Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 1

R-Trees

Integrantes: Bastián Bas
Valeria Franciscangeli
Carlo Merino

Profesor: Benjamín Bustos

Auxiliar: Diego Salas

Fecha de entrega: 19 de octubre de 2023

Índice de Contenidos

1 Introducción **1**

2 Desarrollo **2**
2.1 Métodos de Construcción . . . 2
2.1.1 Método1: Nearest-X: . . . 2
2.1.2 Método 2: Hilbert R-tree . . . 2
2.1.3 Método 3: Sort-Tile-Recursive . . . 2
2.2 Metodología del Experimento . . . 3
2.3 Implementación en C . . . 3

3 Resultados **4**
3.1 Resultados de Nearest-X . . . 4
3.2 Resultados de Hilbert . . . 5
3.3 Resultados de Sort-Tile-Recursive . . . 6
3.4 Comparación de los métodos . . . 7
3.4.1 Intervalos de confianza . . . 9
3.4.1.1 Hilbert . . . 9
3.4.1.2 Nearest-X . . . 11
3.4.1.3 Sort-Tile-Recursive . . . 12

4 Análisis **13**

5 Conclusión **14**

Anexo A Estructuras de datos **15**

Anexo B Código de la Tarea **16**
B.1 Código Método X-Nearest . . . 16
B.2 Código Método Hilbert . . . 16
B.3 Código Método STR . . . 16

1 Referencias **17**

1. Introducción

Los R-tree son, como indica un extracto de A.Guttman (1984), rescatado del paper "Algoritmos y Estrucutras de Datos para Búsqueda de Objetos Similares"(Baeza-Yates, Cuquejo, y Navarro), son diseñados como un "Método de Acceso Espacial (SAMs)", a modo de extensión de los B+-trees en dimensiones mayores a uno, usando la técnica de solapamiento de regiones, y que si bien es balanceado como los B-trees, pueden soportar espacios muertos o áreas densas.

El presente informe tiene por objetivo exponer los principales resultados y conclusiones de una actividad experimental de análisis de algoritmos. El principal objetivo del la actividad experimental consiste en comparar la eficiencia de un algoritmo de búsqueda aplicado a 3 R-trees construidos con diferentes métodos, que se detallarán posteriormente, con especial énfasis en en llevar a cabo un análisis exhaustivo del comportamiento de los árboles tipo R-tree en el contexto de memoria secundaria. Para la realización del análisis es requerido concebir la implementación de tres métodos de construcción de R-trees diferentes, junto con un algoritmo de búsqueda específico para estos árboles. Posteriormente, se realizarán experimentos controlados con el objetivo de obtener datos cuantitativos y cualitativos acerca del tiempo de búsqueda promedio y la cantidad de accesos a bloques de disco de la búsqueda esto nos permitirá sacar conclusiones sólidas y fundamentadas.

A continuación se explicará la implementación de los 3 métodos de construcción a implementar para generar los R-trees sobre los cuáles se busca analizar el rendimiento de búsqueda, los cuáles son:

- **Nearest-X:** Considera sólo la abscisa del centro de cada rectángulo, generando en base a la abscisa, todos los rectángulos sean necesarios para encerrar todos los rectángulos iniciales en "bloques" que contengan M rectángulos, repitiendo el algoritmo sobre el último conjunto de rectángulos generados hasta que sólo se pueda generar un rectángulo, que será el nodo raíz.
- **Hilbert R-tree:** Se ordenan los rectángulos en grupos de tamaño M en función de su valor dentro de la curva de Hilbert, repitiendo el procedimiento sobre los nuevos rectángulos generados
- **Sort-Tile-Recursive:** Se parte ordenando los n puntos de referencia según su coordenada X, dividiendo el resultado en $S = \sqrt{\frac{n}{M}}$ grupos, para que luego cada grupo sea ordenado según su coordenada Y y así se puedan crear S grupos nuevos. De esta manera conseguimos dividir el conjunto de puntos en aproximadamente $\frac{n}{M}$ grupos nuevos. Luego de esto, se itera por cada grupo para construir el árbol.

Como hipótesis, planteamos en primer lugar, que el algoritmo de búsqueda tardará mucho más en el árbol construido con el método Nearest-X que en los demás. En segundo lugar,a pesar de que los árboles construidos por los distintos métodos para un mismo n ocuparán la misma cantidad de memoria, el algoritmo de búsqueda realizará distinto número de operaciones I/O. Por último, que el algoritmo con mejor desempeño en la búsqueda será STR.

