Examen de Programación 3
13 de diciembre de 2018

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

Ejercicio 1 (35 puntos)

Un autor de libros infantiles ha escrito un libro de la serie *Elige tu propia aventura* que consta de n capítulos. En este tipo de libros, el lector comienza en el capítulo 1 y, a partir de allí, cada vez que llega al final de un capítulo debe elegir entre **2 o 3 opciones**. Cada una de las opciones lo dirige hacia un capítulo distinto desde donde continuar con la historia, hasta que eventualmente llega a alguno de los *capítulos finales* en los cuales la historia se desenlaza y no se presentan opciones para continuar. El autor ha sido cuidadoso y no hay forma de que se repita un capítulo a lo largo del desarrollo de una historia. Además, en una planilla ha anotado, para cada capítulo v, la lista de capítulos que pueden seguirse inmediatamente a partir de v.

El revisor de la editorial quiere leer todos los capítulos para hacer un control de calidad, sin repetir ninguno, y nos pide que enumeremos los capítulos para su lectura de manera consistente con las historias. Específicamente, si x, y, son dos capítulos tales que en alguna historia x transcurre antes que y, entonces el revisor quiere que en su orden de lectura x aparezca antes que y.

(a) Dé un algoritmo para resolver el problema que admita una implementación cuyo tiempo de ejecución sea $O(n)$. **Reescriba** cualquier algoritmo que utilice del material teórico.

(b) Demuestre que el algoritmo propuesto puede implementarse de forma que el tiempo de ejecución sea $O(n)$. **Reescriba** cualquier argumento estudiado durante el curso.

Sugerencia: Para analizar la complejidad del algoritmo propuesto note que la cantidad de opciones que se presentan al final de cada capítulo está acotada.

Solución:

(a) Modelamos el problema como un grafo dirigido $G = (V, E)$, donde V es el conjunto de capítulos del libro y existe una arista (u, v) si y solo si desde el capítulo u se puede seguir inmediatamente al capítulo v. Como no hay forma de que se repita un capítulo a lo largo del desarrollo de una historia, el grafo G es un DAG y por lo tanto tiene un ordenamiento topológico, $v_1, v_2, \ldots, v_n$. Si x, y, son dos capítulos tales que en cierta historia x transcurre antes que y, entonces existe un camino desde x a y en G. Como todas las aristas de G son de la forma (v_i, v_j), $j > i$, esto implica que x ocurre antes que y en el ordenamiento topológico. Esta es exactamente la condición requerida por el editor, por lo cual el problema se reduce a encontrar un ordenamiento topológico. Esto se resuelve con el algoritmo de la figura 1.

```
1  Crear el grafo G = (V, E) con la estructura del libro
2  Hacer indeg[u] igual al grado de entrada de u para todo u ∈ V
3  Hace S igual al conjunto de vértices u con indeg[u] = 0
4  Crear una lista vacía orden
5  while S no es vacío do
6      Retirar un vértice u de S y agregarlo al final de orden
7      foreach arista (u, v) saliente de u do
8          Decrementar indeg[v]
9          if indeg[v]=0 then
10             Agregar v a S
11         end
12     end
13 end
14 return orden
```

Figura 1: Algoritmo para determinar un orden de lectura de los capítulos.

(b) La construcción del grafo G, representado mediante listas de adyacencia, se realiza en tiempo $O(n)$, ya que por cada uno de los n vértices hay que construir dos listas de adyacencia (una para aristas salientes y otra para aristas entrantes) con no más de 3 elementos cada una. Para esto recorremos la planilla con la estructura del libro y, por cada capítulo x y cada capítulo y al que se puede pasar directamente desde x, agregamos (x, y) a la lista de aristas salientes de x y a la lista de aristas entrantes de y.

> La representación de grafos dirigidos con listas de aristas entrantes y salientes es la usada en general durante el curso. Para este algoritmo en particular no es necesario contar con listas de aristas entrantes.

El paso 2 se puede implementar asignando inicialmente el valor 0 a todos los elementos de indeg, y recorriendo luego las aristas de G, incrementando el valor de indeg[y] en una unidad por cada arista (x, y) recorrida. Esto requiere tiempo $O(n)$, ya que tanto el tamaño de indeg como la cantidad de aristas de G son $O(n)$.

Para la representación de S usamos una estructura de datos que permite agregar un elemento en tiempo $O(1)$ y también retirar algún elemento en tiempo $O(1)$, por ejemplo una lista encadenada. Para la representación de orden usamos una estructura de datos que permite agregar al final en tiempo $O(1)$, por ejemplo un arreglo con tope. Con esta representación, los pasos 6 y 10 requieren tiempo $O(1)$; la creación de S en el paso 3 requiere tiempo $O(n)$ ya que el arreglo indeg tiene tamaño n.

