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	135/05	hguebel@gmail.com
Rogani, Marcos	520/05	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.

Para la elaboracion del Interval, primero se buscaron datos del mismo para complementar los entregados en el apunte dado por la catedra. Luego de revisar varios papers, se decidio seguir por la implementacion del pseudo-codigo suministrado por la Wikipedia, previa comparacion con los demas.

El arbol de intervalos es una estructura ordenada de datos que contiene intervalos. Sirve, principalmente, para poder encontrar, en una lista de intervalos, aquellos que se solapan con un punto dado o con un rango dado. Este tipo de arboles se utilizan por sobre, por ejemplo, los arboles binarios ordinarios, debido a que, en los ultimos, el costo de una consulta sobre el arbol es del orden de $O(n)$, y en un arbol de intervalos, el orden obtenido es de $O(log_2(n) + m$ (Se explica con mas detalle en la seccion Çalculos de complejidad).

Este arbol se implemento sobre un Arbol Red-Black, para que siempre quede balanceado, ante cada insercion en el mismo. Tambien se podria haber usado un arbol binario o un AVL, pero se prefirio el RB ya que su comportamiento ante la insercion es mas estable que el AVL.

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.

Se procedio a implementar la estructura de la siguiente forma:

Para la creacion, el articulo sugiere tomar el rango total de los intervalos y dividirlo por la mitad, para tomar el elemento medio como pivot de la primera iteracion. Una vez hecho esto, se prodece a dividir la lista de intervalos en 3: aquellos que estan completamente a la izquierda del pivot, aquellos que lo contienen y por ultimo, aquellos que estan posterior al mismo. Una vez hecho esto, se inserta un nodo con el pivot como clave y como dato la lista generada que contiene a ese punto, pero esta ultima aparece en dos oportunidades. La primera, ordenando a los intervalos segun su punto de inicio, y la segunda, ordenandolos segun su punto de fin. Luego, se procede a realizar el algoritmo, pero ya con las dos listas restantes.

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. 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 en un principio. Pero ocurría una contradicción de base: la complejidad del algoritmo de fuerza bruta es también lineal en función de la cantidad de imágenes (ya que debe ir comparando una por una buscando donde los inervalos contengan al punto) y nuestro algoritmo terminaría teniendo una complejidad $\Omega(n)$. Y para eso podríamos haber implementado éste directamente que, total, es más sencillo. Por lo tanto se ideó una especie de *poda* de casos que no valían la pensa revisar, en el algoritmo que ya teníamos.

Ésta fue pensada haciendo que en la primer pasada, en lugar de marcar en x las imágenes cullos intervalos horizontales se intersectaban al punto, se agregaron las imágenes padre a una lista provisoria. Luego se marcaron normalmente los que intersectaban sobre y. Notar que en la lista provisoria hay una referencia a las imágenes, por lo que los últimos cambios influyeron sobre los que ya se encontraban allí. Ahora, en lugar de revisar la lista completa de imágenes sólo se debe revisar la provisoria buscando en donde existe la marca en y, y éstos son agregados. Luego se deberá hacer un recorrido por las intersecciones en y para volver a resetear la marca de todos los padres que fueron alterados (si ésto se hace en el bucle final como en el algoritmo anterior, podrían quedar algunas imágenes que fueron marcadas en y sin resetear, porque éstas podrían no haber sido agregadas a la lista provisoria de las marcas en

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

x). Notar que en el peor caso (aquel donde todas las imágenes intersectan horizontalmente con el punto) la complejidad sigue siendo la misma: $O(n)$, pero en caso promedio éste valor es difícilmente alcanzado[2] , con lo que nos quedamos tranquilos que la implementación de los árboles no fue en vano.

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]Para más información, dirigirse a la sección de resultados.

