Estructuras de Datos y Algoritmos – IIC2133

I3

8 noviembre 2013

1) Se tiene que programar un conjunto de n charlas en varias salas. Cada charla ch_i tiene una hora de inicio s_i y una hora de término f_i , y puede darse en cualquier sala. Queremos programar las charlas de manera de usar el ***menor número posible de salas***.

a) [**2 pts.**] Da un ejemplo que muestre que la solución "obvia" de usar repetidamente el algoritmo codicioso `seleccion` visto en clase ***no funciona***.

b) [**4 pts.**] Da un ***algoritmo codicioso*** de tiempo $O(n + n \log n)$ para determinar cuál charla debería usar cuál sala: parte por ordenar todas las horas de inicio y todas las horas de término en una misma lista, y luego procesa esta lista en orden. Tu descripción puede ser en prosa, pero debe ser clara y precisa; numera pasos, identifica casos.

Respuesta:

a) Si las charlas son [1, 5), [3, 7), [6, 13) y [9, 12), **`seleccion`** toma primero [1, 5) y luego [9, 12), para una misma sala. Como [3, 7) y [6, 13) no son compatibles entre ellas (se traslapan), se necesitan dos salas más, para un total de 3 salas. Sin embargo, se podría haber asignado [1, 5) y [6, 13) a una misma sala, y así quedarían [3, 7) y [9, 12) en otra, para un total de sólo 2 salas.

b) Supongamos que las salas son A, B, C, …, y están en una lista ligada ls de salas disponibles. Primero, ordenamos todas las horas en tiempo $O(n \log n)$; en el caso del ejemplo anterior, las horas ordenadas quedan {1, 3, 5, 6, 7, 9, 12, 13}. Luego, procesamos estas horas así [solo para ayudar a entender el algoritmo + estructura de datos que aparece más abajo]:

– Tomamos la hora 1, inicio de la charla [1, 5), y asignamos la sala A a esta charla, sacándola de ls.
– Tomamos la hora 3, inicio de la charla [3, 7), y asignamos la sala B a esta charla, sacándola de ls.
– Tomamos la hora 5, término de la charla [1, 5), y devolvemos la sala A al comienzo de ls.
– Tomamos la hora 6, inicio de la charla [6, 13), y asignamos la sala A a esta charla, sacándola de ls.
– Tomamos la hora 7, término de la charla [3, 7), y devolvemos la sala B al comienzo de ls.
– Tomamos la hora 9, inicio de la charla [9, 12), y asignamos la sala B a esta charla, sacándola de ls.
– Tomamos la hora 12, término de la charla [9, 12), y devolvemos la sala B al comienzo de ls.
– Tomamos la hora 13, término de la charla [6, 13), y devolvemos la sala A al comienzo de ls.

Es decir, para cada hora, si la hora corresponde al inicio de una charla, tomamos la sala que está al ***comienzo*** de la lista ls y se la asignamos a la charla; si la hora corresponde al término de una charla, devolvemos la sala asignada a esa charla al ***comienzo*** de la lista ls. Esta iteración toma tiempo $O(n)$.

Este algoritmo es, evidentemente, codicioso. Además, produce una solución óptima, porque, cuando asigna una sala a una charla, asigna una que nunca ha sido ocupada ***solo si todas las que han sido ocupadas aún están ocupadas***: las salas se sacan del comienzo de la lista y se devuelven al comienzo de la lista.

2) 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***. Describe 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.

Respuesta:

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

2) 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*.

3) 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.

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

5) 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 4) es una forma concreta de hacerlo.

