Estructuras de Datos y Algoritmos – IIC2133

Examen

25 junio 2013

A) En clase estudiamos los métodos *buildHeap*(*b*), que construye un **max-heap binario** a partir de un arreglo *b*, e *insertObject*(*x*), que inserta un nuevo elemento *x* en un max-heap. Además, en lugar de llamar a *buildHeap*(*b*), podríamos construir un max-heap llamando repetidamente a *insertObject*( ), así:

```
buildHeap'(b):
    heapSize = 1
    for i = 2 to b.length
        insertObject(b[i])
```

1) ¿Cuál es la complejidad de *buildHeap*( ), *insertObject*( ), y *buildHeap'*( ), en función del número $n = b.length$ de elementos procesados? Justifica.

buildHeap( ) hace O(n) operaciones. Este método en la práctica ejecuta **for** $j = n/2$ **downto** 1 *heapify*(j), es decir, hace *heapify*( ) sobre $n/2$ elementos. Cada *heapify*( ) hace un número constante de operaciones más una llamada recursiva. La mitad de estos *heapify*( ), es decir, $n/4$, sólo trabaja en un nivel; en la práctica, no desciende recursivamente. La mitad de los $n/4$ restantes, es decir, $n/8$, descienden recursivamente sólo un nivel; y así sucesivamente.

insertObject(( ) hace O(log$n$) operaciones. Este método coloca el nuevo objeto al final de un heap ya formado, y luego tiene que ubicarlo correctamente según su clave. Para esto, repetidamente, hace un número constante de operaciones en el nivel del árbol en el que está actualmente, y luego sube un nivel; así, hasta llegar a la raíz del max-heap. Hay O(log$n$) niveles en el árbol.

buildHeap'( ) hace O($n$log$n$) operaciones, ya que ejecuta n veces *insertObject*( ).

2) Con respecto a *buildHeap*( ) y *buildHeap'*( ), ¿construyen el mismo max-heap cuando son ejecutados sobre el mismo arreglo *b* de entrada? Demuestra que es así, o da un contraejemplo.

En general, no construyen el mismo heap. P.ej., si el arreglo original tiene los elementos [1, 2, 3], uno construye el heap [3, 1, 2]; y el otro, el heap [3, 2, 1].

3) [*Independiente de las anteriores*] ¿Cuál es el mejor caso para *hespSort*( ) y qué tan bueno es? Explica.

Cuando todos los elementos son iguales, *heapify*( ) no desciende recursivamente desde la raíz, y por lo tanto *heapSort*( ) corre en tiempo O(n).

B) ¿Cuánto cuesta y cómo hay que hacerlo para comprobar si una estructura de datos dada es un **árbol** (de búsqueda binario) **rojo-negro**? En particular,

4) ¿Cuál es la propiedad de árbol de búsqueda binario?
Cualquier clave en el subárbol izquierdo es menor que la clave en la raíz, que a su vez es menor que cualquier clave en el subárbol derecho

5) Escribe un método *esABB*( ), que reciba un nodo como parámetro y devuelva *true* si el nodo es la raíz de un árbol de búsqueda binario; *false*, en otro caso.

Lo más simple puede ser escribir un método recursivo:

```
esABB(x):
    if ( null(x) ) return true
    if ( hoja(x) ) return true
    if ( esABB(x.derecho) && esABB(x.izquierdo) )
        if ( null(x.derecho) ) return x.key < x.izquierdo.key
        if ( null(x.izquierdo) ) return x.derecho.key < x.key
        return x.derecho.key < x.key && x.key < x.izquierdo.key
    else return false
```

6) ¿Qué propiedades adicionales debe cumplir un árbol rojo-negro?

i) Todo nodo es ya sea rojo o negro.
ii) Si un nodo es rojo, entonces sus hijos son negros.
iii) Para cada nodo, todas las rutas desde el nodo a las hojas descendientes contienen el mismo número de nodos negros.

7) Escribe un método *esRojo-Negro*( ), que reciba un nodo como parámetro y devuelva *true* si el nodo es la raíz de un árbol de búsqueda binario rojo-negro; *false*, en otro caso.