2. Desarrollo

2.1. Métodos de Construcción

Se detallan los métodos de construcción a emplear para la actividad experimental. En general, para representar los árboles construidos, utilizamos la siguiente metodología, cambiando solamente el método de ordenamiento:

1. Se recibe un array de nodos hojas con los rectángulos iniciales, un valor `n` con la cantidad rectángulos recibidos y un valor `M` con la cantidad máxima de hijos por nodo.
2. Determinar la cantidad de nodos a construir ($\lceil n/M \rceil$) y verificar que sea mayor a 1.
3. Ordenar los nodos de acuerdo al algoritmo particular.
4. Incorporar los sub-árboles en estructuras `Node` y definir sus respectivos **MBR**.
5. Al definir los **MBR** se vuelve al punto 1, tomando como rectángulos iniciales los **MBR** creados.
6. Cuando solo haya que construir 1 sub-árbol, simplemente se agrupan todos los nodos y se juntan como hijos de una misma estructura `RTree`.

2.1.1. Método1: Nearest-X:

Este método de construcción consiste en agrupar los R rectángulos más cercanos con respecto a sus vecinos más cercanos según la abscisa de forma recursiva, reduciendo la cantidad de R rectángulos a R/M por iteración, hasta llegar a 1. Para ello:

Si se consideran R rectángulos compuestos por 4 enteros, el sobre costo en memoria de almacenar los R rectángulos con el presente método de construcción, considerando la altura del árbol $h = \log_M R$ y la cantidad de nodos en un nivel h $r = \frac{R+R/4}{M^h}$ es de

$$\frac{R}{M(\log_M R)} \tag{1}$$

datos a añadir por cada nueva iteración.

2.1.2. Método 2: Hilbert R-tree

Este método utiliza la curva de Hilbert para ordenar R rectángulos con respecto a la posición de sus centros en dicha curva con el objetivo de agruparlos en grupos de M rectángulos por nodo por cuadrantes en el fractal. Se repite el proceso sobre los $\frac{R}{M}$ MBRs que encierran a los $\frac{R}{M}$ grupos antes calculados hasta que sea posible agrupar a todos los rectángulos en un solo nodo raíz. El procedimiento es similar al primero, sólo que cambia la forma de ordenar los rectángulos en el primer paso. La dificultad radica en obtener los valores de la posicion de un punto en la curva. Al trabajar en C, se pudo calcular en base al código disponible en Wikipedia.

2.1.3. Método 3: Sort-Tile-Recursive

Este método es muy similar al primero, ya que ordena R rectángulos con respecto a su coordenada X, agrupándolos en $S = \sqrt{n/M}$ grupos, cada uno con $M * S$ nodos, para luego ordenar los grupos con respecto a su coordenada Y, volviendo a separar los grupos en S grupos, repitiendo el procedimiento hasta lograr agrupar todos los datos en un sólo nodo.

2.2. Metodología del Experimento

La metodología a utilizar para la realización de los experimentos consistirá, para cada $n = 2^x$ tal que $x \in [10, 25]$, en:

1. Generar un set de R de n rectángulos con valores enteros uniformemente distribuidos en el rango de $[0, 500000]$. Para ello, se generaron los puntos de cada rectángulo de a 4, para así garantizar que los puntos sigan el orden "inferior izquierdo, superior derecho" y ahorrar un poco de tiempo en la construcción de los árboles, cuidando que los lados tengan un tamaño uniformemente distribuido en [0,100].

2. Generar un set Q de 100 consultas con rectángulos cuyos vértices se encuentran aleatoriamente distribuidos en el rango anterior, cuyos lados tengan un tamaño uniformemente distribuido entre [0, 100000].

3. Utilizar el mismo set de datos R para construir 3 R-Trees utilizando los métodos de construcción *Nearest-X*, *Hilbert R-Trees* y *Sort-Tile-Recursive*.

4. Evaluar la eficiencia de los métodos utilizando el mismo set de Q consultas para los 3 árboles.

5. Registrar y analizar los resultados obtenidos para corroborar o refutar nuestras hipótesis iniciales.

Finalmente, nos queda:

1. Comparar los resultados obtenido en cada estructura y graficar la cantidad I/O y el tiempo de búsqueda para las estructuras.

2. Ajustar el valor de la curva de tiempo de búsqueda promedio según la forma a $O(c \cdot n^{\alpha})$, encontrando el valor de los parámetros c, α y una contante utilizando la función *curve_fit* de *scipy* de Python. Esta función ajusta la curva mediante el método de mínimos cuadrados.

3. Graficamos *Tiempo de Búsqueda Promedio y curva ajustada vs x* y *Cantidad de I/Os promedio vs x*.

2.3. Implementación en C

La explicación de la implementación de estos métodos utilizando el lenguaje de programación C está disponible en la sección Anexo.

