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

Tarea 1 - R-Trees

Diseño y Análisis de Algoritmos

Integrantes: Tomás Alvarado
Pablo Skewes
Sebastian Brzovic
Profesor: Gonzalo Navarro
Auxiliar: Asunción Gómez
Ayudantes: Sergio Rojas H.
Valeria Burgos Sanhueza

Fecha de entrega: 19 de Octubre de 2023
Santiago, Chile

Resumen

En esta tarea se implementa un R-Tree, una estructura de datos que permite manipular rectángulos en memoria secundaria y que soporta tanto la inserción como la eliminación de datos. El objetivo principal de la tarea es el estudio de tres formas de construcción de un R-Tree a partir de un conjunto de datos. Los métodos de construcción son los siguientes: Nearest-X, Hilbert R-Tree y Sort-Tile-Recursive.

Nearest-X es un método que agrupa los rectángulos según la posición horizontal de su centro. Primero, se ordena los rectángulos segun su coordenada X y los divide en grupos de tamaño M, que es el número máximo de hijos por nodo.

Este método se basa en el mismo principio que el anterior, pero utiliza la curva de Hilbert para ordenar los centros de los rectángulos según su valor. La curva de Hilbert es una curva fractal que recorre todo el plano y asigna un valor a cada punto según la distancia desde el origen. El algoritmo construye el árbol con nodos de M hijos de forma recursiva, usando el valor de la curva de Hilbert para ordenar los puntos de referencia.

El método consiste en ordenar los puntos de referencia por su coordenada X, y luego agruparlos en S bloques de M elementos cada uno, donde S es la raíz cuadrada de n y M es el número de hojas del árbol. Después, se ordenan los bloques por su coordenada Y y se vuelven a agrupar en S bloques de M elementos. Así, se crean $S^2 = \frac{n}{M}$ nodos que se unen recursivamente hasta formar el nodo raíz.

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Introducción

En esta tarea se aborda el estudio de R-trees, una estructura similar a los B-trees que permiten manipular rectángulos en memoria secundaria y soportar operaciones de inserción, eliminación y búsqueda eficientes. El objetivo principal es implementar y comparar distintos métodos de construcción de R-trees, evaluando su desempeño en términos de costos de búsqueda.

El R-tree se caracteriza por representar conceptualmente rectángulos en sus nodos internos, almacenar los rectángulos de datos en las hojas del árbol y contener un número máximo de hijos por nodo. Además, todos los nodos hojas se encuentran a la misma profundidad en el árbol.

Para esto, se implementan tres métodos de construcción distintos: Nearest-X, Hilbert R-tree y Sort-Tile-Recursive (STR). Posteriormente, se evalua el costo del método de búsqueda en cada uno de estos árboles. Se compara el rendimiento de cada método a través de una serie de experimentos utilizando un conjunto común de datos y consultas.

1.1. Objetivos

Para cumplir con los objetivos de la tarea, se realizarán las siguientes tareas:

1. Implementación de tres métodos de construcción de R-trees: Nearest-X, Hilbert R-tree y Sort-Tile-Recursive (STR).
2. Implementación del método de búsqueda dentro del R-tree.
3. Evaluación comparativa del costo del método de búsqueda utilizando los distintos métodos de construcción.

1.2. Métodos de *Bulk Load* para un RTree

1.2.1. Nearest-X

El método Nearest-X ordena los rectángulos según la coordenada X de su centro y los agrupa formando nodos con un número máximo de hijos. Una vez ordenados, se agrupan recursivamente los rectángulos consecutivos en grupos de tamaño M hasta que los rectangulos restantes quepan en un sólo nodo raiz.

1.2.2. Hilbert R-tree

Similar al método Nearest-X, el Hilbert R-tree ordena los rectángulos según su posición en la curva de Hilbert. La lógica de formación del árbol se mantiene, pero el ordenamiento de los nodos se basa en el valor asignado por la curva de Hilbert.

1.2.3. Sort-Tile-Recursive (STR)

El método Sort-Tile-Recursive comienza ordenando los puntos de referencia según su coordenada X, luego se divide en grupos de tamaño adecuado y se repite el proceso de forma recursiva hasta obtener un único nodo raíz.

1.3. Búsqueda

La operación de búsqueda recorre el R-tree para encontrar todos los rectángulos en los nodos hojas que se intersectan con el rectángulo consultado. Luego, recursivamente busca en los hijos de aquellos nodos cuyo *Minimum Bounding Rectangle* intersectó con el rectángulo consultado. El método deberá devolver una lista de rectángulos y calcular la cantidad de accesos a bloques de disco durante la ejecución.

1.4. Experimentación

Se realizarán experimentos comparativos con valores aleatorios y se evaluará el tiempo de búsqueda promedio y la cantidad de accesos a bloques de disco en cada método de construcción. Se ajustará una curva para estimar los parámetros de complejidad temporal y se discutirá la aproximación obtenida.

1.5. Hipótesis

- **Hipótesis 1:** Se espera que la construcción del R-Tree utilizando el método Nearest-X sea el menos eficiente de los tres métodos. Sin embargo, podríamos pensar que este método tenga un rendimiento relativamente mejor para consultas de puntos en lugar de regiones los rectagunlos de los nodos internos, al ser delgados, descartarían bastante región. Sin embargo en el experimento actual, los rectangulos a user como consultas serán regiones grandes por lo que lo intuitivo es que este método no sea tan eficiente al descartar pocas regiones (ya que fácilmente intersectará con los rectangulos delgados de los nodos internos.
- **Hipótesis 2:** Se espera que el método de construcción Hilbert R-tree ofrezca un mejor rendimiento en términos de costos de búsqueda cuando los datos de entrada presentan una distribución dispersa y heterogénea en el espacio, lo que implica que los puntos de referencia están distribuidos de manera más uniforme en el plano. Dado que el método Hilbert R-tree utiliza la Curva de Hilbert para ordenar los centros de los rectángulos según su valor en la curva, puede manejar eficazmente situaciones en las que los datos están dispersos y distribuidos de manera más uniforme en el espacio. Por otro lado, la eficacia de este método depende fuertemente de encontrar un buen orden de Hilbert para que la curva cubra el espacio correctamente, por lo que también podemos anticipar que la eficacia de este método dependerá fuertemente de su grado.
- **Hipótesis 3:** Se espera que el método de construcción Sort-Tile-Recursive (STR) presente un mejor rendimiento en términos de costos de búsqueda en comparación con los métodos Nearest-X. Dado que el método STR realiza una división y ordenamiento más equitativo de los nodos de referencia en el R-tree, se anticipa que este enfoque proporcionará una distribución más uniforme de los datos en el árbol, lo que resultará en tiempos de búsqueda más eficientes y una menor cantidad de accesos a bloques de disco en comparación con los otros métodos. Si bien, por lo anteriormente mencionado, es muy probable que este método funcione mejor que Nearest X, es dificl saber cómo se comparará con Hilbert Curve.

