CC4102 Diseño y Análisis de Algoritmos

Informe Tarea I

Agustín López Q.
Matías Cisterna M.

22 de septiembre de 2014

Índice

1. Introducción

En esta tarea 1 de Diseño y Análisis de Algoritmos se busca analizar sobre una estructura de datos dada (R-tree) diferentes algoritmos de inserción y como estos repercuten en la operación de búsqueda sobre la estructura de datos.

Un R-tree es una estructura de datos del tipo árbol. Esta estructura es usualmente utilizada para métodos de acceso espacial. Es decir, esta estructura indexa información de localización, para luego ser consultada de forma eficiente. Consultas del tipo: *encontrar todos los puntos en un radio y centro dado, que cumplan cierto requisito*. La idea de la estructura R-tree es a medida que se inserta y se borra ir manteniendo el árbol balanceado y que todos los nodos y hojas cercanos a un dato estén localizados espacialmente cerca. Con el fin de lograr que la búsqueda de un vecindario de un punto en especial sea eficiente.

2. Hipótesis

3. Diseño Experimental

3.1. Implementación

Para la implementación se usó el lenguaje de programación Java. El diseño se dividió en 5 clases:

- *RTree*: Representa el árbol R-Tree.
- *Node*: Representa un nodo dentro del árbol.
- *Rectangle*: Representa los rectángulos guardados en los nodos.
- *StopWatch*: Clase creada para medir tiempos de forma simple.
- *Main*: Clase con método *main*, es desde donde se ejecuta el programa.

Para la representación de los rectángulos se creó la clase *Rectangle*, donde se guardan como variables de instancia los puntos (x_1, y_1) y (x_2, y_2) los que representan la esquina inferior izquierda y superior derecha, respectivamente. Además se guarda como variable de instancia una referencia a un nodo, el que puede ser nulo cuando el rectángulo es real (rectángulo insertado con metodo *insert*), o puede ser un nodo hijo cuando el rectángulo es un *MBR*. El objeto *Rectangle* cuenta con métodos para obtener el área y saber si dos rectángulos se intersectan (para esta parte se considera que si dos rectángulos solo comparten un vértice o una arista entonces no se intersectan). La clase *Node* guarda como variable de instancia una lista con sus rectángulos, la variable t, una referencia a su nodo padre y una variable booleana que indica si es o no una hoja. Por su parte la estructura de datos R-Tree guarda los rectángulos en sus nodos. La cantidad de rectángulos guardados en cada nodo del árbol fluctua entre t y $2t$, donde t es definida previamente y depende a su vez de la siguiente condición: *La cantidad de rectángulos es tal que, un nodo completo debe caber totalmente dentro de un bloque de memoria.* Para hacer esto se usaron clases provistas por el lenguaje para calcular el tamaño que usaría una estructura *Node* y el hecho de que un bloque tiene tamaño $4KB$ (4096 bytes). La implementación de la estructura de datos consta de dos algoritmos, uno de búsqueda y otro de inserción de rectángulos.

- Búsqueda: El algoritmo de búsqueda recibe como parámetro un rectángulo y retorna una lista con todos los rectángulos reales que intersectan a este rectángulo. La implementación es bastante sencilla, pero tiene una pequeña modificación, no retorna la lista de rectángulos que intersectan. Si no que recibe como parámetro una lista vacía y esta se va llenando a medida que se encuentre con rectángulos que intersectan. Para luego retornar el número de lecturas/escrituras que hace el algoritmo. Las que son simuladas y contadas cada vez que se accede a un nodo hijo correspondiente a un rectángulo MBR que intersecta con el rectángulo parámetro.

```
public int searchRectangle(Rectangle r, ArrayList<Rectangle>
    lst)
  int IOs = 0;
  for(Rectangle nr : rectangles){
    if(!r.equals(nr) && r.intersect(nr)){
      if(isLeaf)
        lst.add(nr);
      else
        IOs += 1 + nr.node.searchRectangle(r,lst);
    }
  }
}
return IOs;
```