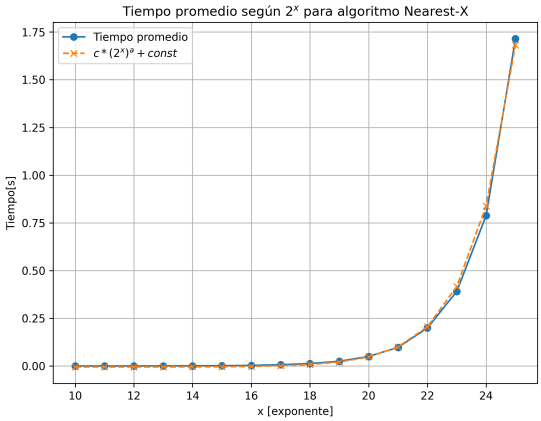
3. Resultados

Se realizaron los experimentos antes mencionados en una máquina con sistema Linux Mint Linux Mint 21.2 Cinnamon, con un procesador Intel© Core™ i5, un disco duro SSD de 256 GB y 16 GB de RAM (2 tarjetas de 8 GB), utilizando el lenguaje C y compilador gcc 11.4. El tamaño de página de disco fue de 4096 bytes, lo que permite guardar $M = 146$ rectángulos por nodo, según la implementación de la escritura de los nodos.

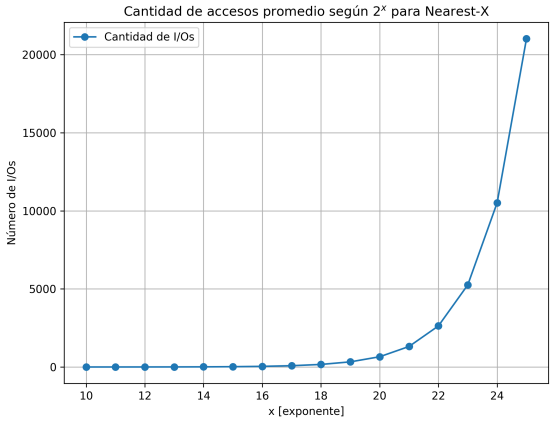
Se observan los siguientes resultados para cada algoritmo.

3.1. Resultados de Nearest-X

Para el R-Tree construido según el algoritmo Nearest-X, se obtuvieron resultados en cuanto a **tiempo de búsqueda promedio** y **cantidad de accesos a bloques de disco** al realizar búsqueda por intersección.



(a) Promedio de tiempo



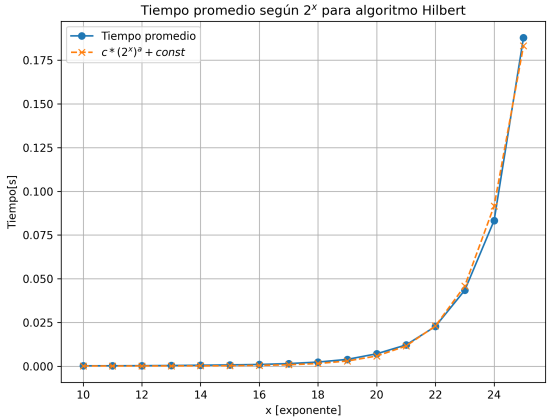
(b) Promedio de accesos

En la figura 1.a se puede comparar los resultados promedio obtenidos y la curva ajustada. Los parámetros encontrados para el ajuste (truncados a 3 decimales) fueron:

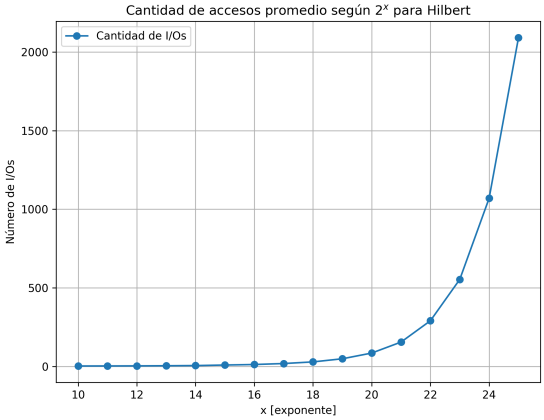
- α: $9.999 \cdot 10^{-1}$
- c: $5.028 \cdot 10^{-8}$
- constante: $-5.017 \cdot 10^{-3}$

3.2. Resultados de Hilbert

Para el R-Tree construido según el algoritmo de Hilbert, se obtuvieron resultados en cuanto a **tiempo de búsqueda promedio** y **cantidad de accesos a bloques de disco** al realizar búsqueda por intersección.



