Parcial de Programación 3

28 de setiembre de 2017

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

Ejercicio 1 (20 puntos)

Roque es un típico personaje principal de telenovela, y por ende sufrió un ataque de amnesia. Lamentablemente, Roque ya no sabe en qué orden ponerse las prendas a la hora de vestirse, y se ha encontrado múltiples veces con los zapatos puestos antes que las medias. Roque está cansado de que esto lo haga llegar tarde al trabajo y decide contratarte a ti para que le soluciones su problema. Asuma que Roque debe ponerse todas las prendas que tiene en su ropero.

(a) Modele la relación de precedencia entre las prendas de vestir mediante un grafo (esto es, que para que Roque se pueda poner un cinturón debe tener puesto antes un pantalón). ¿Qué características tiene el grafo propuesto?

(b) Describa en pseudocódigo un algoritmo que le permita a Roque determinar en qué orden ponerse las prendas, de manera que una vez puesta una prenda no deba sacársela. El algoritmo debe recibir como entrada el grafo propuesto de la parte (a) y devolver una secuencia de prendas.

(c) Roque tiene la peculiar habilidad de ponerse en 10 segundos cualquier número de prendas simultáneamente, mientras no existan relaciones de precedencia entre ellas (si es una sola prenda también le lleva 10 segundos). ¿Qué información debería saber del grafo de la parte (a) para poder determinar el tiempo mínimo que le llevaría a Roque vestirse? ¿Cuál sería ese tiempo?

Solución:

(a) El problema se puede modelar con un grafo dirigido simple sin ciclos (DAG) en el que los nodos representan las prendas y los enlaces representan la relación de precedencia entre las prendas; esto es, como para ponernos los pantalones debemos antes tener puestos los calzoncillos, entonces existe una arista dirigida desde los calzonillos a los pantalones. El grafo es dirigido porque la relación de precedencia establece un orden parcial en el que se pueden poner las prendas. Es simple porque no hay precedencia de una prenda a sí misma; además, la relación de precedencia entre dos prendas distintas es única. Por otra parte, el grafo no tiene ciclos (dirigidos), de lo contrario no modelaría una relación de orden parcial (y no habría forma de verstirse!); debido a esto el grafo debe contener al menos un nodo con solo aristas salientes (nodo fuente) y al menos un nodo con solo aristas entrantes (nodo sumidero).

(b) El problema es equivalente a la obtención de un ordenamiento topológico del grafo que modela la relación de precedencia entre las prendas, dado que el ordenamiento topológico es equivalente a un ordenamiento parcial de las prendas. Una opción para determinar un ordenamiento topológico del grafo es obtener los nodos fuentes del grafo y eliminarlos junto a sus aristas incidentes. Dichos nodos son los primeros en el orden. Luego, sobre el grafo remanente se aplica el mismo proceso hasta ordenar todos los nodos.

```
lista ordentopologico(G){
  l = crearLista()
  // vacio(G) devuelve true cuando el grafo es vacio
  mientras !vacio(G){
    // devuelve un vertice sin aristas incidentes
    v = verticeFuente()
    // agrega v al final de la lista l
    l.add(v)
    // quita el nodo v de G y todas sus aristas salientes
    eliminar(v,G)
  }
  return l
}
```

(c) Lo que se debe encontrar es el largo l del camino más largo en el grafo. Una vez que se tiene ese largo, el tiempo mínimo que le puede llevar a Roque vestirse es $(l + 1) * 10$ segundos. Se suma 1 a l ya que lo que se debe contar es la cantidad de vértices del camino.

Ejercicio 2 (15 puntos)

(a) Considere el grafo conexo no dirigido $G = (V, E)$, cuyas aristas e tienen asociados costos c_e, todos distintos. Sea C un ciclo cualquiera de G, y $e = (v, w)$ la arista más costosa de C. Demuestre que e no pertenece a ningún árbol de cubrimiento mínimo (*minimum spanning tree*) de G.

(b) Sea $G = (V, E)$ un grafo conexo no dirigido con costos $c_e \geq 0$ sobre las aristas $e \in E$. Asuma que se le ofrece un árbol de cubrimiento mínimo (*minimum-cost spanning tree*) T de G. Ahora asuma que a G se le agrega una arista nueva de costo c que conecta los nodos $v, w \in V$.

Escriba un algoritmo eficiente que verifique si T sigue siendo un árbol de cubrimiento mínimo del nuevo grafo G. Asuma que los costos c_e son todos distintos.

Solución:

(a) Prueba: Supongamos que existe T, un árbol de cubrimiento mínimo que contiene e con vértices v y w; como e pertenece a un ciclo C en G, existe un camino de v a w en G que no usa e al que llamamos $C - e$. Supongamos que quitamos e de T, se generan dos componentes conexas T' y T''. El camino $C - e$ tiene al menos una arista que tiene un vértice en T' y otro en T'', llamémosle e'. Por hipótesis $c(e) > c(e')$, por lo tanto el árbol $T' \cup e \cup T''$ tiene un costo mayor que el árbol $T' \cup e' \cup T''$ lo cual contradice la hipótesis de absurdo.