- Inserción: Para la inserción se hace un recorrido del árbol hasta llegar a una hoja, que es donde se insertará el nuevo rectanángulo. El recorrido es como sigue: Se parte de la raíz y se busca el rectángulo en su lista que genere el menor incremento de área al crear el MBR entre este y el nuevo rectángulo. Luego se desciende por el nodo asociado a este rectángulo y se hace el mismo procedimiento, hasta llegar a una hoja. Al igual que en la búsqueda, en la inserción se hace una modificación al algoritmo para que este retorne el número de I/O's que hace a disco. Estas I/O's se cuentan de la siguiente forma: Si el rectángulo que está en el nodo no es una hoja, se suma uno a la cantidad total de I/O's y se hace el llamado recursivo. De lo contrario, si es una hoja, se usa un nuevo método llamado *insertOverflowedRectangle* el que se encarga de insertar el rectángulo y hacer *split* si se produce *overflow*. Para contar la cantidad de I/O's en este método se usa la siguiente estrategia: Si después de insertar no hay *overflow* entonces se retorna 0. De lo contrario primero se verifica si es que estamos en la raíz, de ser así creamos un nuevo nodo. El que será la nueva raíz y le sumamos uno al total de I/O's. Ya que al crear el nuevo nodo se cuenta una nueva escritura. Luego se verifica cual de los 2 métodos se usara para separar la lista de rectángulos y se llama al algoritmo correspondiente (*linearSplit* o *quadraticSlpit*). Luego se aloca en memoria un nuevo nodo (nueva escritura) el que representará al nuevo nodo con la "mitad"de los rectángulos del nodo anterior, y se suma uno más a la cantidad de I/O's. Después se calculan los nuevos MBR de ambos grupos de rectángulos y se hace un llamado recursivo por cada rectángulo nuevo creado. Para así insertarlos en el nodo padre, y la cantidad de I/O's de estos dos llamados recursivos se suman al total de I/O's que se han acumulado, finalemente llamando al método *update* que actualiza todos los MBR desde la hoja hasta el padre.

Los dos métodos para separar nodos son heurísticas para obtener una separación lo más *equitativa* posible de los rectángulos. A Continuación se detalla a grandes razgos la implementación de ambos algoritmos:

- *QuadraticSplit*: Escogemos los dos rectángulos R_1 y R_2 cuyo incremento de área es máximo si es que fueran puestos en el mismo grupo. Es decir, R_1 y R_2 maximizan (área del MBR de R_1 y R_2) $-$ (área de R_1 + área de R_2), sobre todos los pares de rectángulos. Asignamos R_1 y R_2 a grupos distintos.

```
public void quadraticSplit(ArrayList<Rectangle> g1,
    ArrayList<Rectangle> g2){
  Rectangle newMBR;
  // indices finales de los rectangulos elegidos para
     separar en 2 grupos
  int final_i = 0, final_j = 1;
  double incremento = 0.0;
  ArrayList<Rectangle> lst = new ArrayList<Rectangle>();
  // separando en grupos
  for(int i=0;i<rectangles.size()-1;i++){
    lst.add(rectangles.get(i));
    for(int j=i+1; j<rectangles.size();j++){
      lst.add(rectangles.get(j));
      newMBR = makeMBR(lst);
      double newincremento = newMBR.area() - rectangles.get(
          i).area() - rectangles.get(j).area();
      if(incremento < newincremento){
        incremento = newincremento;
        final_i = i;
        final_j = j;
```

```
            }
            lst.remove(rectangles.get(j));
        }
        lst.remove(rectangles.get(i));
    }
    // asignamos a R1 y R2 a grupos distintos
    Rectangle R1 = rectangles.get(final_i);
    Rectangle R2 = rectangles.get(final_j);
    g1.add(R1);
    g2.add(R2);
    rectangles.remove(R1);
    rectangles.remove(R2);
```

Iterativamente asignamos un grupo para el resto de los rectángulos como sigue:

- Si todos los rectángulos tienen grupo asignado, nos detenemos. Si hay un grupo con tan pocos elementos, que la única forma de que tenga al menos t rectángulos es agregando todos los rectángulos restantes a este grupo, entonces los agregamos y nos detenemos.
- Para cada rectángulo R que aún no tiene grupo, calculamos g_1 = incremento en área por agregar R al grupo 1. Similarmente, calculamos g_2. Escogemos R tal que maximiza $|g_1 - g_2|$. Agregamos R al grupo cuyo incremento en área por incluir R es menor. Si hay empate, escogemos el grupo con menor área. Si el área es la misma, escogemos el grupo con menor cantidad de rectángulos. Si sigue habiendo empate, escogemos un grupo arbitrariamente. Volvemos al paso anterior.