2. Desarrollo del Programa

El desarrollo de las funciones principales se implementaron en C++.

2.1. Pipeline del Programa

Dado que la implentación a realizar tiene como objetivo comparar la performance de búsqueda de RTrees creados distintos métodos de *Bulk Load* para inputs con una gran cantidad de rectangulos, resulta fundamental poder guardar de forma confiable los pasos intermediarios del programa. Para esto ideamos la siguiente *pipeline*:

- Primero, se implementa un generador de rectangulos, para poder crear tanto los sets R como el set Q, es decir los rectángulos que se usaran para la construcción de los R-Trees, tomando inputs desde 2^{10} hasta 2^{25}, y los rectángulos que se usarán para las consultas (100 en total). Estos se guardan en archivos *.txt* que contienen los 4 números reales que los caracterizan.
- Una vez creados los rectángulos, se implementan los métodos constructores de R-Trees usando. Para esto creamos las clases RTree y Nodo. Estas construcciones se hacen en memoria.
- Luego podemos construir un RTree en memoria, lo escribimos a disco, en binarios que contendrán la información necesaria para poder hacer las consultas. Hay 45 archivos binarios en total: 15 por cada método.
- A continuación, ejecutamos las consultas en disco, para esto deserializamos los nodos guardados en disco para cada consulta y recuperamos los datos necesarios para saber por cuales ramas continuar la exploración y en qué posición del disco (*offset* se encuentran estas. Esto se ejecuta para cada binario (45 en total) para cada consulta (100 rectangulos), borrando el caché en cada iteración. Los resultados se guardan en formato *.csv* para luego ser analizados.

2.2. Creación de estructuras

En esta sección se muestra la definición de las estructuras fundamentales para el desarrollo del R-tree.

Se implementan 2 tipos de nodos, la clase Nodo sirve para construir los R-Trees en memoria y la estructura Nodo representa los datos que deserializados de un nodo en disco.

Dado que se busca optimizar el R-Tree lo más posible, se quiere obtener un arbol con la áridad mayor posible, es decir queremos maximizar M: el máximo número de hijos por nodo, ya que a mayor M, menor altura del árbol y por lo tanto consultas potencialmente más rápidas.

Sin embargo, se busca poder contar los accesos a discos, por lo que para esto, se impone que un nodo no use más de $B = 4096$ bytes (tamaño de un bloque). De esta forma, se tiene que leer un nodo en disco cuesta exactamente una lectura a disco, o potencialmente 2 si se encuentra a la mitad de 2 bloques, pero en cualquier caso, se puede afirmar que cuesta $O(1)$ accesos a disco.

Por lo tanto, se debe encontrar el M máximo posible que no provoque que el tamaño del nodo sea mayor a B. El tamaño de un nodo es disco se ve representado en la estructura NodeData y se puede ver su uso en bytes en los comentarios de cada uno de sus componentes.

Entonces tenemos: $B = 8 + 1 + 4 + 4 + 32M + 8M => M = 101$ el cual es el valor usado para esta implementación.

Un aspecto importante a mencionar es que la clase Node, que representa un nodo en memoria, tiene un campo *offset*, sin embargo este se encuetra vacío tras construir un BTree con alguno de los métodos. Estos se rellenan haciendo una pasada por el árbol usando un algoritmo de *Depth First Search*, calculando en cada iteración el espacio que cada nodo está usando realmente y con esto computando la posición donde se pondrá en el archivo. Se tomó la decisión de implemetar esto así en lugar de darle una cantidad de espacio fija (B por ejemplo) a cada nodo ya que esto sería muy ineficiente en espacio, ya que por ejemplo, si el nodo raiz tiene pocos hijos, entonces se estaría alocando inutilmente demasiado espacio. Gracias a esto se optimiza el espacio usado en disco, sin embargo la desventaja es que es menos flexible, es decir, agregar un nodo nuevo sería mucho más dificil en este caso. Pero como la implementación actual no busca ser dinámica (no hay inserciones o borrados), no es un problema.

Es por la razón anteriormente mencionada que la estructura Nodo posee también su número de rectángulos y de hijos, ya que es necesario conocer este valor para poder saber hasta dónde leer al momento de deserialización del nodo.

Código 1: Creación de estructuras

```
class Node {
  public:
    Rectangle MBR;
    vector<Node *> children;
    vector<Rectangle> rectangles;
    bool is_leaf;
    long long offset;

    Node(vector<Node *> children, vector<Rectangle> rectangles, bool is_leaf);

    void print();
    int diskSize();
};

struct NodeData {
    long long offset; // 8 bytes
    bool is_leaf; // 1 byte
    int num_rectangles; // 4 bytes
    int num_children; // 4 bytes
    vector<Rectangle> rectangles; // 32*M bytes
    vector<long long> children_offsets; // 8*M bytes
};
```

2.3. Métodos de construcción

Las siguientes secciones se muestran las implementaciones de los métodos de construcción de los R-Trees.