(b) Supongamos que $e = (v, w)$ es la nueva arista agregada. Primero debemos encontrar el camino en T de v a w, la arista e unida a ese camino genera un ciclo en G. Si cada arista en ese camino tiene un costo menor que el de e entonces, por la propiedad de la parte (a), e no es parte de ningún árbol de cubrimiento mínimo y por lo tanto T sigue siendo un árbol de cubrimiento mínimo. Si alguna arista del ciclo tiene costo mayor que e, entonces, por la propiedad de la parte (a) esa arista no puede formar parte de un árbol de cubrimiento mínimo y por lo tanto T ya no es un árbol de cubrimiento mínimo de G.

Versión en que T se trata como un grafo genérico.

```
Entradas:
  visitados es un array de vértices pasado como parametro por referencia;
  v, w son vértices;
  T es un grafo; se trata del árbol T de la letra
  c es un entero; se trata del costo de la nueva arista mencionada en la letra
Salida:
  par de booleanos indicando si T sigue siendo un MST y si alcancé a w
  // siempre hay un camino que alcanza w

(bool,bool) stillMST(T,v,w,visitados, c) {
  if w in adyacentess(T,v)
    return (c >= c(v,w), true); //c(v,w) es el costo de la arista (v,w)
  elsif adyacentess(T,v) in visitados
    return (false,false);
  else {
    still = true;
    encontre = false;
    while !encontre && !(adyacentess(T,v) in visitados) {
       v' = next(adyacentess(T,v));
       append(v', visitados);
       (still, encontre) = stillMST(T,v',w,visitados,c);
       if (encontre)
          still =  still && (c >== c(v, v'));
    }
    return (still,encontre);
}
```

Versión en que T se trata como un árbol y además se usan operaciones del TAD `Iterador`:

- `has_next(S)` devuelve `true` si y sólo si en `S` hay algún elemento no visitado.
- `next(S)` devuelve un elemento de `S` que no habia sido visitado y lo marca como visiado.

```
Entradas:
  u, w, p son vértices;
   w es uno de los vértices entre los que se agrega la arista
      mencionada en la letra
   p es el vértice desde el que se hace la llamada recursiva
  T es un grafo; se trata del árbol T de la letra
  c es un entero; se trata del costo de la nueva arista mencionada en la letra
Salida:
  par de booleanos indicando si T sigue siendo un MST y si se alcanzó w
  // siempre hay un camino que alcanza w
La llamada inicial es stillMST(T, v, w, v, c) donde v es el otro
 vertice entre los que se agrega la arista.

(bool, bool) stillMST(T, u, w, p, c) {
  if (w in adyacentes(T,u))
    return (c >= c(u,w), true);
  else {
    still = false;  // también puede ser true
    encontre = false;
    while ((! encontre) && (has_next(adyacentes(T,u)))) {
       v' = next(adyacentes(T,u);
       if (v' != p) {
         (still, encontre) = stillMST(T, v', w, u, c);
         if (encontre)
           still =  still && (c >= c(u, v'));
       }
    }
    return (still, encontre);
  }
}
```

Ejercicio 3 (15 puntos)

(a) Describa formalmente en que consiste el problema de *"El par de puntos más cercanos"*, tratado en el curso.

La solución vista cumple la relación de recurrencia

$$T(n) \leq \begin{cases} 0 & \text{si } n = 1, \\ 2T(n/2) + c(n \log n) & \text{en otro caso.} \end{cases}$$

siendo c una constante mayor que 0. Explique brevemente la procedencia de esos términos.

(b) La mecánica Mechy tiene una caja con n tuercas y sus correspondientes tornillos. Todas las tuercas tienen diámetros diferentes. No se puede comparar un par de tuercas entre sí ni un par de tornillos entre sí. Pero se puede comparar cualquier tuerca con cualquier tornillo y determinar el tamaño relativo del tornillo con respecto a la tuerca. El propósito es ayudar a Mechy a enroscar cada tornillo con su tuerca correspondiente y se pretende lograrlo con $O(n \log n)$ comparaciones entre tornillos y tuercas en el caso promedio.

Describa en lenguaje natural (no pseudocódigo) el algoritmo *divide & conquer* para lograrlo, dejando claro cuáles son los subproblemas en que se divide el problema original. ¿Cuándo se da el peor caso y cuál es el orden?

Solución:

(a) El problema *"El par de puntos más cercanos"* consiste en, dado un conjunto de puntos $P = \{p_1, \ldots, p_n\}$ en el plano, encontrar un par de esos puntos, $p_i, p_j \in P$ tal que $d(p_i, p_j)$ sea mínima entre todos los pares de puntos, donde $d(\cdot, \cdot)$ es la distancia euclideana.

