Estructuras de Datos y Algoritmos – IIC2133

Examen

8 julio 2014

1) Supongamos que construimos un grafo direccional acíclico $G = (V, E)$ de la siguiente manera. Los vértices representan tareas que deben ser realizadas, y las aristas representan restricciones de orden entre tareas: una arista (u, v) indica que la tarea u debe realizarse antes que la tarea v. Además, asignamos a *cada vértice* un costo, que representa las unidades de tiempo necesarias para realizar esa tarea.

Una ruta en este grafo representa una secuencia de tareas que deben ser realizadas en un orden particular. Una ***ruta crítica*** es una ruta *más larga*, y corresponde al mayor tiempo necesario para realizar una secuencia ordenada de tareas. El costo de la ruta crítica es una cota inferior para el tiempo total necesario para realizar todas las tareas (típicamente, todas las tareas correspondientes a un mismo proyecto).

Da un algoritmo eficiente para encontrar una ruta crítica en G; en particular,

a) ¿Cómo se puede determinar *eficientemente* las rutas *más cortas* desde un vértice s a todos los demás en el grafo direccional acíclico? ¿Cuál es la complejidad de este algoritmo?

En [Cormen et al., 2009], se demuestra que a partir de la ordenación topológica de los vértices de un grafo direccional acíclico (como vimos en clase), se puede determinar eficientemente las rutas *más cortas* desde un vértice s a todos los demás: después de la ordenación topológica, inicializamos el grafo (con *Init*), y luego, recorriendo cada vértice en orden topológico, reducimos (con *Reduce*) una vez cada arista que sale del vértice. Esto tiene sentido, ya que, si recorro los vértices en orden topológico, una vez que llego al vértice v, he reducido todas las aristas que llegan a él y está garantizado al recorrer los restantes vértices que no voy a volver a v (el grafo es acíclico). Este algoritmo toma tiempo $O(V+E)$. *Init* y *Reduce* son tales que, una vez terminada la ejecución del algoritmo, es posible reconstruir fácilmente cada ruta más corta.

b) ¿Cómo se puede convertir el grafo G en otro, G', en que los costos estén en las aristas? Da un algoritmo de tiempo $O(V+E)$, basado en convertir cada vértice en un trío $< \textit{vértice'}, \textit{arista}, \textit{vértice''} >$.

Asignamos costo 0 a cada arista original; y convertimos cada vértice original en un trío $< \textit{vértice'}, \textit{arista}, \textit{vértice''} >$, en que *vértice'* recibe las mismas aristas que el vértice original, asignamos a la *arista* el costo del vértice original, y desde *vértice"* salen las mismas aristas que del vértice original.

c) ¿Cómo se pueden determinar eficientemente las rutas *más largas* en G'? ¿Cuál es la complejidad de tu algoritmo?

Las rutas *más largas* se pueden determinar similarmente a las rutas más cortas (sólo en grafos acíclicos), negando primero los costos de cada arista, o bien reemplazando primero ∞ por $-\infty$ en *Init* y los "$>$" por "$<$" en *Reduce*.

2) Un árbol rojo-negro 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,

a) [**1 pt.**] Dibuja el árbol original, antes de la inserción de G; deduce los colores de los nodos J, N y W.

J, N y W son rojos. Si alguno fuera negro, entonces se violaría la propiedad de la altura negra del árbol: la ruta Q–D–A, en que A es una hoja, tiene sólo dos nodos negros, y cualquiera de las rutas desde la raíz a J, N o W ya tiene dos nodos negros.

b) [**2 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?

G se inserta en la forma habitual en un árbol de búsqueda, queda como hijo izquierdo de J, y se pinta rojo; pero J es rojo, por lo que ahora hay un nodo y su hijo ambos rojos.

c) [**3 pts.**] 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.

Primero, como el hermano N de J también es rojo, cambiamos colores: J y N quedan negros, y su padre, L, rojo. Pero el padre, D, de L también es rojo; es decir, "subimos" el problema de dos nodos rojos consecutivos.

Pero ahora el hermano, T, de D es negro, por lo que no podemos aplicar el cambio de colores anterior. Como L es hijo derecho de D, rotamos D–L a la izquierda, sin cambiar colores, dejando L como hijo izquiedo de la raíz, y D como hijo izquierdo de L.

De nuevo hay un nodo y su hijo rojos, L y D, y el hermano, T, de L es negro. Pero ahora D es hijo izquierdo de L; por lo que rotamos Q–L a la derecha y pintamos Q de rojo.