Tal vez, lo más simple es escribir dos métodos recursivos, uno para verificar ii) y el otro para verificar iii); cualquiera de ellos puede, además, verificar i):

```
esRojo-Negro(x):
    if ( esABB(x) ) return cumple-ii(x) && cumple-iii(x)!=-1
    else return false

cumple-ii(x): —también verifica i)
    if ( null(x) ) return true
    if ( hoja(x) ) return es rojo o negro
    if ( es negro ) return cumple-ii(x.derecho) && cumple-ii(x.izquierdo)
    if ( es rojo )
        if ( null(x.izquierdo) ) return cumple-ii(x.derecho) && x.derecho es negro
        if ( null(x.derecho) ) return cumple-ii(x.izquierdo) && x.izquierdo es negro
        return cumple-ii(x.derecho) && x.derecho es negro && cumple-ii(x.izquierdo) && x.izquierdo es negro
    else return false

cumple-iii(x)
    if( null(x) ) return 0
    if( es rojo && hoja(x) ) return 0
    if( es negro && hoja(x) ) return 1
    p = cumple-iii(x.izquierdo)
    q = cumple-iii(x.derecho)
    if (p!=q) return -1
    if(p== -1 || q==-1) return -1
    if( es rojo ) return p
    if( es negro) return p+1
```

8) [*Independiente de las anteriores*] Un caso particular de árboles-B son los árboles 2-4, en que un nodo puede tener entre 1 y 3 claves y entre 2 y 4 hijos (árbol-B con $t = 2$). Supongamos que hacemos lo siguiente: Los nodos con 2 claves los separamos en dos nodos, con una clave cada uno, en que el nodo con la clave mayor queda como padre del nodo con la clave menor, y distribuimos los tres hijos del nodo original entre estos dos nodos, de modo que ambos queden con dos hijos cada uno. Los nodos con 3 claves los separamos en tres nodos, con una clave cada uno, en que el nodo con la clave central queda como padre de los otros dos, y distribuimos los cuatro hijos del nodo original entre los dos nuevos nodos hijos, de modo que los tres nuevos nodos quedan con dos hijos cada uno. Justifica que el nuevo árbol es un árbol rojo-negro.

Aquí hay que partir por la propiedad de los árboles-B, de que todas las hojas están a la misma profundidad (desde la raíz); es decir, todas las rutas desde la raíz hasta una hoja tienen el mismo número de nodos. No es casualidad que esta propiedad se "parezca" mucho a la propiedad iii), en 6). Por lo tanto, estos nodos, o lo que quede de ellos después de las separaciones sugeridas en el enunciado, serán los nodos negros del nuevo árbol: el mismo número de ellos en cada ruta desde la raíz hasta una hoja.

En el caso de los nodos con dos o tres claves, que en el nuevo árbol separamos en un padre y uno o dos hijos, el nodo que queda como padre mantiene el color negro; los nuevos nodos hijos de este padre negro (uno o dos, que compartían con él el nodo original) serán los nodos rojos del árbol rojo-negro. Es fácil ver que los hijos de estos nodos rojos son negros, tal como lo exige la propiedad ii), en 6).

C) Con respecto a los árboles de cobertura de costo mínimo (MST),

9) Si todos los costos de las aristas son números enteros en el rango 1 a $|V|$, ¿qué tan rápido, en notación O( ), se puede hacer que corra el algoritmo de Kruskal? Toma en cuenta que el algoritmo incluye una inicialización, una ordenación, y finalmente la ejecución del algoritmo propiamente tal.

Kruskal toma O(V) para inicialización, O(E log E) para ordenar las aristas, y O(E $\alpha(V)$) para las operaciones de conjuntos disjuntos (la ejecución del algoritmo propiamente tal); por lo tanto, Kruskal es O(E log E). Ahora, bajo el supuesto de arriba y usando countingSort, podemos ordenar las aristas en O($V + E$) = O(E) —**ya que V = O(E)**. De modo que ahora Kruskal es O(E $\alpha(V)$).

10) Si todos los costos de las aristas son números enteros en el rango 1 a W, en que W es una constante, ¿qué tan rápido, en notación O( ), se puede hacer que corra el algoritmo de Kruskal? Toma en cuenta que el algoritmo incluye una inicialización, una ordenación, y finalmente la ejecución del algoritmo propiamente tal.