- *LinearSlpit*: Se considera un rectángulo $R = [x_1, x_2] \times [y_1, y_2]$. El lado menor y mayor de R en la dimensión x es x_1 y x_2, respectivamente. Similarmente, para la dimensón y será y_1 e y_2. Escogemos dos rectángulos R_1 y R_2 como sigue: Por cada dimensión d, buscamos el rectángulo A_d cuyo lado mayor a_d según d es mínimo, y el rectángulo B_d cuyo lado menor b_d según d es máximo. Calculamos w_d como $b_d - a_d$ dividido por el ancho del conjunto en la dimensión d. Escogemos $R_1 = A_d$ y $R_2 = B_d$ tal que w_d es máximo, sobre $d = \{x, y\}$. Asignamos R_1 y R_2 a grupos distintos.

```
public void linearSplit(ArrayList<Rectangle> g1, ArrayList<
    Rectangle> g2){
```

```java
Rectangle R1, R2;
/* para ambas dimensiones buscamos el rectangulo Ad cuyo
    lado mayor ad es minimo
 * y el rectangulo Bd cuyo lado menor bd es maximo
 */
Rectangle Ax = rectangles.get(0);
double ax = Ax.x2;
Rectangle Ay = rectangles.get(0);
double ay = Ay.y2;
Rectangle Bx = rectangles.get(0);
double bx = Bx.x1;
Rectangle By = rectangles.get(0);
double by = By.y1;
for(Rectangle nr : rectangles){
  if(nr.x2 < ax){
    ax = nr.x1;
    Ax = nr;
  }
  if(nr.y2 < ay){
    ay = nr.y1;
    Ay = nr;
  }
  if(nr.x1 > bx){
    bx = nr.x2;
    Bx = nr;
  }
  if(nr.y1 > by){
    by = nr.y2;
    By = nr;
  }
}
double anchox = (Bx.x2 < Ax.x2 ? Ax.x2 : Bx.x2) - (Bx.x1 >
     Ax.x1 ? Ax.x1 : Bx.x1);
double anchoy = (By.y2 < Ay.y2 ? Ay.y2 : By.y2) - (By.y1 >
     Ay.y1 ? Ay.y1 : By.y1);
double wx = (bx - ax)/anchox;
double wy = (by - ay)/anchoy;
if(wx > wy){
  if(Ax != Bx){
    R1 = Ax;
    R2 = Bx;
  }else{
    R1 = rectangles.get(0);
    R2 = rectangles.get(1);
  }
```

```
}else{
  if(Ay != By){
    R1 = Ay;
    R2 = By;
  }else{
    R1 = rectangles.get(0);
    R2 = rectangles.get(1);
  }
}
g1.add(R1);
g2.add(R2);
rectangles.remove(R1);
rectangles.remove(R2);
```

Para asignar los rectángulos restantes hacemos lo siguiente:

- Si todos los rectsángulos tienen grupo asignado, nos detenemos. Si hay un grupo con tan pocos elementos, que la única forma de que tenga al menos t rectángulos es agregándo todos los rectángulos restantes a este grupo, entonces los agregamos y nos detenemos.
- Escogemos arbitrariamente un rectángulo R que no tiene grupo asignado. Agregamos R al grupo cuyo incremento en área por incluir R es menor. Si hay empate, escogemos el grupo con menor área. Si el área es la misma, escogemos el grupo con menor cantidad de rectángulos. Si sigue habiendo empate, escogemos un grupo arbitrariamente. Volvemos al paso anterior.

3.2. Generación de Instancias

El experimento consiste en medir dos tipos de instancias:

- **Aleatorias**: Se generan rectángulos aleatorios de la siguiente forma: Primero se generan 2 enteros x_1 e y_1 que se distribuyen uniformemente en el intervalo $[0, 500000]$, estos números representan la coordenada de la esquina inferior izquierda del rectángulo. Luego, x_2 se obtiene sumando a x_1 un número aleatorio en el intervalo $[0, 100]$ y finalemente se obtiene y_2 usando las variables anteriores y asumiendo que el área del rectángulo debe estar distribuida aleatoriamente en el intervalo $[0, 100]$.

```
static public Rectangle randomRectangle(){
  /* (x1,y1) es la esquina inferior izquierda del rectangulo
```

```
    *  la que se obtiene de forma uniformemente aleatoria
    */
   int x1 = randomInt(0,500000);
   int y1 = randomInt(0,500000);
   int x2 = x1 + randomInt(0,100);
   int y2 = y1 + randomInt(0,100)/(x2-x1);
   return new Rectangle(x1,x2,y1,y2);
}
```

Con lo anterior se generan dos árboles y conjuntos de rectángulos de tamaño n, con $n \in \{2^9, 2^{12}, 2^{15}, 2^{18}, 2^{21}, 2^{24}\}$. Cada conjunto se agrega a uno de los árboles con uno de los métodos de inserción implementados, el que es elegido a partir de una variable booleana que se pasa como parámetro. Al hacer eso se mide el número de I/O's y el tiempo demorado.

