



Recuperación de Información Multimedia

Índices Multidimensionales (Árboles, Hashing, Filling Curves)

CC5213 – Recuperación de Información Multimedia

Departamento de Ciencias de la Computación

Universidad de Chile

Juan Manuel Barrios – <https://juan.cl/mir/> – 2019



Índices Multidimensionales

- Asumen que los datos son vectores
- Usan los valores de las coordenadas para agrupar vectores
 - Árboles: Agrupar vectores en regiones espaciales ordenadas jerárquicamente
 - Hashing: Asignar vectores a una o más tablas de tamaño fijo
 - Filling Curves: Convertir el espacio multidimensional en un espacio unidimensional

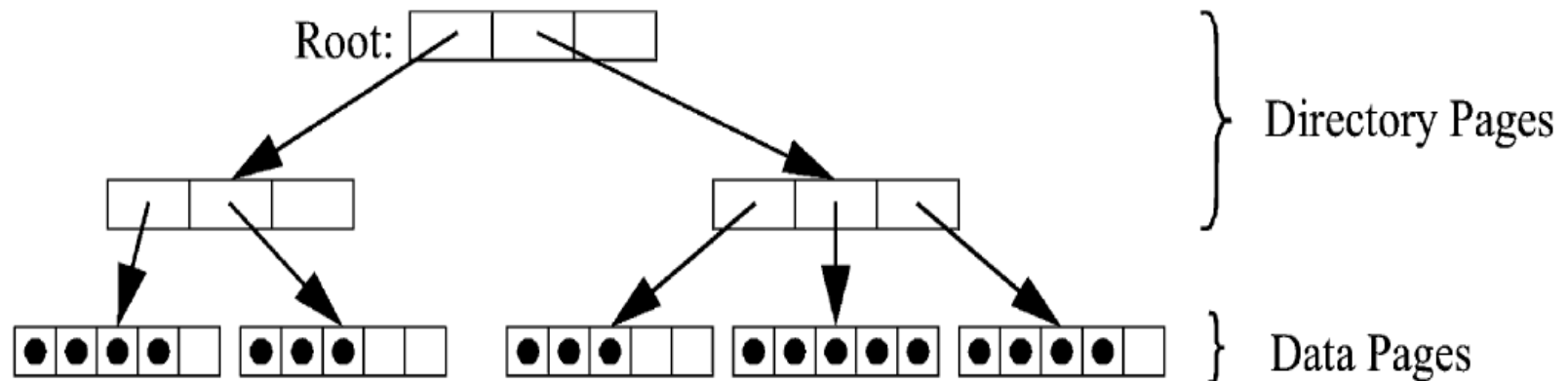


Árboles Multidimensionales

- En general son árboles balanceados
- Contienen dos tipos de nodos:
 - Nodos internos o Páginas de directorio
 - Describen una región espacial
 - Almacenan punteros a nodos hijos (internos/hojas)
 - Nodos hoja o Páginas de datos
 - Almacenan punteros a vectores (descriptores)
- Los nodos tienen una capacidad máxima de almacenamiento de punteros

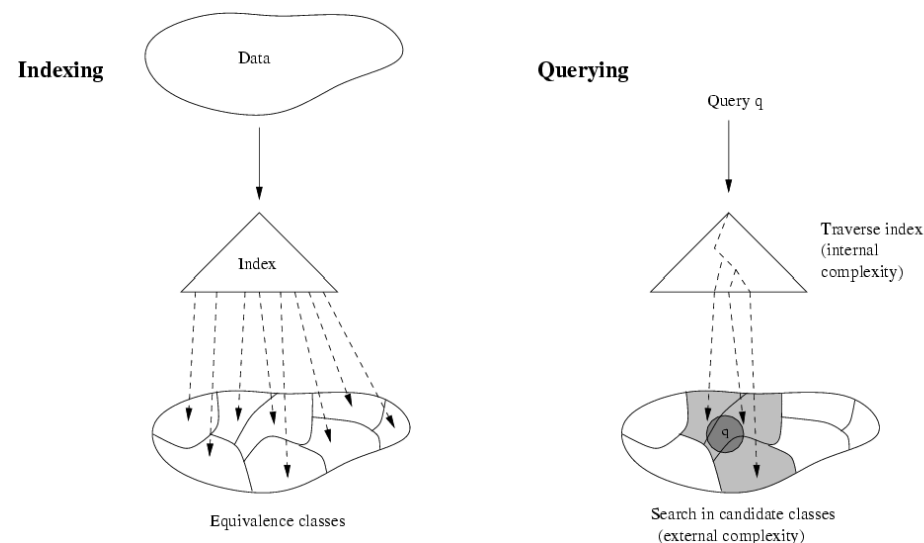
Árboles Multidimensionales

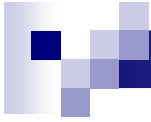
- Estructura básica



Árboles Multidimensionales

- Nodos internos definen regiones espaciales jerárquicas
 - Los elementos de un nodo están contenidos por su padre
- Puntos cercanos idealmente se almacenarán en la misma página de datos o subárbol
- Durante la búsqueda se visitan las regiones que tienen intersección con la bola de consulta





Árboles Multidimensionales

- Cada tipo de índice utiliza distintas formas de región espacial:
 - Hipercubo
 - Hiperesfera
 - Otras regiones convexas (cilindros, intersecciones, etc.)



Índices estáticos y dinámicos

- Fase offline: Se construye el índice con todos los vectores del dataset
- Fase online: El índice se utiliza para responder consultas
- Índice estático: No se modifica su estructura mientras está online
- Índice dinámico: Es posible modificar su estructura mientras está online
 - Operaciones de inserción y borrado incrementales con costo $O(\log n)$
 - Puede iniciar vacío y alimentarlo en la medida que se obtienen nuevos datos

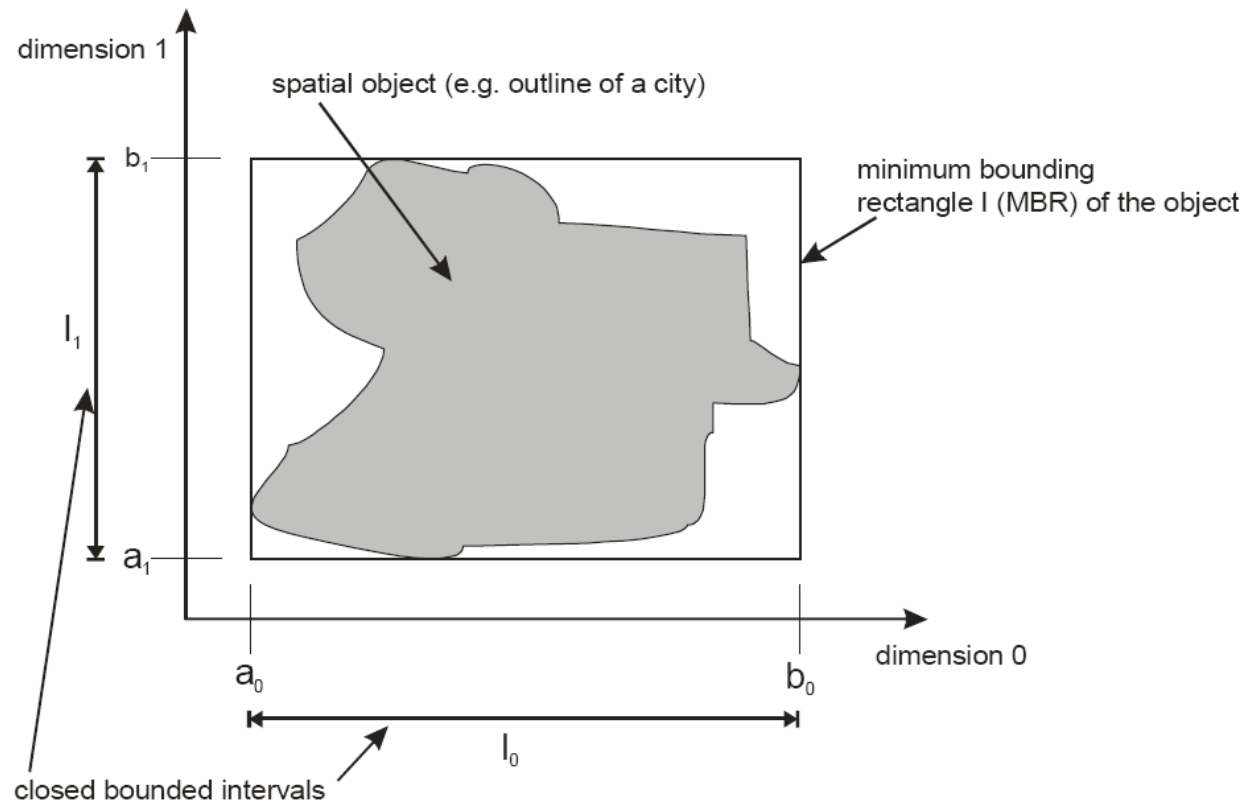


R-Tree

- Permite buscar objetos espaciales (ya sea vectores o figuras) en el espacio
- Árbol balanceado similar a un B-Tree
- Construye *Minimum Bounding Rectangles* (MBR) para agrupar objetos
- Estructura dinámica: inserciones y borrados incrementales

Estructura del R-Tree

- Ejemplo de región espacial (MBR)





Propiedades del R-Tree

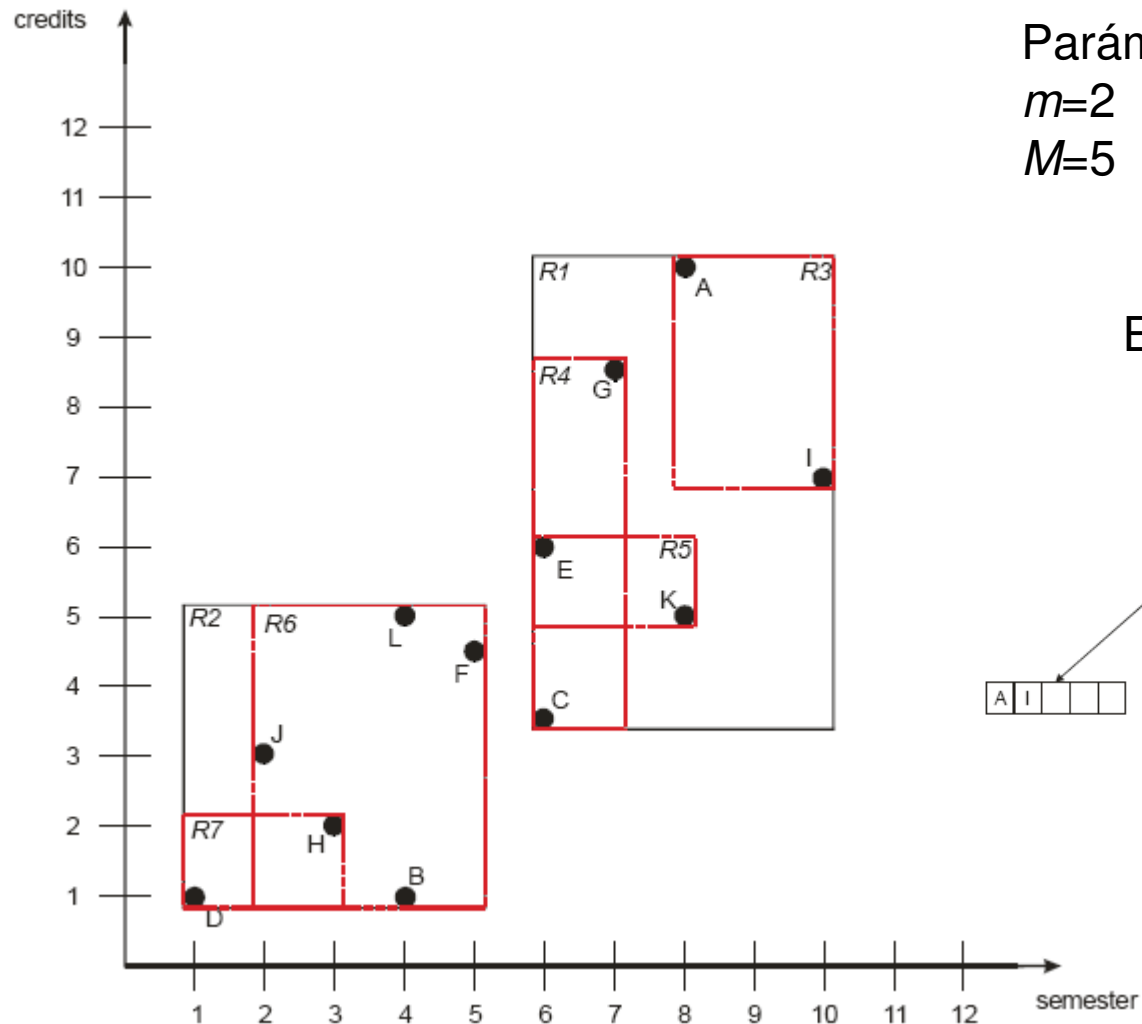
- Cada nodo (excepto la raíz) contiene entre m y M registros
 - M : número máximo de entradas en un nodo
 - m : número mínimo de entradas en un nodo
- Para cada nodo interno se calcula el menor rectángulo que contiene sus objetos hijos
 - Minimum Bounding Rectangle (MBR)
- La raíz contiene a lo menos dos hijos, excepto si es una hoja
- Todas las hojas están al mismo nivel



