Solución Examen de Programación 3 y III (17/12/2012)

Instituto de Computación, Facultad de Ingeniería, UdelaR

Parte Teórica Obligatoria

Ejercicio 1 (10 puntos)

a) Sea $G = (V, E)$ un grafo conexo con una función de costo definido sobre las aristas.
Prop. MST: Sea U un subconjunto de los vértices V, si (u, v) es una arista de costo mínimo tal que $u \in U$ y $v \in V - U$ entonces existe un árbol de cubrimiento de costo mínimo que incluye a (u, v) como una de sus aristas.

Demostración:
Para demostrar la propiedad se supondrá por absurdo que no existe ningún árbol de cubrimiento de costo mínimo para el grafo G que incluya la arista (u, v) entre sus aristas.

Sea entonces, T cualquier árbol de cubrimiento de costo mínimo para G. Si se agrega la arista (u, v) a T, se genera un ciclo (recordar que T es un árbol y por lo tanto existe un camino de u a v).
Debe existir otra arista (u', v') tal que $u' \in U$ y $v' \in V - U$, y que forme parte del camino de u a v.
Si se quita la arista (u', v') se rompe el ciclo y se tiene un árbol de cubrimiento T' en el cual se cumple:

$$Costo(T') = Costo(T) - c(u', v') + c(u, v)$$

Y como (u, v) es de costo mínimo (entre las aristas que tienen un extremo en U y el otro en V-U) $\Rightarrow c(u, v) \leq c(u', v') \Rightarrow Costo(T') \leq Costo(T)$
lo cual contradice la hipótesis de que no existe ningún árbol de cubrimiento de costo mínimo que incluya la arista (u, v).

b) En la propiedad MST se basan los algoritmos de *Kruskal* y *Prim*, que permiten hallar un árbol de cubrimiento de costo mínimo para un grafo G=(V,E) con costos asociados a sus aristas.
Notar que la propiedad indica la estrategia a seguir, la cual es: elegir la arista de mínimo costo entre dos conjuntos de vértices no conectados entre sí y que esta estrategia conduce a la solución optima del problema de hallar el árbol de cubrimiento de costo mínimo.

Ejercicio 2 (10 puntos)

a)

1. arista tree: arista de un vértice del árbol a un hijo.
2. arista forward: arista de un vértice del árbol a un descendiente (no hijo)
3. arista back: arista de un vértice del árbol a un antecesor.
4. arista cross : arista entre dos vértices tales que uno no es ni ancestro ni descendiente del otro.

b) Pueden darse aristas tree, forward y back:

tree: cualquier arista (u,v) puede serlo (por ejemplo: la recorrida empieza por u y sigue por v.

forward: si (u,v) es arista y además hay camino de u a v y los vértices de este camino son visitados antes que v en la recorrida DFS(u), entonces (u,v) es una arista forward.

back: si (u,v) es *tree* o *forward*, entonces la arista (v,u) será *back*.

No puede darse aristas *cross*:

Si (*u*,*v*) fuese arista *cross*, la arista (*v*,*u*) también lo seria, de lo contrario pueden darse dos casos:

1. que (*v*,*u*) sea *tree* o *forward* y entonces (*u*,*v*) seria *back* ó

2. que (*v*,*u*) sea *back* y entonces (*u*,*v*) sería *tree* o *forward*.

Es imposible que ambas aristas sean de tipo *cross*, ya que uno de los dos vértices será visitado en primer lugar y en ese momento el restante no estará marcado. Al existir arista entre ellos, el segundo será descendiente del primero y la arista será *tree* o *forward*.