(a) Promedio de tiempo

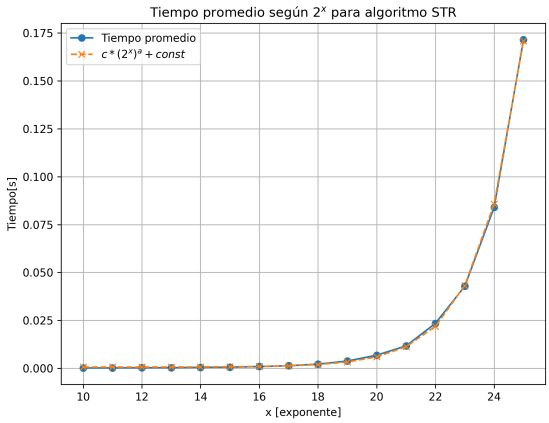


(b) Promedio de accesos

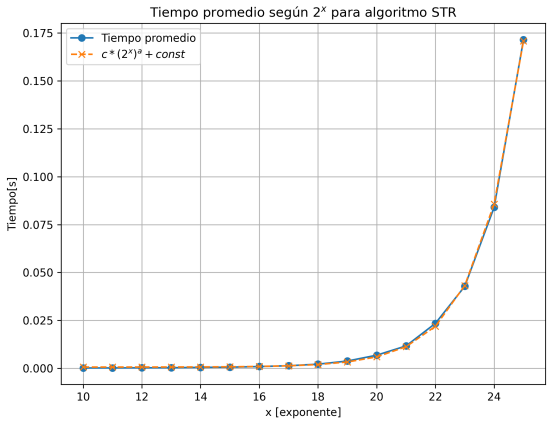
En la figura 2.a se puede comparar los resultados promedio obtenidos y la curva ajustada. Los parámetros encontrados para el ajuste (truncados a 3 decimales) fueron:

- α: $9.999 \cdot 10^{-1}$
- c: $5.454 \cdot 10^{-9}$
- constante: $-1.151 \cdot 10^{-4}$

3.3. Resultados de Sort-Tile-Recursive



(a) Promedio de tiempo



(b) Promedio de accesos

En la figura 3.a se puede comparar los resultados promedio obtenidos y la curva ajustada. Los parámetros encontrados para el ajuste (truncados a 3 decimales) fueron:

- α: $9.999 \cdot 10^{-1}$
- c: $5.070 \cdot 10^{-9}$
- constante: $6.419 \cdot 10^{-4}$

3.4. Comparación de los métodos

En la figura 4 se muestran los valores promedio del tiempo de búsqueda en el R-Tree construido con cada uno de los 3 métodos para cada exponente x de $n = 2^x$.

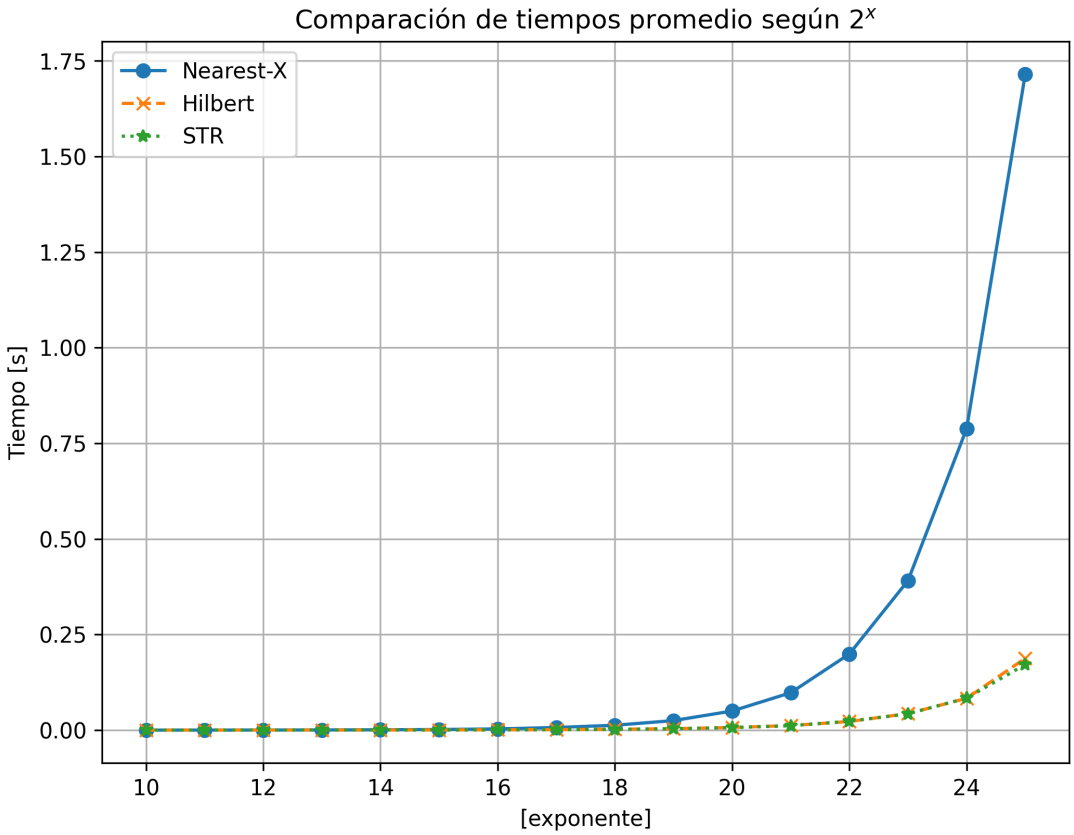


Figura 4: Comparación de tiempos de búsqueda promedio entre los 3 algoritmos.

