Estructuras de Datos y Algoritmos – IIC2133

Examen

28 de noviembre de 2018

1. Árboles binarios de búsqueda

Las claves en un árbol binario de búsqueda están ordenadas. Por lo tanto, debería ser fácil encontrar la k-ésima clave más pequeña o determinar cuántas están en el rango $[a, b]$. Explica cómo podría hacerse:

a) ¿Qué información adicional habría que mantener en cada nodo?

En cada nodo x podemos almacenar en un campo adicional $x.size$ el número total de nodos que tiene el árbol cuya raíz es x, incluyendo a x. Así, p.ej., una hoja h tiene $h.size = 1$, y si el árbol almacena 1000 claves, entonces la raíz r del árbol tiene $r.size = 1000$.

b) Describe los algoritmos necesarios para responder las consultas mencionadas más arriba.

Suponemos que cada nodo x tiene punteros *left*, *right* y p, a sus hijos izquierdo y derecho y a su padre.

```
k-esima(x, k):                    —esta operación es O(altura del árbol)
    r = x.left.size + 1
    if k = r:
        return x
    else: if k < r:
        return k-esima(x.left, k)
    else:
        return k-esima(x.right, k–r)
```

Para determinar cuántas claves están en el rango $[a, b]$, hay que saber qué posición ocupan a y b en un orden lineal de todas las claves (recorrido *inorder* del árbol T) —esto es, el ranking de a y el ranking de b— y luego restar, ranking(b, T) – ranking(a, T):

```
ranking(x, T):                    —esta operación también es O(altura del árbol)
    r = x.left.size + 1
    y = x
    while y ≠ T.root:
        if y == y.p.right:
            r = r + y.p.left.size + 1
        y = y.p
    return r
```

c) ¿Cuánto cuesta mantener la información adicional? En particular, si se hace una inserción en el árbol; y si se hace una eliminación. Justifica.

En ambos casos, el costo es O(altura del árbol). En una inserción, hay que incrementar $x.size$ para cada nodo x en el camino desde la raíz al punto de inserción; el nuevo nodo (que es una hoja) tiene $size = 1$. En una eliminación, si eliminamos una hoja o un nodo con un solo hijo, hay que decrementar $x.size$ para cada nodo x en el camino desde el padre del nodo eliminado hasta la raíz; si eliminamos un nodo con dos hijos, primero lo reemplazamos por su sucesor y luego eliminamos el sucesor (de su posición original).

2. **Ordenación**

Queremos ordenar n datos, que vienen en un arreglo a, en que cada dato es un número entero en el rango 0 a k. Considera el siguiente algoritmo:

```
countSort(a, n, k):
    sea count[0 … k] un arreglo de enteros
    for i = 0 … k: count[i] = 0
    for j = 0 … n–1: count[a[j]] = count[a[j]] + 1
    for i = 0 … k:
        if count[i] > 0:
            for p = 1 … count[i]: print(i)
```

a) ¿Cuál es el *output* del algoritmo?

Una lista con los n números enteros de a, pero ordenados de menor a mayor.

b) ¿Cuál es la complejidad del algoritmo en función de n y k?

El primer **for** i es O(k); el **for** j es O(n); y el siguiente **for** i es O(n), ya que esencialmente imprime los n datos de a, una vez cada uno. **Luego, el algoritmo es O($k + n$)**.

c) Si k es O(n), ¿cuál es la complejidad del algoritmo?

O(n).

d) Esto pareciera contradecir la demostración que vimos en clase de que los algoritmos de ordenación por comparación tienen complejidad O(n log$n$). Explica por qué no hay una contradicción.

Efectivamente, es un algoritmo de ordenación, **pero no ordena por comparación**: en ninguna parte del algoritmo aparece una condición tal como a[i] < a[j]. (O tal vez pdría argumentarse que la asignación *count*[*a*[j]] = *count*[*a*[j]] + 1 hace una comparación de un valor contra otros n–1 en tiempo O(1).)

3. Algoritmos codiciosos

Supongamos que quieres hacer una caminata por un parque nacional, que te va a tomar varios días. Tu estrategia es caminar lo más posible de día, pero acampar cuando oscurece. El mapa proporcionado por la oficina de turismo muestra muchos buenos lugares para acampar, y tú has decidido que cada vez que llegues a uno de estos lugares vas a calcular (correctamente) si alcanzas a llegar al próximo antes de que oscurezca; en caso afirmativo, sigues; de lo contrario, te detienes y acampas.