2. Pseudocódigos

Insertar(k, $valor$): Inserta un nodo en el arbol (como si fuera un árbol binario de búsqueda) y luego reestablece el invariante del RedBlackTree $\longrightarrow$ O($log(n)$)

```
1: elem.construir(k, valor)
2: y ← null
3: x ← this.root
4: while x != null do
5:     y ← x
6:     if k < x.key then
7:         x ← x.izq
8:     else
9:         x ← x.der
10:    end if
11: end while
12: elem.padre ← y
13: if y < null then
14:    this.root ← elem
15: else if k < y.key then
16:    y.izq ← elem
17: else
18:    y.der ← elem
19: end if
20: incrementoeltamaodelarbolen1
21: reestablecerinvariante
```

Constructor($intervalos$): Crea el arbol de intervalos a partir de una lista de ellos $\longrightarrow$ O($nlog(n)$)

```
1: pivot ← (inicioRango − finRango)/2 + inicioRango
2: while i < intervalos.tam do
3:     if intervalos[i] es anterior a pivot then
4:         anteriores.agregar(intervalos[i])
5:     else if intervalos[i] es posterior a pivot then
6:         posteriores.agregar(intervalos[i])
7:     else
8:         centro.agregar(intervalos[i])
9:     end if
10: end while
11: nodo ← crearNodo(centro, pivot)
12: InsertarenRB(nodo)
13: InsertarTodosenRB(Constructor(izquierda))
14: InsertarTodosenRB(Constructor(derecha))
```

La creación del Interval Tree es de la siguiente forma:

Se calcula el rango de las fotos, para saber el menor inicio y mayor final. Una vez calculado esto, se procede a tomar un punto cercano al medio. Con este punto como

pivot, se generan tres listas. Una con aquellos intervalos que contienen al pivot, llamada centro; otra con los intervalos que estan completamente a la izquierda, llamada izquierda; y por ultimo otra con los que estan a la derecha del punto, llamada derecha.

Para ver a que lista pertenece, se utilizan las funciones Contiene, Anterior y Posterior. La primera verifica si el pivot esta entre el inicio y el fin del intervalo. La segunda, si el final del intervalo es menor al punto, entonces, esta a la izquierda del mismo. Y la tercera, si el inicio es mayor al punto, el intervalo, entonces, es posterior. Una vez generadas estas listas, se crea el nodo con: el pivot utilizado para comparar y la lista Centro.

El nodo genera internamente sus miembros: el pivot se mantiene, la lista Inicio es generada ordenando a la lista Centro por sus inicios, y la lista Fin ordenando a la misma lista, pero por sus finales. Al estar creado el nodo, se inserta en una lista.

Para ordenar se utiliza el algoritmo QuickSort, al ser un algoritmo eficiente en caso promedio. Se eligio sobre otras opciones como el SelectionSort, BubbleSort o InsertionSort, ya que estos son peores en la practica al elegido.

El procedimiento se vuelve a repetir con la listas Izquierda y Derecha, agregando los nodos a la misma lista.

Al terminar este proceso recursivo, se agregan los nodos al arbol RB.

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: $intervalos \leftarrow nodo.fin$
2: $i \leftarrow 0$
3: **while** i ¡cantidad(intervalos) $\wedge$ 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: $intervalos \leftarrow nodo.inicio$
2: $i \leftarrow cantidad(intervalos) - 1$
3: **while** i ¿0 $\wedge$ intervalos(i).inicio <= valor **do**
4: agregar intervalo a la solucion
5: $i--$
6: **end while**

Fusionar_SinPoda($p, arbol_x, arbol_y, imagenes$): Devuelve las imágenes en *imagenes* que contienen al punto p utilizando los árboles $\longrightarrow$ $\Theta(n)$

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*

Fusionar_ConPoda($p, arbol_x, arbol_y$): Devuelve las imágenes 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: $imagenes_x$.agregar(int.padre)
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** $imagenes_x$ **do**
10: **if** img está marcado en y **then**
11: $resultados$.agregar(img)
12: **end if**
13: **end for**
14: **for all** int **in** $intersecciones_y$ **do**
15: resetear al padre de int en y
16: **end for**
17: **return** $resultados$

2.1. Respuestas

3. Análisis de complejidad

3.1. Creación

3.1.1. Temporal

El peor caso para la creación del Interval es que todos los intervalos sean disjuntos, lo que provocaría que en la lista Centro de cada nodo, haya solo un intervalo. Vamos a tomar este caso para analizar la complejidad.