Kruskal toma O(V) para inicialización, O(E log E) para ordenar las aristas, y O(E $\alpha(V)$) para las operaciones de conjuntos disjuntos (la ejecución del algoritmo propiamente tal); por lo tanto, Kruskal es O(E log E). Ahora, bajo el supuesto de arriba y usando countingSort, podemos ordenar las aristas en O($W + E$) = O(E) —**ya que W es constante**. De modo que ahora Kruskal es O(E $\alpha(V)$).

11) Si todos los costos de las aristas son distintos, muestra que el MST es único.

Como todos los costos de las aristas son distintos, para cada corte hay una única arista liviana. A partir de las diapositivas #47 a 49, deducimos que el MST es único.

Suponga que el MST no es único, es decir, existen dos MSTs T y T' con T distinto de T'. Considere "e" la arista de menor peso de T que no aparece en T'. A su vez, considere e' la arista de menor peso de T' que no aparece en T. Sin pérdida de generalidad, suponga que "e" tiene menor peso que e'. Entonces, necesariamente se tiene que T' unido con {e} forma un ciclo C. Considere S=C-E(T) el conjunto de todas las aristas del ciclo que no aparecen en T. Notar que todas las aristas de S tienen estrictamente mayor peso que "e", pues todas las aristas son distintas y "e" es la arista con menor peso que no aparecen en ambos árboles T y T'. Sea r una arista de S y se tendrá que T" = T' U {e} – {r} forma un árbol de cobertura. Dado que se intercambió una arista más pesada por otra de menor peso, se tiene que T" tiene menor peso que T', lo que contradice con el hecho de que T' es MST. Luego, debe ocurrir que si todos los costos de las aristas son distintos, el MST es único.

12) Si todos los costos de las aristas son distintos, muestra que el ***segundo mejor*** MST puede no ser único.

Basta con dar un ejemplo.

D) Considera el problema de determinar las **rutas más cortas entre todos los pares de vértices** de un grafo direccional $G = (V, E)$ con costos en las aristas.

13) Enuncia las dos propiedades características de los problemas de optimización que pueden ser resueltos mediante programación dinámica.

Las dos propiedades características son: Propiedad de Subestructura óptima y Propiedad de subproblemas traslapados.

- Subestructura óptima: La solución óptima al problema original puede ser construida con soluciones óptimas de subproblemas.
- Subproblemas traslapados: Si uno utiliza un algoritmo recursivo para resolver el problema, entonces existirá al menos un subproblema que será resuelto varias veces.

14) Muestra que este problema cumple ambas propiedades.

Para la propiedad de subestructura óptima, suponga que p(u,v) es el camino de menor costo desde u hasta v. Sea x el nodo inmediatamente anterior a v en el camino p (por lo que (x,v) es una arista). Entonces suponga que p(u,x) no es el camino más corto desde u hasta x. Entonces existe otro camino p'(u,x) que sí es óptimo. Luego, agregue la arista (x,v) a este nuevo camino. Y por lo tanto, tenemos que:

$$w(p'(u,x))+w(x,v) < w(p(u,x))+w(x,v)$$

Por lo que el camino p' seguido por (x,v) es mucho mejor que el camino p(u,v) original. Luego, p(u,v) no es óptimo y hay una contradicción por suponer que no tiene subestructura óptima. Por lo tanto, este problema exhibe subestructura óptima.

Así, para ir desde u hasta v, considere la siguiente recursión, donde d(x,y) es el costo del camino más corto desde x hasta y:

d(x,x) = 0
d(x,y) = w(x,y) si (x,y) es una arista
d(x,y) = +infinito si y no es alcanzable desde x
d(x,y) = min{d(x,u)+d(u,y) | u está en V}

Para la propiedad de subproblemas traslapados, suponga que quiere encontrar el valor de d(x,y) y para ello, necesita encontrar el valor de d(x,u) y d(u,y). También necesita encontrar el valor de d(x,a) y d(a,y). Para encontrar el valor de d(x,u) necesita de d(x,a) y d(a,u). Luego tiene que calcular al menos dos veces el valor de d(x,a) y este problema tiene subproblemas traslapados.