Después de su creación, el arreglo indeg solo se modifica en el paso 8, donde se decrementa la entrada u. Por lo tanto, como los elemento de indeg nunca incrementan sus valores, la evaluación de la condición del paso 9 inmediatamente después de la ejecución del paso 8 asegura que cada vértice de V es agregado a S a lo sumo una vez, ya sea en el paso 10 o en el paso 3. Esto a su vez implica que la cantidad de iteraciones que se ejecutan del ciclo del paso 5 es a lo sumo n, porque en cada una se retira un elemento de S en el paso 6. Teniendo en cuenta que el ciclo del paso 7 se repite a lo sumo 3 veces cada vez que se ejecuta, vemos que cada iteración del ciclo del paso 5 requiere tiempo $O(1)$, de donde concluimos que el tiempo total de ejecución es $O(n)$.

Ejercicio 2 (15 puntos)

Consideramos el problema de emparejamiento estable entre un conjunto W y otro M, cada uno con n elementos. Para cada $m \in M$ existe una lista de preferencias sobre los elementos de W y viceversa. A diferencia del problema estudiado en el curso, en este ejercicio puede haber *empates* en las listas de preferencia. Por ejemplo, la lista de preferencias para cierto $w \in W$ podría especificar que el elemento de mayor preferencia es m, en segundo lugar de preferencia hay un empate entre m' y m'', etc.. En este ejemplo decimos que w *prefiere estrictamente* a m antes que a m' pero es *indiferente* entre m' y m''.

Decimos que existe una *inestabilidad* en un emparejamiento S si existen $m \in M$, $w \in W$, tales que se prefieren estrictamente entre sí antes que a sus respectivas parejas en S. Un emparejamiento es *estable* si no existe ninguna inestabilidad.

(a) Dé un algoritmo para construir un emparejamiento perfecto estable que admita una implementación cuyo tiempo de ejecución sea $O(n^2)$. Recuerde que puede usar **sin reescribir** cualquier algoritmo estudiado durante el curso. **No es necesario** analizar la complejidad del algoritmo.

Sugerencia: Construya órdenes de preferencia estricta resolviendo los empates de forma arbitraria.

(b) Demuestre la corrección del algoritmo propuesto. Recuerde que **puede utilizar** resultados teóricos estudiado durante el curso.

Solución:

(a) Solucionamos el problema utilizando el algoritmo de Gale-Shapley, para lo cual construimos listas de preferencia estrictas, $MPref'$ y $WPref'$, a partir de las listas de preferencia de nuestro problema, $MPref$ y $WPref$, donde potencialmente hay empates. Eslecíficamente, los pasos a seguir son los siguientes:

1. Construimos $MPref'$, donde $MPref'(m, i) \in W$ es el elemento en lugar i de preferencia para m, $m \in M$, $1 \leq i \leq n$, de forma tal que si m prefiere estrictamente w antes que w' entonces w aparece antes que w' en $MPref'(m, \cdot)$. La construcción de $WPref'$ es análoga.

   Esta construcción se puede hacer en tiempo $O(n^2)$ en una sola recorrida por las listas de preferencia de $MPref$ y $WPref$.

2. Ejecutamos el algoritmo de Gale-Shapley sobre $MPref'$, $WPref'$ y devolvemos el emparejamiento que obtenemos como resultado.

(b) El algoritmo de Gale-Shapley devuelve un emparejamiento estable S con respecto a $MPref'$, $WPref'$. Lo único que hay que probar es que este emparejamiento no presenta inestabilidades (según la definición en la letra) con respecto a $MPref$, $WPref$. Supongamos por absurdo que existen $m \in M$, $w \in W$, tales que se prefieren estrictamente entre sí antes que a sus respectivas parejas en S. En otras palabras, existen $(m, w') \in S$, $(m', w) \in S$ tales que m prefiere estrictamente a w antes que w' según $MPref$ y w prefiere estrictamente a m antes que m' según $WPref$. Entonces, por nuestra definición de $MPref'$ y $WPref'$ en el paso 1, w aparece antes que w' en $MPref'(m, \cdot)$ y m aparece antes que m' en $WPref'(w, \cdot)$. Pero esto configura una inestabilidad en S (según la definición original vista en el curso) con respecto a $MPref'$, $WPref'$, lo cual contradice la corrección del algoritmo de Gale-Shapley.

Ejercicio 3 (35 puntos)

Una empresa de hardware está investigando el desarrollo de un nuevo procesador (CPU) paralelo para dispositivos móviles. Un programa se define como una secuencia de instrucciones.

Sean $X = (x_1, \ldots, x_n)$, $Y = (y_1, \ldots, y_m)$ dos programas que serán ejecutados en paralelo en el dispositivo. Las instrucciones de un programa son ejecutadas en orden. En cada ciclo, si la siguiente instrucción a ejecutar de X es igual a la siguiente instrucción de Y la CPU podrá ejecutarlas en paralelo. En caso contrario la CPU podrá ejecutar una única instrucción, pudiendo ser la de X o la de Y.