La complejidad de las funciones Anterior, Posterior y Contiene es O(1), ya que solo realiza una comparación y devuelve true o false.

La complejidad de BuscoRango es O(n), ya que recorre todos los intervalos para encontrar el menor inicio y el mayor fin. Dentro del while, las operaciones son O(1).

Dentro del algoritmo del constructor, se llama una vez a la funcion BuscoRango y se repiten n veces las funciones Anterior, Posterior y Contiene. Luego, se repite recursivamente $\log_2(n)$ veces. Por lo tanto la complejidad es del orden $n \log_2(n)$.

El tamao de la entrada se calcula de la siguiente manera. La entrada contiene la cantidad de fotos, n, y sus coordenadas. Los valores de las coordenadas estan acotadas por el tamao de la pantalla, lo que hace que sea constante, por lo cual, el tamao de la entrada queda $\log_2(n) + n$, que es del orden de n.

Por lo cual, la complejidad de la creación queda O($\log_2(n) + n$).

3.1.2. Espacial

La complejidad espacial de la creación del Interval Tree es la siguiente: en cada nodo se guardan 2 enteros y 2 listas iguales de enteros con la cantidad de fotos que tocan a ese punto. Esto se repite, en peor caso, n veces. Por lo tanto, el orden de complejidad del algoritmo es O(n).

3.2. BuscarInterseccion

3.2.1. 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.

3.2.2. Espacial

$O(n)$

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

3.3. Fusionar

3.3.1. 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 $O(3n) = O(n)$. Notar que siempre se recorren todas las imágenes, por lo que también es $\Omega(n)$, arrojando $\Theta(n)$. En cambio, la poda puede llegar a ser $O(n)$, pero no tiene cota inferior definida (excepto la trivial de 1).

3.3.2. 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 $O(n)$. Notar que en la versión con poda requiere además de la lista provisoria de imágenes en x, que suma n a la complejidad por lo que no varía. Pero tiene de ventaja (aunque no sea demasiada) que cada imágen no requiere el bit de seleccionado en x.

4. 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

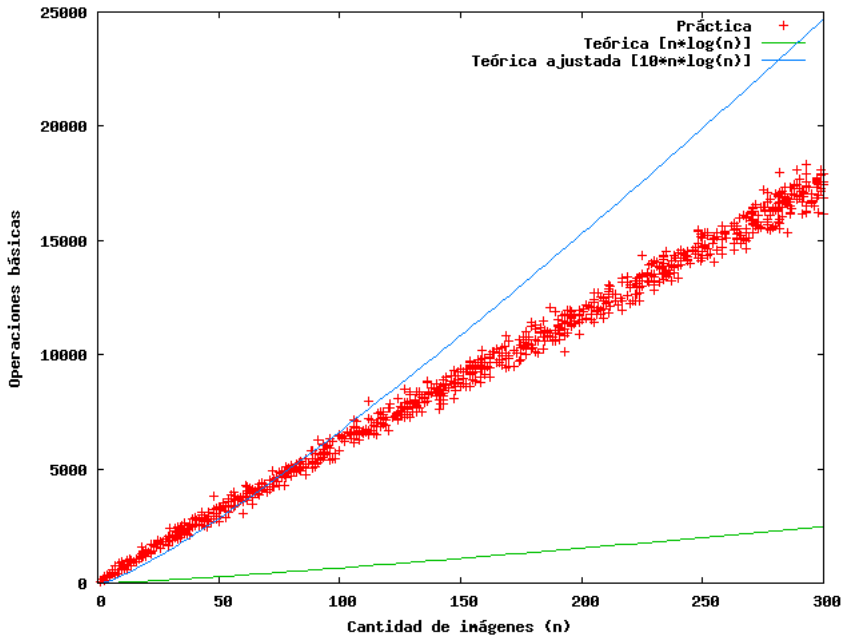

Figura 1: Cantidad de operaciones básicas para el armado en función de cantidad de imágenes