Argumenta rigurosamente que esta estrategia minimiza el número de detenciones para acampar que vas a tener que hacer.

Por contradicción.

Primero, algunos supuestos y definiciones. Sean L la longitud total del camino, d la distancia que puedes caminar en un día, y $x_1, x_2, \ldots x_n$ los puntos de detención que muestra el mapa (distancias desde la entrada del camino). Decimos que un conjunto de puntos de detención es *válido* si
- la distancia entre cualquier par de puntos adyacentes (en el conjunto) es a lo más d,
- la distancia desde la entrada al primer punto es a lo más d, y
- la distancia desde el último punto a la salida del camino es a lo más d.

Sean $R = \{x_{p1}, x_{p2}, \ldots, x_{pk}\}$ el conjunto (válido) de puntos de detención elegidos por tu estrategia codiciosa, y $S = \{x_{q1}, x_{q2}, \ldots, x_{qm}\}$, con $m < k$, un conjunto válido de menos puntos que R.

Primero, los puntos de R están más lejos que los puntos de S; es decir, para cada $j = 1, 2, \ldots, m$, tenemos que $x_{pj} \geq x_{qj}$, lo que demostramos por inducción sobre j.

> Para $j = 1$, sale de la definición de la estrategia codiciosa: tú viajas lo más posible el primer día antes de detenerte.
>
> Para $j > 1$ y suponiendo que la afirmación es válida para todo $i < j$: $x_{qj} - x_{qj-1} \leq d$, y también $x_{qj} - x_{qj-1} \geq x_{qj} - x_{pj-1}$, lo que implica que $x_{qj} - x_{pj-1} \leq d$. Es decir, una vez que tú sales de x_{pj-1}, podrías caminar hasta x_{qj} en un día, por lo que x_{pj}, que es donde finalmente vuelves a detenerte, solo puede estar más lejos que x_{qj}.

Segundo, como $m < k$, entonces $x_{pm} < L - d$; de lo contrario no tendría sentido que te detengas en x_{pm+1}. Como además $x_{qm} \leq x_{pm}$, por la demostración anterior, resulta que $x_{qm} < L - d$, lo que contradice que S sea un conjunto válido de puntos de detención.

4. Programación dinámica

En clases estudiamos el problema de programar el mayor número de tareas en una misma máquina, en que cada tarea k tiene una hora de inicio s_k y una hora de finalización f_k, y la máquina solo puede ejecutar una tarea a la vez. Vimos que el problema se puede resolver usando un algoritmo codicioso en tiempo O(n), si hay n tareas en total y vienen ordenadas por sus horas de finalización.

Considera ahora que cada tarea tiene, además, un valor v_k, y queremos maximizar la suma de los valores de las tareas realizadas (y no el número de tareas realizadas).

a) Muestra que el algoritmo codicioso anterior no resuelve esta versión más general del problema.

Bssta un ejemplo:

k: s_k–f_k, v_k

1: 3–10, 1
2: 5–15, 3
3: 12–20, 1

En este caso, el algoritmo codicioso elige las tareas 1 y 3, con un valor total de 1+1 = 2; pero la elección de la tarea 2 habría producido un valor de 3.

Muestra que el problema se puede resolver mediante programación dinámica. En particular, suponemos nuevamente que las tareas vienen ordenadas por sus horas de finalización: $f_1 \leq f_2 \leq \ldots \leq f_n$. Entonces, para cada tarea j, definimos b[j] como la tarea g que termina más tarde antes del inicio de j; b[j] = 0 si ninguna tarea satisface esta condición.

Sea T es una solución óptima al problema. Revisamos las tareas "de atrás hacia adelante": obviamente, la tarea n pertenece a T, o bien la tarea n no pertenece a T.

b) Argumenta clara y precisamente que en cada una de estas dos situaciones el problema puede resolverse a partir de encontrar la solución óptima a un problema del mismo tipo pero más pequeño.

Si la tarea n no pertenece a T, entonces T es igual a la solución óptima para las tareas 1, …, n–1; es decir, un problema del mismo tipo pero más pequeño.