2.3.1. Nearest-X

Aquí mostramos el código que implementa la construcción del R-Tree ocupando el metodo Nearest-X. La función nearestX() llama a la función _nearestX() recursivamente, la cual reagrupa

los nodos y devuelve la raíz.

Este algoritmo recibe el M (aridad del árbol) y algún set R. Primero crea tuplas de la forma (rectangulo, centro), luego los ordena por el atributo x de su segunda componente y finalmente se itera por las tuplas creando las hojas del arbol, de a grupos de tamaño M. Al crear un nodo, este automáticamente calcula cosas como su *Minimum Bounding Rectangle* y esta lista se entrega a la función recursiva auxiliar que se encarga de seguir agrupándolos hasta que los nodos restantes quepan en la raiz.

Código 2: Construcción Nearest-X

```
1
2  // @brief Uses the NearestX algorithm build the root node of the RTree.
3
4  Node *nearestX(int M, vector<Rectangle> rectangles) {
5      // Create tuple with center of each rectangle
6      vector<tuple<Rectangle, Point>> tuples;
7      for (Rectangle rect : rectangles) { // O(n)
8          tuple<Rectangle, Point> tuple = make_tuple(rect, middle_point(rect));
9          tuples.push_back(tuple);
10     }
11
12     // Sort tuples by x-coordinate of center
13     std::sort( // O(n log n)
14         tuples.begin(), tuples.end(),
15         [](const tuple<Rectangle, Point> &a, const tuple<Rectangle, Point> &b) {
16             return get<1>(a).x < get<1>(b).x;
17         });
18
19     // Create nodes from sorted tuples
20     vector<Node *> leafs;
21     for (int i = 0; i < tuples.size(); i += M) { // O(n)
22         vector<Rectangle> rectangles;
23         for (int j = i; j < i + M && j < tuples.size(); j++) {
24             tuple<Rectangle, Point> tuple = tuples[j];
25             Rectangle rect = get<0>(tuple);
26             rectangles.push_back(rect);
27         }
28         Node *leaf = new Node(vector<Node *>(), rectangles, true);
29         leafs.push_back(leaf);
30     }
31
32     return _nearestX(M, leafs);
33 }
```

2.3.2. Hilbert R-tree

Aquí mostramos el codigo que implementa la construcción del R-Tree ocupando el metodo Hilbert R-Tree. La función hilbertValue() llama a la función_hilbertValue() recursivamente, la cual reagrupa los nodos y devuelve la raíz.

Este algutimo funciona de forma análoga al anterior pero con la diferencia que ordena en función de el valor del centro en la curva de Hilbert. Cabe mencionar que se toma 2^{19} como el grado del

fractal ya que esta es la primera potencia de 2 que logra cubrir apropiadamente un rectángulo de 500.000 x 500.000 (límite donde se sitúan los rectángulos)

Código 3: Construcción Hilbert R-tree

```
1
2  // @brief Auxiliary function to rotate a point.
3
4  void rotate(int n, int *x, int *y, int rx, int ry) {
5      if (ry == 0) {
6          if (rx == 1) {
7              *x = n - 1 - *x;
8              *y = n - 1 - *y;
9          }
10         // Swap x and y
11         int temp = *x;
12         *x = *y;
13         *y = temp;
14     }
15 }
16
17 // @brief Calculates the Hilbert value of a point.
18
19
20 unsigned long long hilbertValue(Point point, unsigned long long n) {
21
22     unsigned long long hilbertVal = 0;
23     int x = point.x;
24     int y = point.y;
25     unsigned long long s = n / 2;
26
27     while (s > 0) {
28         int rx = (x & s) > 0;
29         int ry = (y & s) > 0;
30         hilbertVal += s * s * ((3 * rx) ^ ry);
31         rotate(s, &x, &y, rx, ry);
32         s /= 2;
33     }
34
35     return hilbertVal;
36 }
37
38
39 // @brief Uses the hilbertCurve algorithm build the root node of the RTree.
40
41 Node *hilbertCurve(int M, vector<Rectangle> rectangles) {
42
43     unsigned long long n = pow(2, 19);
44
45     // Create tuple with center of each rectangle and their Hilbert values
46     unsigned long long i = 0;
47     vector<tuple<Rectangle, unsigned long long>> tuples;
48     for (Rectangle rect : rectangles) { // O(n)
```

```
        Point realCenter = middle_point(rect);
        Point center = {round(realCenter.x), round(realCenter.y)};
        unsigned long long hilbertVal = hilbertValue(center, n); //

        // Adjust the order as needed tuple
        tuple<Rectangle, unsigned long long> tuple =
            make_tuple(rect, hilbertVal);
        tuples.push_back(tuple);
    }

    // Sort tuples by Hilbert values
    std::sort( // O(n log n)
        tuples.begin(), tuples.end(),
        [](const tuple<Rectangle, unsigned long long> &a,
           const tuple<Rectangle, unsigned long long> &b) {
            return get<1>(a) < get<1>(b);
        });

    // Create nodes from sorted tuples
    vector<Node *> leafs;
    for (int i = 0; i < tuples.size(); i += M) { // O(n)
        vector<Rectangle> rectangles;
        for (int j = i; j < i + M && j < tuples.size(); j++) {
            tuple<Rectangle, unsigned long long> tuple = tuples[j];
            Rectangle rect = get<0>(tuple);
            rectangles.push_back(rect);
        }
        Node *leaf = new Node(vector<Node *>(), rectangles, true);
        leafs.push_back(leaf);
    }

    return _hilbertCurve(M, leafs);
}
```

2.3.3. Sort-Tile-Recursive(STR)

Aquí mostramos el codigo que implementa la construcción del R-Tree ocupando el metodo Sort-Tile-Recursive. La función sortTileRecursive() llama a la función _sortTileRecursive() recursivamente, la cual reagrupa los nodos y devuelve la raíz.

Este algoritmo también es análogo a Nearest X, con la diferencia de que junta grupos mejor distribuídos que el anterior, al hacer una agrupación por x primero y luego una por y en cada subgrupo.