3) Dada la secuencia de enteros A_1, A_2, …, A_N, posiblemente negativos, queremos encontrar el valor de la suma de aquella subsecuencia que sume más. P.ej., para la secuencia –2, 11, –4, 13,–5, –2, la respuesta es 20 (= 11 + (–4) + 13).

Da un algoritmo del tipo *dividir para conquistar* para resolver este problema en tiempo O(N log$N$); justifica.

Como sabemos, para que un algoritmo dividir para conquistar corra en tiempo O(N log$N$), normalmente tiene que dividir el problema en dos subproblemas (de la misma naturaleza y) del mismo tamaño y luego, al combinar las soluciones a los subproblemas para encontrar la solución al problema original, tiene que tomar tiempo O(N) (similarmente a *mergeSort*).

En este caso, la subsecuencia que suma más es una que está contenida completamente en la mitad izquierda de la secuencia, o bien una que está contenida completamente en la mitad derecha (hasta aquí, la división), o bien una que empieza en la mitad izquierda y termina en la mitad derecha (esta es la combinación). Buscar la subsecuencia que suma más en cada una de las mitades, izquierda y derecha, se puede hacer recursivamente, y es lo que aporta el factior log$N$ al tiempo de ejecución del algoritmo.

¿Cómo encontramos la subsecuencia que suma más y que cruza desde la mitad izquierda a la derecha? La parte izquierda de esta subsecuencia tiene que ser la subsecuencia de la mitad izquierda que sume más y que incluya el elemento en el extremo derecho de esta mitad; y, análogamente, la parte derecha tiene que ser la subsecuencia de la mitad derecha que sume más y que incluya el elemento en el extremo izquierdo de esta mitad. Ambas subsecuencias se determinan revisando linealmente —O(N)— la mitad correspondiente, de izquierda a derecha y de derecha a izquierda, respectivamente.

4) Tenemos dos strings, $X = x_1x_2\ldots x_m$ e $Y = y_1y_2\ldots y_n$. Consideremos los conjuntos {1, 2, …, m} y {1, 2, …, n} que representan las posiciones (de los símbolos) en los strings X e Y, y consideremos un *emparejamiento* de estos conjuntos, es decir, un conjunto de pares ordenados tal que cada item aparece en a lo más un par. P.ej., si X = `hola` e Y = `ollas`, entonces los conjuntos son {1, 2, 3, 4} y {1, 2, 3, 4, 5}, un emparejamiento podría ser { (2,1), (3,2), (4,4) } (que empareja letras iguales entre ellas), y otro { (1,5), (4,1) } (que empareja letras extremas entre ellas).

Decimos que un emparejamiento M es una *alineación* si no hay pares que "se crucen": si (j, k), (j', k') $\in M$, y si $j < j'$, entonces $k < k'$; p.ej., el primer emparejamiento anterior es una alineación, pero el segundo no. Dado M, un *gap* (brecha) es una posición de X o Y que no está en M; p.ej., para la alineación anterior, la posición 1 de X, y las posiciones 3 y 5 de Y son gaps.

Finalmente, podemos asociar un costo a una alineación: cada gap incurre un costo fijo $\delta > 0$; además, si para cualquier par de símbolos u, v del alfabeto del que provienen X e Y hay un costo $\alpha(u, v)$ de emparejamiento, entonces cada par (i, j) de M tiene un costo $\alpha(x_i, y_j)$. El *costo de la alineación* es la suma de los costos debidos a sus gaps más los costos debidos a sus emparejamientos. P.ej., el costo de la alineación anterior es $3\delta + \alpha($`o`, `o`$) + \alpha($`l`, `l`$) + \alpha($`a`, `a`$)$.

En estas condiciones, dados dos strings, X e Y, queremos encontrar la *alineación de costo mínimo*. Muestra que este problema puede ser resuelto mediante programación dinámica. En particular,

a) Muestra que —explica por qué o explica cómo— el problema puede ser visto como el resultado de una secuencia de decisiones.

En una alineación de costo mínimo M, ya sea (m, n) $\in M$ o (m, n) $\notin M$; en este último caso, ya sea la posición m de X o bien la posición n de Y no aparecen emparejadas en M. Esta última afirmación se demuestra por contradicción: de lo contrario, habría pares "cruzados" y no tendríamos propiamente una *alineación*, según la definición de más arriba.

Si (m, n) $\in M$, entonces pagamos el costo $\alpha(x_m, y_n)$ y alineamos lo mejor posible $x_1\ldots x_{m-1}$ con $y_1\ldots y_{n-1}$; es decir, $\text{opt}(m,n) = \alpha(x_m, y_n) + \text{opt}(m-1, n-1)$.

