Estructuras de Datos y Algoritmos – IIC2133

Examen

30 junio 2015

1. Considera un árbol AVL en el que acabamos de insertar una nueva clave; sea *Q* (un puntero a) el nodo en que quedó esta nueva clave, y sea *P* (un puntero a) el padre de *Q*. Si cada nodo tiene un campo *balance* (o simplemente *bal*), además de punteros a su padre e hijos izquierdo y derecho (*p*, *izq*, *der*), …

… **escribe un algoritmo —lo más parecido a código posible— eficiente** que actualice el campo *balance* de cada nodo que lo necesite, y determine el nodo "pivote", o raíz del subárbol que debe ser rebalanceado mediante una rotación, si es que lo hay. ¿Qué complejidad tiene tu algoritmo? Justifica.

[**2/3**] Algoritmo:

```
Q = nodo recién insertado
P = Q.p
if Q es hijo izquierdo de P
    P.balance --
else
    P.balance ++
while P no es la raíz y P.balance ≠ 2 y P.balance ≠ –2
    Q = P
    P = P.p
    if Q.balance = 0
        return
    if Q es hijo izquierdo de P
        P.balance --
    else
        P.balance ++
if P.balance = 2 o P.balance = –2
    P es el nodo pivote
```

[**1/3**] Complejidad = O(log*n*), ya que hace un número constante de operaciones en el nodo que está mirando y luego "sube" al padre del nodo, en donde repite las operaciones; a lo más llega a la raíz, que está a altura log*n*, en que *n* es el número de nodos en el árbol, ya que los árboles AVL son balanceados. **¡ Ojo: Se asigna puntaje sólo si está bien justificado** !

2. *Mergesort natural* es una versión *bottom-up* de mergesort; su paso fundamental es el siguiente: primero encuentra un subarreglo ordenado (incrementa un puntero hasta encontrar un dato que sea menor que su predecesor en el arreglo), luego encuentra el próximo subarreglo ordenado, y luego los mezcla.

Escribe un algoritmo —lo más parecido a código posible— eficiente que implemente mergesort natural para ordenar una *lista ligada* (y no un arreglo; este es el método preferido para ordenar listas ligadas, porque no usa espacio adicional). ¿Qué complejidad tiene tu algoritmo? Justifica.

[**2/3**] Algoritmo (una posibilidad, hay otras):

[1/3] ***—identificar las sublistas a mezclar y la condición de terminación del algoritmo***

```
sea P = puntero al elemento vigente de la lista (inicialmente, al primer elemento)
a partir de P, recorrer la lista mientras los elementos no decrezcan en valor.
sea Q = puntero al último elemento "no decreciente"
sea R = puntero al siguiente elemento en la lista
if R = null
    terminar, la lista está ordenada
else
    a partir de R, recorrer la lista mientras los elementos no decrezcan en valor.
    sea S = puntero al último elemento no decreciente
    elemento vigente = S.next
    Mezclar la sublista de P a Q con la sublista de R a S (*)
    volver a la primera instrucción
```

[1/3] ***—hacer la mezcla (el algoritmo* merge *visto en clase no sirve, porque mezcla arreglos y no listas)***

```
(*) Mezclar la sublista de P a Q con la sublista de R a S:
    T = T3 = new(Node)
    T1 = P, T2 = R
    if T1 < T2
        T3 = T1, T1 = T1.next
    else
        T3 = T2, T2 = T2.next
    if T1 = Q.next
        agregar a a T3 la lista de T2 a S
    else
        if T2 = S.next
            agregar a T3 la lista de T1 a Q
        else
            repetir el primer if
    P = T
```

[**1/3**] Complejidad = O(n log$n$), ya que en el peor caso, cuando la lista está ordenada "al revés", primero se mezclan sublistas de largo 1; luego, de largo 2; luego, de largo 4; luego, de largo 8, etc. Como el largo de las sublistas que se mezclan crece al doble cada vez, a lo más hay log$n$ "pasadas" por la lista completa. En cada pasada, hay O(n) operaciones de mezcla, independientemente del largo de las sublistas que se mezclan, ya que de una manera u otra hay que mezclar todos los elementos de la lista. ¡ **Ojo: Se asigna puntaje sólo si está bien justificado** !