En la figura 5 se muestran los valores promedio la cantidad de accesos a disco en la búsqueda en el R-Tree construido con cada uno de los 3 métodos para cada exponente x de $n = 2^x$.

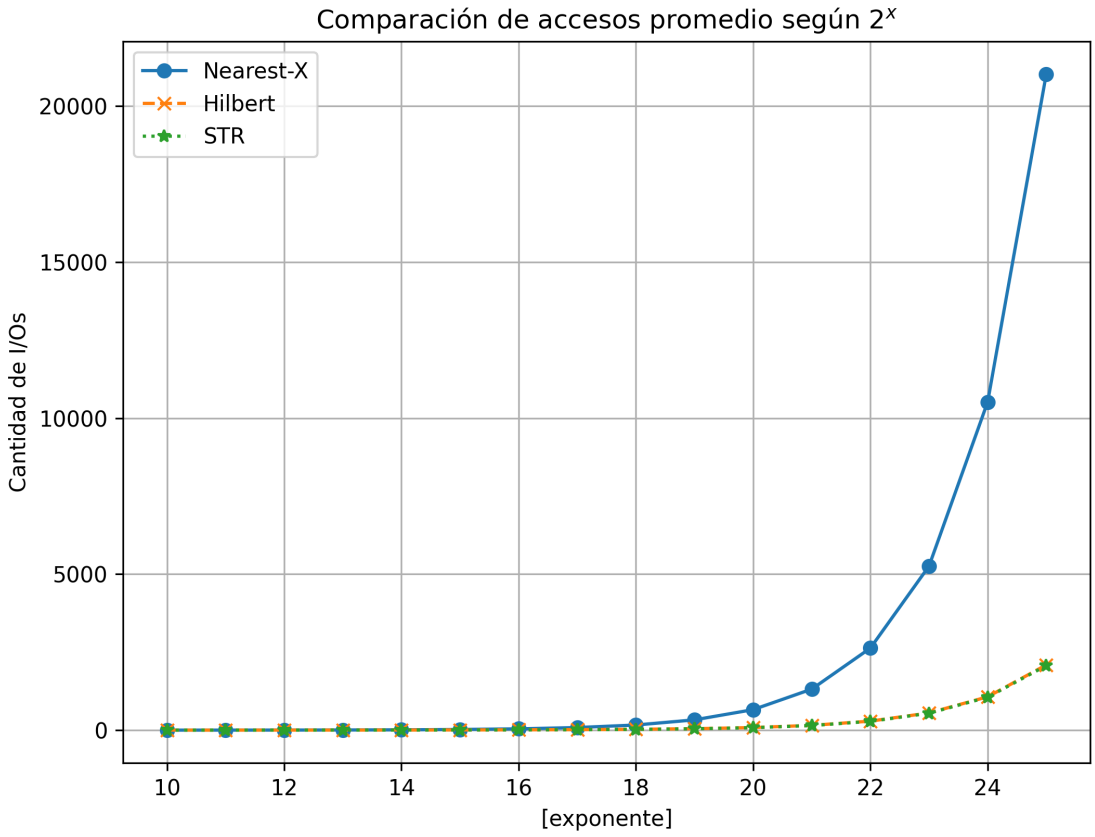


Figura 5: Comparación de cantidad de accesos promedio entre los 3 algoritmos.

3.4.1. Intervalos de confianza

A continuación se muestran los intervalos de confianza obtenidos para cada n al obtener los resultados del tiempo de búsqueda y cantidad de accesos a disco en los 100 experimentos realizados, con una confianza del 67 % (1 σ).

3.4.1.1. Hilbert

Tabla 1: Intervalos de Confianza Tiempo Promedio para Hilbert R-Trees.

	n	Límite Inferior	Promedio	Límite Superior
	2^{10}	0,0002364190339	0,000237	0,0002373917112
	2^{11}	0,0002559578347	0,000256	0,0002569061789
	2^{12}	0,0003090685804	0,000312	0,0003125312447
	2^{13}	0,0003965948175	0,000398	0,0004008452749
	2^{14}	0,0005638080508	0,000566	0,0005685935063
	2^{15}	0,0007512751274	0,000755	0,0007574609975
	2^{16}	0,0009886482202	0,000990	0,0009909915888
*	2^{17}	0,00148750445	0,001494	0,001511088902
	2^{18}	0,002353504067	0,002367	0,002394634645
	2^{19}	0,003860767111	0,003912	0,003935790438
	2^{20}	0,007085815887	0,007107	0,007231559995
	2^{21}	0,0120983376	0,012159	0,01216972203
	2^{22}	0,02255109853	0,022749	0,02308098048
	2^{23}	0,04303942102	0,043429	0,04356366411
	2^{24}	0,0825303732	0,083320	0,08381632627
	2^{25}	0,1857973447	0,187826	0,1892504216