Después de esto se generan $n/10$ nuevos rectángulos y se realizan búsquedas, y en estas también se obtienen la cantidad de I/O's y el tiempo que se demora en buscar, por cada grupo se obtiene el promedio de estas medidas y su respectiva desviación estándar.

3.3. Medidas de Rendimiento

Las medidas de rendimiento usadas son dos:

- La cantidad de I/O's hechas por una inserción o búsqueda. Estas son simuladas en ambos algoritmos. Es decir, en la búsqueda cada vez que se accede a un nuevo nodo hijo se suma uno a la cantidad total de I/O's, y no se agregan por escrituras ya que solo se lee. Por otro lado, para las inserciones se suma uno a la cantidad total de I/O's cada vez que se baja en un nivel en el árbol. Esto se debe a que hay una nueva lectura de un nodo, además se suma uno cada vez que hay *overflow*, ya que se genera una nueva escritura (generar nuevo nodo hermano). Por lo tanto también se suma uno más extra si hay *overflow* en la raíz, ya que debe haber una nueva escritura al crear un nuevo nodo para la raíz.

- El tiempo de ejecución de cada operación. Para esto se creó la clase *StopWatch*, la que se puede iniciar y pausar. Esta incrementa el tiempo total de la ejecución sin ser molestado por ejecuciones de otras partes del código.

Finalmente estas medidas, tanto de búsqueda como de inserción, son almacenadas en un nuevo archivo. Creado con el objetivo de almacenar los datos y luego analizarlos.

4. Presentación de Resultados

A continuación se presenta en una tabla los resultados del experimento de inserción de rectángulos aleatorios en un R-Tree con el método de *QuadraticSplit*:

Tamaño (2^n)	**Cantidad I/O's**	**Tiempo (milisegundos)**
9	755	43
12	10656	115
15	118832	1043
18	1193602	9624
21	11553820	84628
24	141142799	476308

Los gráficos de ambos datos, en escala logarítmica para su mejor visualización se encuentran en la Figura **??** y **??**.

Ahora se presenta en una tabla los resultados del experimento de inserción de rectángulos aleatorios en un R-Tree con el método de *LinearSplit*:

Tamaño (2^n)	**Cantidad I/O's**	**Tiempo (milisegundos)**
9	740	23
12	10471	36
15	113245	451
18	1192441	4683
21	11118402	46764
24	136824746	252177

Los gráficos de ambos datos, en escala logarítmica para su mejor visualización se encuentran en la Figura **??** y **??**.