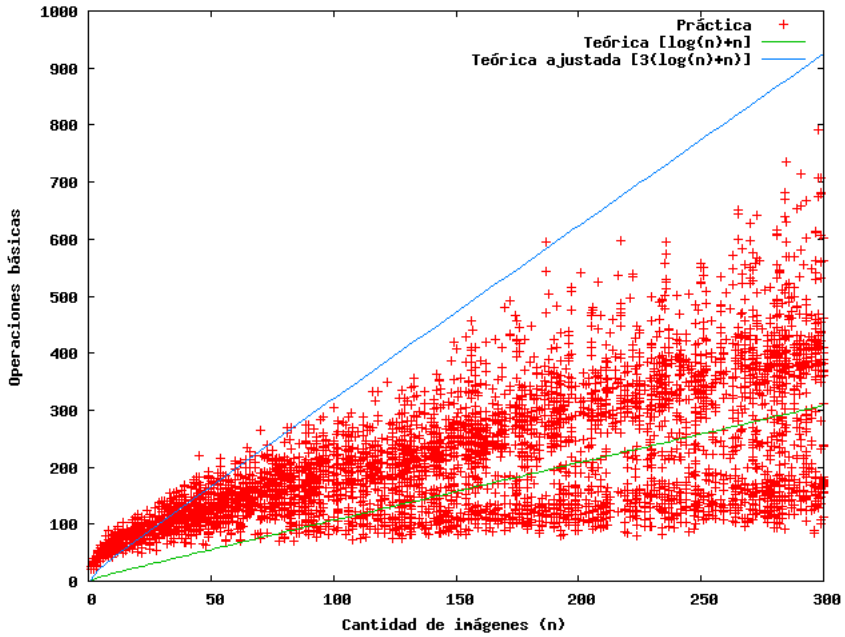

Figura 2: Cantidad de operaciones básicas para una consulta en función de cantidad de imágenes

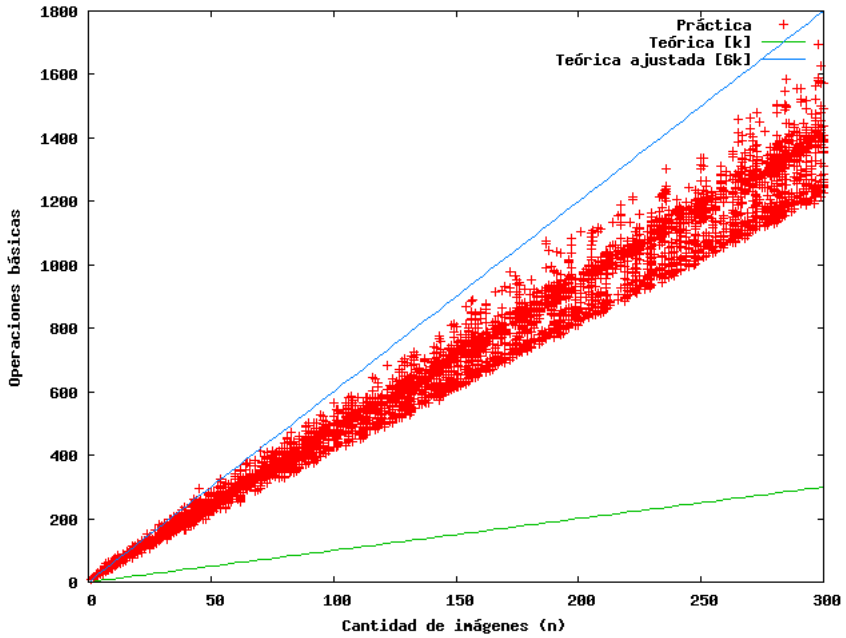

Figura 3: Cantidad de operaciones básicas para la fusión sin poda (elegir la imagen a partir de los intervalos) en función de cantidad de imágenes

