IIC2133 Estructuras de Datos y Algoritmos

Examen

24 noviembre 2015

1. Considera un grafo direccional $G = (V, E)$ con costos y **acíclico** (y que, por lo tanto, puede ser ordenado topológicamente):

a) [**3 pts.**] Da un algoritmo de tiempo $O(V+E)$ para encontrar las rutas más cortas en G desde un vértice de partida s.

1. *ordenar G topológicamente* —visto en clase; necesario para poder hacer un algoritmo eficiente
2. *para cada vértice v de G:* —este es el procedimiento *init*(s) visto en clase
   $d[v] = \infty$
   $\pi[v] = nil$
   $d[s] = 0$
3. *para cada vértice u de G,* ***en el orden producido por la ordenación topológica:*** —esta condición es esencial para la corrección del algoritmo
   para cada vértice v adyacente a u: —este es el procedimiento *reduce*(u, v) visto en clase
   if ($d[v] > d[u] + w(u, v)$)
   $d[v] = d[u] + w(u, v)$
   $\pi[v] = u$

b) [**3 pts.**] Justifica que tu algoritmo encuentra las rutas más cortas y que toma tiempo $O(V+E)$.

[**1.5 pts.**] El algoritmo es de tiempo $O(V+E)$, como se justifica a continuación:
- ordenar G topológicamente (paso 1) es $O(V+E)$, ya que esencialmente es hacer un recorrido DFS
- el ciclo "*para*" que sigue (paso 2) es $O(V)$, por lo que no agrega complejidad
- el ciclo "*para*" que sigue (paso 3) es $O(V+E)$, ya que mira cada vértice u y, para cada uno, mira cada arista que sale de u (los vértices v adyacentes a u); y, en cada caso, hace una operación de complejidad constante (un *reduce*)

[**1.5 pts.**] El algoritmo efectivamente calcula las rutas más cortas desde s a todos los otros vértices (alcanzables desde s); en el paso 3:
- cada arista (u, v) es reducida exactamente una vez, cuando u es procesado (cuando la iteración "pasa" por u), dejando $d[v] \leq d[u] + w(u, v)$
- esta desigualdad se mantiene de ahí en adelante, hasta que el algoritmo termina, ya que $d[u]$ no cambia: como los vértices son procesados en orden topológico, ninguna arista que apunte a u va a ser reducida después de procesar u

2. En clase estudiamos la siguiente versión del algoritmo de Bellman–Ford para encontrar las rutas más cortas desde un vértice de partida s, en un grafo $G = (V, E)$ direccional y con costos:

```
Bellman-Ford(s):
   init(s)
   for k = 1 ... |V|-1:
      for each (u,v) ∈ E:
         reduce(u,v)
```

```
init(s):
   for each v ∈ V:
      d[v] = ∞
      π[v] = null
   d[s] = 0
```

```
reduce(u, v):
   if d[v] > d[u]+ω(u,v):
      d[v] = d[u]+ω(u,v)
      π[v] = u
```

Básicamente, el algoritmo hace $|V|-1$ pasadas por todas las aristas del grafo, tratando de reducirlas (o relajarlas). Sin embargo, las únicas aristas que podrían producir un cambio en `d[]` son aquellas que salen de un vértice cuyo `d[]` cambió en la pasada anterior.

Modifica este algoritmo de manera que intente reducir aristas sólo cuando tenga sentido hacerlo. Para ello, emplea **una cola de vértices** y **un arreglo booleano** de $|V|$ casilleros; además, en lugar de hacer $|V|-1$ pasadas, tu algoritmo debe detenerse cuando la cola quede vacía.

La cola `Q` se usa para guardar los vértices cuyos `d[]` acaban de cambiar.

El arreglo booleano `enQ` es para saber si un vértice está en `Q` y, en ese caso, no guardarlo nuevamente allí

```
Bellman-Ford'(s):
    d[s] = 0 —suponemos que todos los otros d[] son inicialmente ∞
    Q.enqueue(s) —suponemos que Q está inicialmente vacía
    enQ[s] = True —suponemos que todos los otros casilleros de enQ son inicialmente False
    while !Q.empty(): —en lugar de iterar |V|−1 veces, iteramos mientras Q no esté vacía
        u = Q.dequeue()
        enQ[u] = False
        for each v in adj[u]: —reducimos (o tratamos de reducir) cada arista que sale de u
            if d[v] > d[u] + ω(u,v):
                d[v] = d[u] + ω(u,v)
                π[v] = u
                if !enQ[v]: —si pudimos reducir (u,v), entonces guardamos v en Q (si v no está ya en Q)
                    Q.enqueue(v)
                    enQ[v] = True
```

La ventaja de esta versión del algoritmo de Bellman-Ford, que es la que se usa en la práctica, es que es más rápida.