Tabla 2: Intervalos de Confianza Accesos Promedio para Hilbert R-Trees.

	n	Límite Inferior	Promedio	Límite Superior
	2^{10}	2,8228977618128104	2,83	2,8371022381871898
	2^{11}	3,1218090807221235	3,13	3,1381909192778763
	2^{12}	3,5705446706986000	3,58	3,5894553293014000
	2^{13}	4,4461264829012470	4,46	4,4738735170987530
	2^{14}	5,5592136932995880	5,58	5,6007863067004120
	2^{15}	8,9566212190286320	8,99	9,0233787809713690
	2^{16}	12,2532814197991900	12,31	12,3667185802008100
*	2^{17}	18,3772677249361360	18,48	18,5827322750638650
	2^{18}	28,6806394137129600	28,87	29,0593605862870420
	2^{19}	48,4575195902454400	48,81	49,1624804097545650
	2^{20}	84,5231647349296200	85,19	85,8568352650703700
	2^{21}	153,8371681338862000	155,13	156,4228318661138000
	2^{22}	288,2237417097730000	290,74	293,2562582902270400
	2^{23}	548,8594757834953000	553,79	558,7205242165046000
	2^{24}	1060,2892315024906000	1070,02	1079,7507684975094000
	2^{25}	2072,2710049274306000	2091,55	2110,8289950725700000

3.4.1.2. Nearest-X

Tabla 3: Intervalos de Confianza Tiempo Promedio para Nearest-X.

	n	Límite Inferior	Promedio	Límite Superior
	2^{10}	0,0002033149102	0,000204	0,0002059929631
	2^{11}	0,0002630681002	0,000265	0,000267636991
	2^{12}	0,0003957782924	0,000396	0,000398647854
	2^{13}	0,0005985256192	0,000600	0,0006016321673
	2^{14}	0,001083961996	0,001085	0,001087893676
	2^{15}	0,001819377297	0,001821	0,001825754292
	2^{16}	0,00338522657	0,003386	0,00339304778
*	2^{17}	0,007056376475	0,007071	0,007084540073
	2^{18}	0,01294495916	0,012975	0,01300479516
	2^{19}	0,02505058663	0,025122	0,02516268416
	2^{20}	0,050137527	0,050187	0,05022932263
	2^{21}	0,09779556578	0,097988	0,09817144335
	2^{22}	0,1993141228	0,199367	0,1996465964
	2^{23}	0,3909922966	0,391033	0,3917836073
	2^{24}	0,7875060141	0,788091	0,789191064
	2^{25}	1,713188081	1,714886	1,716750836

Tabla 4: Intervalos de Confianza Accesos Promedio para Nearest-X.

	n	Límite Inferior	Promedio	Límite Superior
	2^{10}	2,5245300393883423	2,53	2,5354699606116573
	2^{11}	3,201277954640593	3,21	3,218722045359407
	2^{12}	4,60378684268951	4,62	4,6362131573104906
	2^{13}	7,067922439792848	7,10	7,132077560207152
	2^{14}	12,03682556537447	12,10	12,16317443462553
	2^{15}	23,282551968042174	23,41	23,537448031957826
	2^{16}	43,79776858982718	44,05	44,302231410172816
*	2^{17}	84,59591149358907	85,10	85,60408850641092
	2^{18}	165,81035660145432	166,82	167,82964339854567
	2^{19}	329,5764931688064	331,60	333,62350683119365
	2^{20}	655,2348976358816	659,27	663,3051023641184
	2^{21}	1308,518568652751	1316,60	1324,681431347249
	2^{22}	2615,1174254319435	2631,28	2647,442574568057
	2^{23}	5225,184859263046	5257,50	5289,815140736954
	2^{24}	10448,586705187123	10513,24	10577,893294812877
	2^{25}	20890,27652898836	21019,57	21148,86347101164

3.4.1.3. Sort-Tile-Recursive

Tabla 5: Intervalos de Confianza Tiempo Promedio para Sort-Tile-Recursive.