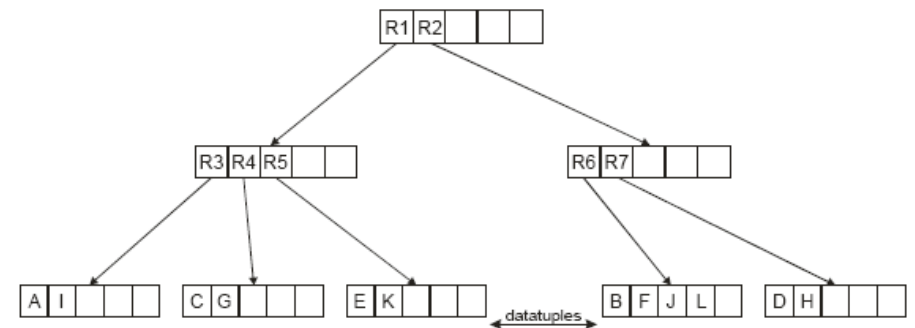
Estructura del R-Tree

- Cada nodo hoja contiene:
 - Una lista de datos (descriptores)
- Cada nodo interno contiene:
 - Una lista de nodos hijos
 - Un rectángulo d -dimensional que contiene espacialmente a todos los nodos hijos (ya sean regiones o datos):
 - Intervalos $[lb_j, ub_j]$ con el valor mínimo y máximo de los hijos a lo largo de la dimensión j

Ejemplo de R-Tree



Estructura del R-Tree



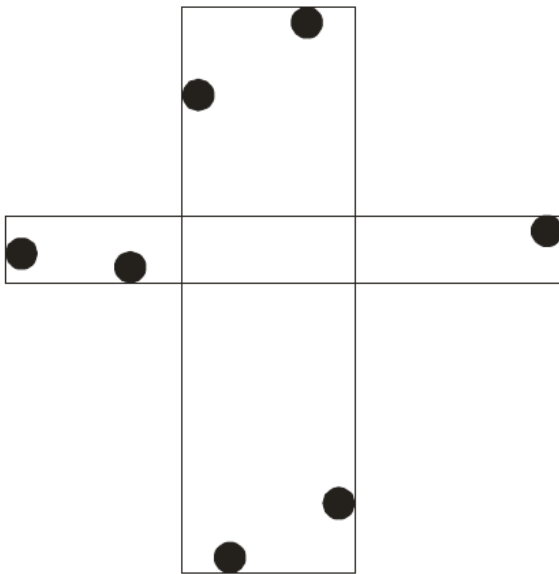


Inserción en R-Tree

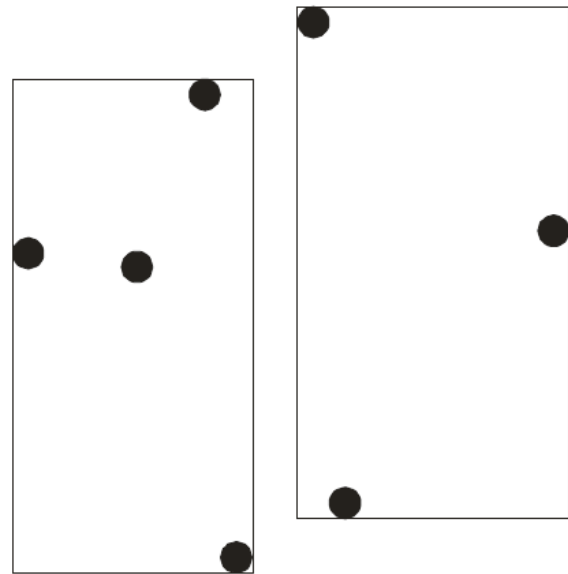
- Similar a insertar en un B-Tree
- Seleccionar una página de datos adecuada y agregar el objeto a esa página
 - Realizar una búsqueda y seleccionar la región donde debería encontrarse el objeto
 - Si ninguna región contiene el objeto elegir la región que requiere agrandarse menos
- Si se excede la capacidad máxima de la página de datos (*overflow*), dividir la página en dos regiones (*split*)
- Modificar la representación del nodo padre:
 - Agregar la nueva página de datos a la lista de hijos (si hubo overflow)
 - Ajustar el tamaño de la región (si se agrandó la página de datos)
- Si el número de hijos excede la capacidad del nodo padre, dividir el nodo padre. Proceder recursivamente hacia arriba en el árbol
- Si la raíz se divide, el árbol crece en un nivel

División de nodos

- *Split*: dividir $M+1$ elementos en dos conjuntos cada uno con al menos m elementos



Criterio 1 "Coverage"
Minimizar área de los MBRs



Criterio 2 "Overlap"
Minimizar intersección de MBRs



División de nodos

- Criterio usado: minimizar la suma de áreas
 - Reduce la probabilidad de visitar ambas regiones en el futuro
- Algoritmo exhaustivo:
 - Probar todas las posibles particiones y seleccionar la óptima (costo exponencial!)
- Algoritmo cuadrático:
 1. Escoger semillas: probar todos los pares de elementos y elegir el par que produce la mayor área.
 2. Cada semilla inicia un grupo.
 3. Mientras queden elementos para elegir:
 - Para cada elemento disponible calcular el aumento requerido por cada grupo para agregarlo.
 - Seleccionar para agregar el elemento con mayor diferencia en el aumento de ambos grupos.



Borrar y Actualizar en R-Tree

■ Borrar un elemento:

- ☐ Buscar el elemento a borrar y eliminarlo de la página de datos
- ☐ Si no se produjo *underflow* ajustar MBRs respectivos
- ☐ Si se produce *underflow* (el nodo queda con menos de m elementos), eliminar el nodo completo y reinsertar todos los datos restantes

■ Actualizar un elemento:

- ☐ Borrar el valor antiguo e insertar nuevo valor



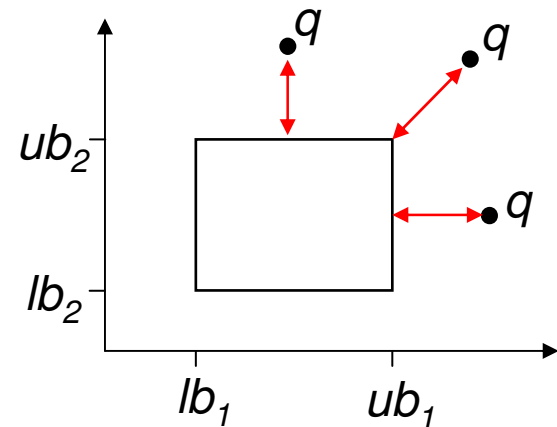
Búsqueda por rango en R-Tree

- Obtener todos los elementos que están dentro de la bola de consulta $B(q,r)$
- Búsqueda recursiva (iniciar en la raíz del árbol):
 - Si está en un nodo interno (contiene otros nodos):
 - Comparar cada nodo hijo con la bola de consulta y si la intersecta visitar el nodo recursivamente
 - Si está en una hoja (contiene elementos):
 - Comparar cada elemento con la bola de consulta y reportar los elementos relevantes, i.e. los que están dentro de $B(q,r)$.
- Notar que la bola de consulta puede intersectar varias regiones simultáneamente
- Una búsqueda por similitud puede requerir recorrer varios subárboles, incluso el árbol completo

MINDIST

- Distancia mínima entre objeto q y algún punto de la región R
- Distancia a la que potencialmente podría existir algún elemento en la región
- Para distancia euclidiana:

$$\text{MINDIST}(q, R) = \sqrt{\sum_{i=1}^n \begin{cases} (lb_i - q_i)^2 & \text{si } q_i < lb_i \\ 0 & \text{si } lb_i \leq q_i \leq ub_i \\ (ub_i - q_i)^2 & \text{si } q_i > ub_i \end{cases}}$$

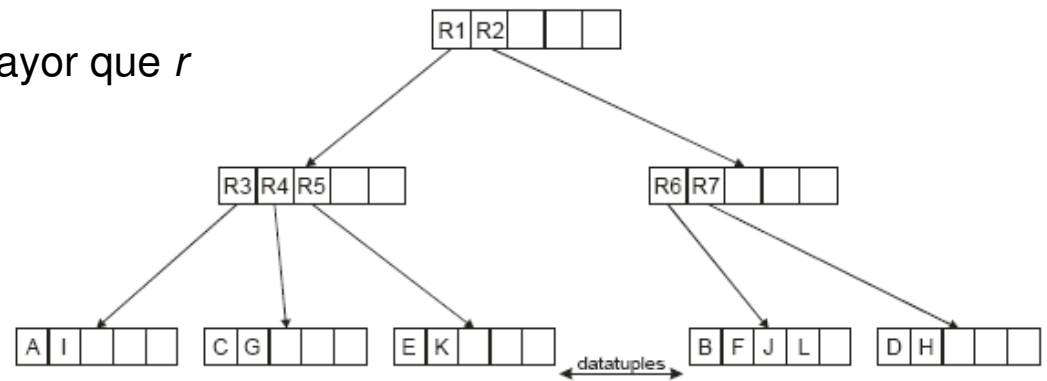
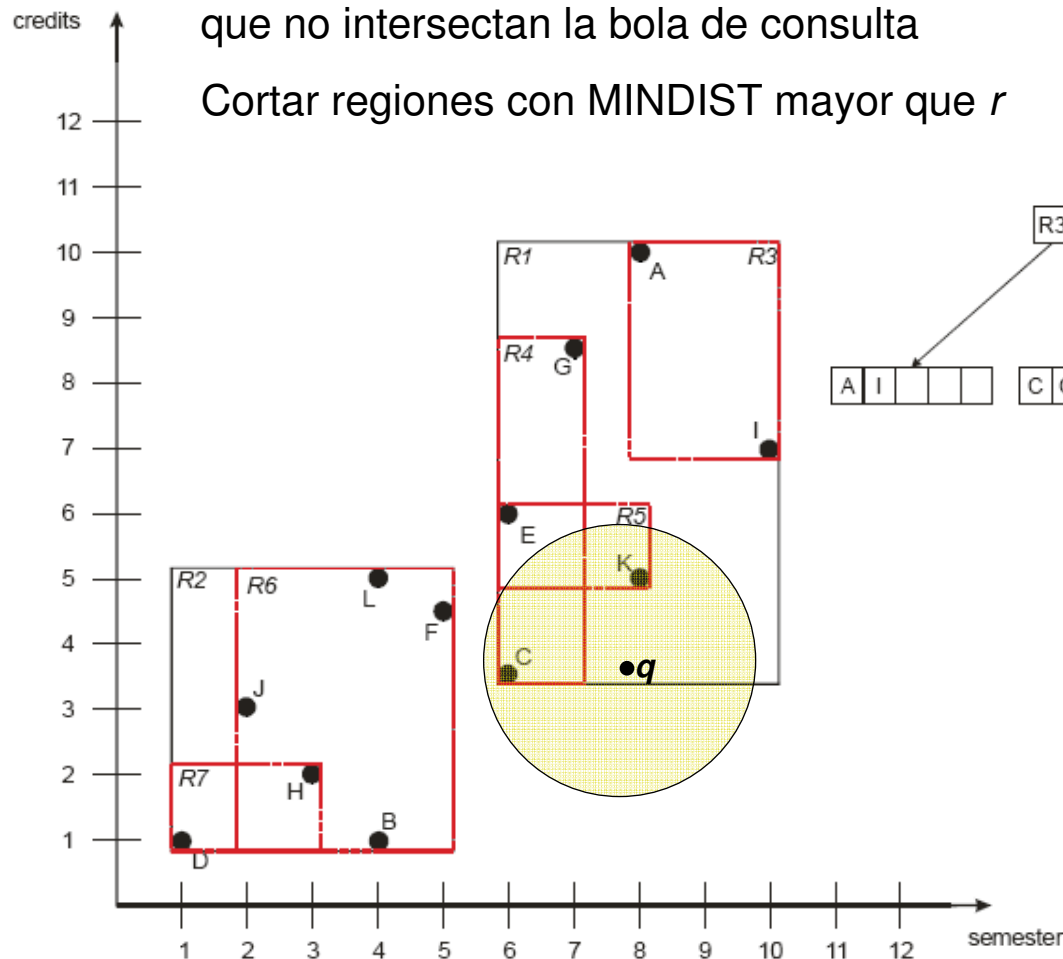