3. Demuestra que dado un grafo no direccional $G = (V, E)$ con costos en las aristas, el problema de encontrar un árbol de cobertura de costo mínimo (MST) para G se puede resolver mediante un algoritmo codicioso —en particular, mediante el algoritmo de Kruskal. Para ello, sigue estos pasos:

a) Formula el problema como uno en el cual haces una elección y te quedas con un subproblema del mismo tipo para resolver (¿qué tipo de elección haces en el caso del algoritmo de Kruskal?).

Una manera de construir el árbol es partir con un árbol vacío e ir agregando una arista a la vez, hasta completar $|V|-1$ aristas que cubran todos los vértices de G (y que no formen un ciclo). [El desafío es encontrar una estrategia para ir eligiendo las aristas, como se explica en **b**) y **c**)].

b) Demuestra que hay una solución óptima al problema original que hace la **elección codiciosa** (¿cuál es la elección codiciosa que haces en el algoritmo de Kruskal?).

La elección codiciosa consiste en elegir la arista más liviana de todas las aristas de G; llamémosla e. ¿Es e parte del MST? Sí, como de demuestra a continuación.

Supongamos que tenemos un MST de G, llamémoslo T, y que T no incluye la arista e. ¿Qué pasa si agregamos e a T? Claramente, formamos un ciclo, ya que T es un árbol que cubre **todos** los vértices de G. De este ciclo podemos sacar cualquier arista y volvemos a obtener un árbol de cobertura. Si sacamos una arista distinta de e, y por lo tanto más pesada que e, el árbol resultante tiene un costo total que no es mayor que el costo de T.

c) Demuestra que, habiendo hecho la elección codiciosa, lo que queda es un subproblema tal que si combinas una solución óptima al subproblema con la elección codiciosa hecha, obtienes una solución óptima al problema original.

4. Un componente esencial del algoritmo de ordenación **`quicksort()`** es el algoritmo de partición. Considera el siguiente algoritmo **`partition()`**:

```
partition(a, p, r):
   x = a[r]
   i = p-1
   for j = p … r-1:
      if a[j] ≤ x:
         i = i+1
         exchange(a[i], a[j])
   exchange(a[i+1], a[r])
   return i+1
```

a) Muestra el funcionamiento de **`partition()`** en el arreglo **`a`** = [B, H, G, A, C, E, F, D].

Inicialmente, p = 0, r = 7. x = 'D', i = –1

Ejecutamos el ciclo for j, con j = 0, 1, …, 6:

j = 0 –> a[j] = 'B' < x = 'D' –> i = 0 e intercambia (los contenidos de) a[0] y a[0] –> a queda igual
j = 1 –> a[j] = 'H' > x = 'D' –> no pasa nada
j = 2 –> a[j] = 'G' > x = 'D' –> no pasa nada
j = 3 –> a[j] = 'A' < x = 'D' –> i = 1 e intercambia a[1] y a[3] –> a = ['B', 'A', 'G', 'H', 'C', 'E', 'F', 'D']
j = 4 –> a[j] = 'C' < x = 'D' –> i = 2 e intercambia a[2] y a[4] –> a = ['B', 'A', 'C', 'H', 'G', 'E', 'F', 'D']
j = 5 –> a[j] = 'E' > x = 'D' –> no pasa nada
j = 6 –> a[j] = 'F' > x = 'D' –> no pasa nada

Así, al salir del ciclo for j, i = 2, y a = ['B', 'A', 'C', 'H', 'G', 'E', 'F', 'D']

Por lo tanto, finalmente intercambia a[3] y a[7] –> **a = ['B', 'A', 'C', 'D', 'G', 'E', 'F', 'H'] y retorna i+1 = 3**

La explicación anterior es la muestra más detallada del funcionamiento de **`partition()`**. Al menos, hay que mostrar las líneas en las que el contenido de **`a`** cambia, y la línea final; es decir, las tres líneas que están en **bold**.

b) Demuestra que en todo momento durante la ejecución de **`partition()`**, un arreglo **`a`** cualquiera está dividido en cuatro sectores: **`a[p]`** … **`a[i]`**, que son valores menores o iguales que el pivote; **`a[i+1]`** … **`a[j-1]`**, que son valores mayores que el pivote; **`a[j]`** … **`a[r-1]`**, que son valores aún no procesados; y **`a[r]`**, que es el pivote.

Los que importan son los sectores **`a[p]`** … **`a[i]`** y **`a[i+1]`** … **`a[j-1]`**; simplemente mirando el código, los otros dos sectores son "obvios".

