Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2007

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 2

Integrante	LU	Correo electrónico
Blanco, Matias	508/05	matiasblanco18@gmail.com
Freijo, Diego	4/05	giga.freijo@gmail.com
Guebel, Hernan	??/??	hguebel@gmail.com
Rogani, Marcos	??/??	marcos.rogani@gmail.com

Palabras Clave

Red-Black Trees, Interval Trees.

Índice

1. Introducción

Lo primero que se pensó fue en como diseñar e implementar los dos tipos de árboles propuestos en el enunciado. La idea fue utilizar el interval tree (árbol de intervalos) manteniendo el equilibrio de sus nodos mediante el red-black tree (árbol rojo-negro) para obtener una complejidad eficiente, aún en el peor caso.

El red-black se diseño sobre punteros. Cada árbol apunta a dos subárboles. En realidad la información se encuentra sobre nodos, pero solamente para mantenerla de forma más sencilla. Los algoritmos se basaron en los dados en clase, realizando las debidas rotaciones cuando, durante cierta inserción, se requiera para mantener el invariante de la estructura.

El interval contiene los algoritmos correspondientes para la inserción y la búsqueda de valores, pero en realidad la información está guardada en un red-black. En cada nodo del árbol tenemos guardadas 2 listas, con los mismos intervalos, pero ordenadas de diferente manera. Una lista est ordena de mayor a menor por el valor de inicio de cada intervalo. La otra lista esta ordena, también de mayor a menor, pero por el valor de fin de cada intervalo. Con una sola lista no alcanza ya que dependiendo el caso, necesito buscar por el valor de inicio o por el valor de fin.

Al realizar una búsqueda y tener la listas ordendas se reduce la cantidad de operaciones para obtener los intervalos deseados(los que contienen al punto dado) ya que en caso de recorrer y tener que devolver una cantidad de intervalos menor a la total no me hace falta recorrerla hasta el final, sino que al ir comparando el valor de inicio o fin, según corresponda, de cada intervalo y al encontrar un valor menor o mayor, tambien según corresponda, puedo devolver los intervalos agregados hasta el momento a la solución y salir de la busqueda sin necesidad de seguir recorriendo porque ya se que mas adelante no voy a encontrar ningún intervalo que contenga al punto dado. Si la lista estaría desordenada si o si debería recorrer todos los intervalos ya que en ninún momento puedo predecir el valor de inicio o fin del intervalo siguiente.

El objetivo es dado un punto x encontrar todos los intervalos que contienen a ese punto. Para esto, la idea del algoritmo consiste en ir comparando el valor x buscado con la clave de la raz del árbol:

- Si el valor de la clave es menor al de x agrego a la solucion los intervalos del nodo que terminan después de x utilizando la lista ordenada de mayor a menor por el valor fin. Utilizo esta lista porque ya se que todos los intervalos del nodo arrancan antes de x y al estar ordenada tendré que hacer k+1 comparaciones (o hasta la cantidad de intervalos) para obtener los k intervalos que contienen a ese punto. Luego de agregar los intervalos, si la rama derecha no es nula continuo con el algoritmo, tomando como el subarbol izquierdo ya que del lado izquierdo estan los intervalos que que finalizan despues del valor de la raiz y x es mayor a la raiz. Si es nula salgo.
- Si el valor de la clave es mayor al de x agrego a la solucion los intervalos del nodo que empiezan antes de x utilizando la lista ordenada de mayor a menor por el valor inicio. Utilizo esta lista porque ya se que todos los intervalos del nodo terminan despus de x, y al igual que en el otro caso, al estar ordenada tendré que hacer k+1 comparaciones (o hasta la cantidad de intervalos) para

obtener los k intervalos que contienen a ese punto. Luego de agregar los intervalos, si la rama izquierda no es nula continuo con el algoritmo, tomando el subrbol derecho ya que del lado derecho estan los intervalos que empiezan antes del valor de la raiz y x es menor al valor de la raiz. Si es nula salgo.

- Si el valor buscado es igual al de la clave, agrego a la solucion todos los intervalos del valor del nodo y si la rama izquierda no es nula contino con el algoritmo, tomando el subarbol izquierdo. Si es nula salgo.