- Determinar si la región R intersecta la bola de consulta $B(q, r)$ consiste en determinar si $\text{MINDIST}(q, R) \leq r$

Búsqueda por rango en R-Tree

Búsqueda en profundidad cortando regiones que no intersectan la bola de consulta

Cortar regiones con MINDIST mayor que r



- R1?** Si intersecta, entrar
- R3?** No intersecta
- R4?** Si intersecta, entrar
 - $d(q, C)$ Es relevante, imprimir
 - $d(q, G)$ No es relevante
- R5?** Si intersecta, entrar
 - $d(q, E)$ No es relevante
 - $d(q, K)$ Es relevante, imprimir
- R2?** No intersecta



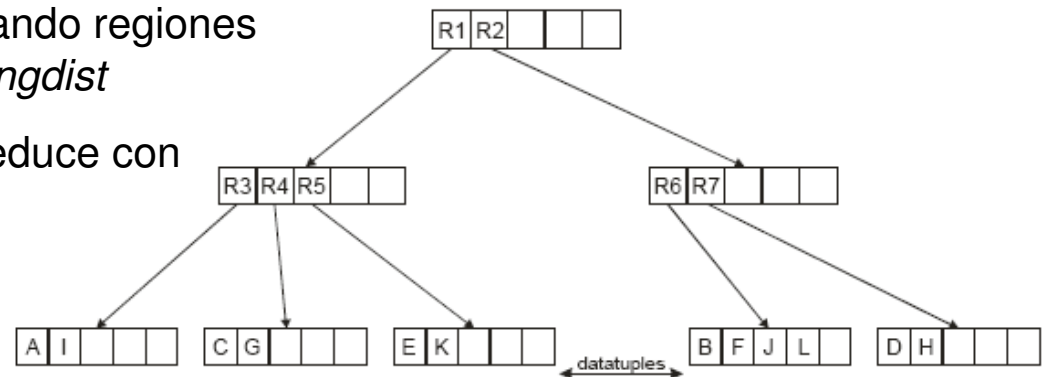
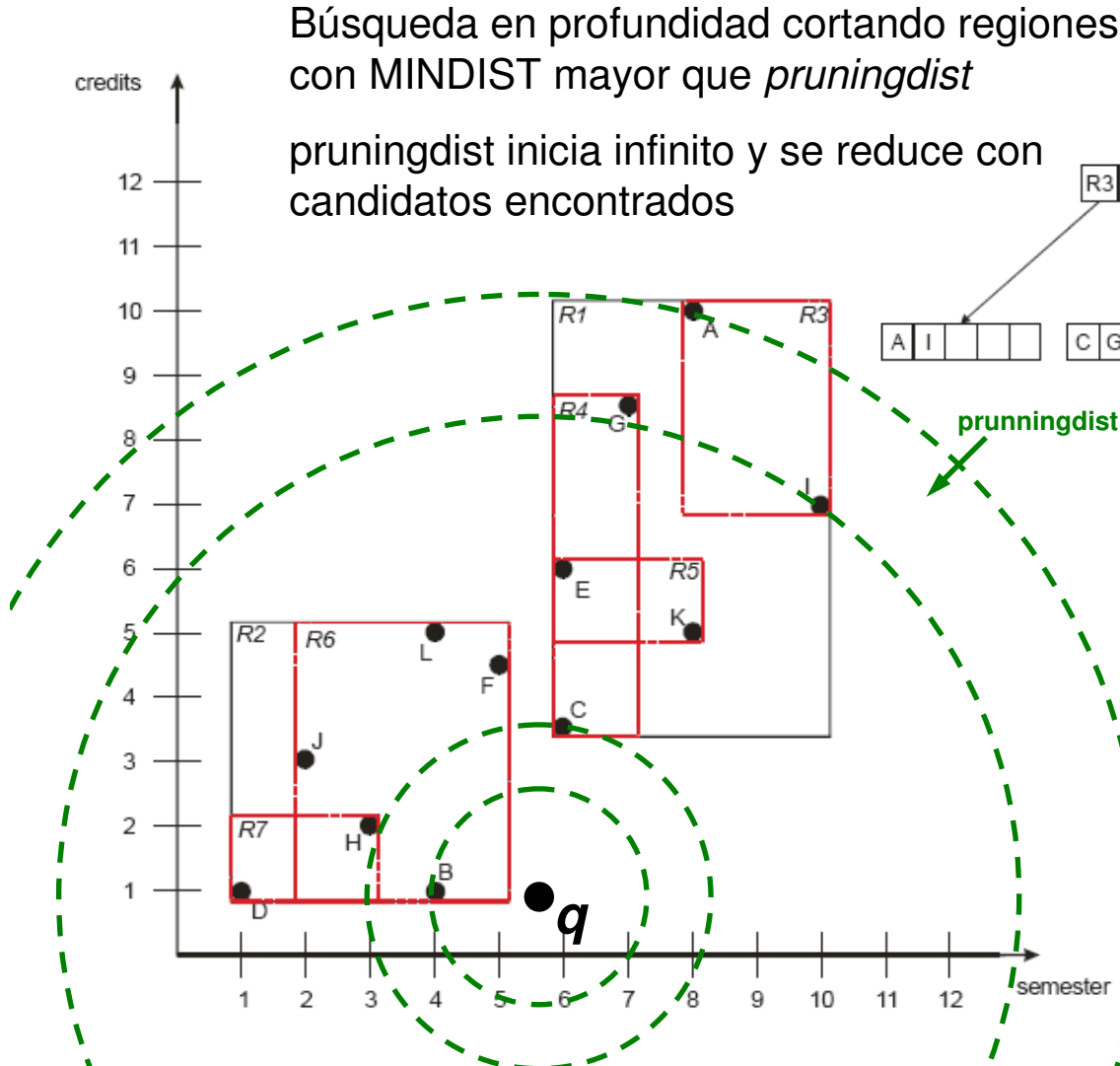
Búsqueda del NN en R-Tree

- Si uno supiera la distancia a la que está el vecino más cercano, bastaría con hacer una consulta por rango con el radio correcto
- Algoritmo recursivo (naive):
 - Aplicar la misma búsqueda en profundidad recursiva de la búsqueda por rango
 - El rango es desconocido, asumir un rango infinito y reducirlo en la medida que se encuentran candidatos
 - ***pruningdist*** es la distancia de corte que se va reduciendo al encontrar mejores candidatos

Búsqueda NN recursiva en R-Tree

Búsqueda en profundidad cortando regiones con MINDIST mayor que *pruningdist*

pruningdist inicia infinito y se reduce con candidatos encontrados



pruningdist=infinito

R1? Si interseca, entrar

R3? Si interseca, entrar

d(q,A) Nuevo candidato → reducir pruningdist

d(q,I) Nuevo candidato → reducir pruningdist

R4? Si interseca, entrar

d(q,C) Nuevo candidato → reducir pruningdist

d(q,G) No es relevante

R5? No interseca

R2? Si interseca, entrar

R6? Si interseca, entrar

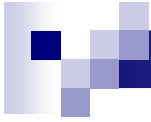
d(q,B) Nuevo candidato → reducir pruningdist

d(q,F) No es relevante

d(q,J) No es relevante

d(q,L) No es relevante

R7? No interseca



Búsqueda del NN en R-Tree

- Desventajas del algoritmo recursivo:
 - Toma un camino predefinido (por ejemplo, la rama izquierda del árbol)
 - El primer objeto candidato encontrado probablemente está lejos de la consulta
 - El algoritmo reduce el espacio de búsqueda lentamente
 - Muchos caminos deben visitarse inútilmente



Búsqueda NN por prioridad en R-Tree

- Búsqueda por prioridad de Hjaltason y Samet
- También llamado algoritmo BBF (Best Bin First)
- No hacer un recorrido recursivo
- Mantener una lista de regiones activas a ser visitadas (*Active Page List* o APL)
 - Una región está activa cuando aún no ha sido visitada pero su padre ya fue visitado
- La APL es una cola de prioridad que mantiene las regiones ordenadas por MINDIST de menor a mayor (un min-heap)
- La distancia al objeto candidato a NN (*pruningdist*) se usa para descartar regiones activas