La afirmación es inicialmente verdadera, ya que los sectores **`a[p]`** … **`a[i]`** y **`a[i+1]`** … **`a[j-1]`** **no tienen elementos**, debido a los valores iniciales de los índices (los *rangos* de los índices son vacíos).

El procesamiento del arreglo **`a`** lo hace el ciclo **`for j`**, que revisa en orden cada uno de los valores **`a[p]`** … **`a[r-1]`**, lo que de paso demuestra que el sector **`a[j]`** … **`a[r-1]`** corresponde a los valores aún no procesados.

La revisión de **`a[j]`** produce un efecto sólo si **`a[j]`** ≤ **`x`** (el pivote **`x`** = **`a[r]`** es constante a lo largo de todo el ciclo):

- el efecto es que el sector **`a[p]`** … **`a[i]`**, de los elementos menores o iguales que el pivote, aumenta su tamaño en 1 (la instrucción **`i = i+1`**) para recibir a **`a[j]`** (por la vía de intercambiarlo con **`a[i]`**);
- en cambio, si **`a[j]`** > **`x`**, entonces lo único que ocurre es que **`j`** aumenta en 1.

Como el sector **`a[p]`** … **`a[i]`** es inicialmente vacío, lo anterior significa que sólo crece para recibir valores menores o iguales que el pivote, demostrando esta parte de la afimación.

Además, como cada valor ≤ **`x`** va a parar al sector **`a[p]`** … **`a[i]`**, y el sector **`a[j]`** … **`a[r-1]`** contiene los elementos aún no procesados, entonces el sector **`a[i+1]`** … **`a[j-1]`** necesariamente contiene valores ya procesados y mayores que **`x`**, demostrando esta otra parte de la afimación.

c) ¿Qué valor devuelve **`partition()`** cuando todos los elementos del arreglo **`a`** tienen el mismo valor?

Si todos los elementos de **`a`** son iguales, entonces para cada **`j`**, efectivamente **`a[j] ≤ x`**. Por lo tanto, para cada **`j`**, se incrementa **`i`** (y se intercambian elementos de **`a`**, pero esto no produce cambios en cómo se ve **`a`**); como **`j`** va de **`p`** a **`r-1`**, entonces **`i`**, que parte en **`p-1`**, llega a valer **`r-2`**.

Y como **`partition()`** siempre devuelve **`i+1`**, entonces **`partition()`** devuelve **`r-1`** cuando todos los elementos de **`a`** son iguales.

5a. Un ladrón entra a una tienda llevando una mochila con capacidad de 10 kg. En la tienda, el ladrón encuentra tres tipos de objetos (aunque hay innumerables objetos de cada tipo): los objetos de tipo 1 pesan 4 kg y tienen un valor de 11; los de tipo 2, pesan 3 kg y valen 7; y los de tipo 3, pesan 5 kg y valen 12. ¿Con cuáles objetos debe llenar la mochila el ladrón para maximizar su valor sin exceder su capacidad? Resuelve este problema empleando **programación dinámica**; en particular:

a) Demuestra que el problema exhibe la propiedad de subestructura óptima.

Subestructura óptima significa que la solución óptima al problema original contiene soluciones óptimas a problemas más pequeños del mismo tipo; en este caso, ya sea mochilas de menor capacidad, o bien sólo uno o dos tipos de objetos, o bien mochilas de menor capacidad y sólo uno o dos tipos de objetos.

La mochila óptima puede contener o no objetos de tipo 1. Hay que analizar ambos casos y quedarse con el que produce el mejor resultado. Supongamos que la mochila óptima contiene al menos un objeto de tipo 1. Esto significa que su valor es 11 más el valor del resto de la mochina. Pero este valor debe corresponder a una mochila óptima de capacidad 6 kg (= 10 kg – 4 kg) con objetos de los tipos 1, 2 y 3: un problema similar al original, pero más pequeño.

Supongamos ahora que la mochila óptima no contiene objetos de tipo 1. Esto significa que es equivalente a una mochila óptima de capacidad 10 kg, pero que sólo contiene objetos de los tipos 2 y 3: nuevamente, un problema similar al original, pero más pequeño.

b) Plantea la solución recursivamente.

Si representamos la solución óptima al problema de una mochila con capacidad C y objetos de los tipos t_1, t_2 y t_3 por $[C, \{t_1,t_2,t_3\}]$, entonces, del análisis de **a**), tenemos que

$[10, \{1,2,3\}] = \max\{ 11 + [6, \{1,2,3\}], [10, \{2,3\}] \}$.