	n	Límite Inferior	Promedio	Límite Superior
	2^{10}	0,0001660018655	0,000167	0,0001678627715
	2^{11}	0,0002049836415	0,000206	0,0002064194603
	2^{12}	0,000269304637	0,000270	0,0002722890601
	2^{13}	0,0003137033731	0,000316	0,0003166657968
	2^{14}	0,0004847335508	0,000489	0,0004890574678
	2^{15}	0,0005862025198	0,000588	0,0005916513891
	2^{16}	0,0009199149883	0,000925	0,0009330595923
*	2^{17}	0,001338829527	0,001348	0,001348408964
	2^{18}	0,002145353963	0,002172	0,002184604146
	2^{19}	0,003783328098	0,003795	0,003851477568
	2^{20}	0,006760852861	0,006854	0,006934661676
	2^{21}	0,01161218703	0,011678	0,01168360048
	2^{22}	0,02336735064	0,023458	0,02347324029
	2^{23}	0,04210673646	0,042787	0,04281167757
	2^{24}	0,08332270531	0,084001	0,08489768071
	2^{25}	0,1705060131	0,171455	0,1733486892

Tabla 6: Intervalos de Confianza Accesos Promedio para Sort-Tile-Recursive.

	n	Límite Inferior	Promedio	Límite Superior
	2^{10}	2,304147047792993	2,31	2,315852952207007
	2^{11}	2,553575441403606	2,56	2,566424558596394
	2^{12}	3,0901122976713133	3,10	3,109887702328687
	2^{13}	3,627132815138771	3,64	3,652867184861229
	2^{14}	4,788638837730511	4,81	4,831361162269488
	2^{15}	7,5475420578021915	7,58	7,612457942197809
	2^{16}	10,696429125565913	10,75	10,803570874434087
*	2^{17}	15,876886253378368	15,97	16,06311374662163
	2^{18}	26,785839029348335	26,97	27,154160970651663
	2^{19}	45,53730398129974	45,88	46,22269601870026
	2^{20}	79,97349217952274	80,62	81,26650782047727
	2^{21}	148,2023684332689	149,47	150,7376315667311
	2^{22}	281,2108074783918	283,69	286,1691925216082
	2^{23}	538,563956609207	543,42	548,2760433907929
	2^{24}	1047,1356666724748	1056,83	1066,524333327525
	2^{25}	2050,677449835714	2069,84	2089,0025501642863

4. Análisis

Analizaremos este experimento en cuanto a los resultados obtenidos y las hipótesis planteadas.

Para cada método, podemos notar que el comportamiento en cuanto a tiempo promedio de búsqueda y cantidad promedio de accesos a disco crece linealmente respecto a la cantidad de rectángulos hoja en el árbol n. Los intervalos de confianza obtenidos muestran que la incerteza en la estimación de estos promedios también tiene un comportamiento lineal con respecto a n. Esto significaría que, a pesar de la aleatoriedad de los experimentos, los intervalos de confianza se mantienen consistentes, dando cuenta que los promedios obtenidos son representativos de los datos.

En las figuras de tiempo de búsqueda para cada método se puede notar que se logró ajustar de forma muy cercana a los valores y la curva obtenida. En los 3 métodos el valor α encontrado fue casi 1, por lo que la función de la curva quedaría de la forma $O(c \cdot n)$, lo que respalda los resultados lineales de los promedios.

Con respecto a los resultados de tiempo promedio de consulta, los mayores tiempos fueron registrados en los R-trees construidos por el método *Nearest-X*, tardando hasta alrededor de 8 veces más que en las estructuras generadas por los otros métodos. Esto tiene relación con el hecho de que *Nearest-X* genera nodos con *MBR*s mucho menos compactos (mayor relación *perimetro*/*area*) que los demás métodos.

Los tiempos promedios de consulta fueron muy similares en las estructuras generadas mediante *Hilbert* y *Sort-Tile-Recursive*. Comparando estos métodos con *Nearest-X*, se puede apreciar que la forma en que éstos métodos construyen los árboles es similar en cuanto a que ambos construyen la estructura de datos disponiendo los datos en 2 dimensiones (x, y) en lugar de sólo considerar x.

En concreto, se puede observar que los 3 métodos de construcción obtienen tiempos de búsqueda muy similares hasta un $n = 2^{17}$. Luego de este valor, *Nearest-X* se vuelve menos eficiente que sus contrapartes, que mantienen desempeños similares. Lo mismo ocurre con la cantidad de accesos a disco promedio.

5. Conclusión

En conclusión, el análisis de los experimentos muestra que disposición de los datos es muy importante al momento de realizar consultas en memoria secundaria. De primera mano se logró corroborar que para estructuras que mantienen la misma información, la forma de organizar dichos datos en memoria secundaria es lo que puede marcar la diferencia entre esperar 10 minutos o más de una hora por consulta.

Además, se muestra que, independientemente del algoritmo de construcción utilizado, los tiempos de búsqueda y accesos a memoria secundaria crecen de forma lineal en relación a la cantidad de datos guardados en el R-Tree.

A pesar de que para muchos datos la diferencia en tiempos de consulta es bastante grande, podría resultar preferible utilizar algoritmos más ineficientes, pero de menor complejidad de implementación, si se sabe de antemano que la cantidad de datos estará por debajo de cierto valor, ya que para cantidades no tan grandes de datos los tiempos son similares y el impacto en los tiempos de consulta podría ser despreciable.

