Estructuras de Datos y Algoritmos – IIC2133

Examen

25 noviembre 2011

1. Queremos encontrar la primera ocurrencia de un string de k caracteres de largo en otro string de n caracteres de largo, en que $n > k$. Da un algoritmo de tiempo esperado $O(k+n)$ para este problema.

   Podemos aplicar una función de hash al string p, con lo que obtenemos Hp.
   Luego, aplicamos la función de hash a cada secuencia de k caracteres consecutivos del string a, partiendo por la que empieza en a1, siguiendo por la que empieza en a2, luego la que empieza en a3, etc. Si el valor de una de estas funciones es igual a Hp, entonces comparamos p con la secuencia correspondiente, carácter por carácter: si son iguales, paramos; de lo contrario, seguimos con la próxima secuencia.
   Calcular la función de hash del string p y de la secuencia a1…ak toma tiempo O(k) cada uno. Pero una vez que hemos calculado la función de hash de la secuencia a1…ak , calcular la función de hash de la siguiente secuencia toma tiempo O(1), ya que sólo cambian dos caracteres de la secuencia: el primero, o más significativo, y el último, o menos significativo.
   Finalmente, simplemente consideramos que el número esperado de veces que la función de hash de una secuencia sea igual a Hp y que esa secuencia no sea igual al string p es muy pequeño.

2. Supón que tienes n votos para presidente del centro de alumnos, en que cada voto es el número de alumno (un entero) del candidato.

   a) Sin saber quiénes son los candidatos ni cuántos candidatos hay, da un algoritmo de tiempo $O(n \log n)$ para determinar al ganador, usando a lo más $O(n)$ memoria extra.

   Ver b).

   b) Si ahora sabes que hay $k < n$ candidatos, da un algoritmo de tiempo $O(n \log k)$ para determinar al ganador, usando a lo más $O(k)$ memoria extra.

   A medida que vamos revisando los votos, los vamos insertando en un ABBB: cada inserción en un ABBB de (a lo más) k elementos toma tiempo O(logk). Si el número de alumno del voto ya está en el ABBB, entonces no lo insertamos (nuevamente), sino que incrementamos un contador.

3. Justifica que los nodos de cualquier árbol AVL T pueden ser pintados "rojo" y "negro" de manera que T se convierte en un árbol rojo-negro.

Hay justificar que en un árbol AVL la ruta (simple) más larga de la raíz a una hoja no tiene más del doble de nodos que la ruta (simple) más corta de la raíz a una hoja.

Esto es así: el árbol AVL más "desbalanceado" es uno en el que todo subárbol derecho es más alto (en uno) que el correspondiente subárbol izquierdo (o vice versa). En tal caso, el número de nodos en la ruta más larga crece según $2h$, y el de la ruta más corta, según $h+1$, en que h es la altura del árbol

Así, para cualquier ruta (simple) desde la raíz a una hoja, sea d la diferencia entre el número de nodos en esa ruta y el número de nodos en la ruta más corta. Si todos los nodos de la ruta más corta son negros, entonces los otros d nodos deben ser rojos: pintamos la hoja roja y, de ahí hacia arriba, nodo por medio hasta completar d nodos rojos.

4. Supón que construimos un grafo direccional acíclico 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 la tarea correspondiente.

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.

Da un algoritmo eficiente para encontrar una ruta crítica en un grafo direccional acíclico; ¿cuál es la complejidad de tu algoritmo?

En [Cormen et al., 1990] y en [Cormen et al., 2001], 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.
Las rutas más largas se pueden determinar similarmente (sólo en grafos acíclicos), negando primero los costos de cada arista, o bien reemplazando primero ∞ por $-\infty$ en *Init* y los ">" por "<" en *Reduce*.
El único tema pendiente es que antes que todo lo anterior, hay que convertir el grafo descrito más arriba en uno en que los costos estén en las aristas. Esto puede hacerse fácilmente en tiempo $O(V+E)$: asignamos costo 0 a cada arista original; y convertimos cada nodo original en un trío <nodo', arista, nodo">, en que nodo' recibe las mismas aristas que el nodo original, asignamos a la arista el costo del nodo original, y desde nodo" salen las mismas aristas que del nodo original.
El algoritmo pedido ejecuta los pasos anteriores en orden inverso. Primero, convierte el grafo original a uno con los costos en las aristas [**2 pts.**]; luego, niega los costos de las aristas, o cambia ∞ por $-\infty$ y los ">" por "<" [**2 pts.**]; y, finalmente, ejecuta la ordenación topológica, seguida por la inicialización y la serie de reducciones [**2 pts.**].