Código 4: Construcción Sort-Tile-Recursive

```

// @brief Uses the Sort-Tile-Recursive algorithm build the root node of the RTree.

Node *sortTileRecursive(int M, vector<Rectangle> rectangles) {
    int n = rectangles.size();
    int S = (int) ceil(sqrt(n / M));
```

```
7
8   // Create tuples of (rectangle, center)
9   vector<tuple<Rectangle, Point>> tuples;
10  for (int i = 0; i < rectangles.size(); i++) {
11      Rectangle rectangle = rectangles[i];
12      Point center = middle_point(rectangle);
13      tuples.push_back(make_tuple(rectangle, center));
14  }
15
16  // Sort tuples by x-coordinate of center
17  std::sort( // O(n log n)
18      tuples.begin(), tuples.end(),
19      [](const tuple<Rectangle, Point> &a, const tuple<Rectangle, Point> &b) {
20          return get<1>(a).x < get<1>(b).x;
21      });
22
23  // Group tuples into S groups of size M*S
24  vector<vector<tuple<Rectangle, Point>>> groups;
25  for (int i = 0; i < tuples.size(); i += M * S) {
26      vector<tuple<Rectangle, Point>> group;
27      for (int j = i; j < i + M * S && j < tuples.size(); j++) {
28          group.push_back(tuples[j]);
29      }
30      groups.push_back(group);
31  }
32
33  // Sort each group by y-coordinate of center
34  for (int i = 0; i < groups.size(); i++) {
35      std::sort( // O(n log n)
36          groups[i].begin(), groups[i].end(),
37          [](const tuple<Rectangle, Point> &a,
38             const tuple<Rectangle, Point> &b) {
39              return get<1>(a).y < get<1>(b).y;
40          });
41  }
42
43  // Group each group into S groups of size M
44  vector<vector<vector<tuple<Rectangle, Point>>>> groups2;
45  for (int i = 0; i < groups.size(); i++) {
46      vector<vector<tuple<Rectangle, Point>>> group2;
47      for (int j = 0; j < groups[i].size(); j += M) {
48          vector<tuple<Rectangle, Point>> group3;
49          for (int k = j; k < j + M && k < groups[i].size(); k++) {
50              group3.push_back(groups[i][k]);
51          }
52          group2.push_back(group3);
53      }
54      groups2.push_back(group2);
55  }
56
57  // Create nodes from each group (S*S leaf nodes)
58  vector<Node *> leafs;
```

```
    for (int i = 0; i < groups2.size(); i++) {
        for (int j = 0; j < groups2[i].size(); j++) {
            vector<Rectangle> rectangles;
            for (int k = 0; k < groups2[i][j].size(); k++) {
                tuple<Rectangle, Point> tuple = groups2[i][j][k];
                Rectangle rect = get<0>(tuple);
                rectangles.push_back(rect);
            }
            Node *leaf = new Node(vector<Node *>(), rectangles, true);
            leafs.push_back(leaf);
        }
    }

    return _sortTileRecursive(M, leafs, S);
}
```

2.4. Busqueda

Aquí se muestra la implementación del metodo que devuelve una lista con todos los rectángulos pertenecientes a la solución, calculando tambien la cantidad de accesos a los bloques de disco realizados durante la ejecución.

El método readNodeFromDisk es aquel que toma un *offset* y que lee y deserializa una estructura de nodo con la información relevante. Dado que un nodo usa a lo más B bytes, podemos contar una de estas lecturas como 1 I/O.

Código 5: Búsqueda

```

void _query(RTree *rtree, long long offset, Rectangle query_rect,
            vector<Rectangle> &result, ifstream &input_file, int *readCounter) {
    NodeData node_data = rtree->readNodeFromDisk(offset);
    *readCounter += 1;

    if (node_data.is_leaf) {
        for (int i = 0; i < node_data.rectangles.size(); i++) {
            Rectangle rectangle = node_data.rectangles[i];
            if (intersects(rectangle, query_rect)) {
                result.push_back(rectangle);
            }
        }
    } else {
        for (int i = 0; i < node_data.rectangles.size(); i++) {
            Rectangle rectangle = node_data.rectangles[i];
            if (intersects(rectangle, query_rect)) {
                _query(rtree, node_data.children_offsets[i], query_rect, result,
                       input_file, readCounter);
            }
        }
    }
}
```

3. Resultados

A continuación los resultados de las mediciones de las experiencias realizadas. Estas se presentan en tablas, gráficos e imágenes.

3.1. Características del equipo usado para los tests

- **Procesador:** AMD 5700G 3,8 GHz
- **RAM instalada:** 16,0 GB (15,3 GB utilizable)
- **Tipo de sistema:** Sistema operativo de 64 bits, procesador x64

3.2. Resultados de búsqueda

En esta sección se muestran los resultados de busqueda para los 3 tipos de árboles. Para esto se construyeron un set de 100 rectangulos en donde sus valores son enteros uniformemente distribuidos en el rango de $[0, 500000]$, el tamaño de cada lado de los rectángulos de R estan uniformemente distribuido entre [0, 100]. Teniendo estos rectangulos se realizaron busquedas en cada rectángulo, y se tomo el tiempo de demora al mismo tiempo que se contó el numero de lecturas que realizó en el disco. Luego se graficaron los resultados.

