {Fórmula explicada en la sección 3.1}
3: $n_i = 2^{b_i}$, $1 \leq i \leq d$ {Cantidad de puntos de partición en la dimensión i, aparte del punto $pc[i][0]$}
4: Sea pc una matriz con d filas en donde cada fila, de tamaño $n_i, 1 \leq i \leq d$, almacena los puntos de partición de la dimensión i
5: **for** $i = 1$ to d **do**
6: Sort(i, p) {Ordena los puntos del arreglo p según la dimensión i}
7: $pc[i][0]$ = límite inferior del espacio en la dimensión i
8: $pc[i][n_i]$ = límite superior del espacio en la dimensión i
9: let $np = \left\lfloor \frac{size\ of\ p}{n_i} \right\rfloor$
10: $k = 1$ {Subíndice de los sucesivos puntos de partición}
11: $j = np$ {Subíndices de los puntos de p que se van tomando para obtener los puntos de partición}
12: **while** $j \leq$ (size of p) $- np$ **do**
13: $pc[i][k] = p[j].coordenada[i]$ {El k-ésimo punto de partición según la coordenada i se toma como el valor de la coordenada i del j-ésimo punto de p}
14: $k = k + 1$
15: $j = j + np$
16: **end while**
17: **end for**
18: **return** pc

Alg. 4 Algoritmo de filtrado

```
1: Filtrar(Bucket buck[], NodeOfRtree n, int k, int b)
2: Sea nbuckets = 2^b el número de buckets (total de particiones)
3: Sea m = min{MINMINDIST(buck[k].MBR, n.entries[i].MBR) }, 1 ≤ i ≤ n.MaxEntry
4: for i = 1 to n.MaxEntry do
5:    for j = 1 to nbuckets do
6:       if j ≠ k and MINMAXDIST(n.entries[i].MBR, buck[j].MBR) < m then
7:          Descartar todos los objetos del bucket buck[j] almacenados en disco
8:          return true
9:       end if
10:   end for
11: end for
12: return false
```

La segunda etapa (ver Alg. 5) toma como entrada el R-tree del conjunto indexado y las particiones (buckets) generadas en la etapa 1. El algoritmo parte con un valor infinito para la distancia D, la cual progresivamente se va ajustando y converge a la distancia del entre los dos puntos más cercanos. Para ello desciende por el R-tree desde la raíz hacia las hojas. Por cada nodo interno del R-tree cuyos hijos no son hojas, se ajusta la distancia D de la solución candidata por medio de la métrica MINMAXDIST entre todos los posibles pares de MBRs, donde uno de los MBRs pertenece a una de las entradas del nodo y el otro al definido en un bucket (líneas 9 y 10 Alg. 5). El valor de D se utiliza para podar el R-tree descartando aquellas ramas cuyo MBR (definido en la correspondiente entrada del nodo interno del R-tree) tenga valores de MINMINDIST mayores que la distancia D para todos los MBRs de los bucket. Para el caso en que los hijos del nodo interno correspondan a nodos hojas el algoritmo inserta en un heap h, con nodos ordenados desde la raíz a las hojas en orden creciente de MINMINDIST, todos los pares $\langle e, b\rangle$, con e una entrada del nodo y b el bucket y cuyo $MINMINDIST(e.MBR, b.MBR) \leq D$ (líneas 17-19 Alg. 5). Luego todos los pares de h que cumplen la condición $MINMINDIST(e.MBR, b.MBR) \leq D$ se procesan calculando la distancia de cada posible par de puntos y eventualmente mejorando la solución alcanzada hasta este punto. Notar que a partir del momento en que $MINMINDIST(e.MBR, b.MBR) > D$ no es necesario considerar los restantes pares en h, pues no hay posibilidad que mejoren la solución (líneas 21-33 Alg. 5). Finalmente el algoritmo continua revisando ramas del R-tree que no han sido descartadas.