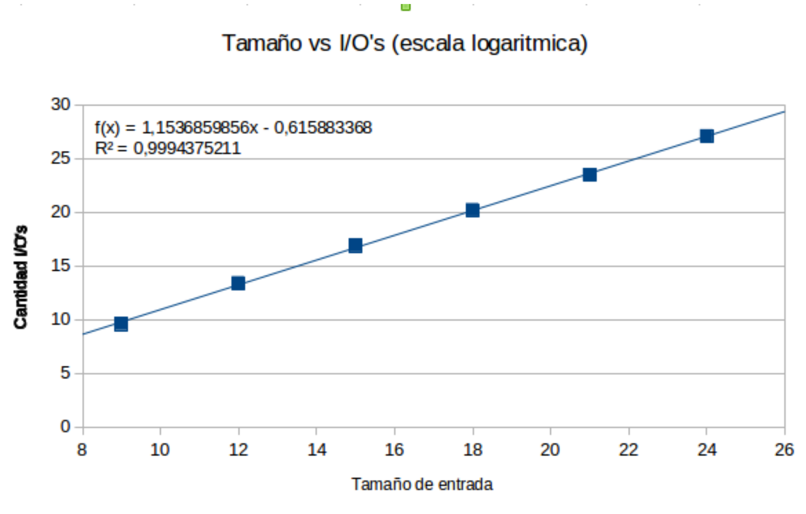

Figura 1: Gráfico Tamaño vs I/O's para QuadraticSplit

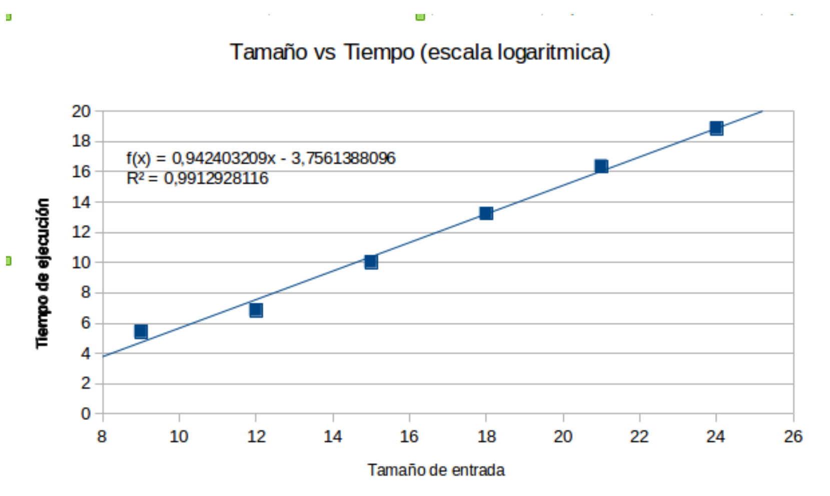

Figura 2: Gráfico Tamaño vs Tiempo para QuadraticSplit

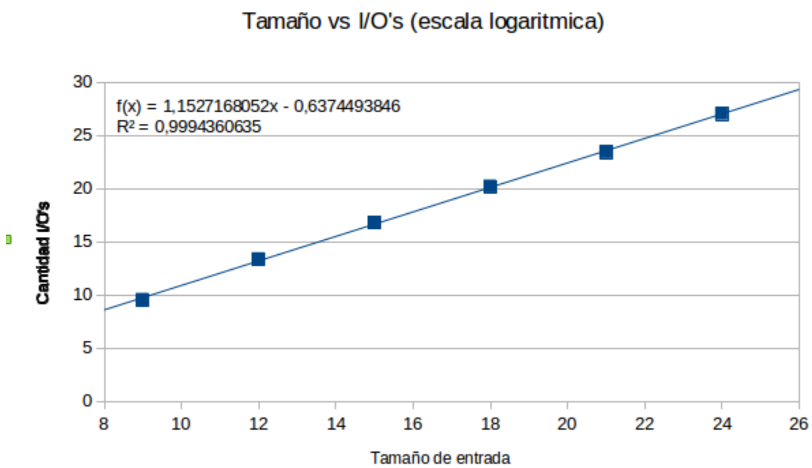

Figura 3: Gráfico Tamaño vs I/O's para LinearSplit

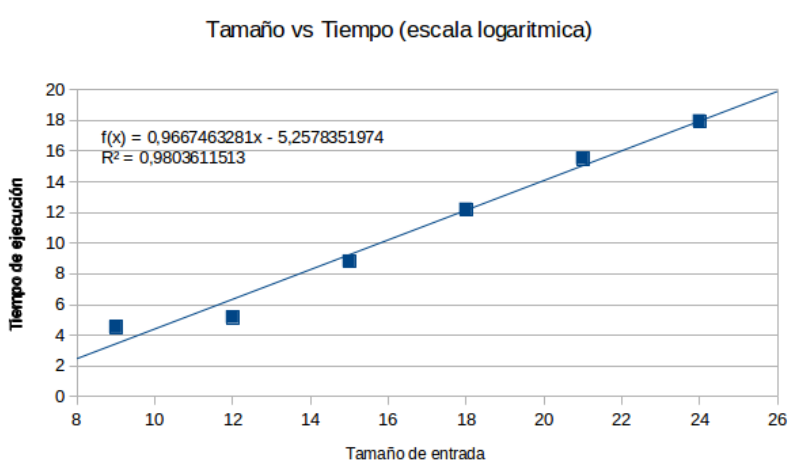

Figura 4: Gráfico Tamaño vs Tiempo para LinearSplit

5. Análisis e Interpretación