Estructuras de Datos y Algoritmos – IIC2133

I3

15 de noviembre, 2010

1) Se tiene un grafo direccional en que las aristas que salen desde el vértice de partida *s* pueden tener costos negativos, todas las otras aristas tienen costos no negativos, y no hay ciclos con costo acumulado negativo. Prueba que, a pesar de lo que dijimos en clase, el ***algoritmo de Dijkstra*** encuentra correctamente las rutas más cortas desde *s* en este grafo.

Respuesta:

Dijkstra resuelve el problema porque lo costos negativos, de las aristas que salen de *s*, son los primeros que ve, y, por lo tanto, los toma en cuenta antes de tomar en cuenta cualquier otro costo no negativo.

2) 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 $G = (V, E)$, y considera el número de aristas en cada una de estas rutas más cortas. Sea m el número de aristas de la ruta más corta con más aristas. Modifica el algoritmo, de manera que termine después de m+1 pasadas (en lugar de $|V| + 1$ pasadas), aún cuando m no se conozca por adelantado.

```
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
}
```

Respuesta:

Bellman-Ford itera $|V|-1$ veces (primer **for** exterior), porque la ruta más corta con más aristas tiene a lo más $|V|-1$ aristas (si tuviera más aristas, contendría un ciclo). El segundo **for** exterior hace una última pasada por todas las aristas, pero solo para verificar que en G no haya ciclos con costo acumulado negativo. (Este sería el caso si esta última pasda pudiera "mejorar" alguna ruta más corta.)

Por lo tanto, bajo el supuesto de que la ruta más corta con más aristas tiene m aristas, la modificación al algoritmo consiste en cambiar el primer **for** exterior por un **while** que itere mientras sea posible mejorrar alguna ruta más corta, es decir, mientras la ejecución de algún **Reduce** haya producido una mejora. (El segundo **for** exterior desaparece, porque si G contuviera ciclos con costo acumulado negativo, el **while** no se detendría jamás.)

```
BellmanFord-m( Vértice s )
{
   Init(s)
   mejora = true
   while ( mejora =  true )
   {
      mejora = false
      for ( cada arista (u,v) ∈ E )
         if ( d[v] > d[u]+ω(u,v) )
         {
            d[v] = d[u]+ ω(u,v)
            π[v] = u
            mejora = true
         }
   }
}
```

3) 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) [**1/3**] Da un ejemplo que muestre que la solución "obvia" de usar repetidamente el algoritmo codicioso `seleccion` visto en clase ***no funciona***.

b) [**2/3**] 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.

Respuesta:

a) [**1/3**] 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 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 2 salas.

b) [**2/3**] 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.

4) Se tiene que programar un conjunto de n actividades en una máquina. Cada actividad a_i tiene una hora de inicio s_i , una hora de término f_i , y un valor v_i . El objetivo es maximizar el valor total de las actividades programadas (y no necesariamente el número de actividades programadas).

Sea C_{ij} el conjunto de actividades que empiezan después que la actividad a_i termina y que terminan antes que empiece la actividad a_j . Sea A_{ij} una solución óptima a C_{ij} , es decir, A_{ij} es un subconjunto de actividades mutuamente compatibles de C_{ij} que tiene valor máximo.

a) Supongamos que A_{ij} incluye la actividad a_k . Demuestra que este problema tiene la propiedad de ***subestructura óptima***: una solución óptima al problema contiene soluciones óptimas a subproblemas.

b) Sea $val[i, j]$ el valor de una solución óptima para el conjunto C_{ij}. Demuestra que este problema tiene la propiedad de ***subproblemas traslapados***: si usamos un algoritmo recursivo para resolver el problema, este tiene que resolver los mismos subproblemas repetidamente.

Respuesta:

a) Podemos descomponer la solución óptima A_{ij} como $A_{ik} \cup \{a_k\} \cup A_{kj}$; es decir, la actividades que, estando en A_{ij} , terminan antes que empiece a_k , junto a las actividades que, estando en A_{ij} , empiezan después que termina a_k , junto a a_k . Así, el valor de la solución óptima A_{ij} es igual al valor de A_{ik} más el valor de A_{kj} más v_k . Siguiendo un razonamiento similar a los vistos en clase para otros problemas, concluimos que A_{ik} debe ser una solución óptima al problema definido por C_{ik} , y análogamente para A_{kj} y C_{kj} .

b) De acuerdo con a), $val[i, j] = val[i, k] + val[k, j] + v_k$. Pero en realidad no sabemos que la solución óptima al problema definido por C_{ij} incluye la actividad a_k , por lo que debemos mirar todas las actividades en C_{ij} para averiguar cuál elegir; es decir,

$$val[i, j] = 0, \qquad \text{si } C_{ij} = \varnothing$$
$$val[i, j] = \max_{a_k \in S_{ij}} \{ val[i, k] + val[k, j] + v_k \}, \qquad \text{si } C_{ij} \neq \varnothing$$

Aquí aparecen los subproblemas traslapados.