4 Experimentación.

En esta sección se presentan los resultados de una serie de experimentos llevados a cabo con el objeto de demostrar el rendimiento de nuestro algoritmo. Se comparó VR-BCP con el algoritmo básico, que llamaremos CPN, propuesto en [3, 4] el cual encuentra el par de vecinos más cercano sobre dos conjuntos

Alg. 5 Etapa 2. Considera como entrada un R-tree y un conjunto de buckets

oneIndexCP(Rtree r**, Bucket** $buck[]$**)** {Suponemos que las entradas de cada nodo interno del R-tree se almacenan en un arreglo *entries* donde cada una de estas entradas tiene la estructura $\langle MBR, ptr\rangle$, donde *MBR* es el rectángulo mínimo que encierra a todos los puntos en el subárbol cuyo nodo raíz es apuntado por *ptr*}
Sea n el número de buckets y m el número de entradas en r
Sea $pcp = null$ el par de vecinos más cercanos y $D = \infty$ la distancia inicial
Si r es una hoja, calcular el par de vecinos más cercanos entre los objetos de r y el conjunto de buckets. Retornar el par de puntos y terminar.
Sea s una Pila
Push(s, r)
while not Vacio(s) **do**
  r_1 = Pop(s)
  $D_1 = \min\{MINMAXDIST(entries[i].MBR, buck[j].MBR)\}$, con $1 \leq i \leq m, 1 \leq j \leq n$ {Nótese que D_1 es una cota superior de la distancia mínima}
  $D = \min\{D, D_1\}$
  if los hijos de r_1 no son nodos hojas **then**
    for cada entrada $entries[i] \in r_1$, $1 \leq i \leq m$ tal que existe $buck[j]$, $1 \leq j \leq n$, con $MINMINDIST(entries[i].MBR, buck[j].MBR) \leq D$ **do**
      Push($s, entries[i].ptr$)
    end for{Los nodos del R-tree que no son empilados, se descartan}
  else
    Sea h un heap de tamaño $n \cdot m$ ordenado por MINMINDIST entre los MBRs de una entrada de r y un bucket
    for cada par $\langle entries[i], buck[j]\rangle$, $1 \leq i \leq m$ y $1 \leq j \leq n$, con $MINMINDIST(entries[i].MBR, buck[j].MBR) \leq D$ **do**
      Insertar(h,$\langle entries[i], buck[j]\rangle$)
    end for
  end if
  while not Vacio(h) **do**
    $\langle e, b\rangle$ =EliminarMin(h)
    if $MINMINDIST(entry.MBR, bucket.MBR) \leq D$ **then**
      pcp_1 = pcp($e.ptr, D, b.pFile$){Se calcula el par de vecinos más cercanos entre dos conjuntos de puntos que han sido llevados a memoria principal}
      D_1 =Distance($pcp_1.p, pcp_1.q$)
      if $D_1 < D$ **then**
        $pcp = pcp_1$
        $D = D_1$
      end if
    else
      break
    end if
  end while
end while
return *pcp*

indexados por un R-tree de la misma altura. Como unidad de rendimiento se utilizó la cantidad de operaciones de entrada/salida (accesos a disco) que cada algoritmo necesita realizar para resolver el problema.

4.1 Parámetros.

Se consideraron conjuntos de puntos de tamaño 100K[1], 300K, 500K y 700K distribuidos en el espacio de dos dimensiones $[0, 2] \times [0, 1]$ (ver Fig. 3). En cada prueba se utilizó la misma cantidad de puntos para ambos conjuntos. Se estudiaron 5 porcentajes de intersección (solapamiento) de las áreas de los conjuntos, a saber: 0%, 25%, 50%, 75% y 100%. Dichos porcentajes indican las fracciones de cada área que se intersectan. Se experimentó con varias cantidades de buckets (64, 128, 256 y 512) y se usaron 1.000 objetos (sample) para establecer límites de las particiones. Se utilizaron bloques de tamaño 1024 bytes. Para los R-tree se consideraron dos tipos de nodos, nodos internos y nodos hojas con un máximo de entradas de 28 y 51 respectivamente. Para resolver el problema con CPN, en primer lugar se crea un índice (R-tree) para el conjunto sin índice, y posteriormente se aplica el algoritmo a los dos conjuntos indexados. En la ejecución de CPN se contabilizaron todos los accesos a disco los que incluyen la lectura de los puntos, los requeridos para construir el R-tree y los accesos para encontrar el par de vecinos más cercanos a partir de los dos R-tree.