3) Considera el algoritmo de Prim, que se muestra, para determinar un árbol de cobertura de costo mínimo (MST) para un grafo no direccional con aristas con costos. ***Deduce paso a paso la complejidad del algoritmo***, en notación O( ), en función del número de vértices, *V*, y del número de aristas, *E*, del grafo; explicita las suposiciones que hagas sobre las estructuras de datos usadas en el algoritmo

```
void prim(Vertex r)
    Queue q = new Queue(V)
    for ( each u in q ) u.key = ∞
    r.key = 0; π[r] = null
    while ( !q.empty() )
        Vertex u = q.xMin()
        for ( each v in α[u] )
            if ( v ∈ q ∧ w(u,v) < v.key )
                π[v] = u
                v.key = w(u,v)
```

Respuesta:

Suponiendo, como vimos en clase, que implementamos la cola priorizada **`q`** como un min-heap, las líneas antes del **`while`** corresponden a la construcción inicial del heap, que toma tiempo $O(V \log V)$: hay que poner V datos en el heap, y poner cada uno toma tiempo $O(\log V)$ —aunque más estrictamente se puede demostrar que el tiempo total es $O(V)$.

A este tiempo, hay que sumar el tiempo que toma el **`while`**.

En cada iteración, se saca un dato (el que tiene la menor clave y por lo tanto está en la primera posición de la cola) del heap (**`q.xMin()`**) y no se inserta ninguno, por lo que en total se ejecutan $|V|$ iteraciones; cada ejecución de **`xMin()`** toma tiempo $O(\log V)$, ya que hay que restaurar el heap cada vez, por lo que el tiempo total para las $|V|$ llamadas a **`xMin()`** es $O(V \log V)$.

Además, en cada iteración se ejecuta un **`for`**, que lo que hace es revisar cada uno de los vértices adyacentes al vértice **`u`** que se acaba de sacar del heap, o, equivalentemente, revisar cada una de las aristas conectadas al vértice **`u`**. Como finalmente se sacan todos los vértices del heap, entonces la totalidad de los **`for`** dentro del **`while`** lo que hace es revisar todas las aristas del grafo, dos veces cada una; es decir, el **`for`** ejecuta $2|E|$ veces en total. Como cada vez potencialmente se actualiza la clave de un vértice en el heap (**`v.key = w(u,v)`**), lo que toma $O(\log V)$, las $2|E|$ ejecuciones del **`for`** toman en total tiempo $O(E \log V)$. (La condición **`v ∈ q`** se puede implementar eficientemente en tiempo $O(1)$ agregando un bit a cada vértice.)

Así, el **`while`** en total toma tiempo $O(V \log V + E \log V) = O(E \log V)$.

Por lo tanto, el algoritmo toma en total tiempo $O(V) + O(E \log V) = O(E \log V)$.

4) Considera el ***algoritmo de Bellman-Ford***, que determina las rutas más cortas desde s a cada uno de los vértices v de un grafo direccional $G = (V, E)$:

```
boolean BellmanFord( Vértice s )
    Init(s)
    for (k = 1; k < |V|; k++)
        for (cada arista (u,v) ∈ E) Reduce(u,v)
    for (cada arista (u,v) ∈ E)
        if (d[v] > d[u] + ω(u,v)) return false
    return true
```

Aplícalo al grafo representado por la siguiente matriz de adyacencias, para determinar las longitudes de las rutas más cortas desde el vértice 1 (las casillas vacías representan ∞):

	1	2	3	4	5	6	7
1	0	6	5	5			
2		0			–1		
3		–2	0		1		
4			–2	0		–1	
5					0		3
6						0	3
7							0

Muestra los valores del vector **d** después de cada iteración.

Respuesta:

$k = 1$:	0	6	5	5	∞	∞	∞
$k = 2$:	0	3	3	5	5	4	∞
$k = 3$:	0	1	3	5	2	4	7
$k = 4$:	0	1	3	5	0	4	5
$k = 5$:	0	1	3	5	0	4	3
$k = 6$:	0	1	3	5	0	4	3

Si bien las respuestas podrían diferir en cuanto a la progresión de los valores de **d**, las respuestas correctas deben cumplir lo siguiente:

- Para $k = 1$, **d** debe corresponder a los costos de las aristas (directas) desde el vértice 1 a los otros vértices.
- El algoritmo ha encontrado todas las distancias mínimas cuando $k = 5$ (ya que la ruta más corta que tiene más aristas tiene 4 aristas) y, por lo tanto, **d** no cambia de $k = 5$ a $k = 6$.
- Los valores de **d** no pueden aumentar a lo largo de las iteraciones.