Ejercicio 3 (10 puntos)

a)
Θ define la relación de equivalencia R entre las funciones, f R g $\Leftrightarrow$ f $\in \Theta$(g)

Recordando las definiciones de O y Ω:

i) f $\in$ O(g) si y solo si $\exists$ $c_1 \in R^+$, $n_1 \in N$ / $\forall$ n > n_1 f(n) $\leq c_1$.g(n)

ii) f $\in \Omega$(g) si y solo si $\exists$ $c_1 \in R^+$, $n_1 \in N$ / $\forall$ n > n_1 f(n) $\geq c_1$.g(n)

$f \in \Theta(g)$ si y sólo si f $\in$ O(g) y f $\in \Omega$(g)

Propiedad de dualidad:
f $\in$ O(g) $\Leftrightarrow$ g $\in \Omega$(f).

Utilizando las definiciones de O y Ω:

($\rightarrow$)

A partir de i) se tiene que: $\forall$ n > n_1 $\frac{1}{c_1} * f(n) \leq g(n)$

Si se elige $n_2 = n_1$ y $c_2 = 1/c_1$:

g(n) $\geq c_2$.f(n) $\forall$ n > n_2 entonces g$\in \Omega$(f)

($\leftarrow$)
La demostración en el otro sentido es análoga.

Reflexiva) Con $n_1 = 0$ y $c_1 = 1$ se cumple que f $\in$ O(f) y f $\in \Omega$(f) . Por lo tanto $f \in \Theta(f)$ y entonces f R f .

Simétrica) Si f R g entonces g R f .
Prueba)
$fRg \Leftrightarrow f \in \Theta(g) \Leftrightarrow f \in O(g)$ y $f \in \Omega(g)$
Por dualidad:
$f \in O(g)$ sii $g \in \Omega(f)$ y $f \in \Omega(g)$ sii $g \in O(f)$

Entonces
$fRg \Leftrightarrow g \in \Omega(f)$ y $g \in O(f)$ o sea $fRg \Leftrightarrow g \in \Theta(f) \Leftrightarrow gRf$.

Transitiva) Si f R g y g R h entonces f R h,

f R g entonces y $f \in O(g)$ y g R h entonces $g \in O(h)$ desarrollando la definición de O en ambos casos se llega a que $f \in O(h)$.

De la misma forma f R g entonces y $f \in \Omega(g)$ y g R h entonces $g \in \Omega(h)$ desarrollando la definición de Ω en ambos casos se llega a que $f \in \Omega(h)$.

Luego $f \in \Theta(h)$ y f R h.

b) Partiendo de las *definiciones* dadas en el curso y siendo $f(n) = n^2 - 4n + 4$ y $g(n) = n^3 + 1$ probar que $f \notin \Theta(g)$
Se va a probar que $f \notin \Omega(g)$ entonces $f \notin \Theta(g)$.
Por absurdo, suponiendo que $f \in \Omega(g)$
si y sólo si $\exists\, c_1 \in R^+, n_1 \in N / \forall\, n > n_1\ n^2 - 4n + 4 \geq c_1.\ (n^3 + 1)$