El programa principal mantiene dos árboles, cargando en uno la información horizontal (el intervalo $[x_i, x_f]$) y en el otro la vertical (el intervalo $[y_i, y_f]$) de cada imágen. A la hora de realizar la consulta de un punto $p = (p_x, p_y)$, se busca el valor p_x en el árbol de intervalos horizontales y el p_y, en el de verticales. Cada búsqueda devolverá una lista de intervalos a los cuáles pertenece la respectiva componente de p. Notar que, para que una imágen sea considerada como resultado, deberá suceder que p pertenezca a cada intervalo. Es decir, ambos intervalos de ésta deberán estar en las listas devueltas por el interval.

El primer algoritmo pensado para armar la lista de imágenes fue el de armar una lista con las imágenes que tienen los intervalos (horizontales) donde estába p_x. Luego ésta lista se la filtraría con los intervalos (verticales) donde estaba p_y, obteniéndose la lista de imágenes deseadas. Como en el peor caso se debería recorrer toda la lista de imágenes para cada intervalo horizontal (se supone que la cantidad de ellas es n), la complejidad de éste algoritmo sería $O(nk_x + k_xk_y)$, donde k_x es la cantidad de intervalos horizontales satisfactorios y k_y la de intervalos verticales satisfactorios[1].

Una segunda idea fue de asignar un valor de identificación a cada intervalo (llamado *id*) de manera que permita ubicar sencillamente a cual imágen pertenece cada intervalo. Cuando se realiza la división de una imágen en sus dos intervalos, éstos recibirán el mismo id, y será la posición de la imágen actual en la lista imágenes de entrada. De ésta forma, al tener ambas listas solamente resta comparar para cada elemento de una lista si el mismo id se encuenta en la otra. De así serlo, se agrega la imágen a la lista de resultados (lo cual se hace en tiempo constante por ya tener la posición en la lista de la misma). Por lo tanto el algoritmo sería $O(k_xk_y)$. Se puede ver que, como en el peor caso $k_x = k_y = n$, las complejidades de ambos algoritmos serían iguales (cuadrática en función de la cantidad de imágenes) pero no nos quedamos del todo satisfechos. Lo implementamos y, cuando tuvimos los gráficos listos, vimos que éste algotimo era un gran cuello de botella. La cantidad de operaciones para una consulta aumentaba considerablemente y hasta terminaba siendo peor que por fuerza bruta. Por lo tanto, nos pusimos a pensar otra alternativa.

La tercer y última opción consistió en agregar otro tipo de información adicional a las clases ya existentes. Al intervalo se le agrego una referencia a su *padre* (es decir, a la imagen que contenia ése inervalo), y a la imágen se le agregaron dos bits que marcan si fueron seleccionados en x e y. La idea del algoritmo es recorrer ambas listas de intervalos en donde se generan las intersecciones e ir marcando a sus padres el correspondiente bit. Luego se realiza una pasada por la lista de imágenes en busca de aquellas donde estén tildados en los 2 bits. Éstas son agregadas a la lista de resultados. Notar que, a su vez, mientras se recorren las imágenes se van destildando los bits para que la estructura pueda volver a ser utilizada en la próxima consulta.

[1] Notar que éstas cantidades podrían ser n en el peor de los casos.

Como en el peor caso la cantidad de intervalos intersecados es n, la complejidad del algoritmo sería $O(3n) = O(n)$. Dado que es parecida a la del resto de los algoritmos utilizados en la búsqueda, se lo terminó elijiendo.

Notar que de ésta forma las clases Intervalo e Imagen terminan teniendo cierto sabor a *ad-hoc*, ya que se les agregan valores que carecen de sentido fuera del contexto que debe tener (el representar un intervalo como estructura matemática y una imágen como una fotografía). Igualmente fue preferida ya que

- Mejoraba notablemente la complejidad del algoritmo.
- No nos pusimos en una posición tan *purista*.

2. Detalles de implementación

- En BuscarInterseccion, la lista de imágenes satisfactorias se alojan en una LinkedList ya que tienen inserción en tiempo constante.
-
-

3. Pseudocódigos

```
1: elem ← newNodo2()
2: elem.construir(k, valor)
3: y ← null
4: x ← this.root
5: while x != null do
6:     y ← x
7:     if k < x.key then
8:         x ← x.izq
9:     else
10:        x ← x.der
11:    end if
12: end while
13: elem.padre ← y
14: if y < null then
15:    this.root ← elem
16: else if k < y.key then
17:    y.izq ← elem
18: else
19:    y.der ← elem
20: end if
21: this.tamanio++
22: arreglarInv(elem)
```

