Estructuras de Datos y Algoritmos – IIC2133

I2

18 octubre 2010

1. Dado un grafo direccional *G*, explica cómo construir en tiempo $O(V + E)$ otro grafo direccional, *G'*, que tenga las mismas componentes fuertemente conectadas y el mismo grafo de componentes de *G*, pero que tenga el mínimo número posible de aristas; justifica tu respuesta.

Respuesta:

[**1 pt.**] Observación: Las SCC's son conjuntos de vértices; luego, para que *G'* tenga las mismas SCC's que *G*, *G'* tiene que tener los mismos vértices que *G*, pero no necesariamente las mismas aristas.

[**2.5 pts.**]
[**2 pts.**] Para cada SCC de *G*, en *G'* formamos un ciclo simple con los mismos vértices; así, si la SCC en *G* tiene *k* vértices, en *G'* habrá un ciclo con *k* aristas (el menor número posible de aristas de una SCC de *k* vértices).
[**0.5 pts.**] Identificar las SCC's de *G* toma tiempo $O(V + E)$; y agregar las aristas a *G'* para formar ciclos simples para todas las SCC's toma tiempo $O(V)$.

[**2.5 pts.**]
[**0.5 pts.**] Teniendo las SCC's de *G*, se puede formar el grafo de componentes de *G* en tiempo $O(V + E)$.
[**2 pts.**] Finalmente, para que *G'* tenga el mismo grafo de componentes que *G*, hay que hacer lo siguiente: por cada arista del grafo de componentes de *G* (aristas que unen pares de SCC's), en *G'* hay que agregar una arista que una un vértice (cualquiera) de la primera SCC con un vértice (cualquiera) de la segunda SCC. Esto toma tiempo $O(E)$.

2. Con respecto a los siguientes algoritmos de ordenación, (**a**) ¿cuáles son estables y cómo se sabe que lo son? Para los que no lo son, (**b**) explica cómo se los puede hacer estables y a qué costo. Recuerda que un algoritmo de ordenación es ***estable*** si los datos con igual valor aparecen en el resultado en el mismo orden que tenían al comienzo.

i) Insertionsort. ii) Mergesort. iii) Heapsort. iv) Quicksort.

Respuesta:

a) [**3 pts.**]
[**1 pt.**] *Insertionsort* y *Mergesort* son estables.
[**1 pt.**] *Insertionsort* (diapositiva #19 de los apuntes) es estable porque al comparar las claves de b y $a[j\text{-}1]$, "subimos" (o avanzamos hacia $a[0]$) solo si la clave de b es estrictamente menor que la de $a[j\text{-}1]$.
[**1 pt.**] *Mergesort* (diapositiva # 26 de los apuntes) es estable porque al comparar el $a[p]$, de la primera "mitad", con el $a[q]$, de la segunda mitad, colocamos $a[p]$ en el resultado temporal (*tmp*) solo si su clave es estrictamente menor que la de $a[q]$.

b) [**3 pts.**]
Heapsort y *Quicksort* no son estables.
[**2 pts.**] Una forma de hacerlos estables es guardar el índice de cada elemento (la ubicación del elemento al comienzo) junto con el elemento. Así, cuando comparamos dos elementos, los comparamos según sus valores (*key*) y, si son iguales, usamos los índices para decidir.
[**1 pt.**] Por cada elemento, se necesita almacenar adicionalmente su índice. Si los índices van entre 0 y n, cada uno se puede almacenar en $\log n$ bits. Así, en total, se necesita $n \log n$ espacio adicional.

3. Considera el algoritmo de Prim, estudiado en clase, para encontrar un árbol de cobertura de costo mínimo para un grafo no direccional $G = (V, E)$, a partir del vértice r en E.

```
for ( cada vértice u en V ) u.key = ∞
r.key = 0
formar una cola Q con todos los vértices en V, priorizada según el campo key de cada vértice
π[r] = null
while ( !Q.empty( ) )
{
    u = Q.extractMin( ) —esta operación modifica la cola Q
    for ( cada vértice v en listadeAdyacencias[u] )
        if ( v ∈ Q ∧ costo(u,v) < v.key )
        {
            π[v] = u
            v.key = costo(u,v) —esta operación modifica la cola Q
        }
}
```

Recuerda que el desempeño de Prim depende de cómo se implementa la cola Q; p.ej., si Q es un heap binario, entonces Prim toma tiempo $O(E \log V)$: el *for* dentro del *while* mira cada arista de G y para cada una realiza una operación *decreaseKey* sobre un vértice (cuando actualiza $v.key$, en la última línea).

Si los costos de todas las aristas de G son números enteros entre 0 y una constante W (tal que es factible declarar un arreglo de tamaño W+1), describe una forma de implementar la cola Q, tal que Prim corra en tiempo $O(E)$, es decir, que cada operación *descreaseKey* tome tiempo O(1). [Recuerda que si W es constante, es decir, no depende ni de E ni de V, entonces recorrer un arreglo de tamaño W+1 toma tiempo $O(1)$.]

Respuesta:

[**2 pts.**] Implementamos Q como un arreglo Q[0], Q[1], …, Q[W], Q[W+1]. Cada elemento del arreglo es una lista doblemente ligada de vértices, en que Q[k] contiene todos los vértices cuyo *key* vale k; Q[W+1] contiene los vértices cuyo *key* vale infinito (∞).

[**2 pts.**] Entonces, *extractMin* consiste simplemente en recorrer Q buscando el primer elemento no vacío; por lo tanto, toma $O(W) = O(1)$.

[**2 pts.**] También, *decreaseKey* toma tiempo $O(1)$: basta sacar el vértice de la lista en que está (la lista tiene que ser doblemente ligada para que sacar un vértice tome $O(1)$) y agregarlo al comienzo de la nueva lista que le corresponda según su nuevo valor de *key*.

4. Escribe una versión no recursiva (y, por lo tanto, iterativa) de *Quicksort*, que use la versión de *partition* estudiada en clase. Supón que dispones de un stack de números enteros con las operaciones *push*, *pop* y *empty* habituales. La idea es que después de llamar a *partition*, en lugar de llamar recursivamente a *Quicksort* dos veces, coloques en el stack los índices correspondientes a cada una de las dos partes. Así, tu versión iterativa debe iterar mientras el stack no esté vacío.

Respuesta:

```
void iterQuicksort(int[] a)
{
   Stack s = new Stack();
   s.push(0); s.push(a.length-1);     [1 pt.] Iniciar el stack con los índices extremos del arreglo original.
   int e, m, w;
   while (!s.empty())
   {
      w = s.pop(); e = s.pop();       [2 pts.] Sacar del stack los índices extrremos de la parte a procesar.
      if (e < w)                      [1 pt.] Verificar si la parte tiene al menos dos elementos.
      {
         m = partition(a, e, w);
         s.push(e); s.push(m-1);      [1 pt.] Colocar en el satck los índices extremos de la parte izquierda.
         s.push(m+1); s.push(w);      [1 pt.] Colocar en el stack los índices extremos de la parte derecha.
      }
   }
}
```