3.2.1. Resultados de búsqueda método Nearest-X

n	Duration mean [µs]	Duration CI	Reads mean	Reads CI
2^{10}	70.32	[59.72, 80.92]	3.01	[2.87, 3.15]
2^{11}	99.61	[87.91, 111.31]	4.11	[3.87, 4.35]
2^{12}	73.74	[54.95, 92.53]	6.19	[5.72, 6.66]
2^{13}	115.97	[93.80, 138.14]	10.42	[9.48, 11.36]
2^{14}	195.88	[165.61, 226.15]	19.89	[18.03, 21.75]
2^{15}	354.14	[308.77, 399.51]	36.84	[33.11, 40.57]
2^{16}	659.24	[583.93, 734.55]	70.93	[63.46, 78.40]
2^{17}	1299.68	[1154.46, 1444.90]	138.94	[124.02, 153.86]
2^{18}	2542.60	[2262.53, 2822.67]	274.67	[244.91, 304.43]
2^{19}	4966.170	[4418.26, 5514.08]	545.64	[486.09, 605.19]
2^{20}	9949.27	[8841.67, 11056.87]	1090.13	[970.96, 1209.30]
2^{21}	19778.06	[17580.60, 21975.52]	2174.46	[1936.25, 2412.67]
2^{22}	38973.23	[34638.30, 43308.16]	4346.89	[3870.30, 4823.48]
2^{23}	79268.39	[70426.75, 88110.03]	8691.53	[7738.43, 9644.63]
2^{24}	158073.80	[140441.21, 175706.39]	17380.64	[15474.07, 19287.21]
2^{25}	322614.74	[286670.79, 358558.69]	34748.74	[30936.85, 38560.63]

3.2.2. Resultados de búsqueda método Hilbert R-tree

n	Duration mean [µs]	Duration CI	Reads mean	Reads CI
2^{10}	24.74	[21.30, 28.18]	2.86	[2.70, 3.02]
2^{11}	38.04	[27.41, 48.67]	3.63	[3.43, 3.83]
2^{12}	47.75	[36.01, 59.49]	4.02	[3.76, 4.28]
2^{13}	67.29	[47.97, 86.61]	5.21	[4.85, 5.57]
2^{14}	108.59	[76.18, 141.00]	8.17	[7.60, 8.74]
2^{15}	159.85	[122.34, 197.36]	11.35	[10.37, 12.33]
2^{16}	244.02	[191.53, 296.51]	16.61	[14.88, 18.34]
2^{17}	436.57	[339.78, 533.36]	26.88	[23.74, 30.02]
2^{18}	730.96	[596.90, 865.02]	43.98	[38.03, 49.93]
2^{19}	1248.66	[1053.21, 1444.11]	76.64	[65.31, 87.97]
2^{20}	2130.33	[1798.58, 2462.08]	140.02	[118.19, 161.85]
2^{21}	3927.19	[3313.58, 4540.80]	260.33	[218.01, 302.65]
2^{22}	6977.46	[5881.47, 8073.45]	491.58	[408.90, 574.26]
2^{23}	12776.84	[10631.22, 14922.46]	948.76	[785.57, 1111.95]
2^{24}	26300.96	[21700.06, 30901.86]	1846.39	[1523.42, 2169.36]
2^{25}	51940.62	[42885.70, 60995.54]	3625.87	[2985.38, 4266.36]

3.2.3. Resultados de búsqueda método Sort-Tile-Recursive

n	Duration mean [µs]	Duration CI	Reads mean	Reads CI
2^{10}	27.13	[20.45, 33.81]	2.58	[2.43, 2.73]
2^{11}	31.51	[24.38, 38.64]	2.94	[2.74, 3.14]
2^{12}	46.57	[33.18, 59.96]	3.73	[3.47, 3.99]
2^{13}	89.97	[49.71, 130.23]	4.40	[4.04, 4.76]
2^{14}	130.19	[96.13, 164.25]	7.55	[6.93, 8.17]
2^{15}	202.65	[117.26, 288.04]	10.41	[9.42, 11.40]
2^{16}	287.17	[220.23, 354.11]	15.59	[13.96, 17.22]
2^{17}	465.71	[353.20, 578.22]	25.38	[22.18, 28.58]
2^{18}	830.30	[642.05, 1018.55]	43.45	[37.47, 49.43]
2^{19}	1444.62	[1142.96, 1746.28]	76.10	[64.88, 87.32]
2^{20}	2595.97	[2057.91, 3134.03]	142.20	[120.46, 163.94]
2^{21}	4826.49	[3930.53, 5722.45]	268.91	[226.12, 311.70]
2^{22}	9228.42	[7513.75, 10943.09]	511.93	[427.92, 595.94]
2^{23}	17665.32	[14555.26, 20775.38]	983.89	[818.32, 1149.46]
2^{24}	34010.72	[28098.97, 39922.47]	1904.15	[1577.64, 2230.66]
2^{25}	63655.19	[53190.55, 74119.83]	3706.07	[3061.52, 4350.62]

3.3. Gráficos

A continuación se presetan los gráficos correspondientes a los resultados experimentales. Los gráficos representan tanto los tiempos de lectura, duración y tiempos de ejecución en base a los distintos tamaños de los inputs.