BuscarInterseccion(*valorBusq*): Devuelve la lista de intervalos que contienen al valor buscado $\longrightarrow O(log(n) + k)$

```
1: while !salir do
2:     if nodo.key < valorBusq then
3:         agrego a la solucion los intervalos que terminan despues de valorBusq
4:         if derecha no es null then
5:             voy para la derecha
6:         else
7:             salir
8:         end if
9:     else if nodo.key > valorBusq then
10:        agrego a la solucion los intervalos que empiezan antes de valorBusq
11:        if izquierda no es null then
12:            voy para la izquierda
13:        else
14:            salir
15:        end if
16:    else
17:        agrego todos los intervalos a la solucion
18:        if derecha no es null then
19:            voy para la derecha
20:        else
21:            salir
```

```
22:             end if
23:         end if
24: end while
```

dameTermDespues(*nodo*, *valor*): Devuelve una lista con los intervalos del nodo que terminan despues de valor, utilizando la lista ordenada por el valor fin de mayor a menor $\longrightarrow$ O($+k$)

```
1: intervalos ← nodo.fin
2: i ← 0
3: while i ¡cantidad(intervalos) ∧ intervalos(i).fin >= valor  do
4:     agregar intervalo a la solucion
5:     i++
6: end while
```

dameEmpAntes(*nodo*, *valor*): Devuelve una lista con los intervalos del nodo que empiezan antes de valor, utilizando la lista ordenada por el valor inicio de mayor a menor $\longrightarrow$ O($+k$)

```
1: intervalos ← nodo.inicio
2: i ← cantidad(intervalos) − 1
3: while i ¿0 ∧ intervalos(i).inicio <= valor  do
4:     agregar intervalo a la solucion
5:     i−−
6: end while
```

Fusionar($p, arbol_x, arbol_y, imagenes$): En realidad llamado BuscarInterseccion, devuelve las imágenes en *imagenes* que contienen al punto p utilizando los árboles $\longrightarrow$ O(n)

```
1: intersecciones_x = arbol_x.BuscarInterseccion(p)
2: intersecciones_y = arbol_y.BuscarInterseccion(p)
3: for all int in intersecciones_x do
4:     marcar al padre de int en x
5: end for
6: for all int in intersecciones_y do
7:     marcar al padre de int en y
8: end for
9: for all img in imagenes do
10:     if img está marcado en x e y then
11:         resultados.agregar(img)
12:     end if
13:     limpiar marcas de img
14: end for
15: return resultados
```

4. Análisis de complejidad

4.1. BuscarInterseccion

4.1.1. Temporal

O($log(n) + k$)

Siendo n la cantidad de nodos totales del árbol de intervalos y k la cantidad de intervalos obtenidos en la búsqueda.

En realidad en cada nodo, como se explicó anteriormente, si la cantidad de intervalos obtenidos es menor a la cantidad de intervalos de la lista se realizan $k + 1$ comparaciones pero la constante 1 es despreciable. Si se devuelven todos los intervalos se realizan k comparaciones.

$log(n)$ ya que es un árbol balanceado y lo recorro desde la raiz hasta alguna hoja.

4.1.2. Espacial

O(n)

Ya que cada segmento es guardado en dos listas y el árbol es balanceado.

4.2. Fusionar

4.2.1. Temporal

Como se explicó anteriormente en la introducción del presente, en el peor de los casos éste algoritmo deberá recorrer 3 veces todas las imágenes (caso en el que el punto esté contendio en toda imágen de la pantalla). Por lo que la complejidad será O($3n$) = O(n).

4.2.2. Espacial

El algoritmo sólo requiere (sin contar el espacio para la lista de resultados) los bits de seleccionado en las imágenes y las referencias a los padres en cada intervalo. Por lo que se requieren cantidad de números del órden de la cantidad de imágenes, siendo la complejidad O(n).

5. Resultados

Los casos de prueba fueron generados automáticamente con valores azarosos:

- Cantidad de instancias: 1000
- Ancho de la pantalla: 400
- Alto de la pantalla: 400
- Cantidad máxima de imágenes: 300
- Cantidad de consultas: 5

En el directorio *in* del proyecto se pueden encontrar los correspondientes archivos de entrada.