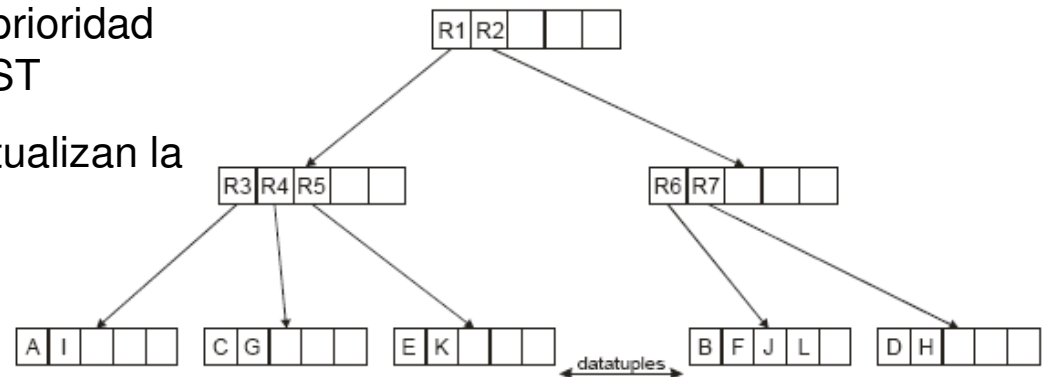
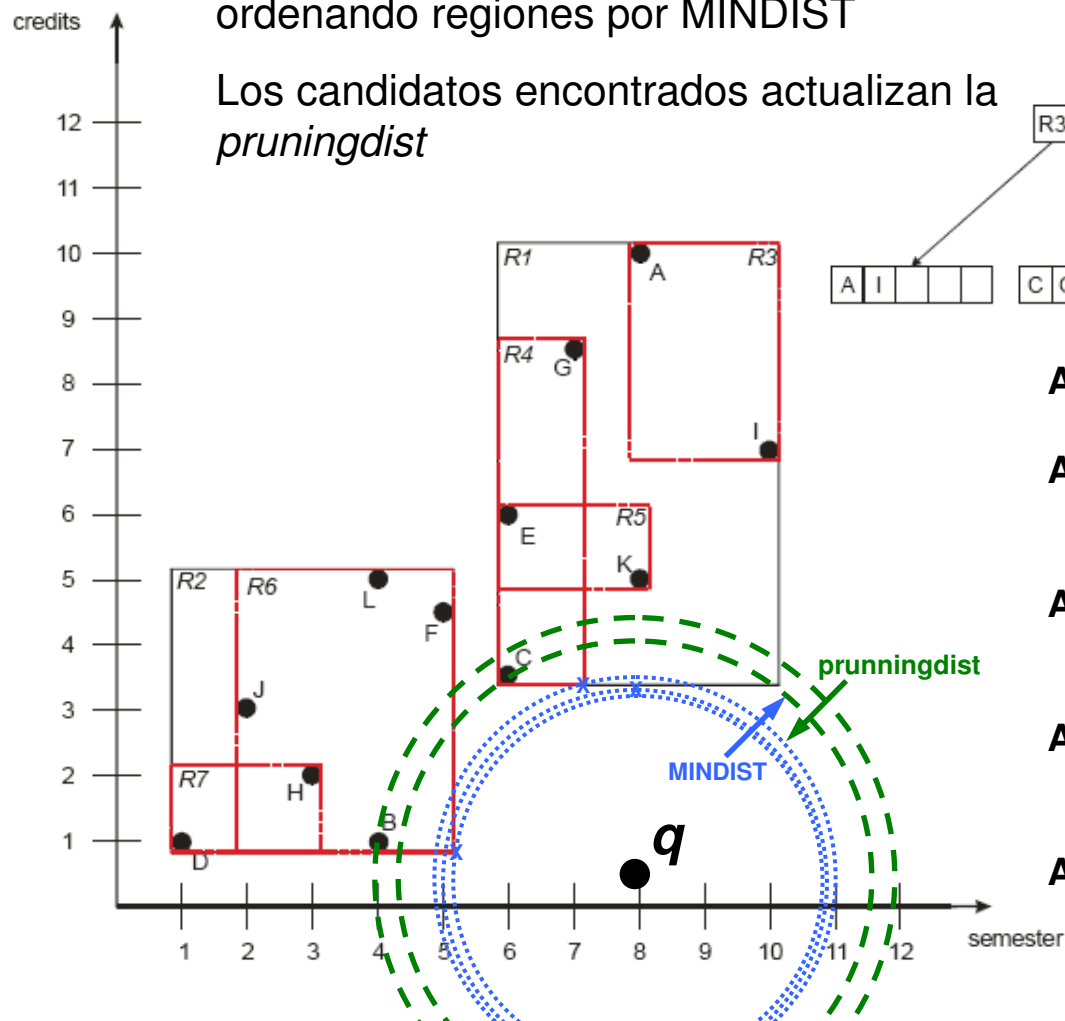
Algoritmo de Búsqueda NN por prioridad

- Iniciar la APL con la raíz del árbol
- Sacar de la APL la región con mejor prioridad (con menor MINDIST)
 - Si es hoja (con datos), se comparan todos sus elementos contra la consulta. Si corresponde, se actualiza el NN candidato y *pruningdist*
 - Si es nodo interno (con regiones), se obtienen sus hijos y los nodos con MINDIST menor a *pruningdist* se insertan en la APL
- El algoritmo termina cuando la APL está vacía o cuando la región sacada de la APL tiene un MINDIST mayor o igual a *pruningdist*

Búsqueda NN por prioridad en R-Tree

Búsqueda usando una cola de prioridad
ordenando regiones por MINDIST

Los candidatos encontrados actualizan la
pruningdist



APL: **R2**, R1

→ visitar R2: insertar R6, R7

APL: **R6**, R1, R7

→ visitar R6: comparar **B, F, J, L**

→ candidato: **B** (nuevo **pruningdist**)

APL: **R1**, R7

→ visitar R1: insertar R4, no insertar R3 ni R5 porque MINDIST > **pruningdist**

APL: **R4**, R7

→ visitar R4: comparar **C, G**

→ candidato: **C** (nuevo **pruningdist**)

APL: **R7**

→ R7 no es relevante



Búsqueda NN por prioridad

- Las páginas se acceden en orden creciente de MINDIST (círculos azules)
- *pruningdist* (círculos rojos) va decreciendo en la medida que se encuentran puntos más cercanos
- El algoritmo se detiene cuando ambos círculos se encuentran
- Requerimientos de espacio:
 - Puede suceder que todas las regiones del último nivel se inserten en la APL
 - La complejidad en espacio (peor caso) es $O(n)$, mucho peor que la búsqueda en profundidad $O(\log n)$



Optimalidad del algoritmo

- El algoritmo NN por prioridad es óptimo con respecto al número de regiones visitadas:
 - Visita la mismas cantidad de regiones que una consulta por rango de tamaño exacto para obtener el NN
 - Visita todas las regiones con MINDIST menor a la distancia del NN (asegurar que es el NN)
 - No visita regiones con MINDIST mayor que la distancia del NN
- Notar que la búsqueda no necesariamente termina cuando aparece como candidato el NN real, porque se debe asegurar que no existe un candidato mejor (visitar regiones con MINDIST menor)



Búsqueda k -NN en R-Tree

- Para resolver una búsqueda k -NN se requieren dos colas de prioridad:
 - APL (min-heap ordenando regiones por MINDIST)
 - Lista de k candidatos (max-heap ordenando elementos por su distancia a q)
- *pruningdist* corresponde a la distancia del peor candidato (k -ésimo)
 - La cabeza del max-heap de candidatos
 - infinito si hay menos de k candidatos



Búsqueda k -NN aproximada

- La búsqueda por prioridad visita páginas en orden de distancia creciente a la consulta
 - Idea: Realizar una **detención anticipada** (early stop) y retornar los candidatos hasta ese momento
- Parámetro: c número máximo de regiones con datos a visitar
- Realizar una búsqueda k -NN por prioridad pero finalizar la búsqueda cuando se han visitado c regiones con datos y retornar los candidatos hasta ese momento



Variantes del R-Tree

■ MINMAXDIST

- Distancia al punto más lejano de la cara más cercana
- Un MBR garantiza que existe al menos un elemento a esa distancia, se puede usar como *pruningdist*

■ R* Tree.

- Cambia criterio de creación de MBR: hojas minimizan intersección, nodos internos minimizan área total
- Evita splits resolviendo overflows con re-inserciones

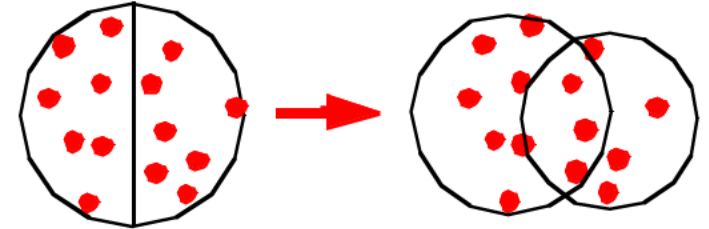
■ R+ Tree

- Elimina la necesidad de recorrer varios árboles en una búsqueda insertando un mismo elemento en más de una región.

■ X-Tree

- Si no existe un buen split, se crea un “super-nodo” donde se van agregando los objetos en una lista enlazada de nodos.
- Si hay mucho overlap X-Tree deriva en búsqueda secuencial

SS-Tree



■ SS-Tree (Similarity Search Tree):

- Usar hiper-esferas para definir las regiones espaciales (en vez de MBRs).
- Las esferas tienen ventaja con respecto a la probabilidad de acceso si las consultas también son regiones esféricas.
- Split más complicado
 - Cuando se divide una esfera en dos, no se obtiene como resultado dos esferas
 - Se utiliza la esfera de cobertura mínima para encerrar a los puntos de las regiones resultantes
 - Produce traslape alto
 - El centroide (centro de masa) de los puntos en la región se utiliza como centro de la esfera, y se busca el radio mínimo tal que todos los objetos de la región queden cubiertos

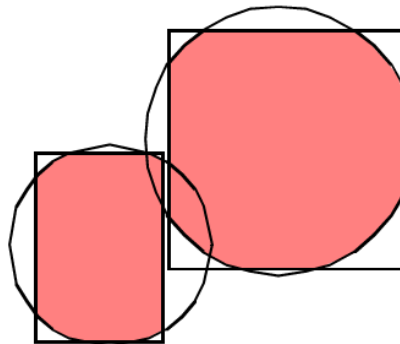


SS-Tree

- Algoritmo de inserción
 - El objeto se inserta en el nodo hijo cuyo centroide tenga la mínima distancia al nuevo punto
- Manejo de overflow
 - Se reinserta un 30% de los puntos (los más lejanos al centroide)
- Criterio de split
 - Para la posición de corte se prueban todas las posibilidades que garantizan la utilización mínima de espacio
 - El punto de corte minimiza la suma de las varianzas de ambos nodos resultantes

SR-Tree

- SR-Tree (Sphere-rectangle tree)
 - Utiliza la combinación (intersección) de un MBR y una esfera como región espacial
 - Busca combinar las ventajas de ambos métodos
 - Menor probabilidad de acceso de una esfera
 - Mejor “particionabilidad” de un MBR
 - Experimentalmente se muestra un mejor rendimiento que el SS-tree