5. Un sistema está compuesto por n dispositivos conectados en serie; cada uno representa una etapa del sistema. Sea r_i la confiabilidad del dispositivo de la etapa i; entonces, la confiabilidad de todo el sistema es $\prod r_i$.

Para aumentar la confiabilidad del sistema, conviene duplicar dispositivos en las etapas: en cada etapa, conectamos en paralelo varias copias del mismo tipo de dispositivo. Si la etapa i contiene m_i copias de un dispositivo, entonces la probabilidad de que todas las copias fallen es $(1 - r_i)^{m_i}$; y la confiabilidad de la etapa i es $1 - (1 - r_i)^{m_i}$. Ahora, la confiabilidad del sistema es $\prod(1 - (1 - r_i)^{m_i})$.

Por otra parte, sea $c_i > 0$ el costo de cada copia del dispositivo de la etapa i, y sea c el costo máximo permitido para todo el sistema. Entonces, nuestro objetivo es maximizar $\prod(1 - (1 - r_i)^{m_i})$ sujeto a $\sum c_i m_i \leq c$, con m_i un entero ≥ 1, $1 \leq i \leq n$. Observa que el mayor valor posible para m_i es $\lfloor (c + c_i - \sum c_k)/c_i \rfloor$.

a) Muestra que este problema tiene subestructura óptima (satisface el principio de optimalidad).

Una solución óptima es un conjunto de valores para los m_i. La última decisión consiste en elegir el valor de m_n. Una vez elegido m_n, las otras decisiones deben ser tales que los fondos restantes, $c - c_n$, sean usados de una manera óptima para elegir los números de copias de cada dispositivo de las etapas 1, 2, …, n–1 (claramente, si no fuera así, la solución original no sería óptima y tendríamos una contradicción: habría una mejor manera de gastar el monto $c - c_n m_n$, que el que hay actualmente).

b) Plantea una formulación recursiva *top-down* para resolver el problema.

Si la solución óptima es $f_n(c)$ = maximizar, sobre todos los valores posibles para m_n, el producto de la confiabilidad de la etapa n por el valor óptimo de la confiabilidad de las n–1 primeras etapas (considerando un fondo de $c - c_n m_n$), es decir, $f_n(c) = \max_{1 \leq m_n \leq u_n} \{ [1 - (1 - r_n)^{mn}] \, f_{n-1}(c - c_n m_n) \}$, entonces, en general,

$f_k(x) = \max_{1 \leq m_k \leq u_k} \{ [1 - (1 - r_k)^{mk}] \, f_{k-1}(x - c_k m_k) \}$.

6. Sea A un arreglo de n claves, en que cada una es un número entero en el rango $[1, n]$. Queremos decidir si hay claves repetidas en A.

a) ¿Cómo se puede hacer esto en tiempo $O(1)$ en un computador PRAM CRCW de n procesadores? ¿Qué versión de PRAM CRCW habría que usar?

Todas las celdas de memoria están inicialmente en 0. Cada procesador lee un dato de A: el procesador k lee $A[k]$. Luego, cada procesador escribe un 1 en la celda de memoria correspondiente al valor que leyó: el procesador k escribe un 1 en la celda $M[A[k]]$. CRCW común es suficiente, ya que todos los procesadores escriben el mismo valor. Finalmente, cada procesador lee su celda de memoria; si algún procesador lee un 0, entonces había claves repetidas en A.

b) Supón ahora que las claves son números enteros en el rango $[0, n^c]$, en que c es constante. Muestra cómo resolver el problema en tiempo $O(1)$ usando n procesadores PRAM CRCW y $O(n^c)$ memoria.