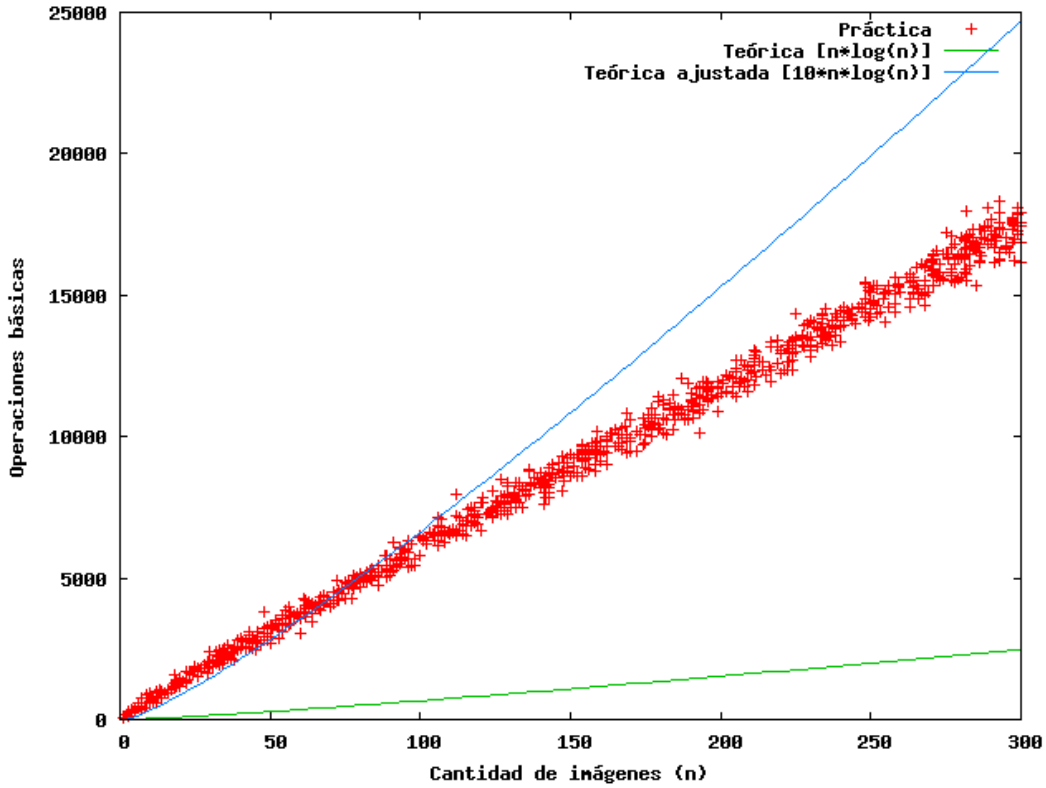

Figura 1: Cantidad de operaciones básicas para el armado en función de cantidad de imágenes