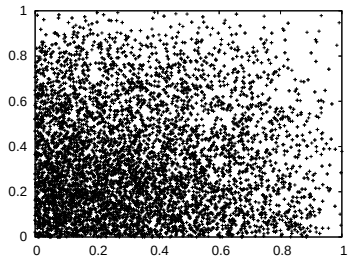

Fig. 3. Ejemplo de un conjunto de puntos utilizados en los experimentos

4.2 Discusión.

La Fig. 4 muestra la cantidad de accesos que VR-BCP y CPN requieren para resolver el problema. Se puede ver que VR-BCP supera a CPN en todos los casos requiriendo entre un 0.4% y 35% del número de accesos realizados por CPN. También podemos ver que en la medida que el porcentaje de

[1] 1K = 1000 puntos

intersección disminuye la ventaja de VR-BCP sobre CPN aumenta. Esto se debe principalmente al efecto de la regla de filtrado la cual es más efectiva conforme disminuye el porcentaje de intersección entre los conjuntos, lo cual se puede corroborar en la Fig. 5.

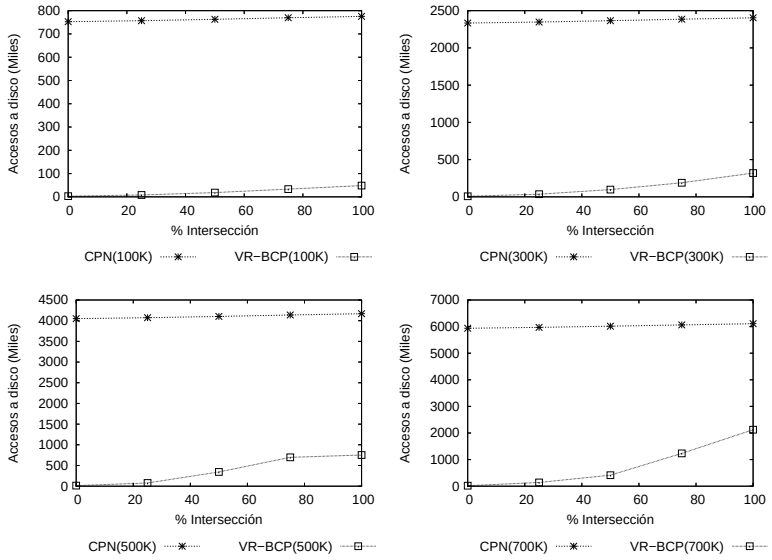

Fig. 4. Número de accesos a disco para conjuntos de 100K, 300K, 500K y 700K considerando 256 buckets

También medimos la efectividad de la regla de filtrado cuyos resultados se pueden ver en la Fig. 5. En dicha figura es posible apreciar que el porcentaje de filtrado varía entre 0% y 68%, y que en la medida que el porcentaje de intersección entre los conjuntos aumenta, la efectividad de la regla disminuye alcanzando 0% cuando el porcentaje de intersección se encuentra entre 50% y 75%. También se puede ver que en la medida que la cantidad de buckets aumenta también lo hace la efectividad de la regla. Por ejemplo, en la Fig. 5 (c) y (d), considerando un 50% de intersección, se puede pasar de 0% a un 10% de filtrado aproximadamente al aumentar el número de buckets de 64 a 256.

5 Conclusiones

En este trabajo se describe el algoritmo VR-BCP para obtener el par de vecinos más cercanos considerando que sólo uno de los conjuntos se encuentra almacenado en un índice espacial R-tree. Los resultados obtenidos a partir de una serie de experimentos preliminares que consideraron datos sintéticos, muestran que VR-BCP necesita entre un 0.4% y 35% del costo requerido por el algoritmo

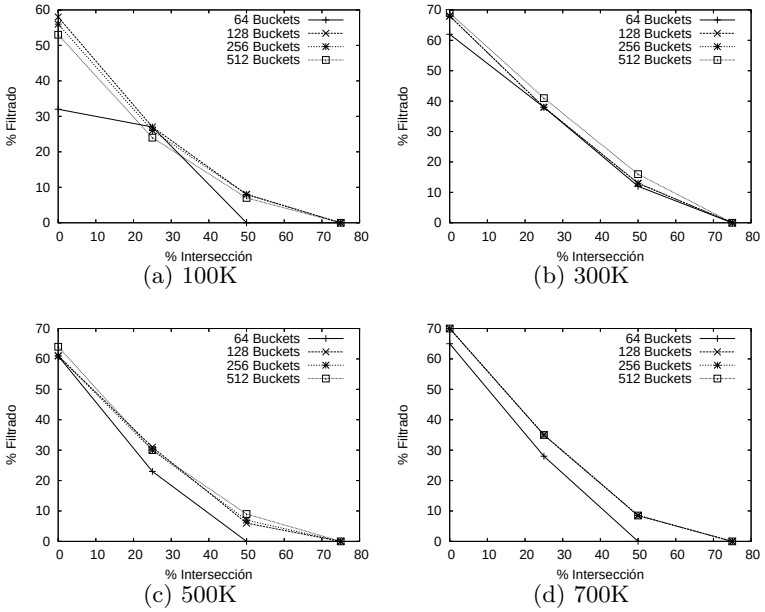

Fig. 5. % de descarte de la regla de filtrado para conjuntos de 100K, 300K, 500K y 700K considerando 64, 128, 256 y 512 buckets

