Solución del Segundo Parcial de Programación 3

28 de noviembre de 2014

Nota previa: los logaritmos utilizados a lo largo de esta solución son en base 2.

Ejercicio 1 (22 puntos)

1.
   - **Forma de la solución**
     Tupla de largo fijo R, $t =< x_1, \cdots, x_R >$, donde x_R, con $1 \leq r \leq R$, es la cantidad de unidades de la artesanía de tipo r que se van a fabricar.
   - **Restricciones explícitas**
     La cantidad de unidades de cada tipo de artesanía no puede superar $E, x_r \in \{0, \cdots, E\}$, con $1 \leq r \leq R$.
   - **Restricciones implícitas**
     - El costo de fabricación total no puede superar C: $\sum_{r=1}^{R} c_r \times x_r \leq C$.
     - El tiempo total de fabricación no puede superar H: $\sum_{r=1}^{R} t_r \times x_r \leq H$.
   - **Función objetivo**
     El objetivo es maximizar la ganancia:

     $f = max_{t \in T}(g(T))$, donde $T = \{t =< x_1, \cdots, x_R > \mid t$ es solución$\}$, con $g(t) = \sum_{r=1}^{R} (v_r - c_r) \times x_r$

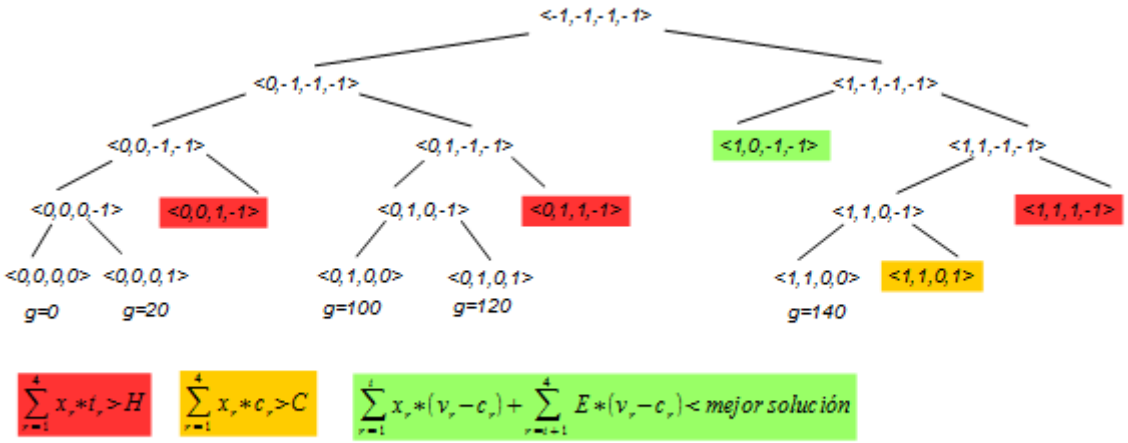
2. a) No es un predicado de poda. Podría descartar soluciones óptimas.
   b) Es un predicado de poda. La máxima ganancia que se obtendría no superaría a la de una tupla que ya se obtuvo.
   El valor más optimista VO es el que se obtiene al incoroporar E unidades de cada una de las artesanías para las que todavía no se hizo la planificación. Formalmente: $VO = \sum_{i=r+1}^{R} (v_i - c_i) \times E$
   c) Poda el espacio de soluciones pero no se la considera un predicado de poda. Esto se debe a que deriva de la primera de las restricciones implícitas.

3. A continuación se presenta una solución posible.

   Parámetros:

   - $R = 4$ (requerimiento de la letra)
   - $E = 1$
   - $H = 24$
   - $C = 35$
   - Tiempos: $< t_1, t_2, t_3, t_4 >=< 5, 3, 80, 10 >$
   - Costos: $< c_1, c_2, c_3, c_4 >=< 10, 1, 20, 25 >$
   - Precios de venta: $< v_1, v_2, v_3, v_4 >=< 50, 101, 21, 45 >$

Árbol de solución:



Ejercicio 2 (23 puntos)

1. a) Sea el grafo $G = (V, A)$ conexo y no dirigido, el subgrafo $T = (V, A')$ es un **árbol de cubrimiento de** G si $A' \subseteq A$ es tal que $|A'|$ es mínima para que T sea conexo.

   b) Se define **árbol de cubrimiento de costo mínimo**, cuando las aristas están ponderadas, la suma de los valores de las elegidas debe ser la mínima posible.