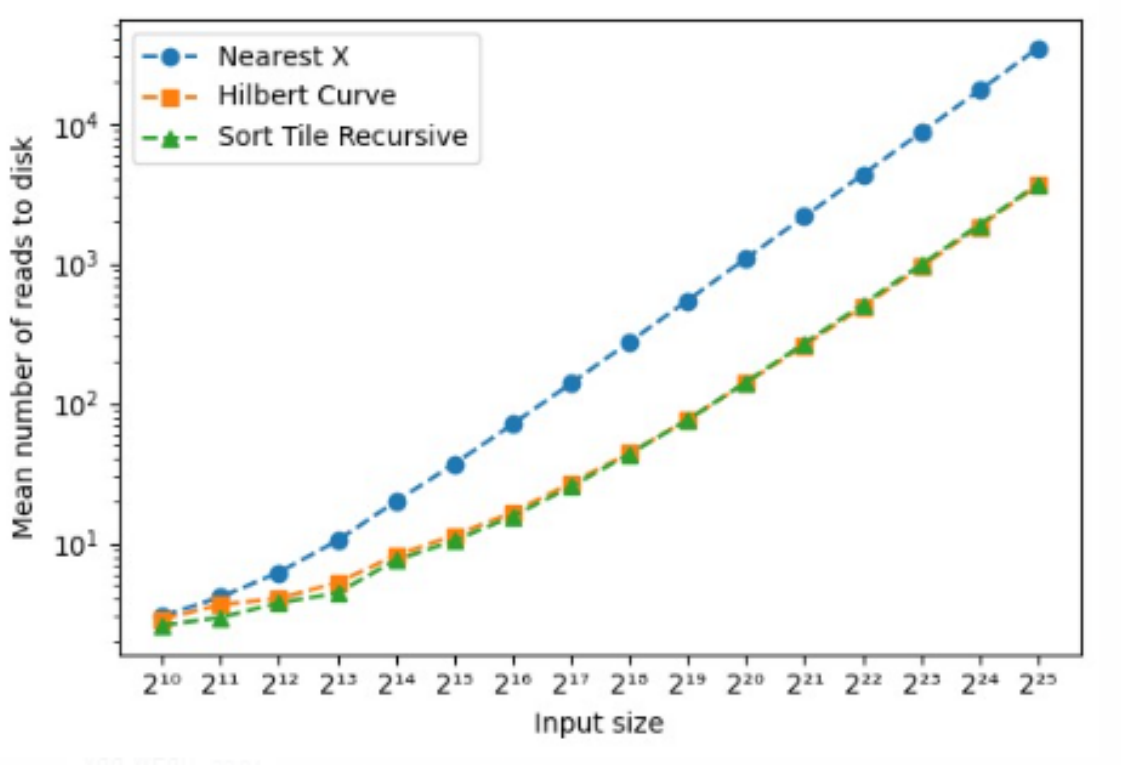

Figura 1: Promedio de número de Lecturas para cada método

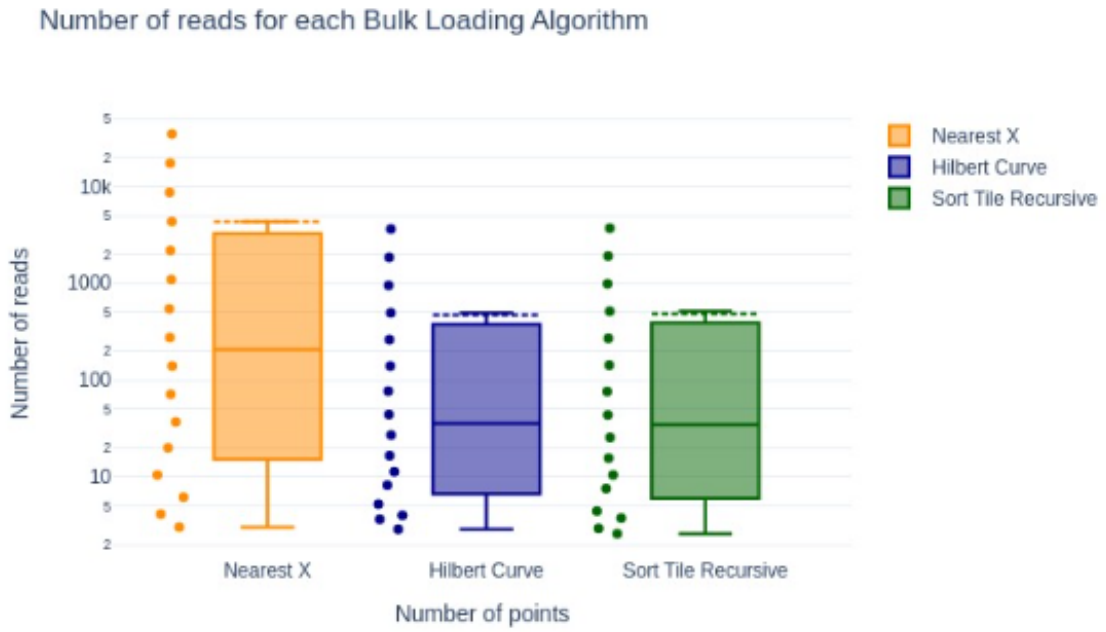

Figura 2: Comparación de los 3 algoritmos: número de lecturas

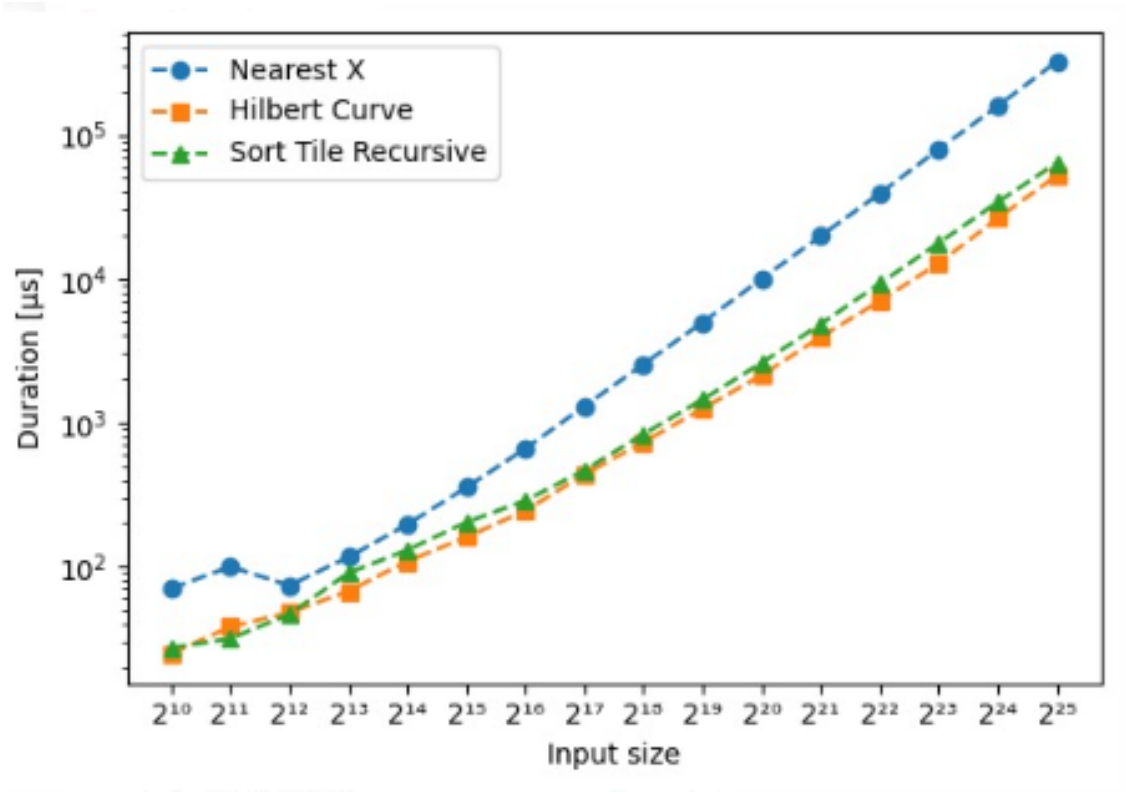

Figura 3: Tiempo de demora de la Busqueda

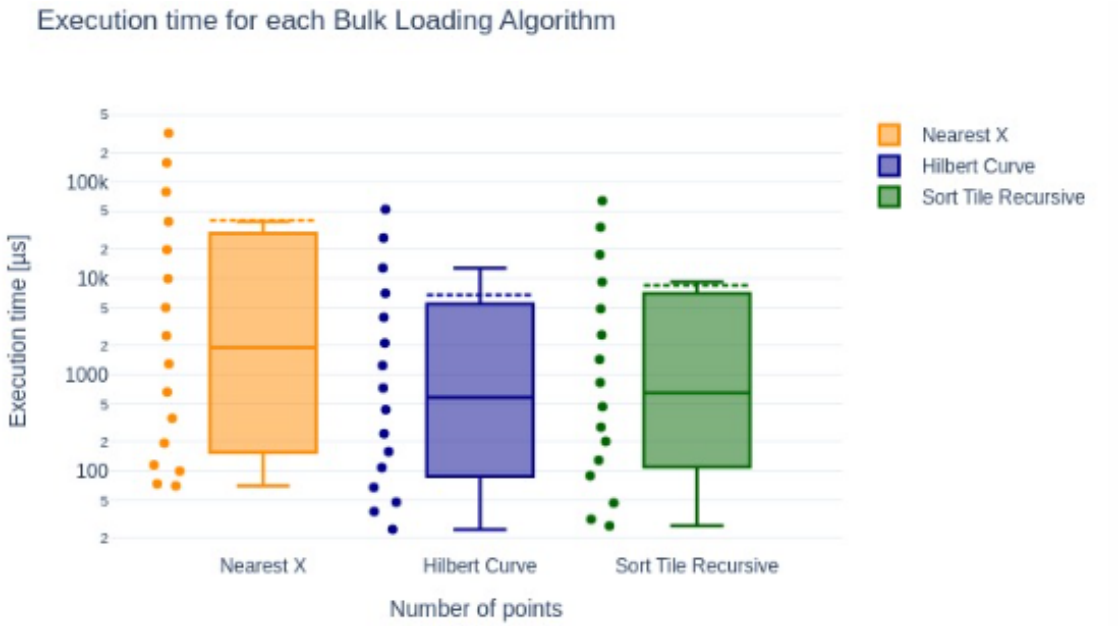

Figura 4: Comparación de los 3 algoritmos: tiempo de ejecución

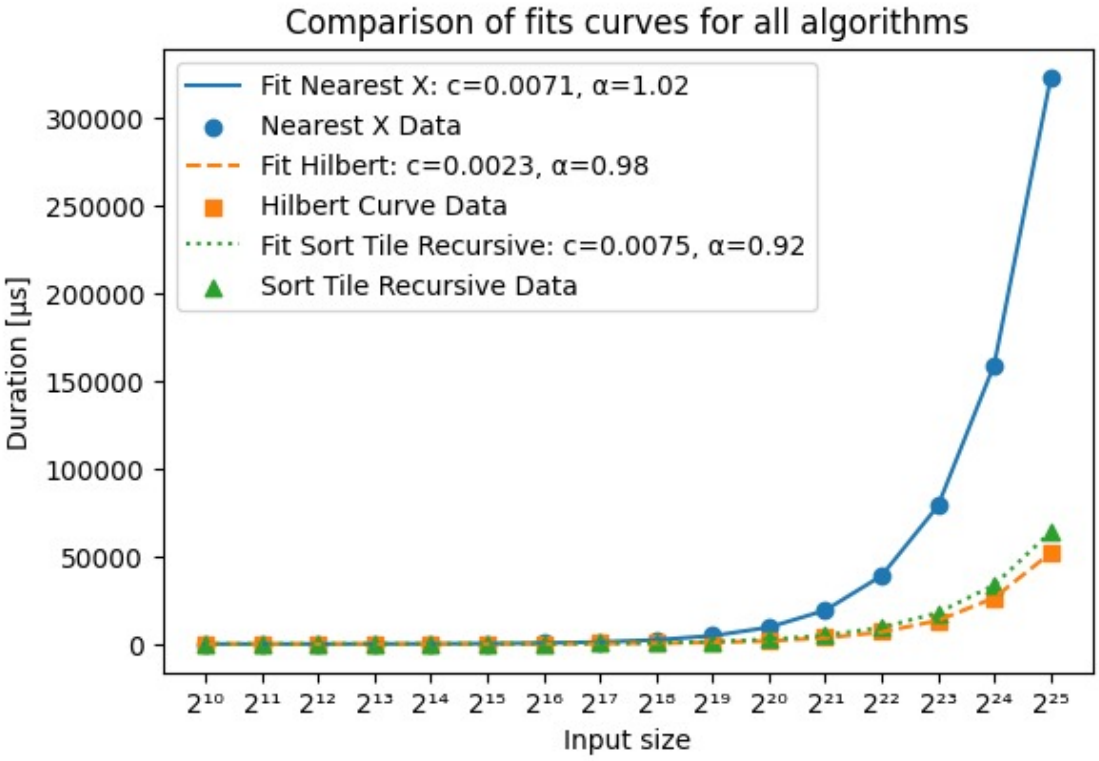

Figura 5: Comparación curvas de ajuste para los 3 algoritmos

4. Análisis de Resultados

Los resultados de los experimentos revelan patrones distintos de desempeño para cada uno de los métodos de construcción de R-trees. Estos patrones proporcionan información valiosa sobre la eficiencia relativa de cada método en diferentes contextos de uso. A continuación, se presentan los análisis y conclusiones derivadas de los resultados obtenidos.