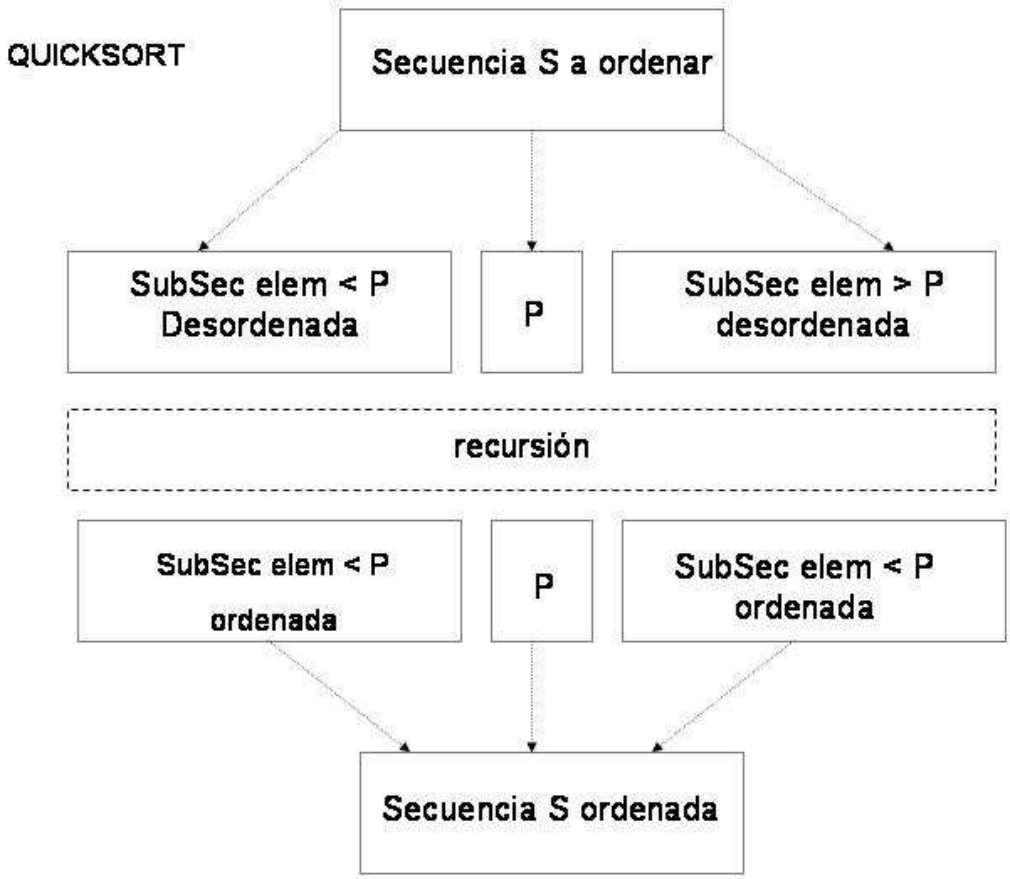
si y sólo si $\exists\, c_1 \in R^+, n_1 \in N / \forall\, n > n_1\ 0 \geq c_1.\ (n^3 - n^2 - 4n - 3)$

Lo cual es contradictorio ya que a partir de cierto número natural la expresión de la derecha es positiva.

Ejercicio 4 (10 puntos)

1)
QuickSort utiliza Divide & Conquer como estrategia. En este caso el énfasis está puesto en la división en subproblemas de forma de que, una vez resueltos los subproblemas en forma recursiva, no será necesario mover la secuencia para combinar (como puede suceder en otros algoritmos de ordenación)

2)
La idea es que dada la secuencia A = [a_1, a_2, a_3, …, a_n] se elija algún elemento a_i, que llamaremos Pivote para reordenar la secuencia ubicando a todos los elementos menores que el Pivote antes que él en la secuencia y los mayores después de él. De este proceso quedarán armadas dos subsecuencias: S1 que tiene todos sus elementos menores que el pivote (pero desordenados aún) y S2 que tiene todos sus elementos mayores que el pivote. Notar que S1 y S2 no contienen al Pivote ya que éste queda ubicado en su lugar definitivo al reordenar los elementos.
Aplicando recursivamente el proceso a las dos subsecuencias S1 y S2 hasta que haya un único elemento en cada subsecuencia se resolverá la ordenación al volver de la recurrencia.



3)

```
void QuickSort( arreglo &A, int ini, int fin){
  int PosPivote;
  if (ini<fin){ // si hay más de un elemento
      AUXILIAR(A, ini, fin, &PosPivote);
      QuickSort(A, ini, PosPivote-1);
      QuickSort(A, PosPivote+1, fin);
  }  // if
}  // fin QuickSort
```

4)

Complejidad de QuickSort

PEOR CASO

Las comparaciones se hacen en AUXILIAR. La **cantidad** es siempre igual al largo de la subsecuencia con que se invoca: **fin-ini+1 = L**

Por lo anterior el peor caso estará dado por las instancias de la entrada que provoquen la mayor cantidad de invocaciones a AUXILIAR.