2. Para demostrar la propiedad se supondrá por absurod que no existe ninún 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 de T' en el cual se cumple que $costo(T') = costo(T) - c(u', v') + c(u, v)$, siendo $c(u, v)$ la ponderación de la arista (u, v).

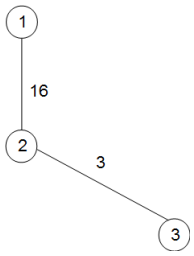
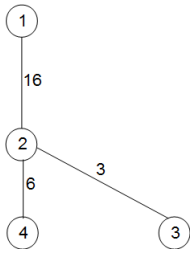
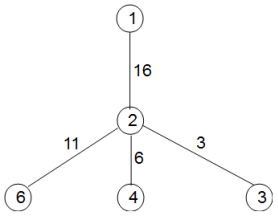
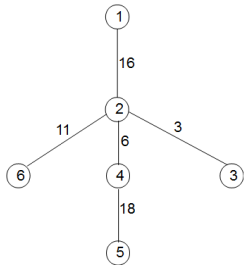
   Y como (u, v) es de costo mínimo entre las aristas que tienen un extremo en U y el otro en $V - U \Rightarrow c(u, v) \leq c(u', v') \Rightarrow costo(T') \leq costo(T)$, lo cual contradice la hipótesis de que no existe ningún árbol de cubrimiento de costo mínimo que incluya la arista (u, v).

3. a) El algoritmo de Prim permite hallar el árbol de cubrimiento de costo mínimo de un grafo a partir de un vértice arbitrario teniendo en cada paso un único árbol que va creciendo al agregarse aristas.

   b)
```
Prim (Grafo G, Grafo &arbol)
{
    ListaVertices U;
    arbol = CrearGrafo();
    U = CrearLista();
    Agregar(U,1);
    mientras (U != V)
        sea <u,v> la arista de costo mínimo / u in U y v in V-U
        AgregarArista(arbol,<u,v>);
        Agregar(U,v);
    fin mientras
}
```

   c) La técnica utilizada es **Greedy**. Como en cada iteración se toma una arista de costo mínimo entre U y $V - U$, por la demostración de la parte anterior se ve que cada decisión es óptima ya que existe un *MST* que contiene a (u, v) de costo mínimo, para cualquier $U \subseteq V$, por lo que se puede aplicar *Greedy*.

4. Se decidió empezar por el vértice 1.

Paso	Formalización	Esquema gráfico
1	$U = \{1\}$ $V - U = \{2,3,4,5,6\}$ $\Rightarrow min\{c(1,2), c(1,6), c(1,5)\}$ $\Rightarrow min\{16, 21, 19\} = 16$ $\Rightarrow$ Se agrega el vértice 2.	
2	$U = \{1,2\}$ $V - U = \{3,4,5,6\}$ $\Rightarrow min\{c(1,6), c(1,5), c(2,6), c(2,4), c(2,3)\}$ $\Rightarrow min\{21, 19, 11, 6, 3\} = 3$ $\Rightarrow$ Se agrega el vértice 3.	
3	$U = \{1,2,3\}$ $V - U = \{4,5,6\}$ $\Rightarrow min\{c(1,6), c(1,5), c(2,6), c(2,4), c(3,4)\}$ $\Rightarrow min\{21, 19, 11, 6, 10\} = 6$ $\Rightarrow$ Se agrega el vértice 4.	
4	$U = \{1,2,3,4\}$ $V - U = \{5,6\}$ $\Rightarrow min\{c(1,6), c(1,5), c(2,6), c(4,6), c(4,5)\}$ $\Rightarrow min\{21, 19, 11, 14, 18\} = 11$ $\Rightarrow$ Se agrega el vértice 6.	
5	$U = \{1,2,3,4,6\}$ $V - U = \{5\}$ $\Rightarrow min\{c(1,5), c(4,5), c(6,5)\}$ $\Rightarrow min\{19, 18, 33\} = 18$ $\Rightarrow$ Se agrega el vértice 5.	