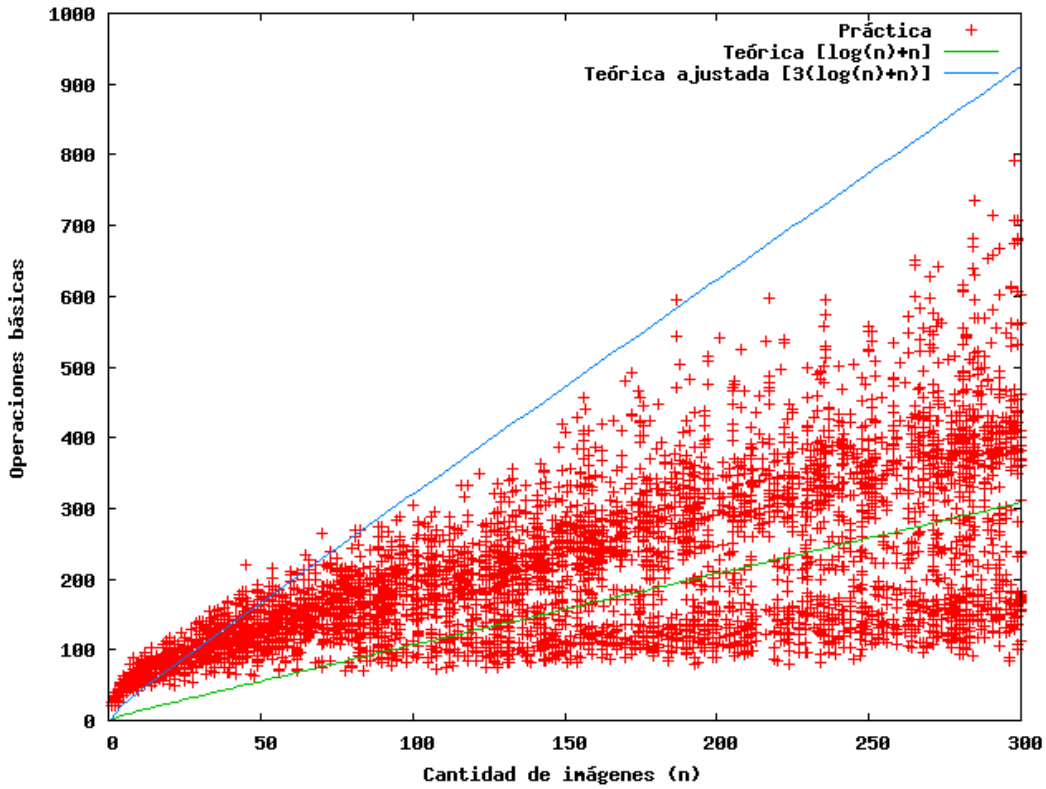

Figura 2: Cantidad de operaciones básicas para una consulta en función de cantidad de imágenes

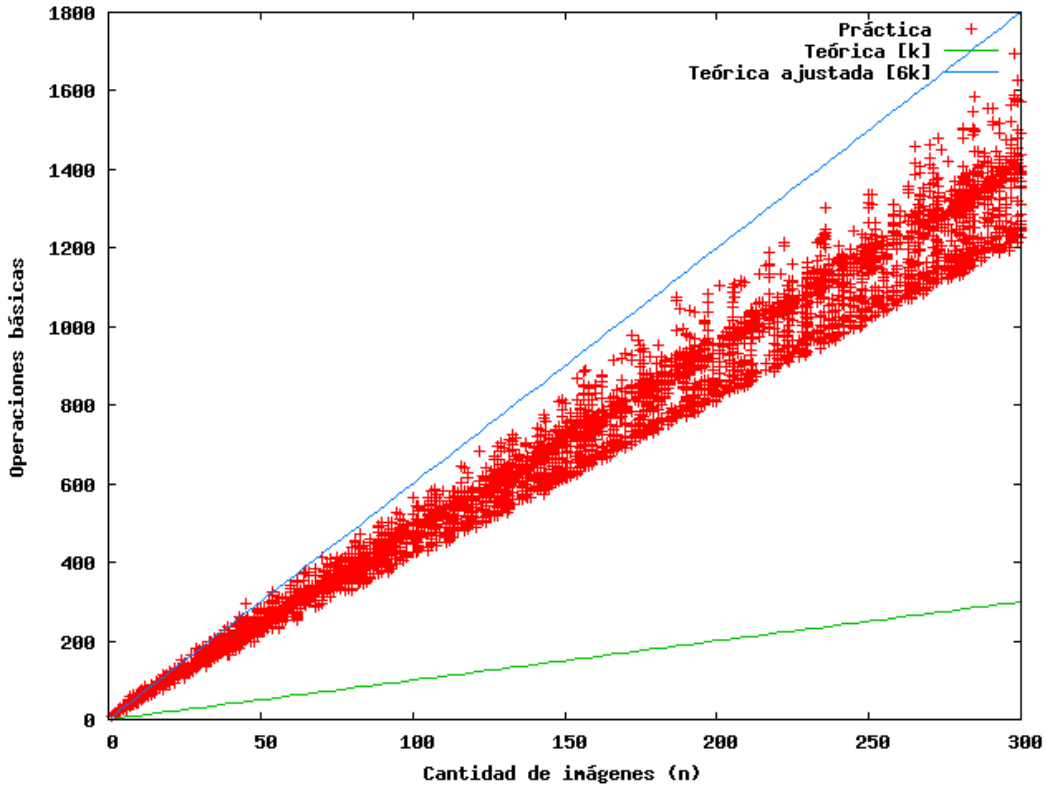

Figura 3: Cantidad de operaciones básicas para la fusión (elegir la imagen a partir de los intervalos) en función de cantidad de imágenes