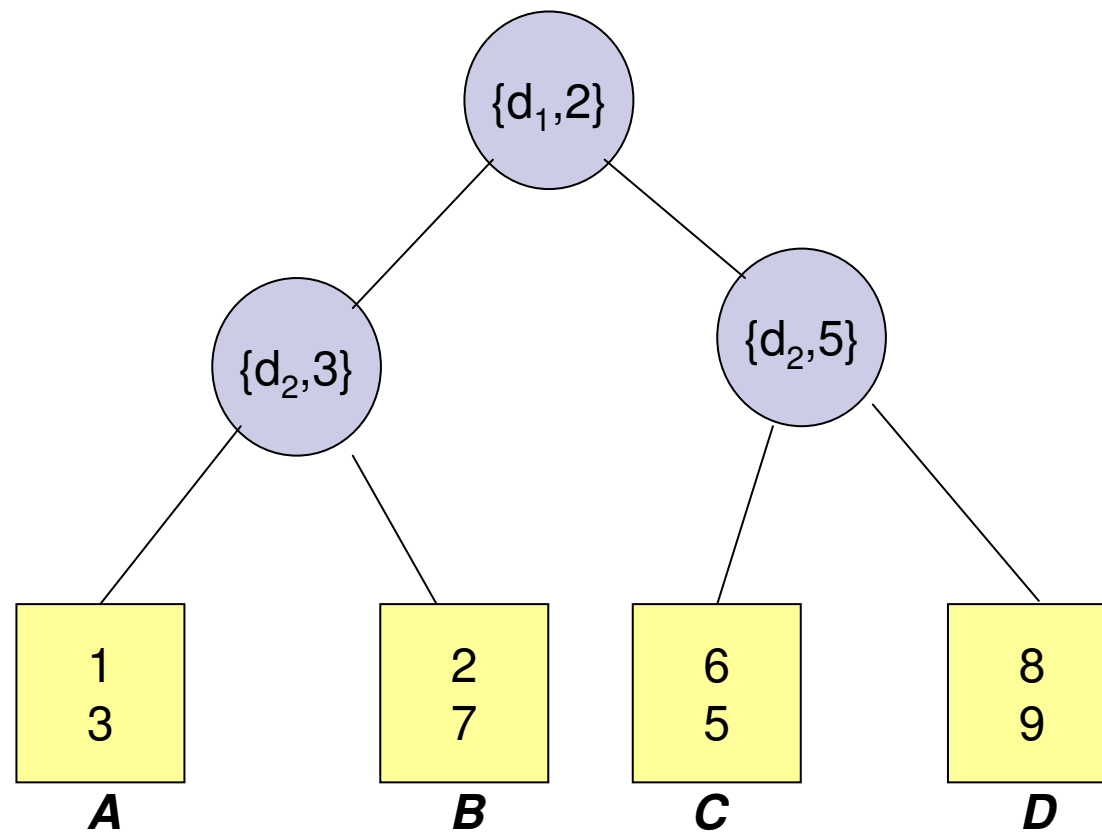




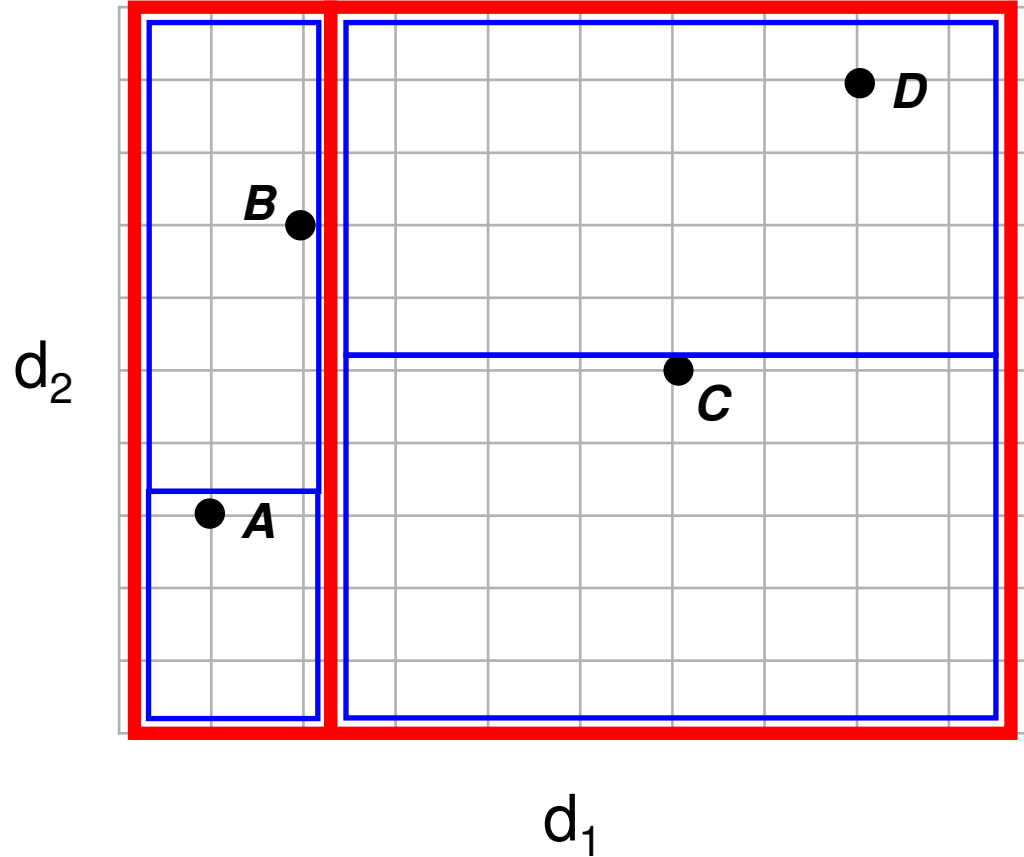
KD-Tree

- Árbol binario balanceado
- Particiona recursivamente el espacio por medio de hiper-planos paralelos a los ejes
- Nodo Interno: almacena un número de dimensión k y un valor umbral x
 - El subárbol izquierdo es la región con coordenada k menor o igual a x
 - El subárbol derecho es la región con coordenada k mayor o igual a x
- Nodo externo: almacena los vectores que se encuentran dentro de la región definida por los hiperplanos de sus ancestros

Ejemplo kd-Tree



Ejemplo kd-Tree

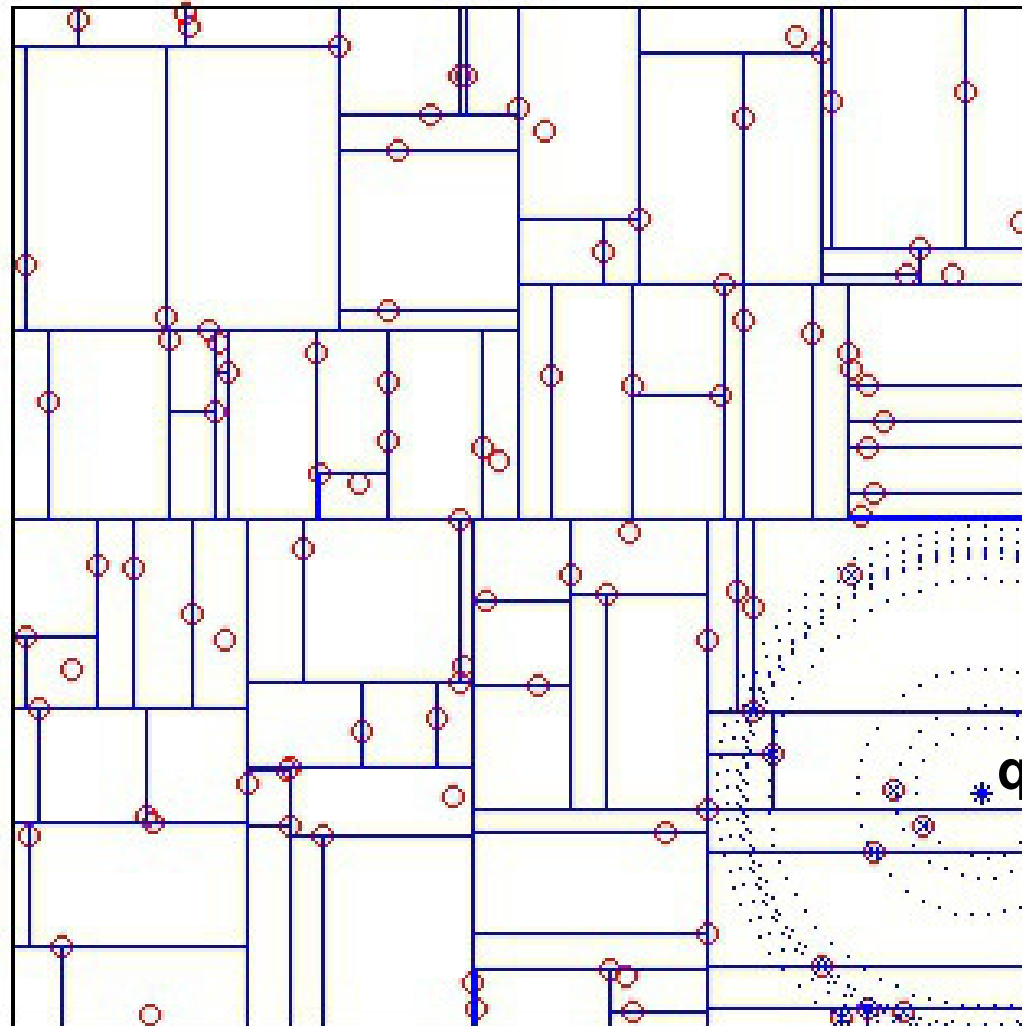




Construcción de un kd-Tree

- Sea **R** un conjunto de **n** vectores de **d** dimensiones, es decir, $\mathbf{R}=\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ donde \mathbf{v}_i es un vector con coordenadas (v_{i1}, \dots, v_{id})
- Calcular la varianza de cada dimensión, es decir, para cada dimensión **j** calcular la varianza σ_j^2 de los **n** valores $\{v_{1j}, \dots, v_{nj}\}$
- Comparar las varianzas $\{\sigma_1^2, \dots, \sigma_d^2\}$ y elegir la dimensión **k** que tiene mayor varianza
- Particionar **R** en subconjuntos **S** y **T** según la mediana de la dimensión **k**, es decir, sea λ_k la mediana de $\{v_{1k}, \dots, v_{nk}\}$, \mathbf{v}_i está en **S** si $v_{ik} \leq \lambda_k$
- Dividir recursivamente ambos subconjuntos **S** y **T**

Ejemplo kd-Tree





Búsqueda del NN en kd-Tree

- Opción Naive (búsqueda recursiva): bajar recursivamente por el árbol hasta la hoja que contiene a \mathbf{q} , comparar vectores en esa hoja y obtener un candidato, continuar el recorrido recursivo descartando regiones más lejanas que el candidato actual
- Mejor (búsqueda por prioridad): Análogo a la búsqueda en el R-Tree. Utilizar una cola de prioridad (APL) que ordena regiones por distancia a \mathbf{q} (MINDIST) y visitar regiones en orden ascendente



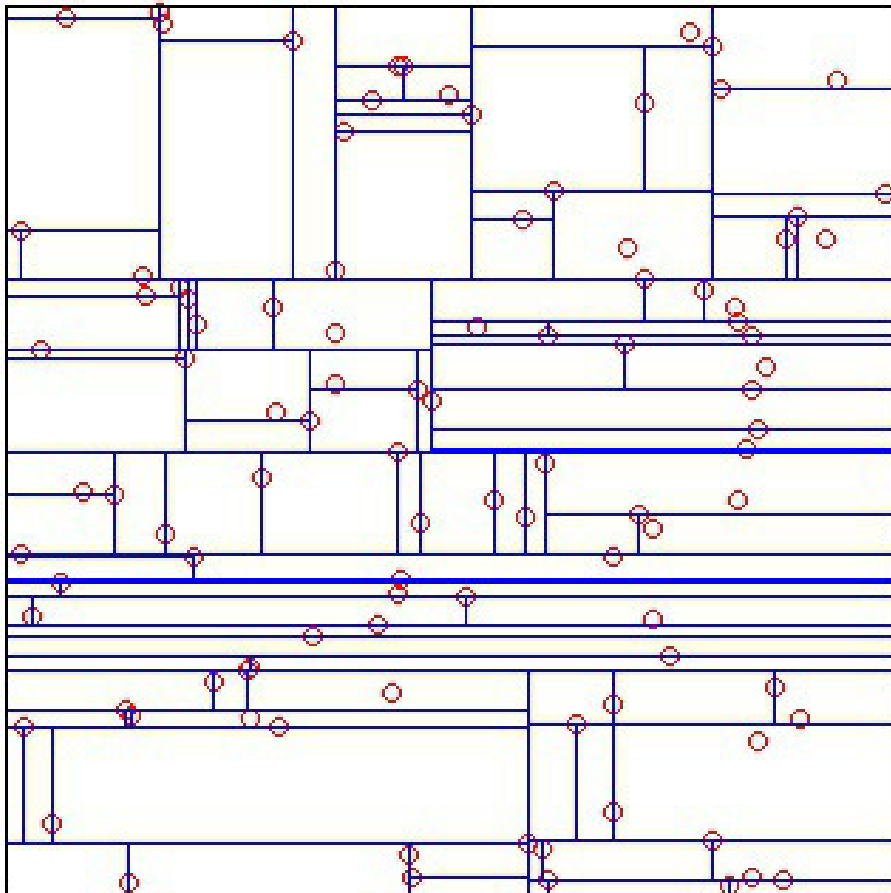
Randomized kd-Trees

- Para un conjunto de vectores el algoritmo de construcción del kd-tree produce un único árbol
- Se pueden crear varios kd-tree (un kd-forest) si en cada nivel se particiona por una dimensión al azar entre las D dimensiones de mayor varianza (por e.j. $D=5$)
- Algoritmo de Búsqueda del NN:
 - Búsqueda por prioridad utilizando **una única cola de prioridad** para las regiones de todos los árboles
 - La búsqueda va visitando todos los árboles al mismo tiempo
 - Limitar hojas visitadas → búsqueda aproximada