15) Plantea un algoritmo eficiente para resolverlo; justifica que tu algoritmo es eficiente.

(No basta con decir que el algoritmo corresponde al de Floyd-Warshall, debe plantearse el algoritmo)

Suponga que tenemos una matriz W donde W(i,j) indica el costo de ir desde el nodo i hasta el nodo j. Sea D otra matriz, inicialmente vale D=W y que tenemos N nodos. El algoritmo es:

```
D=W;
for k=1..N{
  for i=1..N{
    for j=1..N{
      if(D(i,j)>D(i,k)+D(k,j))
        D(i,j) = D(i,k)+D(k,j)
    }
  }
}
```

El algoritmo toma tiempo O(n^3). Es más eficiente que ejecutar N veces Dijkstra (que toma tiempo O((V+E)log V)) y más eficiente que ejecutar N veces Bellman-Ford (que toma tiempo O(VE))

16) El ***diámetro*** de G es la más larga de las rutas más cortas en G. A partir de tu algoritmo para 15), da un algoritmo eficiente que imprima la *secuencia de vértices* del diámetro de G.

Si uno simplemente ejecuta el algoritmo anterior, obtiene una matriz D con las distancias más cortas entre todos los pares de vértices; el mayor valor en esta matriz es la magnitud del diámetro del grafo; pero falta la secuencia de vértices.

Para esto, entre otras posibilidades, [Cormen et al., 2001] sugiere construir una matriz P iterativamente, similarmente y en paralelo a la de las distancias más cortas, pero que represente las rutas propiamente tales. Si en la iteración k me conviene incluir el vértice k para ir de u a v, entonces $P[u, v] = P[k, v]$ de la iteración k–1; de lo contrario, $P[u, v] = P[u, v]$ de la iteración k–1. Así, al finalizar Floyd-Warshall, $P[u, v]$ contiene el vértice anterior a v en la ruta más corta de u a v.

Finalmente, buscamos en D la más larga de las distancias más cortas, y usamos P para reconstruir la ruta correspondiente.

E) Una ***cola concurrente*** es una estructura de datos lineal con capacidad n, con dos extremos —en que los datos son ingresados por un extremo y son extraídos por el otro extremo (en el orden en que fueron ingresados)— y que permite que haya ingresos y extracciones concurrentemente: varios procesos están ingresando datos y varios procesos están extrayendo datos "al mismo tiempo".

17) Describe la implementación de la estructura de datos propiamente tal.

P.ej., un arreglo con n casillas, con un puntero *front* que almacena el índice del primer dato en la cola, y otro *tail*, que almacena el índice de la casilla siguiente al último dato en la cola; ambos punteros se incrementan en 1, *front* cuando se extrae un dato y *tail* cuando se ingresa otro (si los índices de las casillas van de 0 a n–1, entonces los incrementos son módulo n, para usar el arreglo "circularmente"). Se puede agregar un contador de datos puestos en la cola, para saber cuando está vacía y cuando está llena (o se puede usar alguna relación entre *front* y *tail*).

18) Describe las sincronizaciones que deben existir entre los procesos para que la cola funcione correctamente: no se puede ingresar un dato, si la cola está llena, ni extraer un dato, si la cola está vacía.

-Hay que asegurarse de que un proceso no extraiga un dato si la cola está vacía, ni ingrese un dato si la cola está llena.
-A lo más, un proceso a la vez puede ingresar datos a la cola. De manera similar, a lo más un proceso a la vez puede extraer datos de la cola.

19) Implementa las sincronizaciones de 18) usando *spin locks* o bien *semáforos*.

Una solución posible con semáforos es la siguiente: Considere 4 semáforos: empty inicializado en n, full inicializado en 0, mutexI y mutexE ambos semáforos binarios inicializados en 1. Empty y full forman un semáforo dividido.
El productor hace:

```
while(true){

P(empty);
P(mutexI);
//ingresar dato a la cola
V(mutexI);
V(full);
}
```

El consumidor hace:

```
while(true){

P(full);
P(mutexE);
//extraer dato de la cola
V(mutexE);
V(empty);
}
```

Tiempo: 150 minutos