En un paso cualquiera genérico la entrada a AUXILIAR será: A[ini]…A[fin], el pivote seleccionado será A[ini]. Si este elemento es el menor de la secuencia no será cambiado de lugar (no habrá ningún cambio de elementos) y devolverá PosPivote=ini.
Las subsecuencias que se generen en ese caso tendrán largos 0 y L-1 respectivamente. Si cada vez que se invoca a AUXILIAR se da esta situación, hay n-1 invocaciones recursivas y en cada una el largo de la secuencia no vacía se va decrementando en una unidad.
Este es el peor caso para QuickSort y es el caso en que la secuencia ya está ordenada como se desea.

$$T_W(n) = \sum_{L=2}^{n}(L-1) = \sum_{L=1}^{n} L - 1 - \sum_{L=2}^{n} 1 = \frac{n(n+1)}{2} - 1 - (n-2+1)$$

$$T_W(n) = \frac{n(n+1)}{2} - 1 - n + 1 = \frac{n(n+1) - 2n}{2} = \frac{n(n-1)}{2}$$

Optimalidad: como la cota inferior del Problema de Sorting es $\Omega(n\log n)$ y $T_w(n) \in \Theta(n^2)$ entonces Quicksort no es óptimo en su peor caso.

Problemas

Problema 1 (30 puntos)

a) El proceso de serializar sigue una recorrida en amplitud del árbol, dado el orden requerido de los vértices en el formato serial. El vértice raíz se procesa en forma separada incluyéndolo anticipadamente en la cola. Para cada vértice procesado de la cola del BFS, se accede a todos sus adyacentes no visitados (hijos), se lleva una cuenta de ellos, y se los pone en la cola de procesamiento. Luego de acceder a todos los adyacentes a un vértice, final de su procesamiento, se genera la línea del formato serail con la pareja del identificador y la cantidad de hijos.

El proceso de des-serializar lee ordenadamente el formato serial generando el árbol y mediante un mecanismo de construcción por BFS almacena en colas los vértices y la cantidad de hijos a ser procesados posteriormente. El primer vértice leído, la raíz, se procesa en forma separada incluyéndolo anticipadamente en las colas de indentificadores y cantidad de hijos. Luego para cada vértice que se procesa (de la cola) se leen la cantidad correspondiente de hijos; para cada una de las parejas leídas se genera, en el árbol, un vértice y una arista al progenitor correspondiente, y se colocan su identificador y cantidad de hijos en las colas correspondientes.

b)

```
#define n …

struct parejaSerial {
      int identificador;
      int canthijos;
};
typedef parejaSerial * Serial;

Serial S[n] = {NULL};
…

void *  Serializar_Arbol(Grafo * G, int raiz, Serial * S) {
      int i = 0;
      ListaEnteros Cola = CrearLista();
      int procesado[n] = {0};

      procesado[raiz] = 1;
      ColaInsertarUltimo(Cola, raiz);
      while ( ! ListaEsVacia(Cola)) {
            parejaSerial * pS;
            int progenitor;
            int canthijos = 0;
            ListaEnteros Ady;

            Progenitor = ListaSacarPrimero(Cola);
            Ady = GrafoAdyacentesVertice(G, progenitor);
            while ( ! ListaEsVacia(Ady)) {
                  int hijo = ListaSacarPrimero(Ady);

                  if ( ! procesado[hijo]) {
                        procesado[hijo] = 1;
                        ColaInsertarUltimo(Cola, hijo);
                        canthijos++;
                  }
            }
            S[i] = new parejaSerial;
            S[i]->identificador = progenitor;
            S[i]->canthijos = canthijos;
            i++;
      }
}
```

