Estructuras de Datos y Algoritmos – IIC2133

I3

14 junio 2013

1) En clase resolvimos el problema de optimización de programar tareas con plazos: Dado un conjunto de tareas, c/u con un plazo y una ganancia, encontrar un subconjunto de tareas que pueden ser hechas dentro de sus plazos (subconjunto *factible*) y que maximiza la suma de las ganancias (subconjunto *óptimo*); las tareas sólo pueden ser hechas usando una única máquina por una unidad de tiempo —por lo tanto, sólo se puede hacer una tarea a la vez— y las ganancias se obtienen sólo si las tareas son hechas dentro de sus plazos.

Demostramos que el problema puede resolverse si ordenamos las tareas de mayor a menor ganancia, y empleamos la estrategia codiciosa de agregar a la solución la próxima tarea que siendo factible hace aumentar más la ganancia acumulada. Esta solución requiere poder determinar la factibilidad de un subconjunto de tareas, lo que resolvimos demostrando que basta probar la factibilidad de una permutación del subconjunto en que las tareas estén ordenadas crecientemente por plazos.

a) ¿Qué complejidad tiene este algoritmo? Justifica.

Resp. $O(n^2)$. Para cada una de las n tareas —ya ordenadas de mayor a menor ganancia en tiempo $O(n\log n)$— hay que ver si es compatible con las otras tareas que ya están en la solución. Esto requiere ordenar las m tareas en la solución por plazos —en tiempo $O(m)$, usando ordenación por inserción en un arreglo ordenado— y luego verificar la factibilidad del conjunto, es decir, determinar si cada tarea se está haciendo dentro de su plazo: como el conjunto tiene m tareas, esta verificación toma tiempo $O(m)$. Como este tiempo $O(m)$ se repite para cada una de las n tareas, el tiempo total es $O(nm)$, que en el peor caso es $O(n^2)$.

Es posible usar otro método para determinar la factibilidad de un subconjunto de tareas. Si J es un subconjunto factible de tareas, entonces podemos asignar los tiempos de procesamiento de cada tarea de la siguiente manera: si para la tarea i aún no hemos asignado un tiempo de procesamiento, entonces asignémosle el *slot* $[k–1, k]$, en que k es el mayor entero menor o igual que el plazo de i y el *slot* $[k–1, k]$ está vacío —es decir, postergamos la tarea i lo más que podemos. Así, al construir J de a una tarea a la vez, no movemos las tareas que ya tienen sus *slots* asignados para acomodar una nueva tarea: si para esta nueva tarea no encontramos un k como el que definimos antes, entonces la tarea no puede programarse.

b) Demuestra esta última afirmación.

Resp. La única forma en que la tarea t, con plazo d_t, no pueda programarse es que todos los slots $[0, 1]$, $[1, 2]$, …, $[d_{t-1}, d_t]$ estén ocupados. En este caso, no hay ninguna forma de hacer espacio para la tarea t, ya que todas las tareas programadas en estos slots están postergadas lo más posible, considerando incluso slots posteriores a $[d_{t-1}, d_t]$: si el slot $[k–1, k]$, con $k > d_t$, estuviera vacío, y la tarea s programada en el slot $[j–1, j]$, con $j < d_t$, se hubiera podido programar en el slot $[k–1, k]$, entonces la tarea s se hubiera programado en este slot al considerarla por primera vez.

c) Explica cómo se puede implementar esta nueva forma de programar las tareas usando una estructura de conjuntos disjuntos.

Resp. Llamemos k al slot $[k-1, k]$; y sea n_k el mayor entero tal que $n_k \leq k$ y el slot n_k está vacío. Dos slots p y q están en el mismo conjunto si $n_p = n_q$; claramente, si $p < q$, entonces p, $p+1$, $p+2$, …, q están todos en el mismo conjunto; el valor n_p es un valor importante para el conjunto, por lo que lo almacenamos en un campo f en el representante del conjunto. Para programar una tarea con plazo d, buscamos el (representante del) conjunto al que pertenece el slot d, y obtenemos su valor f, que es el slot más cercano (y no mayor que d) disponible; luego, unimos este conjunto con el conjunto correspondiente al slot $f-1$.

d) ¿Qué complejidad tiene este nuevo algoritmo? Justifica.

Resp. La complejidad es $O(n\alpha(2n,n))$, si usamos la estructura más eficiente conocida para manejar conjuntos disjuntos: inicialmente, hay n conjuntos disjuntos, y luego realizamos n *find*'s (uno por cada tarea) y a lo más n *union*'s.

2) Considera el siguiente algoritmo de ordenación topológica un grafo direccional acíclico (DAG):

> Identifica un vértice *fuente* (un vértice al que no "llega" ninguna arista), asígnale el número 0, sácalo del grafo (junto a las aristas que salen de él). Repite este proceso para el grafo resultante, pero ahora usa el número 1; luego, el 2; y así sucesivamente.

Sugiere las estructuras de datos necesarias y su funcionamiento para implementar este algoritmo eficientemente. En particular:

a) Justifica que todo DAG tiene al menos una fuente.