ingenuo. En parte el rendimiento alcanzado por VR-BCP se explica por la regla de filtrado la cual alcanzó entre un 0% y 68% de efectividad. Como trabajo futuro pretendemos extender nuestro algoritmo para $k > 1$ y medir su rendimiento considerando datos reales y sintéticos.

References

1. Stephen Blott and Roger Weber. A simple vector-approximation file for similarity search in high-dimensional vector spaces. Technical report, Institute of Information Systems, ETH Zentrum, Zurich Switzerland, 1997.
2. Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *ACM SIGMOD Conference on Management of Data*, pages 237–246, Washington, DC, USA, 1993. ACM.
3. Antonio Corral. *Algoritmos para el Procesamiento de Consultas Espaciales utilizando R-trees. La Consulta de los Pares Más Cercanos y su Aplicación en Bases de Datos Espaciales.* PhD thesis, Universidad de Almería, Escuela Politécnica Superior, España, Enero 2002.
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. Closest pair queries in spatial databases. In *SIGMOD '00:*

Proceedings of the 2000 ACM SIGMOD international conference on Management of data, pages 189–200, New York, NY, USA, 2000. ACM Press.
6. Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
7. Luis Gajardo and Gilberto Gutiérrez. El par de vecinos más cercanos: hacia propuestas de algoritmos en escenarios en que uno de los conjuntos no se encuentra indexado. In *Encuentro Chileno de Computación, Santiago (Chile)*, 2009.
8. Oliver Günther. Efficient computation of spatial joins. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 50–59, Washington, DC, USA, 1993. IEEE Computer Society.
9. Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD Conference on Management of Data*, pages 47–57. ACM, 1984.
10. Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *ACM SIGMOD Conference on Management of Data*, pages 237–248, Seattle, WA, 1998.
11. Yun-Wu Huang, Ning Jing, and Elke A. Rundensteiner. A cost model for estimating the performance of spatial joins using r-trees. In *SSDBM*, pages 30–38, 1997.
12. Edwin H. Jacox and Hanan Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7, 2007.
13. Nikos Mamoulis and Dimitris Papadias. Slot index spatial join. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):211–231, 2003.
14. Lo Ming-Ling and Ravishankar Chinya. Spatial joins using seeded trees. In *ACM SIGMOD Conference on Management of Data*, pages 209–220, Minneapolis, Minnesota, USA, 1994.
15. Lo Ming-Ling and Ravishankar Chinya. Spatial hash-joins. In *ACM SIGMOD Conference on Management of Data*, pages 247–258, Montreal, Canada, 1996.
16. Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 259–270, New York, NY, USA, 1996. ACM Press.
17. Miguel Pincheira, Gilberto Gutiérrez, and Luis Gajardo. Closest pair query on spatial data sets without index. In *XXIX International Conference of the Chilean Computer Society. (to appear)*, 2010.
18. Shaojie Qiao, Changjie Tang, Jing Peng, Hongjun Li, and Shengqiao Ni. Efficient k-closest-pair range-queries in spatial databases. In *Proceedings of the 2008 The Ninth International Conference on Web-Age Information Management*, WAIM '08, pages 99–104, Washington, DC, USA, 2008. IEEE Computer Society.
19. Kim Sang-Wook, Cho Wan-Sup, Lee Min-Jae, and Whang Kyu-Young. A new algorithm for processing joins using the multilevel grid file. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 115–123. World Scientific Press, 1995.
20. Jing Shan, Donghui Zhang, and Betty Salzberg. On spatial-range closest-pair query. In Thanasis Hadzilacos, Yannis Manolopoulos, John Roddick, and Yannis Theodoridis, editors, *Advances in Spatial and Temporal Databases*, volume 2750 of *Lecture Notes in Computer Science*, pages 252–269. Springer, 2003.
21. Shashi Shekhar and Sanjay Chawla. *Spatial databases - a tour*. Prentice Hall, 2003.