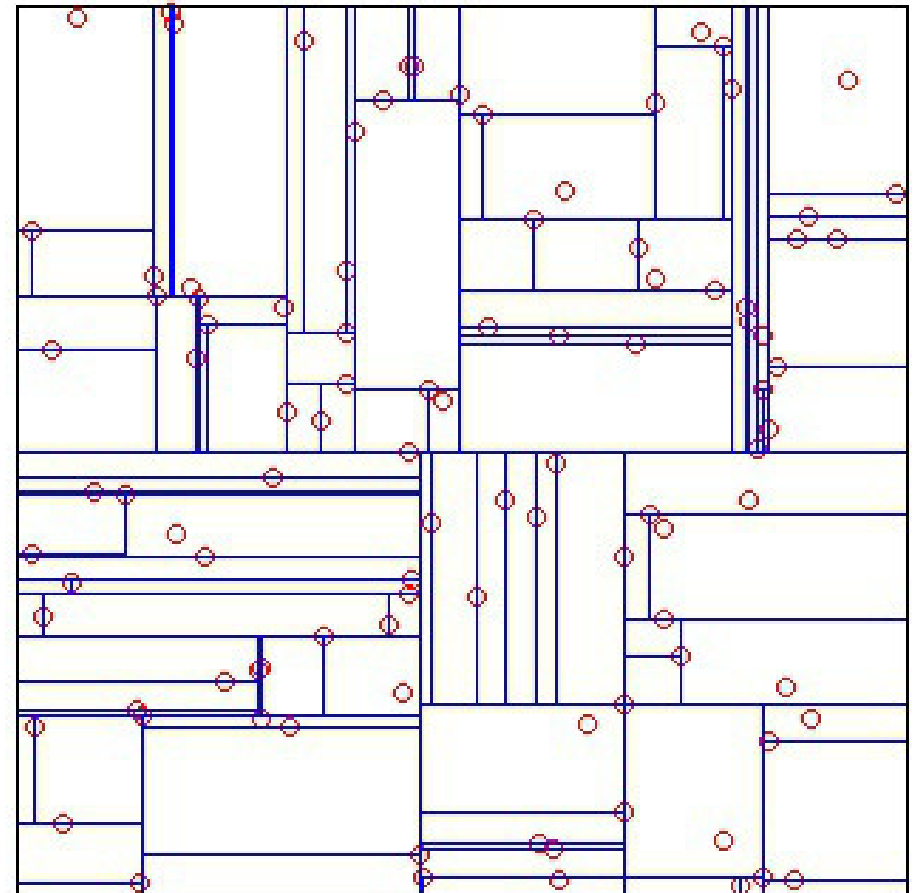


Ejemplo Randomized k-d Trees

Tree number 1

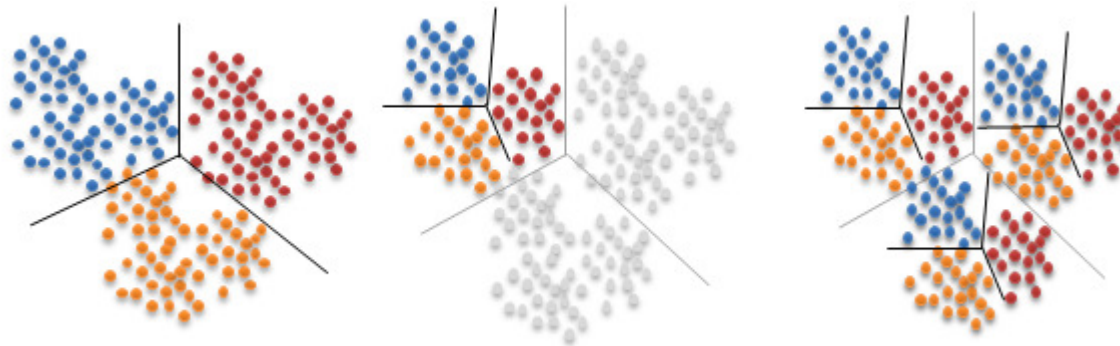


Tree number 2



Hierarchical K-Means Tree

- Utilizar k -means para dividir un conjunto en k grupos de objetos cercanos entre sí
 - División recursiva, terminar cuando quedan menos de k vectores en un grupo
 - Árbol balanceado k -ario (k es un parámetro).



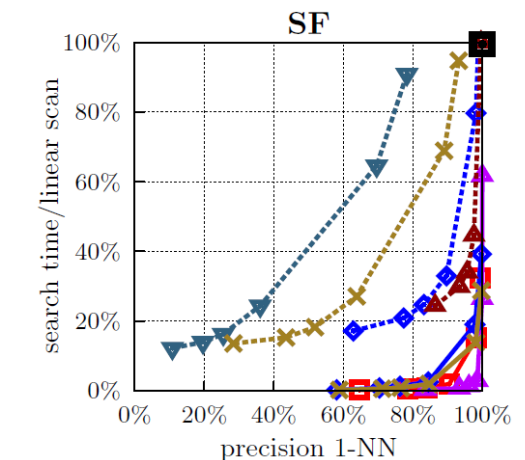
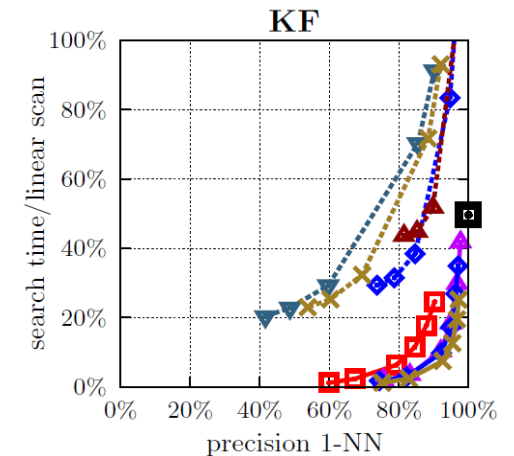


Hierarchical K-Means Tree

- Búsqueda aproximada con cola de prioridad
- Algoritmo de Búsqueda del NN:
 - Búsqueda por prioridad donde la regiones se ordenan por distancia de q al centroide
 - No hay gran cambio al ordenar por distancia de q al borde de celda de Voronoi
 - No hay gran cambio si se realizan múltiples k-means trees

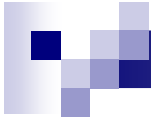
Efectividad versus Eficiencia

- La búsqueda lineal (linear scan) logra la máxima efectividad (retorna el 100% de los NN correctos) pero es lento
- Los métodos aproximados permiten mejorar la eficiencia (búsquedas más rápidas) con costo en la efectividad (bajar la calidad de la respuesta)
 - En algunos casos puede ser tolerable o incluso imperceptible retornar otro objeto cercano que no sea necesariamente el NN
 - Se debe medir experimentalmente cuanto decrece la calidad de la respuesta al acelerar la búsqueda
 - Esta relación depende el espacio de búsqueda (distribución de los descriptores)



Legend:

- kd-tree
- multi kd-tree ($n=10$)
- k-means tree ($k=10$)
- k-means tree ($k=20$)
- LaesaR ($|\mathcal{P}|=10$)
- LaesaR ($|\mathcal{P}|=20$)



Locality Sensitive Hashing



Locality Sensitive Hashing (LSH)

- Algoritmo aleatorio de búsqueda por similitud
- Búsqueda **no es exacta**
 - Su grado de aproximación se ajusta en la fase offline
- Se definen varias funciones de hash que producen colisiones para objetos similares
- Cada función de hash consiste en unir varias proyecciones aleatorias de los datos
- Una función de hash se dice locality-sensitive cuando la probabilidad de colisión de dos objetos disminuye cuando la distancia entre ellos aumenta



LSH para Espacios de Hamming

- Descriptores binarios y distancia de Hamming
- Si los descriptores son vectores se deben convertir a cadenas de bits
- Notación unaria, vector de coordenadas x_i :
 - Cada x_i debe ser un entero entre 0 y M
 - Cada x_i se mapea a un “1” repetido x_i veces seguido de un “0” repetido $(M-x_i)$ veces
 - Se concatenan las cadenas de todos los x_i
 - Ej.: vector (2,5) para $M=6$ se mapea a la cadena “110000111110”



Construcción del Índice

- Una función de hash h convierte el espacio s -dimensional a un subespacio m -dimensional
 - La función $h(o)$ escoge en forma aleatoria m posiciones entre 1 y s (s es el largo de la cadena de bits)
 - Si x e y son cercanos probablemente $h(x)=h(y)$
 - Si x e y son lejanos probablemente $h(x)\neq h(y)$
- La probabilidad total de cometer un error se reduce al utilizar L funciones de hash $\{h_1, \dots, h_L\}$
 - Para cada función h_i construir una tabla T_i asignando cada objeto o_j a la celda $T_i[h_i(o_j)]$
 - Dos objetos muy parecidos debieran coincidir en varias tablas



Búsqueda k -NN con LSH

- Para cada una de las L tablas:
 - Calcular la proyección correspondiente $h(q)$
 - Determinar los objetos que tienen colisión con q , i.e., localizar todos los o que $h(o)=h(q)$
 - Para cada objeto o sumar uno a su contador de ocurrencias
- Retornar los k objetos con más ocurrencias
- Opcional: realizar una búsqueda k' -NN ($k' \geq k$), calcular la distancia real de los k' candidatos y retornar los k -NN

LSH para Espacios de Hamming

Números entre 1 y 14 al azar



			HS = UNARY(X) & UNARY(Y)		PROJECTION INSTANCES			
			UNARY(X)	UNARY(Y)	{2,9,13}	{7,10,14}	{1,5,11}	{8,12,14}
Chicago	2	3	1100000	1110000	110	010	100	100
Mobile	4	0	1111000	0000000	100	000	100	000
Toronto	4	6	1111000	1111110	111	010	101	110
Buffalo	6	5	1111110	1111100	110	010	111	110
Denver	0	3	0000000	1110000	010	010	000	100
Omaha	2	2	1100000	1100000	110	000	100	100
Atlanta	6	1	1111110	1000000	100	000	110	100
Miami	7	0	1111111	0000000	100	100	110	000
$q = \text{Reno}$	0	4	0000000	1111000	010	010	001	100



Una cadena de
14 bits de largo

Parámetros $m=3$ $L=4$

Ver Samet, pág 713

LSH para Espacios de Hamming

Colisiones

Object	Chicago	Mobile	Toronto	Buffalo	Denver	Omaha	Atlanta	Miami	$q = \text{Reno}$
Chicago		1	1	11	11	111	1		11
Mobile	3.6					1	11	111	
Toronto	3.6	6.0		1					1
Buffalo	4.5	5.4	2.1						1
Denver	2.0	5.0	5.0	6.3					111
Omaha	1.0	2.8	4.5	5.0	2.2				1
Atlanta	4.5	2.2	5.4	4.0	6.3	4.1		1	1
Miami	5.8	3.0	6.7	5.1	7.6	5.4	1.4		
$q = \text{Reno}$	2.2	5.7	4.5	6.1	1.0	2.8	6.7	8.1	