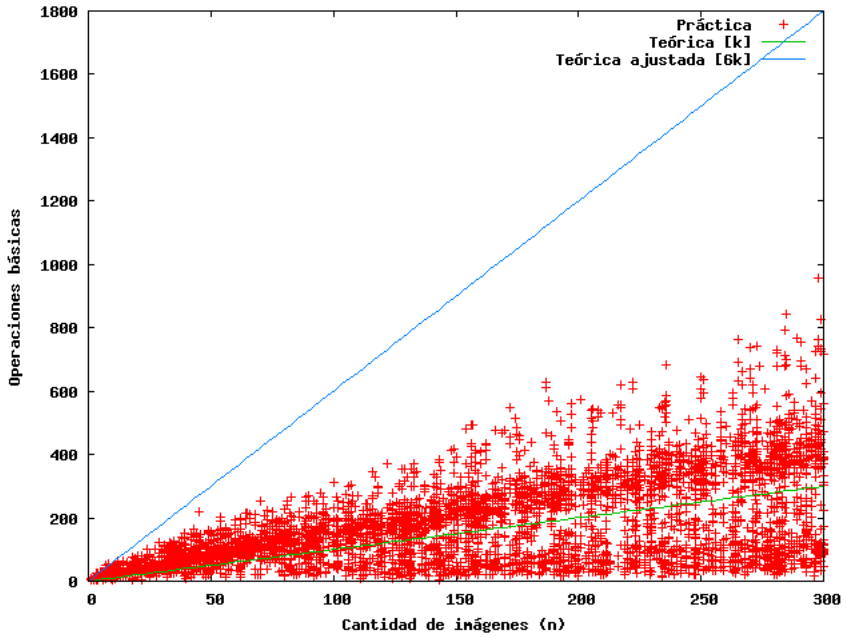

Figura 4: Cantidad de operaciones básicas para la fusión con poda (elegir la imagen a partir de los intervalos) en función de cantidad de imágenes

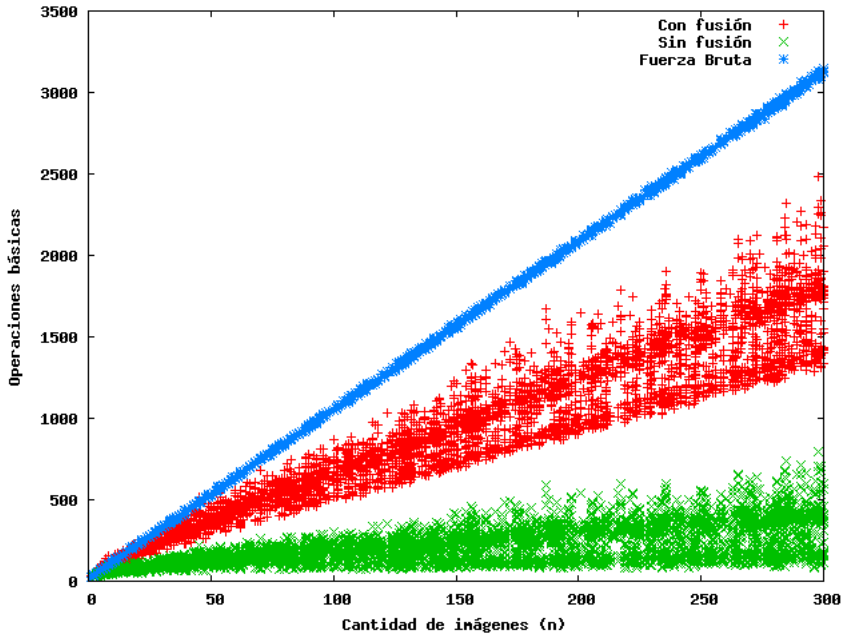

Figura 5: Cantidad de operaciones básicas para una consulta por fuerza bruta, por nuestro algoritmo sin realizar la fusión y por el mismo pero realizándola (sin poda) en función de cantidad de imágenes