En cambio, si la tarea n pertenece a T, entonces ninguna tarea q, b[n] < q < n, puede pertenecer a T, por lo que T debe incluir, además de la tarea n, una solución óptima para las tareas 1, …, b[n]; es decir, nuevamente, un problema del mismo tipo pero más pequeño.

c) Generalizando, podemos decir que si T_j es la solución óptima al problema de las tareas 1, …, j, y su valor es OPT(j), entonces el problema original consiste en buscar T_n y encontrar su valor OPT(n). Escribe una ecuación recursiva para encontrar OPT(j), en función de v_j, b[j] y j–1.

La generalización se traduce a

si j pertenece a T_j , entonces OPT(j) = v_j + OPT(b[j])

si j no pertenece a T_j , entonces OPT(j) = OPT(j–1)

Por lo tanto, **OPT(j) = max{ v_j + OPT(b[j]) , OPT(j–1) }** … (falta agregar condición de borde)

d) Explica claramente cuál sería la dificultad práctica de tratar de resolver el problema aplicando directamente la ecuación recursiva anterior.

La ecuación recursiva hace dos llamadas recursivas, pero (a diferencia de *quicksort* o *mergesort*, en que las llamadas recursivas son sobre conjuntos disjuntos de datos) estas pueden ser llamadas para resolver el mismo problema, o para resolver problemas que ya han sido resueltos; es decir, las llamadas son sobre conjuntos de datos no necesariamente disjuntos y, por lo tanto, pueden terminar resolviendo un mismo problema muchas veces. Así, **el número total de problemas efectivamente resueltos puede ser mucho mayor que el número total de problemas diferentes**.

e) Explica precisamente cómo la programación dinámica ayuda a resolver la dificultad de d).

La P.D. parte resolviendo los problemas más pequeños (problemas que no dan origen a llamadas recursivas) primero y va almacenando estos problemas (convenientemente codificados) y sus resultados en una tabla. Así, cuando aparece un problema para resolver, primero se busca en la tabla a ver si ese problema ya está resuelto, en cuyo caso se usa el resultado que está en la tabla y no se vuelve a resolver el problema.

5. Rutas disjuntas

Un conjunto de rutas es disjunto si sus conjuntos de aristas son disjuntos, es decir, ningún par de ru-tas comparte una arista (aunque muchas rutas pueden pasar por algunos de los mismos nodos).

Dado un grafo direccional acíclico $G = (V, E)$, en que un nodo s no tiene aristas que llegan a él, y otro nodo t no tiene aristas que salen de él, el problema de las rutas direccionales disjuntas es encontrar el número máximo de rutas disjuntas de s a t en G.

El problema puede resolverse usando flujos en redes. Primero, construimos una red de flujo a partir de G: s y t son la fuente y el sumidero, respectivamente, y cada arista tiene capacidad 1.

a) Demuestra que si hay k rutas disjuntas en el grafo original G, entonces el valor del flujo máximo de s a t en la red construida a partir de G es a lo menos k.

   Podemos hacer que cada una de las rutas disjuntas lleve una unidad de flujo: el flujo en una arista vale 1 si la arista pertenece a una de las rutas, y vale 0 para todas las otras aristas. Este flujo cumple con las condiciones de (i) no sobrepasar las capacidades de las aristas, y (ii) no producir ni acumular flujo en los nodos distintos de s y t.

b) ¿Qué otra propiedad es necesario poder demostrar para poder concluir que el valor del flujo máximo en la red corresponde al número de rutas disjuntas en G? Solo enuncia la propiedad, de manera clara y precisa; no se pide que la demuestres.

   En a) demostramos que si hay k rutas disjuntas, entonces el flujo es a lo menos k. Por lo tanto, lo que falta por ser demostrado es que **si en la red hay un flujo de valor k, entonces en G hay k rutas disjuntas**.

c) ¿Qué propiedad del problema del flujo máximo da pie para la siguiente afirmación: "El número máximo de rutas disjuntas de s a t en G es igual al número mínimo de aristas (de G) que es necesario sacar para separar s de t (es decir, para que no queden rutas de s a t en G)."?

   La propieded de que el flujo máximo es igual a la capacidad del corte de capacidad mínima (*max-flow min-cut*).