Consulta

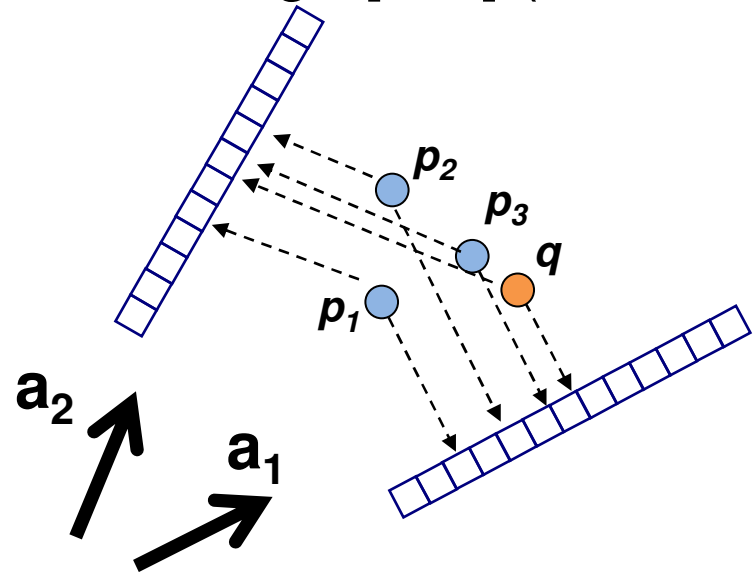
Distancias L_2

q tiene muchas colisiones con su NN

LSH para Espacios Euclidianos

- Descriptores vectoriales y distancia L2
- Proyección aleatoria:
 - Escoger vector aleatorio \mathbf{a} y valor aleatorio b
 - \mathbf{a} sigue distribución normal con media 0 y varianza 1
 - b sigue distribución uniforme en el rango $[0, w]$ (w : tamaño del bin)


$$h_{a,b}(v) = \left\lfloor \frac{\mathbf{a} \cdot \mathbf{v} + b}{w} \right\rfloor$$





Indexamiento LSH

- Una función $h_{a,b}(o)$ entrega un valor numérico
- Concatenar m funciones $h_{a,b}$ para definir un hash de vector
- Calcular L valores de hash por vector
- Para descriptores SIFT (128-d) se ha recomendado:
 - $w=700$; $m=24$; $L=32$



Búsqueda k-NN de LSH

- Para cada una de las L tablas:
 - Calcular el hash de q aplicando las m funciones $h_{a,b}$
 - Obtener los vectores v que coinciden en el hash $h(v)=h(q)$
- Reportar como NN los vectores que coinciden en uno o más de los L hash



Space-Filling Curves

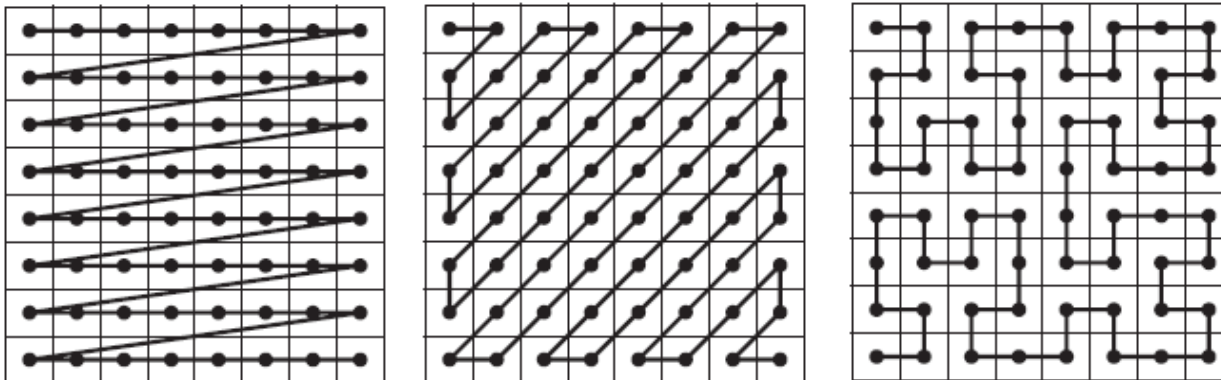


Space-Filling Curves

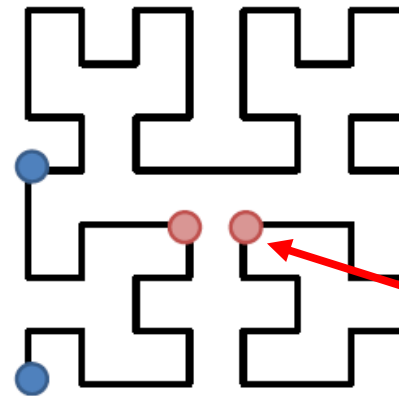
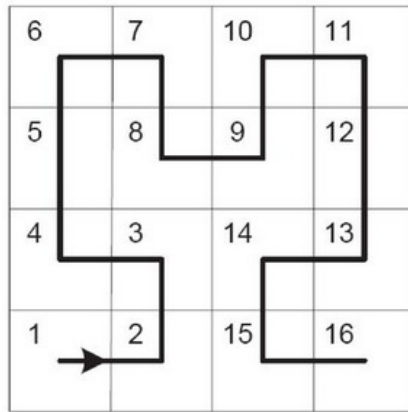
- Un espacio multidimensional se puede recorrer ordenadamente en forma lineal
- Un espacio de n -dimensiones se convierte en un espacio unidimensional
- Vectores cercanos en el espacio n -dimensional debieran quedar cerca en el espacio unidimensional

Space-Filling Curves

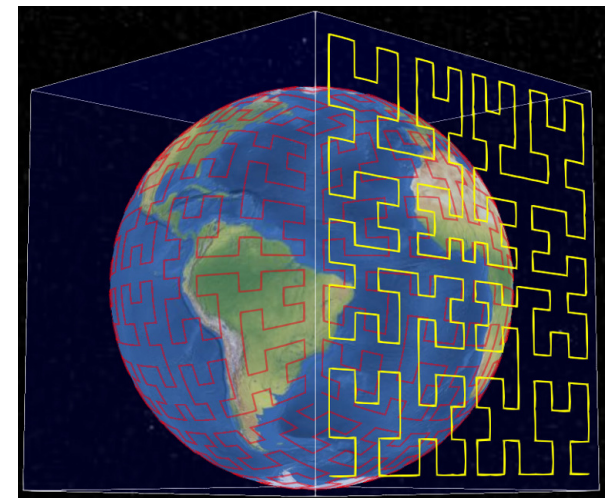
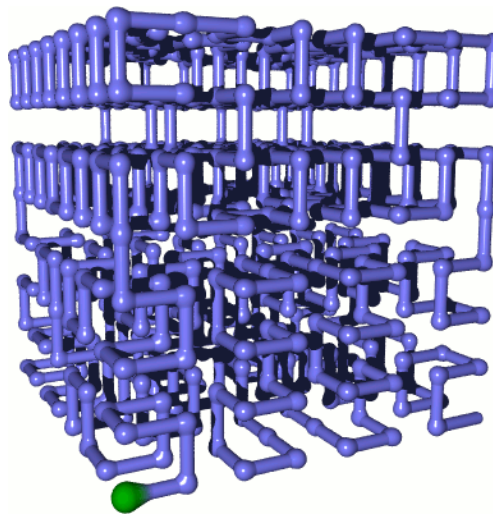
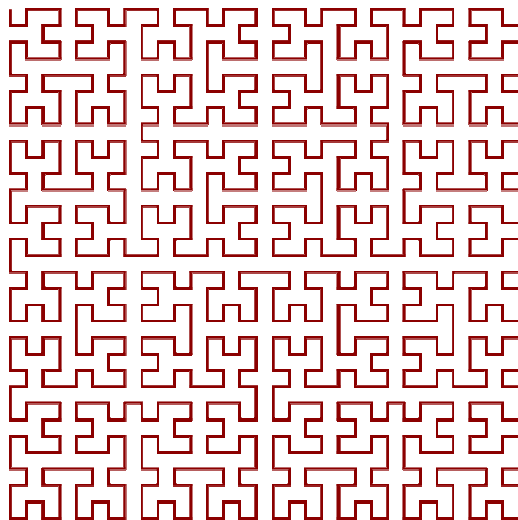
- El espacio se divide en celdas y se define un orden de recorrido de las celdas
- Propiedades deseables del recorrido:
 - Visitar cada celda una y solo una vez
 - Celdas cercanas deben quedar cerca en la curva
 - Celdas lejanas deben quedar lejos en la curva
 - La función debe ser rápida de calcular



Curva de Hilbert

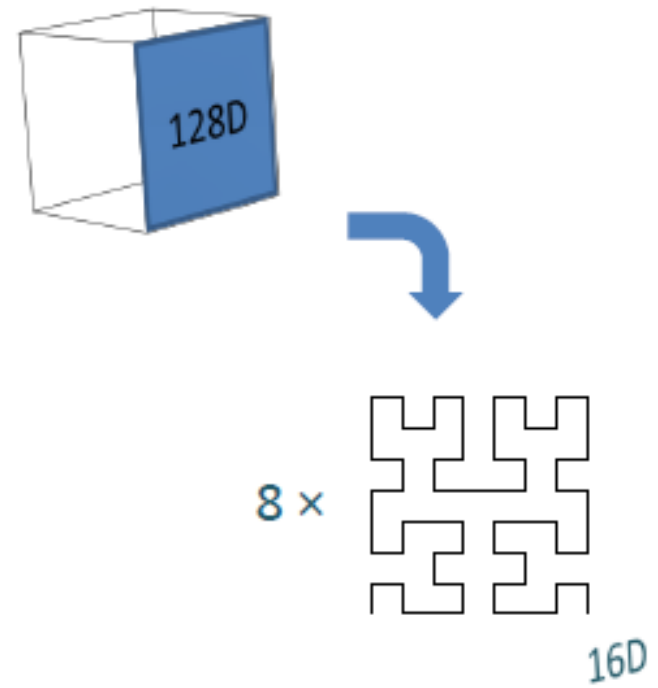
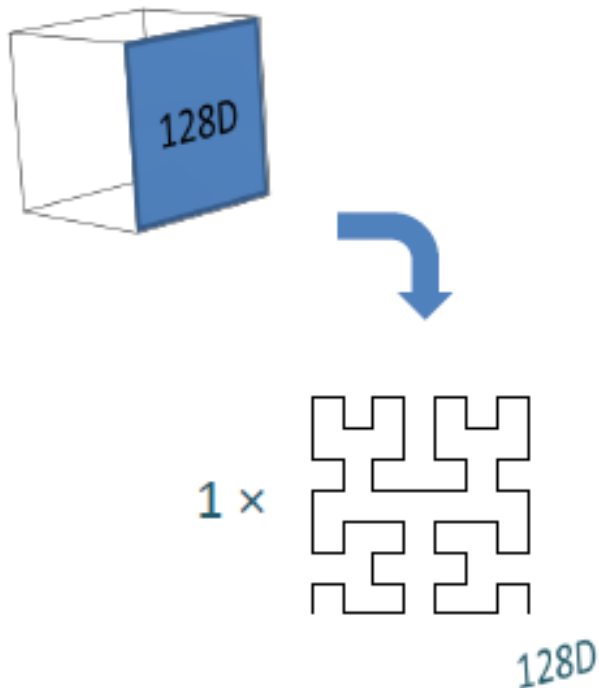


Puntos cercanos
en el espacio pero
lejanos en la curva



Multi-curves

- En vez de crear una única curva, crear varias curvas para subespacios





Curse of Dimensionality

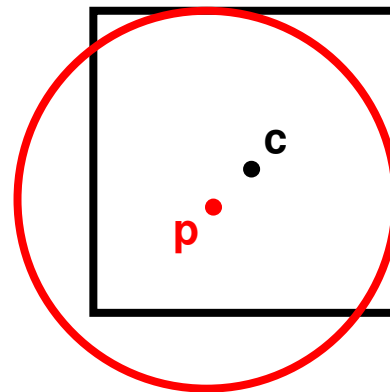


Alta Dimensionalidad

- “Curse of Dimensionality” o “Maldición de la Dimensionalidad”
- Se refiere a los efectos adversos que suceden cuando aumenta la dimensión de los datos
 - La distancia entre cualquier par de objetos tiende a un valor constante (desaparece la varianza en las distancias)
 - Todos los índices comienzan a fallar (todas las regiones intersectan la bola de consulta)
 - El volumen de las regiones aumenta exponencialmente con la dimensión del espacio
 - No hay imaginación geométrica, falla la intuición

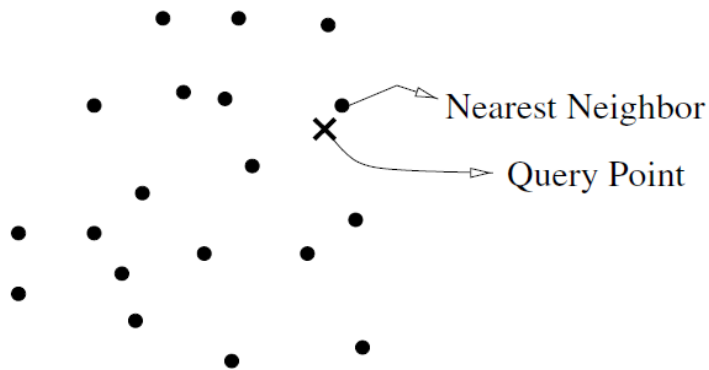
Alta Dimensionalidad

- “Si un círculo toca todos los bordes, siempre incluye el punto central?” No!
 - Hiper-cubo 16-dimensional unitario
 - $c = (0.5, 0.5, \dots, 0.5)$ (punto central)
 - $p = (0.3, 0.3, \dots, 0.3)$
 - Círculo centrado en p y radio 0.7 (toca todas las caras)
 - Distancia $L_2(p, c) = 0.8$!!