Anexo A. Estructuras de datos

Para el modelamiento del problema, se diseñaron 4 estructuras de datos: "Point", "Rect", "Node" y "RTree":

- `typedef struct Point`: Representa un punto en el plano cartesiano. Cuenta con un atributo `x (float)` e `y (float)` para guardar sus coordenadas.
- `typedef struct Rect`: Representa un rectángulo. Cuenta con los atributos `p1 (Point)` y `p2 (Point)` para representar al rectángulo a través de sus vértices inferior/izquierdo y superior/-derecho respectivamente. Si el rectángulo tiene altura 0, `p1` será aquel con menor valor en `x`, y de manera análoga, si el rectángulo tiene ancho 0, `p1` será aquel con menor valor en `y`.
- `typedef struct Node`: Representa un rectángulo en contexto del RTree. Se utiliza para emparejar a un rectángulo (MBR), con el `RTree` que contiene los rectángulos contenidos por el MBR. Cuenta con los atributos `MBR (Rect)` y `child (RTree *)`.
- `typedef struct RTree`: Representa un nodo del RTree. Cuenta con los atributos `nodes (Node **)`, que es el listado de hijos del nodo, además de un atributo `n` para guardar la cantidad de hijos.

Junto con la creación de estas estructuras, se implementaron varias funciones para facilitar la construcción de los árboles:

- `Point getMaxPoint(Point p1, Point p2)`: Dados dos puntos, retorna el punto que agrupe la mayor de las coordenadas de ambos puntos. Ej: `getMaxPoint((2, 1), (3, 0)) -> (3, 1)`
- `Point getMinPoint(Point p1, Point p2)`: Dados dos puntos, retorna el punto la menor de las coordenadas de ambos puntos. Ej: `getMaxPoint((2, 1), (3, 0)) -> (2, 0)`
- `void initRect(Rect *r, float x1, float y1, float x2, float y2)`: Dado un puntero a un rectángulo y 4 coordenadas, inicializa el rectángulo generando y asignando sus respectivos `p1` y `p2`.
- `Point getRectCenter(Rect r)`: Dado un rectángulo, retorna el punto correspondiente a su centro.
- `int areIntersecting(Rect r1, Rect r2)`: Dados dos rectángulos, retorna 1 si los rectángulos intersectan, o 0 si no.
- `void initNode(Node *node)`: Dado un puntero a un nodo, inicializa un nodo hoja sin MBR.
- `int isLeaf(Node node)`: Dado un nodo, retorna 1 si el nodo es hoja (`child` apunta a `NULL`) o 0 si no.
- `void setNodeMBR(Node *node)`: Dado un puntero a un nodo, calcula el rectángulo correspondiente al MBR del árbol en `child` y lo define en `MBR`.
- `void initRTree(RTree *rt, int size)`: Dado un puntero a un árbol y una cantidad máxima de hijos, inicializa un árbol.
- `void addNode(RTree *rt, Node *node)`: Dado un puntero a un árbol y un puntero a un `Node`, inserta el `Node` como hijo del árbol.

- `Rect calcMBR(RTree rt)`: Dado un árbol, retorna el `Rect` que representa su MBR.
- `int treeHeight(RTree *prt)`: Dado un puntero a un árbol, retorna su altura.
- `int isValidTree(RTree *prt)`: Dado un puntero a un árbol, retorna 1 si es válido, y 0 si no. Para efectos de este estudio, solo se revisó recursivamente que cada hijo de un árbol tuviese la misma altura.
- `void freeRTree(RTree *rt)`: Dado un puntero a un árbol, se encarga de liberar la memoria reservada para sus componentes.

Anexo B. Código de la Tarea

Para probar la tarea:

- Descomprimir el fichero.
- Abrir una terminal dentro del fichero comprimido o navegar al fichero.
- escribir 'sudo bash' para permitir la compilación y ejecución de la tarea.
- Para compilar, ejecutar "gcc -Wall -g -o main main.c nearest_x.c hilbert.c STR.c RTree.c -lm"
- Para ejecutar las pruebas: ".\main". Los resultados deberían empezar a mostrarse en pantalla.

B.1. Código Método X-Nearest

Ver archivo "nearest_x.c" en el fichero comprimido adjunto al informe.

B.2. Código Método Hilbert

Ver archivo "hilbert_x.c" en el fichero comprimido adjunto al informe.

B.3. Código Método STR

Ver archivo "STR.c" en el archivo comprimido adjunto al informe.

.

1. Referencias

1. V. Cuquejo, R. Baeza-Yates and G. Navarro “Algoritmos y Estructuras de Datos para Búsqueda de Objetos Similares”

2. Hilbert Curve, https://en.wikipedia.org/wiki/Hilbert_curve".