5. Sirve para encontrar los caminos de costo mínimo de un vértice a todos los demás.

```
Procedure Dijkstra (int origen)
    S = {origen};
    Para (i in 1..n)
        D[i] = C[origen][i];
    Fin
    Para (i in 1..n-1)
        elegir un vértice w in V-S / D[w] sea mínimo
        S = S + {w}; // Se considera la suma como operador unión de conjuntos
        Para (v in V-S)
            D[v] = mínimo{D[v], D[w]+C[w][v]}
        Fin
    Fin
Fin;
```

Ejercicio 3 (15 puntos)

1.
```
void MergeSort(arreglo &A, int p, int q){
    int medio;
    if (p<q) {
        medio = (p+q) / 2;
        MergeSort(A, p, medio);
        MergeSort(A, medio+1, q);
        Merge(A, p, medio, medio+1, q);
    }
}
```

2. **Peor Caso**
   MergeSort en sí no realiza explícitamente comparaciones de elementos; toda comparación se hace en el `Merge`. En cada invocación a *MergeSort* se hace una invocación a `Merge` y dos llamadas recursivas a *MergeSort* lo cual implica el planteo de una ecuación de recurrencia para calcular $T_w(n)$.

   `Merge` recibe dos secuencias, $S1$ de largo $L1$ y $S2$ de largo $L2$, devolviendo una secuencia de largo L con $L = L1 + L2$. La cantidad de comparaciones en el peor caso del algoritmo `Merge` es de $L - 1$.

   Para MergeSort queda entonces:

$$T_w(n) = T_w(n/2) + T_w(n/2) + (n-1)$$
$$T_w(1) = 0$$

   Considerando n potencia de 2: $n = 2^p$

$$T_w(n) = 2 \times T_w(n/2) + (n-1)$$
$$2^1 T_w(n/2^1) = 2^2 \times T_w(n/2^2) + (n - 2^1)$$
$$\vdots$$
$$2^{p-1} T_w(n/2^{p-1}) = 2^p \times T_w(n/2^p) + (n - 2^{p-1})$$
$$2^p T_w(n/2^p) = 2^p \times T_w(1) = 0$$

   Sumando y eliminando términos queda:

$$T_w(n) = \sum_{i=0}^{p-1} (n - 2^i) = np - \sum_{i=0}^{p-1} (2^i) = np - 2^p + 1$$

   Como habíamos tomado $n = 2^p, p = log(n) \Rightarrow T_w(n) = n\ log(n) - n + 1$. Por lo tanto, $T_w(n) \in O(n\ log(n))$ en el peor caso.

   Como la cota inferior para el problema de Sorting es $\Omega(n\ log(n)) \Rightarrow T_w(n) \in \Omega(n\ log(n)) \Rightarrow T_w(n) \in \Theta(n\ log(n))$.

3. **Caso medio**
   Observar que:

   - Como el problema de Sorting en su caso medio es $\Omega(n\ log(n))$, entonces *MergeSort* no puede hacer menos que $n\ log(n)$ comparaciones en su caso medio (asumiendo las mismas condiciones para caso medio, lo que incluye la ausencia de elementos repetidos).
   - *MergeSort* es $O(n\ log(n))$ en su peor caso, su caso medio solo podría ser mejor o igual, nunca peor.

   Entonces:

$$T_A(n) \in O(n\ log(n)) \text{ y } T_A(n) \in \Omega(n\ log(n)) \Rightarrow T_A(n) \in \Theta(n\ log(n))$$