Resp. Parémonos en un vértice cualquiera, v_0, y recorramos "hacia atrás" una arista que llega a él; como el grafo es acíclico, el vértice al que llegamos, v_1, es distinto de v_0. Hagamos lo mismo en v_1; vamos a llegar a otro vértice, v_2, que, por la misma razón, debe ser distinto de v_0 y v_1. Sigamos así hasta tener una secuencia de $n = |V|$ vértices distintos (suponiendo que en los n–1 primeros vértices de la secuencia siempre encontramos al menos una arista que llega al vértice). Si en este punto el vértice actual, v_n, tuviera una arista que llegara a él, ésta sólo podría venir de alguno de los vértices ya visitados, y por lo tanto formaría un ciclo; como el grafo es acíclico, este último vértice no puede tener ninguna arista que llegue a él.

b) ¿Cómo identificamos los vértices fuente la primera vez?

Resp. Supongamos que el grafo está representado por sus listas de adyacencias. Definamos un arreglo x, indexado según los vértices del grafo, que va a contar el número de aristas que llegan a cada vértice; inicializamos todos los elementos de x están en 0. Ahora recorremos las listas de adyacencias una a una; cada vez que llegamos a un vértice v, incrementamos el contador $x[v]$. Al terminar de recorrer las listas de adyacencias, los elementos de x que estén en 0 corresponden a los vértices fuente iniciales. Este procedimiento toma tiempo $O(V+E)$.

c) ¿Dónde guardamos los vértices fuente que vamos identificando a lo largo de la ejecución del algoritmo?

Resp. Cada vez que procesamos un vértice fuente (le asignams el número 0, 1, 2, …, lo sacamos del grafo, y también sacamos las aristas que salen de él), producimos nuevos vértices fuente: elementos de x que quedan en 0 (ver d). ¿Cómo hacemos para identificarlos y procesarlos? Una posibilidad es que cada vez que vamos a buscar un nuevo vértice fuente, recorramos todo el arreglo x buscando elementos que estén en 0 y que no sean vértices fuente procesados anteriormente. Esto *no* es muy eficiente.

Es mejor usar una cola (o un *stack*) de vértices fuente aún no procesados. Inicializamos la cola con los vértices fuente identificados en b). Para procesar el próximo vértice fuente, simplemente sacamos el primero de la cola. Durante el procesamiento de un vértice fuente, cuando producimos uno nuevo, ponemos éste en la cola.

d) ¿Cómo identificamos los (nuevos) vértices fuente en el grafo que queda después de que sacamos un vértice fuente?

Resp. Sacamos un vértice fuente de la cola y recorremos su lista de adyacencias: para cada vértice v que encontramos en esta lista, decrementamos $x[v]$; si $x[v]$ queda en 0, agregamos v a la cola. Este procedimiento toma tiempo $O(V+E)$.

3) Hay que programar un conjunto de n tareas en una máquina que sólo puede realizar una tarea a la vez. Cada tarea a_i tiene una hora de inicio s_i , una hora de término f_i , y un valor v_i . Sea A un subconjunto de tareas mutuamente compatibles (para cualquier par de tareas de A, la hora de inicio de una de ellas es mayor que la hora de término de la otra); el *valor de A* es la suma de los valores de las tareas de A. El objetivo es encontrar un subconjunto A de valor máximo, es decir, maximizar la suma de los valores de las tareas programadas (y no necesariamente el número de tareas programadas).

En particular, sea C_{ij} el conjunto de tareas que empiezan después que la tarea a_i termina y que terminan antes que empiece la tarea a_j . Sea A_{ij} una solución óptima a C_{ij} , es decir, A_{ij} es un subconjunto de tareas 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.

Resp. 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) 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.

Resp. 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 determinar 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.

4) Con respecto a las rutas más cortas en grafos direccionales:

a) Un compañero de curso te dice que implementó el algoritmo de Dijkstra. El programa produce las distancias d y los padres π para cada vértice del grafo. Da un algoritmo de tiempo $O(V+E)$ para verificar el resultado del programa de tu compañero. Supón que todas las aristas tienen costos no negativos. En particular,

i) ¿qué debes verificar con respecto al vértice de parida s?

ii) ¿qué debes verificar con respecto a los otros vértices?

iii) ¿cómo te puedes asegurar que el programa de tu compañero efectivamente encontró las rutas más cortas desde s?

Resp.

i) Para s, hay que verificar $s.d = 0$ y $s.\pi = nil$.

ii) Para los otros vértices v, hay que verificar $v.d = v.\pi.d + w(v.\pi, v)$; o bien $v.d = \infty$ si y solo si $v.\pi = nil$.

Si cualquiera de las verificaciones anteriores, **i**) o **ii**), falla, entonces el resultado del programa del compañero es incorrecto.

iii) De lo contrario, todavía hay que asegurarse que haya encontrado las rutas más cortas desde s. Para esto, basta con aplicar **`reduce`** a cada arista una vez: si algún $v.d$ cambia, entonces el resultado es incorrecto; de lo contrario, es correcto.

b) Considera el *algoritmo de Bellman-Ford* y, en particular, el número de aristas en cada una de las 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
}
```

Resp. *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 la ruta tuviera más aristas, contendría un ciclo). El segundo `for` exterior hace una última pasada por todas las aristas, 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 mejorar 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 )
  {
    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
          }
  }
}
```

Tiempo: 120 minutos