De acuerdo con los resultados de los tres gráficos, se observa que el método Nearest-X presenta una duración promedio notablemente mayor en comparación con los otros dos métodos, lo cual se evidencia en el grafico 1. Por otro lado, se destaca que la diferencia de duración promedio entre los otros dos métodos es de tan solo 10 segundos, y sus medias de lecturas son muy similares, lo que sugiere que ambos métodos son prácticamente igualmente eficientes.

Por otro lado, en el gráfico 2 se observa una situación similar, ya que los resultados en el número de lecturas muestran una similitud marcada entre el método de construcción basado en la *Curva de Hilbert* y el método *Sort-Tile-Recursive*, mientras que se evidencia que el método *Nearest-X* requiere considerablemente más lecturas a disco para las búsquedas.

En cuanto al tiempo de demora de las búsquedas, grafico 3, entre los distintos métodos, inicialmente se observa un mejor desempeño del método *Sort-Tile-Recursive*; sin embargo, a medida que avanzan las pruebas, el método basado en la *Curva de Hilbert* tiende a ser más rápido en las búsquedas. Por otro lado, se evidencia que el método *Nearest-X* es significativamente más lento que los otros dos.

En el gráfico 4, se puede observar una tendencia similar, en la cual el método basado en la *Curva de Hilbert* muestra un rendimiento ligeramente mejor en comparación con el método *Sort-Tile-Recursive*, a pesar de que ambos son bastante similares entre sí. Sin embargo, es notable que el método *Nearest-X* exhibe consistentemente un peor desempeño en los tiempos de ejecución en general.

El gráfico 5 muestra el análisis de ajuste de curvas de tiempo usando el modelo: $f(n) = c * n^{\alpha}$. Se puede ver que las curvas se ajustan correctamente a los datos reales para cada algoritmo, por lo que se encontraron valores empíricos que pueden servir para describir matemáticamente la duración de las consultas para cáda método en función del tamaño del input. Por lo tanto se puede decir la duración de las consultas tiene un comportamiento exponencial donde los valores de α y c encontrados (se pueden ver en la figura) nos indican su comportamiento. Estos resultados se condicen para Nearest X ya que es aquel que tiene el valor de α más grande, lo cual indica que el método es menos eficiente. Sin embargo, los valores de α y c de Hilbert Curve y STR parecieran indicar que STR es más costoso para inputs más pequeños (mayor c) pero menos costoso para valores más grandes (α más pequeño).

5. Conlusiones

En esta tarea, se exploraron tres métodos de construcción de R-trees: Nearest-X, Hilbert R-tree y Sort-Tile-Recursive (STR). Estos métodos se evaluaron en términos de su eficiencia en las operaciones de búsqueda, específicamente, el tiempo promedio de búsqueda y la cantidad de accesos a bloques de disco. A continuación, presentamos las conclusiones específicas para cada método y sus resultados.

Los resultados experimentales sugieren que el método Nearest-X tiende a ser menos eficiente en términos de tiempo de búsqueda y accesos a bloques de disco en comparación con los otros métodos, especialmente cuando se realizan consultas con rectángulos grandes. Aunque se planteó la hipótesis de que Nearest-X podría ser eficiente para consultas de puntos, en el experimento actual, los rectángulos utilizados como consultas eran regiones grandes. Esto indica que Nearest-X no es óptimo para manejar la intersección de regiones grandes, ya que no puede descartar eficazmente regiones irrelevantes.

Los resultados sugieren que el método Hilbert R-tree ofrece un mejor rendimiento en términos de costos de búsqueda cuando los datos de entrada presentan una distribución dispersa y heterogénea en el espacio. Este método se beneficia de su capacidad para ordenar los rectángulos según la Curva de Hilbert, lo que permite una distribución más uniforme de los datos en el árbol. Sin embargo, la eficacia de Hilbert R-tree depende en gran medida de encontrar un buen orden de Hilbert que cubra el espacio de manera óptima.

El método Sort-Tile-Recursive (STR) presenta un mejor rendimiento en términos de costos de búsqueda en comparación con el método Nearest-X. STR realiza una división y ordenamiento más equitativo de los nodos de referencia en el R-tree, lo que resulta en una distribución más uniforme de los datos en el árbol. Esto se traduce en tiempos de búsqueda más eficientes y una menor cantidad de accesos a bloques de disco en comparación con Nearest-X. Si bien se esperaba que STR funcionara mejor que Nearest-X, es difícil determinar cómo se comparará con Hilbert Curve, ya que este último también muestra un buen rendimiento en ciertos escenarios.

En resumen, la elección del método de construcción del R-tree debe basarse en la distribución de los datos y las características de las consultas que se realizarán. Nearest-X tiende a ser menos eficiente en la mayoría de los casos, mientras que Hilbert R-tree es adecuado para datos dispersos y heterogéneos. STR ofrece un buen rendimiento en general y puede considerarse una opción sólida en ausencia de información específica sobre la distribución de los datos.

En una futura instancia, sería intersante analizar realizar este experimento para distintos valores de M. Si bien el M se calculó para maximizar la aridad del árbol sin hacer que un nodo use más de $B = 4096$ bytes, cabe cuestionarse si la duración de las consultas efectivamente empeoraría la duración de las consultas. Sabemos que en teoría, el modelo de complejidad en tiempo con memoria secundaria indica que lo más costoso son las lecturas y escrituras a disco, sería interesante probar si, al no tener restricciones sobre el tamaño del nodo, se obtienen mejores o peores resultados en la práctica. Este estudio sería pertinente ya que el M es un parámetro fundamental en la construcción del R-Tree.