- El término 0 corresonde al caso base en el cual no se hace nada.
- El término $2T(n/2)$ corresponde a dos llamadas a problemas con aproximadamente la mitad de los puntos cada una, separados según la coordenada x por una línea vertical L.
- El término $c(n \log n)$ corrresponde a ordenar según la coordenada y los puntos de una franja central alrededor de la línea L cuyo ancho está determinado por el mínimo de los resultados de las dos llamadas recursivas.

El problema se resuelve encontrando la coordenada x de una recta vertical, L, que divide los puntos en dos conjuntos de aproximadamente el mismo tamaño, $n/2$. Se resuelve recursivamente el problema con cada uno de los dos conjuntos, tras lo cual se obtiene una cota superior, δ, que es el mínimo de las distancias de los pares encontrados en las dos llamadas recursivas. En la fase de combinar se busca el par de puntos más cercanos entre los pertenecientes a la franja central compuesta por todos los puntos cuya distancia a la recta L es menor o igual a δ. El par buscado es el que tiene menor distancia entre los dos pares obtenidos en las llamadas recursivas y el que se encuentra en la franja central. Se puede demostrar que para cada punto de la franja hay $O(1)$ puntos que están a distancia menor a δ, por lo que se si se ordenan los puntos de la franja según su coordenada y se comprueba en $O(1)$ si el punto pertenece al par buscado. Por lo tanto se obtiene en tiempo $O(n)$ el par más cercano en la franja. Entonces, el costo de las fases dividir y combinar es $O(n \log n)$ para ordenar los elementos de la franja según la coordenada y, que domina el costo $O(n)$ de encontrar el par de menor distancia una vez que ya están ordenados. Previamente, en la primera fase de dividir se ordenan los puntos según la coordenada x. El término $2T(n/2)$ proviene de resolver los dos problemas de tamaño $n/2$.

Si se preordenara el conjunto de puntos tanto por la coordenada x como por la coordenada y las fases de dividir y combinar tendrían costo $O(n)$ por lo que el costo total del algoritmo seria $O(n \log n)$.

(b) Como caso base, se toma $n = 1$ y el problema está resuelto.

Para el caso general hay que separar los conjuntos de tuercas en (al menos) dos subconjuntos S_p y S_g y el conjunto de tornillos en dos subconjuntos T_p y T_g. Se generan dos subproblemas, uno con S_p y T_p y otro con S_g y T_g. Para que los subproblemas mantengan las condiciones del problema original las tuercas de S_p deben corresponder a los tornillos de T_p, y las tuercas de S_g a los tornillos de T_g. El tiempo de ejecución pedido sugiere separar el problema en dos subproblemas en un tiempo $O(n)$ en las fases de dividir y combinar.

Se toma una de las tuercas, s_m, como pivote y se la compara con cada tornillo del conjunto T. El conjunto de tornillos queda dividido en 3: el conjunto T_p con los que son más pequeños que s_m, el conjunto T_g con los son más grandes que s_m y el tornillo t_m que coincide con s_m. Con el tornillo t_m como pivote se aplica un procedimieto similar con el conjunto S de tuercas, obteniendo los conjuntos S_p y S_g. Se encontró la coincidencia entre s_m y t_m y luego se aplica de manera recursiva el mismo procedimiento dos veces, en una instancia a los conjuntos S_p y T_p y en otra instancia a los conjuntos S_g y T_g. En promedio el tamaño de los conjuntos S_p y S_g serán cercanos a $n/2$.

Se da el peor caso si en cada subproblema la tuerca s_m es la menor o la mayor del conjunto S. En ese caso sólo hay que resolver uno de los subproblemas, pero el tamaño del conjunto es $n - 1$. El tiempo de ejecución sería $O(n^2)$.

```
Enroscar(S, T)
  Si S = {s}
    R ← {(s,t)}
  en otro caso
    Sea s_m ∈ S
    T_p, T_g ← ∅
    Para cada t ∈ T
      comp ← comparar(s_m, t)
      Si EsIgual = comp
        t_m ← t
        R ← {(s_m, t_m)}
      en otro caso si EsMenor = comp
        T_p ← T_p ∪ {t}
      en otro caso
        T_g ← T_g ∪ {t}
    S_p, S_g ← ∅
    Para cada s ∈ S − {s_m}
      comp ← comparar(s, t_m)
      Si EsMenor = comp
        S_g ← S_g ∪ {s}
      en otro caso
        S_p ← S_p ∪ {s}
    Si S_p ≠ ∅
       R ← R ∪ Enroscar(S_p, T_p)
    Si S_g ≠ ∅
       R ← R ∪ Enroscar(S_g, T_g)
    Devolver R
```