Examen de Programación 3

11 de diciembre de 2017

En recuadros con este formato aparecerán aclaraciones que cumplen una función explicativa pero que no eran requeridos como parte de la solución.

Parte obligatoria

Esta parte es eliminatoria. Para la aprobación del examen debe obtenerse un mínimo del **50 % de esta parte (20 puntos)**. En caso de no llegar a dicho mínimo, **NO** se corregirán los problemas.

Pregunta 1 (15 puntos)

Sea $G = (V, E)$ un grafo no dirigido y conexo. Suponga que se toma un vértice $u \in V$ y se realiza una recorrida DFS desde u obteniéndose el árbol T. Suponga luego que se realiza una recorrida BFS desde u y se obtiene el mismo árbol T. Demuestre que $G = T$. Sugerencia: demuestre por el absurdo asumiendo que G tiene un ciclo.

Solución:

Lo demostraremos por absurdo. Supongamos que $G \neq T$, dado que $E_T \subset E_G$ y T es un árbol de cubrimiento de G, G no es un árbol y por lo tanto tiene un ciclo $(v, w, ..., x, v)$, siendo v el primer vértice del ciclo que es visitado por ambas recorridas (esto no tiene por qué suceder siempre pero no se pierde generalidad asumiéndolo).

La recorrida DFS de este ciclo producirá un subárbol que no contendrá una de las aristas (v, w) y (v, x). Sin embargo la recorrida BFS producirá un subárbol que incluirá a ambas. Esto contradice la hipótesis de que, en este caso, DFS y BFS generan el mismo árbol.

Pregunta 2 (15 puntos)

Considere las funciones f y g tales que $f(n)$ es $O(g(n))$. Para cada una de las siguientes afirmaciones, indique si son verdaderas o falsas, demuestre o de un contraejemplo.

(a) $f(n)^2$ es $O(g(n)^2)$.

(b) $2^{f(n)}$ es $O(2^{g(n)})$.

(c) $log_2 f(n)$ es $O(log_2 g(n))$.

Solución:

(a) Es verdadero. Ya que $f(n) \leq cg(n)$ para todo $n \geq n_0$, entonces $(f(n))^2 \leq c^2(g(n))^2$ para todo $n \geq n_0$.

(b) Es falso. Suponga $f(n) = 2n$ y $g(n) = n$. Entonces $2^{f(n)} = 4^n$ y $2^g(n) = 2^n$.

(c) En el caso general esto es falso. Si $g(n) = 1$ para todo n, $f(n) = 2$ para todo n, entonces $\log_2 g(n) = 0$ y por lo tanto no podemos decir que $\log_2 f(n) \leq c \log_2 g(n)$.

En el caso particular en el que $g(n) \geq 2$ para todos los $n \geq n_1$, sí es cierto. Esto se debe a que como $f(n) \leq cg(n)$ para todo $n \geq n_0$, entonces $\log_2 f(n) \leq \log_2 c + \log_2 g(n) \leq (\log_2 c)(\log_2 g(n))$ con $n \geq max(n_0, n_1)$.

Pregunta 3 (10 puntos)

Indique si la siguiente afirmación es **verdadera** o **falsa**. Si es verdadera, de una breve justificación. Si es falsa, describa un contraejemplo.

En toda instancia del problema del emparejamiento estable (stable matching), hay un emparejamiento estable que contiene un par (m,w) tal que m esta ranqueado primero en la lista de preferencia de w y w esta ranqueado primero en la lista de preferencia de m.

Solución:

Es falso. Considere el siguiente ejemplo:

m prefiere a w sobre w'
m' prefiere a w' sobre w
w prefiere a m' sobre m
w' prefiere a m sobre m'

note que no hay ningún par en el que ambas personas *rankean* al otro en primer lugar, de modo que no puede haber un emparejamiento estable con un par de ese tipo.

