Estructuras de Datos y Algoritmos – IIC2133

I2 - pauta

14 octubre 2015

(***Algoritmo***: Secuencia ordenada y finita de pasos para llevar a cabo una tarea; cada paso es descrito con la precisión suficiente —p.ej., en un pseudo lenguaje de programación similar a C— para que un computador lo pueda ejecutar.)

1. **a**) ¿Es todo árbol AVL un árbol rojo-negro? Justifica.

Sí. Hay justificar que en un árbol AVL la ruta (simple) más larga de la raíz a una hoja no tiene más del doble de nodos que la ruta (simple) más corta de la raíz a una hoja. Esto es así:

- *El árbol AVL más "desbalanceado" es uno en el que todo subárbol derecho es más alto (en uno) que el correspondiente subárbol izquierdo (o vice versa); en tal caso, el número de nodos en la ruta más larga crece según 2h, y el de la ruta más corta, según h+1, en que h es la altura del árbol.*
- *Así, para cualquier ruta (simple) desde la raíz a una hoja, sea* d *la diferencia entre el número de nodos en esa ruta y el número de nodos en la ruta más corta. Si todos los nodos de la ruta más corta son negros, entonces los otros* d *nodos deben ser rojos: pintamos la hoja roja y, de ahí hacia arriba, nodo por medio hasta completar* d *nodos rojos.*

b) ¿Cuántos cambios de color y cuántas rotaciones pueden ocurrir a lo más en una inserción en un árbol rojo-negro? Justifica.

Recuerda que al insertar un nodo, *x*, lo insertamos como una hoja y lo pintamos rojo. Si el padre, *p*, de *x* es negro, terminamos. Si *p* es rojo, hay dos casos: el hermano, *s*, de *p* es negro; *s* es rojo. Si *s* es negro, hacemos algunas rotaciones y algunos cambios de color. Si *s* es rojo, sabemos que el padre, *g*, de *p* y *s* es negro; entonces, cambiamos los colores de *g*, *p* y *s*, y revisamos el color del padre de *g*.

Hacemos O(log n) cambios de color y O(1) rotaciones. Como dice arriba, esto ocurre solo cuando p *es rojo.*

Si s *es negro, hacemos exactamente una o dos rotaciones, dependiendo de si* x *es el hijo izquierdo o el hijo derecho de su padre; y hacemos exactamente dos cambios de color.*

Si s *es rojo, no hacemos rotaciones, solo cambios de color. Inicialmente, cambiamos los colores de tres nodos:* g*, que queda rojo, y sus hijos* p *y* s*, que quedan negros. Como* g *queda rojo, hay que revisar el color de su padre: si es negro, terminamos; si es rojo, repetimos estos últimos cambios de color, pero más "arriba" en el árbol.*

2. **a**) Considera un grafo no direccional. Queremos pintar los vértices del grafo, ya sea de azul o de amarillo, pero de modo que ***dos vértices conectados por una misma arista queden pintados de colores distintos***.

Da un algoritmo eficiente que pinte los vértices del grafo según la regla anterior, o bien que se dé cuenta de que no se puede; justifica la corrección de su algoritmo.

La idea es usar DFS, que es O(V+E):

1) Recordemos que al aplicar DFS a un grafo no direccional sólo aparecen aristas de árbol y hacia atrás; no hay aristas hacia adelante ni cruzadas.

2) Por lo tanto, en el árbol DFS resultante, si a partir de un vértice v *surgen varias ramas, significa que en el grafo original cualquier ruta que va de los vértices en una de esas ramas a los vértices en otra necesariamente pasa por* v.

3) Por lo tanto, a partir de la raíz del árbol DFS podemos pintar los vértices alternando colores a medida que bajamos por cada rama.

4) Finalmente, hay que revisar las aristas hacia atrás del árbol DFS, cuya presencia refleja la existencia de ciclos en el grafo original:

- *Si alguna arista hacia atrás conecta (directamente) vértices pintados del mismo color, entonces el grafo* ***no se puede pintar*** *como se pide en el enunciado.*
- *Pero si el árbol DFS no tiene aristas hacia atrás, o si ninguna arista hacia atrás conecta vértices pintados del mismo color, entonces es posible pintar los vértices del grafo original como se pide en el enunciado; la asignación de colores definida en el paso 3) es una forma concreta de hacerlo.*

b) Considera el siguiente grafo direccional G, representado mediante listas de vértices adyacentes:

$[0] \to 5, 1$	$[4] \to 3, 2$	$[8] \to 7, 9$	$[12] \to 9$
$[1] \to$	$[5] \to 4$	$[9] \to 11, 10$	
$[2] \to 0, 3$	$[6] \to 9, 4, 0$	$[10] \to 12$	
$[3] \to 5, 2$	$[7] \to 6, 8$	$[11] \to 4, 12$	

Determina las componentes fuertemente conectadas de G ***ejecutando el algoritmo estudiado en clase***:

1) Realizamos DFS de G, para calcular los tiempos de finalización, $u.f$, de cada vértice
2) Determinamos G^T
3) Realizamos DFS de G^T, pero en el ciclo principal consideramos los vértices en orden decreciente de $u.f$ calculado antes