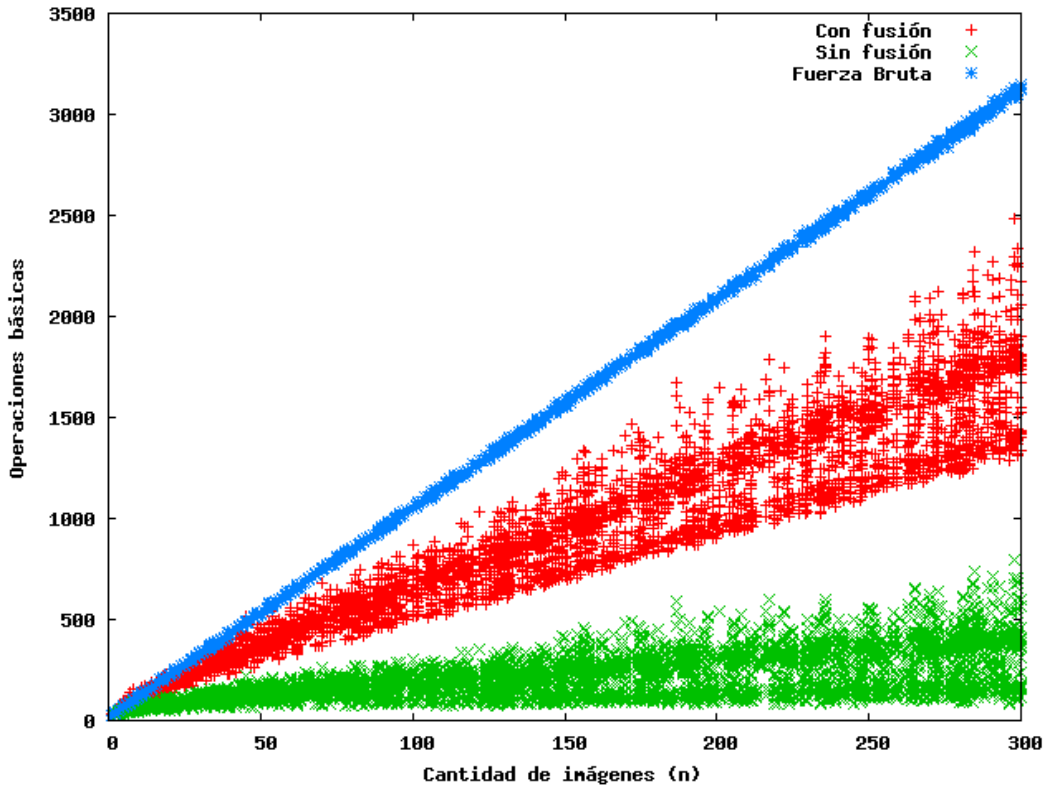

Figura 4: Cantidad de operaciones básicas para una consulta por fuerza bruta, por nuestro algoritmo sin realizar la fusión y por el mismo pero realizándola en función de cantidad de imágenes

6. Discusión

El primer gráfico muestra la cantidad de operaciones básicas necesarias para armar los árboles en función de la cantidad de imágenes. Como se puede ver, la complejidad teórica es correcta. La curva recién se logró ajustar con una constante de 10, lo cual indica que las constantes en la complejidad no son menores y, dependiendo del uso que se le dará al algoritmo, deberían tomarse con cuidado.

El segundo cuenta las operaciones básicas para realizar una consulta. Notar que por consulta se entiende a la búsqueda en los árboles, pero no a la fusión (es decir, a obtener la imagen a partir de los intervalos). La complejidad teórica es válida como cota superior. Notar que los valores se encuentran notablemente distribuídos, sin formar una figura clara. Ésto se debe a que, dependiendo del punto que se pida buscar y de la distribución de los árboles, no siempre se mantiene una complejidad *pareja*.

En el tercero vemos que el algoritmo de fusión también representa una complejidad teórica correcta. Se puede ver que existe una línea como valor mínimo de la complejidad. Éstos valores corresponden a los mejores casos, donde pocos intervalos son devueltos por los árboles. Pero como siempre, aún en éstos casos, se deberá recorrer toda la lista de imágenes, siempre existirá ésa cota mínima.

El cuarto gráfico intenta demostrar dos cosas. Por un lado, notar que el algoritmo de fusión es realmente un cuello de botella ya que si se cuentan solamente la cantidad de operaciones que requieren los árboles para devolver los intervalos de intersección, éstas son mucho menores que cuando se agrega éste último algoritmo. Pero por otro lado se vé que, a pesar de lo costoso que es la fusión, la complejidad total es menor que la de fuerza bruta. Por lo que las estructuras y algoritmos implementados podríamos considerarlos *eficientes*. Notar que el mismo patrón notado en el gráfico anterior de la cota mínima se aplica aquí, ya que se suman los mismos valores a los del costo de la consulta.

7. Referencias

Referencias

[1] Algoritmo de Strassen para la multiplicación de matrices cuadradas: *http://en.wikipedia.org/wiki/Strassen_algorithm*