Problemas

Problema A (30 puntos)

*C*onsiderese G un grafo conexo no dirigido con aristas ponderadas. Se busca construir un árbol de cubrimiento mínimo, y para ello se propone el siguiente algoritmo:

```
HallarMST(Grafo G)
    Ordenar todas las aristas del grafo de manera decreciente. // MergeSort(?)
    Inicializar arreglo de aristas // Para usarse como marcas.
    Repetir hasta que esten todas las aristas marcadas
        Tomar la arista de mayor valor $a$ no marcada
        Si FormaCiclo(G, a)
            G = G - {a}
        Marcar(a)
    Retornar G

FormaCiclo(Grafo G, arista a)
    bfs = BFS(G - {a}) // Es una recorrida entera BFS
    Si bfs tiene una sola componente conexa
        retornar Verdadero
    Sino
        retornar Falso
```

(I) ¿Cuál es el peor caso del algoritmo?

(II) ¿Qué orden de complejidad tiene el algoritmo?

(III) ¿Se obtiene un árbol de cubrimiento de costo mínimo?

(IV) ¿Es un algoritmo *greedy*?

(V) Escriba el pseudocódigo de un algoritmo *greedy* que permita hallar un árbol de cubrimiento mínimo.

Solución:

(I) El peor caso del algoritmo se da cuando el grafo es completo. Esto se debe a que el loop principal del algoritmo recorre todas las aristas del grafo, el grafo con mayor cantidad de aristas posibles es el grafo completo.

(II) El algoritmo recorre todas las aristas (O(m)) y por cada una de ellas ejecuta un BFS (O(m+n)), por lo tanto se ejecuta en un tiempo de orden $m \times (m + n)$.

(III) Si. Se trata del algoritmo presentado en el punto 4.21 del Kleinberg “Reverse-Delete Algorithm”.

(IV) Si. Resuelve el problema tomando la decisión localmente óptima en cada paso.

(V) Por ejemplo, cualquiera de los algoritmos presentados en la página 143 del Kleinberg.

Problema B (30 puntos)

En una empresa existe una hilera de n baldosas para hacer transitable un camino entre dos de sus edificios. Debido a restricciones presupuestales se desea vender algunas de esas baldosas que según su estado tienen valores de venta diferentes: el valor de la i-ésima baldosa es $b[i]$. Para que el camino continúe siendo transitable no se pueden remover dos baldosas seguidas del camino.

(I) Desarrolle la relación de recurrencia que obtiene el máximo valor de venta que se puede lograr.

(II) Implemente la función

```
int * removidas (int *b, int n, int & c);
```

que devuelve un arreglo con las baldosas que deben removerse para obtener la máxima ganancia. El arreglo devuelto debe estar en orden decreciente. En el parámetro c se debe devolver la cantidad de baldosas que se remueven. Los valores de las n baldosas se pasan en las posiciones 1 a n del arreglo b.

Solución:

(I)

$$C(i) = \begin{cases} 0 & \text{si } i = 0, \\ b_1 & \text{si } i = 1, \\ \max\{C_{i-1}, b_i + C_{i-2}\} & \text{si } i > 1, \end{cases}$$

(II)

```
int * removidas (int *b, int n, int &c) {
  int C[n+1];   // costo
  bool Tb[n+1]; // traceback
  C[0] = 0, C[1] = b[1];
  Tb[1] = true;
  for (int i = 2; i <= n; i++)
    if (b[i] + C[i-2] > C[i-1]) {
      C[i] = b[i] + C[i-2];
      Tb[i] = true;
    } else {
      C[i] = C[i-1];
      Tb[i] = false;
    }
  int *R = new int [n / 2]; // es el máximo posible
  c = 0;
  int i = n;
  while (i > 0) {
    if (Tb[i] == true) {
      c++;
      R[c] = i;
      i -= 2;
    } else {
      i -= 1;
    }
  }
  return R;
}
```

Esta es una versión simplificada del problema de intervalos poderados. En este caso cada baldosa representa un intervalo y se solapa con la inmediata anterior y la inmediata posterior y sólo con ellas.