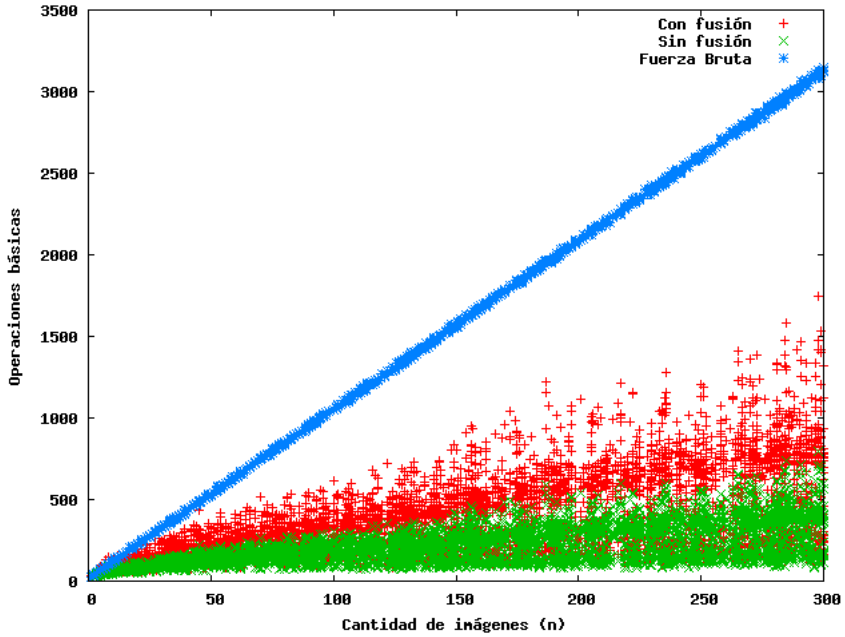

Figura 6: Cantidad de operaciones básicas para una consulta por fuerza bruta, por nuestro algoritmo sin realizar la fusión y por el mismo pero realizándola (con poda) en función de cantidad de imágenes

5. 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 (sin poda) 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. En cambio, en el cuarto gráfico se ve que la poda en éste algoritmo dá notables resultados, realizando una cantidad de operaciones mucho menor. Notar que aquí los valores se encuentran más dispersos. Y es de esperar ya que éste algoritmo no tiene ninguna cota inferior bien definida (a excepción de 1), a diferencia del primero donde se debían recorrer todas las imágenes obligatoriamente, siendo $\Omega(n)$.

El quinto 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. En el sexto gráfico se muestran los mismos resultados con el algoritmo de poda. Como era de esperarse por las figuras 3 y 4, la complejidad total es menor en éste que en el anterior. Además se aprecia el mismo comportamiento en los datos al estar muy dispersos ya que se apoya en forma directa en la complejidad graficada anteriormente.

6. Respuestas

En este caso, se analiza como seria el programa, si se agregasen fotos al mapa.

El problema que se presenta es que el Interval Tree esta pensado para, una vez creado, no ser modificado. Al intentar ingresar un intervalo nuevo, este podria agrandar el rango, y asi no quedaria la idea original plasmada, ya que ya no seria el pivot el elemento elegido.

Para resolverlo, se propone, en la funcion "Insertar Foto", crear nuevamente ambos arboles de intervalos, conteniendo los nuevos intervalos, para que se balancee nuevamente y asi no modificar su complejidad.

Otra alternativa, es, por ejemplo en el caso en que la nueva foto no modifique el rango total, ir viendo nodo por nodo para ver en cual se puede insertar, y ahi ponerlo en las listas, ordenandolas nuevamente.

En el primer caso, la complejidad seria la de crear otra vez los dos arboles, o sea $O(n \log_2(n))$.

En el segundo, la complejidad seria similar a la busqueda, recorriendo el arbol, y una vez encontrado el nodo indicado, agregar a las listas y ordenar. En este caso es $O(\log_2(n) + 2n \log_2(n))$. Se remarca que en la parte de ordenar, la lista va a estar practicamente ordenada, lo que achica, en practica, la complejidad del algoritmo.

Una vez realizado esto, la interfaz superior del programa no se modifica, a excepción de agregar la funcion insertar. El resto de las complejidades se mantienen como son.

Hay que remarcar que esta es una variante al Interval Tree, ya que este no esta pensado, en su version standard, para realizar inserciones.

7. Referencias

- http://www.dgp.toronto.edu/people/JamesStewart/378notes/22intervals/
- http://en.wikipedia.org/wiki/Interval_tree