3. El algoritmo de Dijkstra para determinar las rutas más cortas desde un vértice a todos los otros vértices en un grafo direccional con costos no negativos es el siguiente:

```
void Dijkstra(Vertex s)
   Init(s)
   S = ∅
   Queue q = new Queue(V)
   while ( !q.empty() )
      Vertex u = q.xMin()
      S = S ∪ {u}
      for ( each v in α[u] )
         reduce(u,v)
```

Este algoritmo es un algoritmo codicioso; **demuestra que efectivamente resuelve el problema**:

a) Demuestra que el problema tiene subestructura óptima.

Trivial. Lo hicimos varias veces en clases.

b) Demuestra que después de hacer una elección, te quedas con un problema del mismo tipo, pero más pequeño.

Como vimos en clase, un algoritmo codicioso funciona en **etapas** y usa una **medida de optimización**. P.ej., en este caso, determinemos las rutas más cortas una por una (las etapas) y sumemos los costos de todas las rutas encontradas hasta ahora (la medida); para que esta suma sea minimizada, cada ruta individual debe ser de mínimo costo. Así, si hasta el momento hemos construido k rutas más cortas, entonces (usando nuestra medida de optimización) la próxima ruta a ser construida deberá ser la próxima ruta de longitud mínima más corta.

c) Demuestra que cada vez que tienes que hacer una elección, la elección codiciosa es óptima.

Lo primero que hay que establecer es **cuál** elección codiciosa (p.ej., elegir la arista de menor costo no sirve). Por supuesto, en el caso de Dijkstra la respuesta es elegir, entre aquellos vértices que aún no han sido elegidos, el vértice que está más cerca del vértice s de partida (siguiendo la estrategia explicada en **b**). La demostración de que esta elección es correcta aparece en Cormen et al. [2009] y se basa en el hecho de que los costos de las aristas son no negativos; la hacemos por contradicción.

Sea u el primer vértice que al ser seleccionado no cumple que $u.d = \delta(s,u)$, según la notación vista en clase —es decir, la elección codiciosa no es óptima. ¿Cuál es la situación justo antes de seleccionar a u? La ruta más corta, p, entre s y u pasa por un vértice w, el primer vértice en p que aún no ha sido seleccionado, y sea x el predecesor de w en p: entonces, podemos descomponer p en un tramo de s a x, seguido por la arista (x,w), seguido por el tramo de w a u.

Sabemos que $x.d = \delta(s,x)$ cuando x fue seleccionado; en ese momento, el algoritmo aplica *reduce* a (x,w), por lo que $w.d = \delta(s,w)$ cuando u es seleccionado.

Entonces, como w aparece antes que u en la ruta más corta de s a u y los costos de todas las aristas son no negativos —en particular, los del tramo w a u— tenemos que $\delta(s,w) \leq \delta(s,u)$, por lo que $w.d = \delta(s,w) \leq \delta(s,u) \leq u.d$. Pero ni u ni w habían sido seleccionados cuando seleccionamos a u, por lo que si seleccionamos (codiciosamente) primero a u, significa que $u.d \leq w$.d, lo que sólo es posible si $w.d = \delta(s,w) = \delta(s,u) = u.d$, lo que efectivamente contradice nuestra forma de seleccionar a u.

4. [**Este problema vale doble**] Considera el siguiente grafo direccional con costos, con vértices *a*, *b*, *c*, *d* y *e*, representado mediante sus listas de adyacencias:

[*a*]: [*b*, 3] – [*c*, 8] – [*e*, –4] [*b*]: [*d*, 1] – [*e*, 7] [*c*]: [*b*, 4] [*d*]: [*a*, 2] – [*c*, –5] [*e*]: [*d*, 6]

El algoritmo de Floyd-Warshall para determinar las rutas más cortas entre todos los pares de vértices en un grafo direccional con costos es el siguiente:

```
D = matriz de adyacencias
for k = 1 … n —n es el número de vértices
    for i = 1 … n —para cada vértice i
        for j = 1 … n —para cada vértice j
            dij = min( dij, dik+dkj )
return D
```

a) Ejecuta el algoritmo de Floyd-Warshall sobre el grafo anterior; muestra el contenido de la matriz D después de cada iteración del índice k, y encuentra tanto las longitudes de las rutas más cortas **como las rutas propiamente tales**.

