Estructuras de Datos y Algoritmos – IIC2133

I2 - pauta

5 octubre 2011

1. Sea $a[1 \ldots n]$ un arreglo de n números distintos. Si $j < k$ y $a[j] > a[k]$, entonces el par (j, k) se llama una inversión de a.

a) ¿Cuál es exactamente el máximo número de inversiones que puede tener un arreglo de n números distintos? Justifica.

Este caso se da cuando el arreglo está ordenado justamente "al revés", es decir, de mayor a menor; entonces, hay una inversión por cada par de elementos: (1, 2), (1, 3), …, (1, n), (2, 3), (2, 4), …, (2, n), …, (n–1, n). Así, hay exactamente $(n–1) + (n–2) + \ldots + 1 = n(n–1)/2$ inversiones.

b) Justifica que cualquier algoritmo de ordenación que ordena intercambiando elementos adyacentes —por ejemplo, ordenación por inserción (*insertionSort*)— toma tiempo $\Omega(n^2)$ en promedio para ordenar n elementos.

Al intercambiar dos elementos adyacentes, sólo resolvemos una inversión. Por lo tanto, hay que justificar que un arreglo tiene $\Omega(n^2)$ inversiones en promedio. En realidad, hay $n(n\text{-}1)/4$ inversiones en promedio: tomemos un arreglo $a[1]$, $a[2]$, …, $a[n]$ y su inverso $a[n]$, $a[n\text{-}1]$, …, a[1]; el par (j, k) será una inversión en uno o en el otro; luego, cada dos arreglos, se dan todas las inversiones posibles, es decir, $n(n\text{-}1)/2$ inversiones.

2. Supón que queremos usar *Quicksort* para ordenar un arreglo con muchas claves duplicadas; por ejemplo, ordenar el archivo de alumnos de la universidad por año de nacimiento. En este caso, el desempeño de *Quicksort* puede mejorarse bastante; por ejemplo, un subarreglo que tiene sólo datos iguales (todos con la misma clave) no necesita seguir siendo procesado, pero *Quicksort* va a seguir particionándolo en subarreglos cada vez más pequeños.

Una idea es particionar el arreglo en tres partes: las claves que son menores que el pivote, las que son iguales al pivote, y las que son mayores que el pivote.

a) [**4 pts.**] Da un algoritmo, llamado *Partition3*, de complejidad $O(n)$ —es decir, equivalente al *Partition* estudiado en clase— para realizar esta nueva partición en tres (sobre un subarreglo de n datos).

La idea es que se hace una pasada por el arreglo $a[e \,..\, w]$ de izquierda a derecha que mantiene tres punteros:
- un puntero lt tal que $a[e \,..\, lt\text{-}1]$ es menor que el pivote
- un puntero gt tal que $a[gt\text{+}1 \,...\, w]$ es mayor que el pivote
- un puntero k tal que $a[lt \,...\, k\text{-}1]$ es igual al pivote
- $a[k \,...\, gt]$ aun no han sido examinados

> ***Partition3***:
> Se elige el pivote $v = a[e]$, y los punteros $lt = e$, $gt = w$
> Recorremos el arreglo con k, a partir de $k = e\text{+}1$ y hasta $k = gt$:
> - si $a[k] < v$, intercambiamos $a[lt]$ con $a[k]$; incrementamos lt y k
> - si $a[k] > v$, intercambiamos $a[k]$ con $a[gt]$; decrementamos gt
> - si $a[k] = v$, incrementamos k

b) [**2 pts.**] Escribe la nueva versión de *Quicksort*, que hace uso de *Partition3*.

```
Quicksort(a, e, w):
if ( e < w )
    —aquí va el código de Partition3
    Quicksort(a, e, lt-1)
    Quicksort(a, gt+1, w)
```

3. Considera el problema de ordenar topológicamente un grafo direccional acíclico (DAG).

a) [**2 pts.**] Justifica que todo DAG tiene al menos una ***fuente*** (un vértice al que no "llega" ninguna arista) y al menos un ***sumidero*** (un vértice del que no "sale" ninguna arista).

Respuesta: Demostramos la existencia del sumidero. Parémonos en un vértice cualquiera, v_0, y recorramos una arista que sale de 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 salía). Si en este punto el vértice actual, v_n, tuviera una arista que sale, ésta sólo podría ir a alguno de los vértices ya visitados, formando un ciclo; como el grafo es acíclico, este último vértice no puede tener ninguna arista que salga.

La demostración de la existencia de la fuente es análoga (recorremos las aristas "hacia atrás").

b) [**Se puede resolver sin haber hecho la demostración pedida en a)**] Considera el siguiente algoritmo de ordenación topológica: Identifica un vértice fuente, 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:

i) [**1 pt.**] ¿Cómo identificamos los vértices fuente la primera vez?

Respuesta: Supongamos que el garfo está representado por sus listas de adyacencias. Sea x un arreglo que cuenta el número de aristas que llegan a cada vértice; inicializamos los elementos de x en 0. Ahora recorremos las listas de adyacencias una a una; cada vez que llegamos a un vértice v, aumentamos 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 $O(V+E)$.

ii) [**1 pt.**] ¿Dónde guardamos los vértices fuente que vamos identificando a lo largo de la ejecución del algoritmo?

Respuesta: Cada vez que procesamos un vértice fuente (esto es, le asignamos 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 **iii**). ¿Cómo hacemos para identificarlos y procesarlos? Una posibilidad es que cada vez que vamos a identificar (y procesar) un nuevo vértice fuente recorramos todo el arreglo x buscando elementos que estén en 0 pero 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 **i**). 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.

iii) [**2 pts.**] ¿Cómo identificamos los (nuevos) vértices fuente en el grafo que queda después de que sacamos un vértice fuente?

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

4. Considera el algoritmo de Kruskal para encontrar un árbol de extensión de costo mínimo (MST) para un grafo direccional $G = (V, E)$ en que cada arista tiene asociado un costo. Si los costos de todas las aristas de G son números enteros en el rango de 1 a $|V|$, ¿qué tan rápido se puede hacer la ejecución del algoritmo de Kruskal? Recuerda que el algoritmo de Kruskal toma tiempo $O(V)$ para inicialización, $O(E \log E)$ para ordenar las aristas, y $O(E\ \alpha(V))$ para las operaciones de conjuntos disjuntos.

Respuesta. Primero, el tiempo total del algoritmo de Kruskal 'básico' es $O(V) + O(E \log E) + O(E\ \alpha(V)) = O(E \log E)$. Si ahora los costos de todas las aristas son números enteros en el rango de 1 a $|V|$, entonces podemos ordenarlas en tiempo $O(V + E)$ —en lugar de $O(E \log E)$— usando *countingSort*; además, como G es conexo, $V = O(E)$, y el tiempo de ordenación se reduce a $O(E)$. Esto da un tiempo total para el algoritmo de $O(V) + O(E) + O(\mathrm{E}\ \alpha(V)) = \boldsymbol{O(E\ \alpha(V))}$ —de nuevo, ya que $V = O(E)$. (Es decir, el término dominante cambia de ser el tiempo que toma ordenar las aristas al tiempo que toma procesarlas una vez ordenadas. Además, sólo la ordenación de las aristas se hace más rápido, porque ninguna otra parte del algoritmo usa los costos de las aristas.)