Y para una instancia intermedia cualquiera, gracias a la propiedad de subestructura óptima, suponiendo objetos de tipos $t_j, \ldots, t_k$ con valores $v(t_i)$ y pesos $w(t_i)$,

$[c, \{t_j, \ldots, t_k\}] = \max\{ v(t_j) + [c - w(t_j), \{t_{j+1}, \ldots, t_k\}], [c, \{t_{j+1}, \ldots, t_k\}] \}$

c) Desarrolla la formulación recursiva de la solución, de modo de responder la pregunta anterior.

Si se desarrolla la formulación anterior a partir de la primera ecuación y aplicando la segunda en los pasos intermedios, finalmente obtenemos que la solución óptima es un objeto de tipo 1 y dos de tipo 2: llenan la mochila exactamente y su valor total es 25.

5b. Supongamos que tienes los votos emitidos por los estudiantes de la universidad para elegir al presidente de la federación; cada voto contiene simplemente el RUT del candidato. Hay n votos emitidos y sabemos que hay k candidatos, pero no sabemos quiénes son.

a) [**2/3**] Describe completamente un algoritmo y las estructuras de datos correspondientes que permitan determinar al ganador de la elección en tiempo $O(n \log k)$; tus estructuras de datos pueden usar a lo más $O(k)$ memoria.

b) [**1/3**] ¿Qué suposición razonable debes hacer para que el tiempo sea efectivamente $O(n \log k)$?

6. Con respecto a los árboles rojo-negros:

a) [**1 pt.**] Considere un árbol formado únicamente mediante la inserción de n nodos usando el método de inserción visto en clases. Justifica que si $n > 1$, el árbol tiene a lo menos un nodo rojo.

b) [**1 pt.**] Muestra con un ejemplo que un árbol de más de un nodo puede tener sólo nodos negros si su construcción ha incluido eliminaciones.

c) [**2 pts.**] Supón que insertamos un nodo x en un árbol rojo-negro y luego lo eliminamos inmediatamente. ¿Es el árbol resultante el mismo que el inicial? Justifica tu respuesta.

d) [**2 pts.**] Un árbol con 8 nodos tiene la siguiente configuración: la raíz, con clave Q, es negra; sus hijos tienen las claves D y T, y son rojo y negro, respectivamente; D tiene dos hijos negros: A y L; T tiene un único hijo, W; y L tiene dos hijos: J y N. Inserta en este árbol la clave G; en particular

i) [**0.5 pts.**] Dibuja el árbol original, antes de la inserción de G; deduce los colores de los nodos J, N y W.

ii) [**0.5 pts.**] Dibuja el árbol justo después de la inserción de G, pero antes de cualquier operación de restauración de las propiedades del árbol. ¿Cuál propiedad **no** se cumple?

iii) [**1 pt.**] Dibuja el árbol después de cada operación de cambio de colores y de cada operación de rotación, explicando qué problema se produce en cada caso, hasta llegar al árbol final.

7. Encuentra los costos de las rutas más cortas entre todos los pares de vértices para el siguiente grafo direccional representado por su matriz de adyacencias, **empleando el algoritmo de Floyd–Warshall**. P.ej., el costo de la arista que va del vértice 1 al vértice 2 es 51; y no hay arista del vértice 3 al vértice 4. En particular, muestra cada una de las siguientes **tres** matrices que produce el algoritmo.

	0	**1**	**2**	**3**	**4**	**5**
0	0	41				29
1		0	51		32	
2			0	50		
3	45			0		38
4			32	36	0	
5		29			21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41				29
1		0	51		32	
2			0	50		
3	45	86		0		38
4			32	36	0	
5		29			21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41	92		73	29
1		0	51		32	
2			0	50		
3	45	86	137	0	118	38
4			32	36	0	
5		29	80		21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41	92	142	73	29
1		0	51	101	32	
2			0	50		
3	45	86	137	0	118	38
4			32	36	0	
5		29	80	130	21	0

8. En clase resolvimos el siguiente problema de optimización: Dado un conjunto de tareas, cada una con un plazo y una ganancia, encontrar un subconjunto de tareas tal que todas las tareas del subconjunto 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 explicando que **basta probar la factibilidad de una sola permutación de las tareas del subconjunto: cualquiera en que las tareas estén ordenadas crecientemente por plazos**.

a) ¿Qué complejidad tiene este algoritmo? Justifica.

Es posible mejorar este desempeño usando 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 t aún no hemos asignado un tiempo de procesamiento, entonces le asignamos el *slot* $[k-1, k]$, en que k es el mayor entero menor o igual que el plazo de t y el *slot* $[k-1, k]$ está vacío —es decir, postergamos la tarea t 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.

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

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