Ver p. 696 de Cormen et al. [2009].

b) ¿Cómo podemos usar el resultado del algoritmo de Floyd-Warshall para detectar la presencia de un **ciclo con costo acumulado negativo**? Justifica.

Una posibilidad es ejecutar una iteración adicional, con $k = n+1$, y ver si alguno de los valores de la matriz D cambia. Si hay CCANs, entonces el costo de alguna ruta más corta deberá disminuir más allá del "mínimo" entontrado después de n iteraciones.

Lo otro es mirar los valores de la diagonal de D: hay un CCAN si y sólo si alguno de esos valores es negativo. **Demostración**. … (si la necesitan, díganme).

Este algoritmo es un algoritmo de programación dinámica; **muestra que es aplicable al problema**:

c) Demuestra que el problema tiene subestructura óptima.

Trivial. Lo hicimos varias veces en clases.

d) Demuestra que una formulación recursiva de la solución (~~p.ej., a partir de **b***~~[este "hint" no tenía *nada que ver*]) presenta la característica de *subproblemas traslapados*.

La formulación recursiva útil (que también da origen al algoritmo) es la siguiente: Consideremos rutas más cortas que sólo usan los vértices 1, 2, …, k como vértices intermedios. Una ruta más corta de i a j (la *ruta*) puede incluir o no al vértice k; como no sabemos, debemos calcular ambas posibilidades y quedarnos con la mejor (similarmente a como lo hicimos en otros problemas):

- si no lo incluye, entonces la *ruta* es idéntica a la ruta más corta de i a j que sólo usa los vértices 1 a k–1 como vértices intermedios;

- si lo incluye, entonces (por subestructura óptima) la *ruta* es la concatenación de la ruta más corta de i a k con la ruta más corta de k a j, ambas usando sólo los vértices 1 a k–1 como vértices intermedios.

5. Considera el algoritmo de Kruskal para encontrar un árbol de cobertura de costo mínimo (MST) para un grafo $G = (E, V)$ no direccional con costos. Recuerda que Kruskal primero ordena las aristas de menor a mayor costo. Responde como preguntas independientes:

¡ Tanto en **a**) como en **b**), si no hay justificación, el puntaje es **0** !

a) Si todos los costos de las aristas son número enteros en el rango 1 a $|V|$, ¿qué tan rápida puede ser la ejecución de Kruskal? Justifica.

Kruskal toma tiempo $O(V)$ para inicialización, $O(E \log E)$ para ordenar las aristas, y $O(E\ \alpha(V))$ para las operaciones de conjuntos disjuntos; en total, $O(E \log E)$, que es el término "más grande".

Ahora, sabiendo que los costos de las aristas son enteros en el rango 1 a $|V|$, podemos ordenarlas en tiempo $O(V+E) = O(E)$ con *countingSort*, por lo que ahora el término más grande es $O(E\ \alpha(V))$.

b) Si usamos la implementación basada en listas ligadas para representar y manejar conjuntos disjuntos, ¿qué tan rápida puede ser la ejecución de Kruskal? Justifica.

Nuevamente, Kruskal toma tiempo $O(V)$ para inicialización, $O(E \log E)$ para ordenar las aristas, y $O(E\ \alpha(V))$ para las operaciones de conjuntos disjuntos; en total, $O(E \log E)$, que es el término "más grande".

Como vimos en clase, si usamos listas ligadas para representar los conjuntos disjuntos, el tiempo para procesar los conjuntos disjuntos cambia a $O(m+n\log n)$, en que m es el número total de operaciones y n es $|V|$. Entonces, el término más grande seguirá siendo $O(E \log E)$.