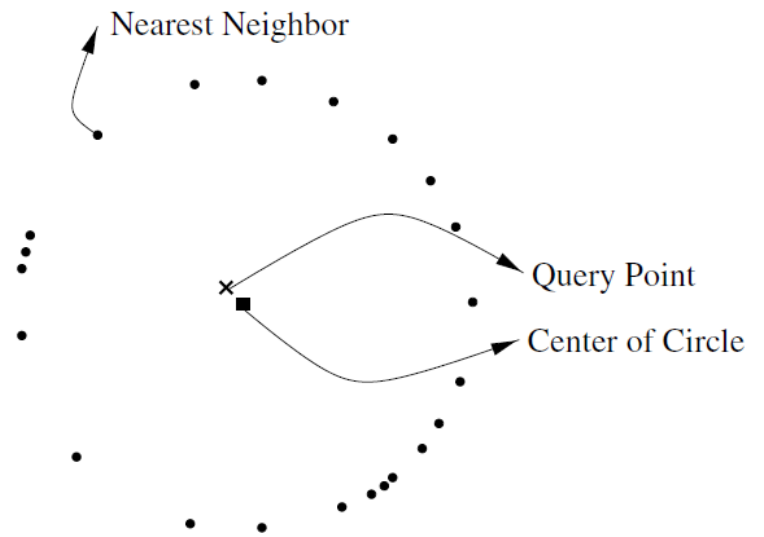


Alta Dimensionalidad

- Al aumentar las dimensiones, los objetos tienden a ubicarse a una misma distancia



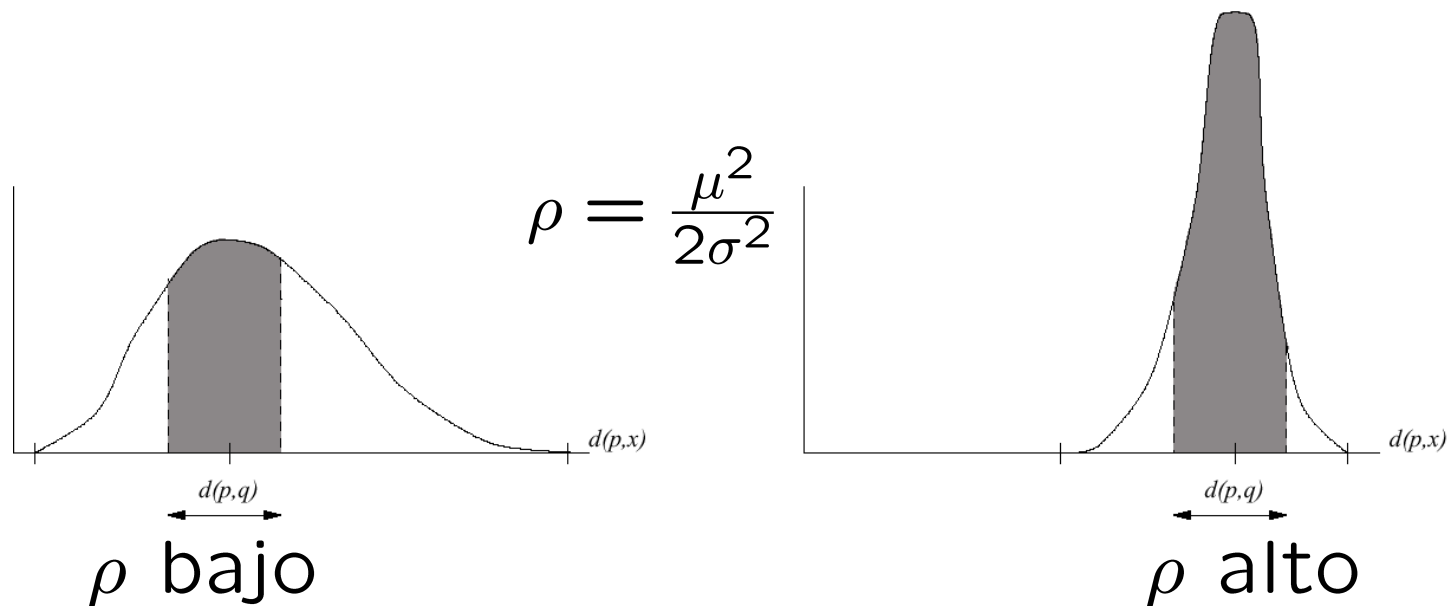
Dimensión Baja



Dimensión Alta

Histograma de distancias

- Histograma de $d(a,b)$ con a y b aleatorios del conjunto R
- Al aumentar dimensiones las distancias tienden a concentrarse a un valor fijo
- La dificultad de indexar un conjunto tiene relación con poca varianza en valores de distancias





Volumen Bola de Consulta

- Volumen crece exponencialmente:

- Hiper-cubo

$$Vol_{cube}(edge, D) = edge^D$$

$$Diag_{cube}(edge, D) = edge \cdot \sqrt{D}$$

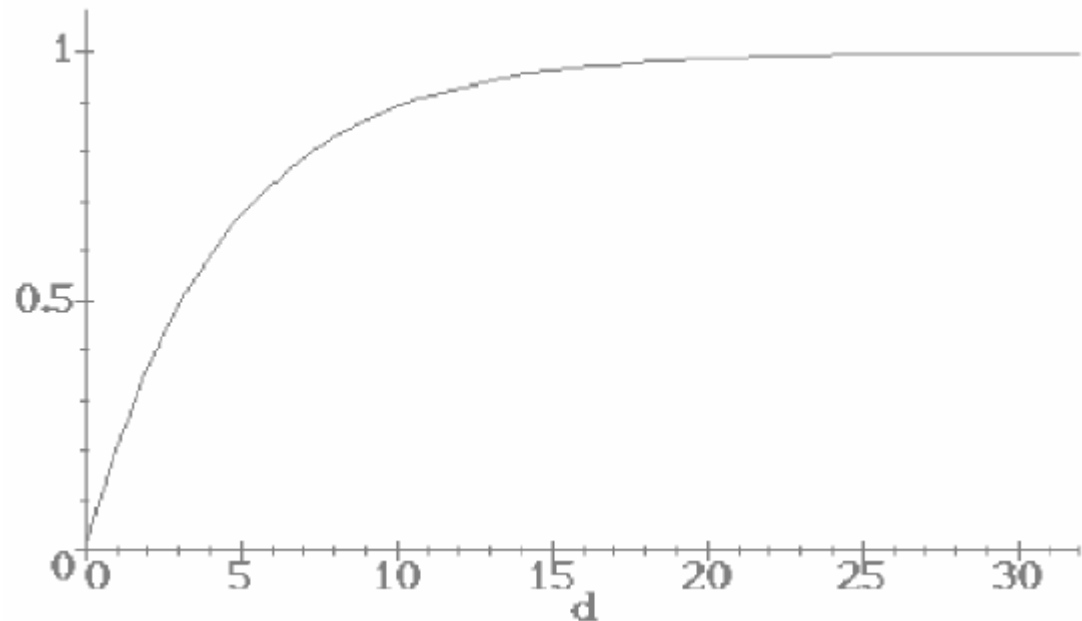
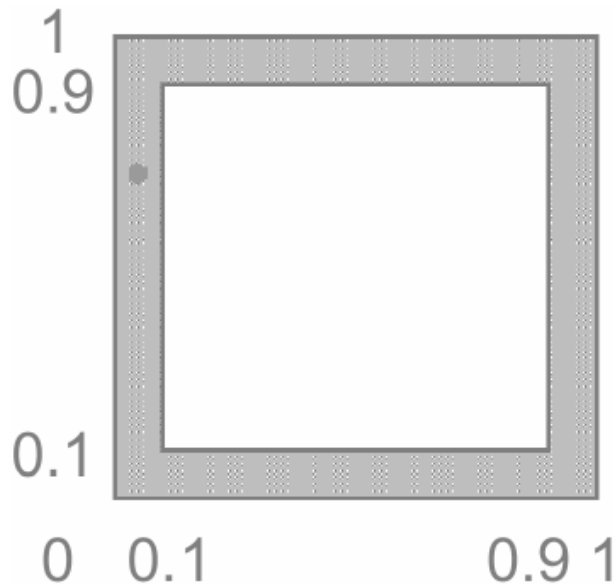
- Hiper-esfera

$$Vol_{sphere}(radius, D) = radius^D \cdot \frac{\sqrt{\pi^D}}{\Gamma\left(\frac{D}{2} + 1\right)}$$

- Cuando la dimensión aumenta, el volumen de la hiper-esfera de radio 1 comparado con el hipercubo de lado 1 tiende a 0.

Volumen en las superficies

- Todo el volumen está en la superficie!
 - Probabilidad que un punto aleatorio esté cerca del borde en un MBR





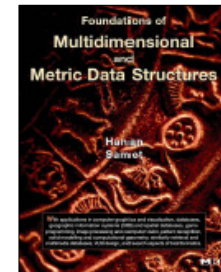
Alta Dimensionalidad

- No se puede evaluar el desempeño de un índice con datos aleatorios
 - Seguro fallará porque es imposible indexar datos aleatorios de dimensión alta
 - No es realista
- Los datos reales no son aleatorios!
 - Cada dataset tiene consultas y datos relevantes para esas consultas (existen grupos de objetos cercanos)
- Al capturar datos se debe evitar incluir datos irrelevantes ya que sólo se conseguirá igualar los registros

Bibliografía

- **Foundations of Multidimensional and Metric Data Structures. Samet 2006.**

- Cap 1.5 (Kd-Tree)
- Cap 2.1.5.2 (R-Tree)
- Cap 4.7.4 (LSH)





Papers

- Guttman. **R-trees: A dynamic index structure for spatial searching.** 1984.
- Hjalton and Samet. **Ranking in Spatial Databases.** 1995.
- Roussopoulos, Kelley and Vincent. **Nearest Neighbor Queries.** 1995.
- Böhm, Berchtold and Keim. **Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases.** 2001.
- Muja and Lowe. **Scalable Nearest Neighbor Algorithms for High Dimensional Data.** 2014.
- Beyer, Goldstein, Ramakrishnan and Shaft. **When is “nearest neighbor” Meaningful?** 1998.
- Hinnenburg, Aggarwal and Keim. **What is the nearest neighbor in high dimensional spaces?.** 2000.
- Shaft and Ramakrishnan. **Theory of Nearest Neighbors Indexability.** 2006.



Implementaciones

- Implementación de Kd-Tree y K-Means Tree:

<http://www.cs.ubc.ca/research/flann/>

- Implementación de LSH:

<http://www.mit.edu/~andoni/LSH/>

<https://github.com/pixelogik/NearPy>

- Implementación para curva de Hilbert:

https://en.wikipedia.org/wiki/Hilbert_curve