El DFS del ***paso 1*** *se puede realizar a partir de cualquier vértice de G; el resultado del paso 1, en términos de los tiempos u.f asignados a cada vértice, va a depender de cuál vértice partimos. Supongamos que partimos de* 0*, luego de* 6*, y finalmente de* 7:

0.d = 1 5.d = 2 4.d = 3 3.d = 4 2.d = 5 2.f = 6 3.f = 7 4.f = 8 5.f = 9 1.d = 10 1.f = 11 0.f = 12	6.d = 13 9.d = 14 11.d = 15 12.d = 16 12.f = 17 11.f = 18 10.d = 19 10.f = 20 9.f = 21 6.f = 22 7.d = 23 8.d = 24 8.f = 25 7.f = 26

El ***paso 2*** *consiste en transponer G, es decir, invertir la dirección de las aristas; vale tanto el dibujo del grafo, como la representación mediante listas de vértices adyacentes*:

[0] → 2, 6	[4] → 5, 6, 11	[8] → 7	[12] → 10, 11
[1] → 0,	[5] → 0, 3	[9] → 6, 8, 12	
[2] → 3, 4	[6] → 7	[10] → 9	
[3] → 2, 4	[7] → 8	[11] → 9	

En el ***paso 3*** *hacemos DFS del grafo transpuesto, pero en el ciclo principal de DFS consideramos los vértices en orden decreciente de u.f calculado en el paso 1, es decir, en este caso:*

partimos con el vértice 7 (7.f = 26)*, a partir del cual sólo podemos llegar al* 8*, de modo que estos dos vértices forman una componente;*

seguimos con el vértice 6 (6.f = 22)*, a partir del cual no podemos ir a ningún otro vértice (excepto* 7*, pero ya fue considerado), de modo que* 6 *forma una componente por sí solo;*

seguimos con 9 (9.f = 21)*, a partir del cual podemos llegar a* 12*,* 11 *y* 10 *(también a* 6*, pero ya lo consideramos), de modo que* 9*,* 10*,* 11 *y* 12 *forman otra componente;*

… similarmente encontramos la componente 0*,* 2*,* 3*,* 4 *y* 5*;*

… y por último, la componente formada sólo por 1.

3. **a**) Da un algoritmo que, dados n enteros en el rango 0 a k, los preprocesa en tiempo $\Theta(n+k)$ y luego responde cualquier consulta acerca de cuántos de los n enteros están en el rango $[a \,..\, b]$ en tiempo $O(1)$.

Suponemos que los n enteros vienen en un arreglo A. *La idea es usar la primera parte de countingSort, es decir, los primeros tres ciclos* `for`*, que son los que arman el arreglo* C:

```
for i = 0 to k: C[i] = 0
for j = 0 to n–1: C[A[j]] = C[A[j]] + 1
for i = 1 to k: C[i] = C[i] + C[i–1]
```

Con esto, el número de enteros en el rango [a .. b] *es* C[b] – C[a–1].

b) Justifica que *insertionSort* toma tiempo $\Omega(n^2)$ en promedio para ordenar n elementos.

insertionSort *ordena por la via de intercambiar la posición de dos elementos adyacentes (es decir, corregir una inversión), repitiendo esta acción todas las veces que sea necesario. La pregunta es, ¿cuántas inversiones hay en promedio en un arreglo de n elementos? Si consideramos el arreglo* $[a_1, a_2, \ldots, a_n]$ *y su inverso,* $[a_n, \ldots, a_2, a_1]$*, dos elementos cualquiera,* a_i *y* a_j*, están desordenados (invertidos) en el primer arreglo o en el segundo; es decir, cada dos arreglos, tenemos todas las inversiones posibles entre n elementos:* $n(n-1)/2 = \Omega(n^2)$.

c) Escribe una versión de mergeSort que tome tiempo $O(n)$ para ordenar un arreglo de n elementos ***que ya está ordenado***. Justifica tu respuesta.

`mergeSort()` *mezcla dos subarreglos ordenados —*r[e .. m] *y* r[m+1 .. w]*— en un tercer subarreglo ordenado —*tmp[e .. w]*— que luego copia de vuelta a* r[e .. w]: *el algoritmo* `merge()`. *Sin embargo, si los elementos de* r[e .. m] *son todos menores que los elementos de* r[m+1 .. w]*, entonces no es necesario realizar la mezcla, ahorrando una cantidad de tiempo de ejecución proporcional a* w – e*, la cantidad total de elementos en los dos subarreglos originales. Para esto, en* `mergeSort()` *basta con probar si* r[m] < r[m+1] *justo antes de llamar a* `merge()`*; la llamada sólo se hace si* r[m] > r[m+1]. *Es decir:*

```
mergeSort(r, tmp, e, w):
   if e < w:
      int m = (e+w)/2
      mergeSort(r, tmp, e, m)
      mergeSort(r, tmp, m+1, w)
      if r[m] > r[m+1]:
         merge(r, tmp, e, m+1, w)
```

Si hacemos este cambio en `mergeSort()`*, entonces, si* `r` *está inicialmente ordenado,* `mergeSort()` *no va a hacer llamadas a* `merge()` *y la ejecución de todo el algoritmo va a tomar un tiempo proporcional a* n: *primero, divide* `r` *hasta formar subarreglos de tamaño 1, y, luego, mezcla pares de subarreglos ordenados, sólo que sin llamar a* `merge()`.