Si, en cambio, la posición m de X no está emparejada, pagamos el costo δ del gap y alineamos lo mejor posible $x_1\ldots x_{m-1}$ con $y_1\ldots y_n$; es decir, $\text{opt}(m,n) = \delta + \text{opt}(m-1, n)$. Y similarmente si la posición n de Y no está emparejada, obteniendo $\text{opt}(m,n) = \delta + \text{opt}(m,n-1)$.

b) Muestra que —explica por qué o explica cómo— se verifica el *principio de optimalidad*, cuando se aplica al estado del problema que resulta al tomar una decisión.

De la argumentación de **a**), cuando decimos "y alineamos lo mejor posible".

Si M es una alineación óptima y sacamos el par que contiene las posiciones m y/o n, entonces el resto de los pares, M', es una alineación óptima para los demás símbolos de X e Y. Si no fuera así y hubiera una alineación M'' mejor que M' para el resto de los símbolos, entonces podríamos cambiar M' por M'' en M y obtener una alineación mejor que M, contradiciendo que M sea óptima.

c) A partir de **a**) y **b**), plantea una ecuación recursiva para calcular el costo de la alineación de costo mínimo (la ecuación debe estar planteada, entre otros, en términos de los costos de subalineaciones óptimas); incluye las condiciones de borde —es decir, cuando la recursión ya no aplica más— que permiten resolver la ecuación.

Para $i \geq 1$ y $k \geq 1$,

$\text{opt}(i, k) = \min\{ \alpha(x_i, y_k) + \text{opt}(i-1, k-1), \delta + \text{opt}(i-1, k), \delta + \text{opt}(i, k-1) \}$

con $\text{opt}(i, 0) = \text{opt}(0, i) = i\delta$, para todo i (ya que la única forma de alinear una palabra de i letras con una de 0 letras es usar i gaps)

5) Tú quieres conducir un auto desde Santiago hasta Pueto Montt. Cuando el estanque de bencina del auto está lleno, te permite viajar n kilómetros. Tú tienes un mapa con las distancias entre estaciones de bencina en la ruta. Tu propósito es hacer el menor número posible de detenciones en el viaje.

a) Muestra que este problema tiene *subestructura óptima*.

(Suponemos que partimos con el estanque lleno y que las distancias entre pares consecutivos de estaciones de bencina son todas $\leq n$.)

Supongamos que tenemos la solución óptima, y que en esta solución, la primera detención es a los m kilómetros ($m \leq n$); en este punto, que llamaremos R, llenamos el estanque. El resto del recorrido tiene que ser una solución óptima (minimiza el número de detenciones) al problema de viajar desde R (m kilómetros al sur de Santiago) hasta Pueto Montt. La razón es que si no lo fuera, entonces podríamos encontrar otra secuencia de detenciones para viajar desde R hasta Puerto Montt que tuviera menos detenciones, y podríamos usar esta secuencia en el viaje desde Santiago hasta Puerto Montt, después de parar en R, reduciendo el número total de detenciones para este viaje, contradiciendo así la suposición de que la solución que tenemos es óptima.

b) Plantea una solución recursiva.

Dada la propiedad de subestructura óptima, podemos plantear la solución como sigue: Elegir de la mejor manera la próxima estación, y, de allí, encontrar una solución óptima al problema de viajar desde esa estación hasta Pueto Montt.

c) Justifica que en cualquier etapa de la recursión, una de las elecciones óptimas es la *elección codiciosa*.

La elección codiciosa es recorrer la mayor distancia que podamos antes de parar en una estación de bencina (y allí llenar nuevamente el estanque).

Supongamos que acabamos de reanudar el viaje depués de detenernos para llenar el estanque. Supongamos también que las siguientes estaciones son S, …, T, …, U, en orden de cercanía a nuestra posición actual, y que T es la más lejana que no está a más de n kilómetros de distancia. Y supongamos que en una solución óptima la próxima detención es en la estación S (más cercana que T).

Es fácil ver que en esta solución óptima podemos cambiar la elección de detenernos en S por la de detenernos en T, sin aumentar el número total de detenciones: si en la solución óptima, la próxima detención más allá de T es U, a la cual pudimos llegar desde S, entonces también podemos llegar a U desde T, ya que T está más cerca de U que S; y podemos llegar desde nuestra posición actual a T sin detenernos antes, ya que la distancia a T no es mayor que n kilómetros.