c)

```
void *  Desserializar_Arbol(Serial * S, Grafo * G, int * raiz) {
      int i = 0;
      ListaEnteros ColaId = CrearLista(); // Cola identificadores
      ListaEnteros ColaCantHijos = CrearLista(); // Cola cant. hijos
      Grafo* G = CrearGrafo();
      int progenitor;
      int hijosprog; // Cantidad de hijos

      raiz = S[i]->identificador;
      hijosprog = S[i]->canthijos;
      i++;
      GrafoInsertarVertice(G, *raiz);
      ColaInsertarUltimo(ColaId, *raiz);
      ColaInsertarUltimo(ColaCantHijos, hijosprog);

      while ( ! ListaEsVacia(ColaId)) {
            progenitor = ListaSacarPrimero(ColaId);
            hijosprog = ListaSacarPrimero(ColaCantHijos);

            for (int h = 1; h <= hijosprog; h++) {
                  int hijo, hijoshijo;

                  hijo = S[i]->identificador;
                  hijoshijo = S[i]->canthijos;
                  i++;
                  GrafoInsertarVertice(G, hijo);
                  GrafoInsertarArista(G, progenitor, hijo);
                  ColaInsertarUltimo(ColaId, hijo);
                  ColaInsertarUltimo(ColaCantHijos, hijoshijo);
            }
      }
}
```

Problema 2 (30 puntos)

a) No siempre se obtiene la solución óptima.

Ej:
$p_1 = 10$, $t_1 = 1$, $v_1 = 1$
$p_2 = 15$, $t_2 = 3$, $v_2 = 3$

$p_1/t_1 = 10$, $p_2/t_2 = 5$, por lo que se procesa primero ***1***. Después no puede agregarse ***2*** porque se necesitaría 4 días para terminar ambos. Por lo tanto el ingreso es $p_1 = 10$.

Pero la solución óptima es implementar el módulo ***2*** y obtener $p_2 = 15$.

b)
Forma de la solución:
tupla t = $<m_1,..., m_k>$ de largo variable, compuesta por los identificadores de los módulos a implementar. *k* es la cantidad de esos módulos.

Restricciones explícitas:
Los componentes identifican un módulo:
$1 \leq m_i \leq n$, para $1 \leq i \leq k$.

Restricciones implícitas:
- Ningún módulo se implementa dos veces:
$i \neq j \Rightarrow m_i \neq m_j$ para $1 \leq i, j \leq k$.

- Sea $\pi_1, \ldots, \pi_k$ una permutación de $m_1,..., m_k$ ordenada según la fecha límite de forma no decreciente. O sea $v_{\pi_i} \leq v_{\pi_{i+1}}$ para $1 \leq i < k$.
Entonces para cada componente de la tupla $< \pi_1, \ldots, \pi_k >$, la fecha límite no puede ser inferior a la suma de los tiempos de todos los módulos del prefijo de tupla:

Llamemos T_i a la suma de los tiempos de implementación de los primeros i módulos aceptados: $T_i = \sum_{h=1}^{i} t_{\pi_h}$. Se debe cumplir $T_i \leq v_{\pi_i}$ para $1 \leq i \leq k$.

Función objetivo:
Se quiere maximizar la suma de los precios de los módulos a implementar:

f = max ($\sum_{i=1}^{k} p_{m_i}$, para todos las $t = < m_1,..., m_k >$ que son solución).

Predicados de poda:
Se poda si la suma de los precios de la solución parcial más la suma de los precios de los módulos que todavía no se han descartado no supera el precio total de la mejor solución encontrada hasta el momento:

Supongamos que el h-ésimo componente de la tupla es el i-ésimo de los n módulos.

Sea $s = < s_1,..., s_k >$ la solución con máximo precio encontrado hasta ahora, y $M = \sum_{j=1}^{k} p_{s_j}$.

Se poda si
$\sum_{j=1}^{h} p_{m_j} + \sum_{j=i+1}^{n} p_j \leq M$.

Si alguno de los módulos que todavía no fueron considerados tiene fecha límite inferior a la suma de los tiempos de implementación del prefijo de tupla construido más su propio tiempo de implementación, entonces ese módulo puede excluirse de la suma.
Sea $\delta_{hj} = 1 \text{ si } T_h + t_j \leq v_j$ o 0 en otro caso. Entonces se poda si

$\sum_{j=1}^{h} p_{m_j} + \sum_{j=i+1}^{n} p_j . \delta_{hj} \leq M$.