Se desea determinar la cantidad mínima de ciclos necesarios para ejecutar ambos programas.

Ejemplo. Supongamos que la CPU cuenta con cuatro instrucciones: `ADD`, `SUB`, `DIV` y `MUL`. Sean X = (`ADD, MUL, DIV, SUB, ADD, SUB, MUL`), Y = (`ADD, SUB, MUL, ADD, SUB`).

Una posible ejecución en paralelo de X e Y, que requiere 9 ciclos, es la siguiente.

X	ADD	-	MUL	-	DIV	SUB	ADD	SUB	MUL
Y	ADD	SUB	MUL	ADD	-	SUB	-	-	-

Otra posible ejecución, que requiere menos ciclos que la anterior, es la siguiente.

X	ADD	-	MUL	DIV	SUB	ADD	SUB	MUL
Y	ADD	SUB	MUL	-	-	ADD	SUB	-

Definimos OPT(i, j) como la cantidad mínima de ciclos necesarios para ejecutar las primeras i instrucciones de X y las primeras j instrucciones de Y, donde $0 \leq i \leq n$, $0 \leq j \leq m$.

(a) ¿Cuánto vale OPT(0, j), $0 \leq j \leq m$? ¿Cuánto vale OPT(i, 0), $0 \leq i \leq n$?

(b) Muestre que OPT satisface una relación de recurrencia y especifique cuál es esa relación. Justifique.

(c) Dé un algoritmo **iterativo** de programación dinámica que permita calcular la cantidad de ciclos mínima para ejecutar los programas X e Y usando la recurrencia de la parte anterior. El algoritmo debe admitir una implementación cuyo tiempo de ejecución es polinomial en n y m (no es necesario demostrarlo).

Solución:

(a) La CPU puede ejecutar las primeras j instrucciones de Y en j ciclos, y no lo puede hacer en menos porque no puede ejecutar más de una instrucción de un mismo programa en un ciclo. Por lo tanto, $\text{OPT}(0, j) = j$. Análogamente tenemos que $\text{OPT}(i, 0) = i$.

(b) Consideramos ahora una ejecución óptima (en la menor cantidad posible de ciclos) de las instrucciones $x_1, \ldots, x_i, y_1, \ldots, y_j$. Distinguimos los siguientes casos:

- Si x_i, y_j no se ejecutan simultáneamente, alguna de ellas se ejecuta al final de forma independiente. Supongamos que la instrucción x_i es la última en ejecutar. Entonces, la CPU ejecuta primero las instrucciones $x_1, \ldots, x_{i-1}, y_1, \ldots, y_j$ y finalmente ejecuta la instrucción x_i. La cantidad total de ciclos es entonces la cantidad mínima de ciclos requeridos para ejecutar $x_1, \ldots, x_{i-1}, y_1, \ldots, y_j$, dada por $\text{OPT}(i-1, j)$, más el ciclo requerido para ejecutar x_i. Por lo tanto, tenemos $\text{OPT}(i, j) = 1 + \text{OPT}(i-1, j)$. Análogamente, si se ejecuta por último y_j, tenemos que $\text{OPT}(i, j) = 1 + \text{OPT}(i, j-1)$.
- Si x_i, y_j se ejecutan simultáneamente, lo cual solo puede ocurrir si $x_i = y_j$, entonces la CPU ejecuta primero las instrucciones $x_1, \ldots, x_{i-1}, y_1, \ldots, y_{j-1}$ y finalmente las instrucciónes x_i, y_j en paralelo. La cantidad total de ciclos es entonces la cantidad mínima de ciclos requeridos para ejecutar $x_1, \ldots, x_{i-1}, y_1, \ldots, y_{j-1}$, dada por $\text{OPT}(i-1, j-1)$, más el ciclo adicional requerido para ejecutar a x_i, y_i en paralelo. Por lo tanto, en este caso tenemos $\text{OPT}(i, j) = 1 + \text{OPT}(i-1, j-1)$.

Como esta distinción de casos cubre todas las alternativas posibles, deducimos que la cantidad mínima de ciclos necesarios para ejecutar las primeras i instrucciones de X y las primeras j instrucciones de Y está dada por la alternativa que redunda en la menor cantidad de ciclos. Teniendo en cuenta además la parte a, concluimos que OPT satisface la recurrencia

$$\begin{aligned}
&\mathrm{OPT}(0,j) = j, \quad 0 \le j \le m.\\
&\mathrm{OPT}(i,0) = i, \quad 0 < i \le n.\\
&\mathrm{OPT}(i,j) = 1 + \text{mín}\left\{\mathrm{OPT}(i-1,j), \mathrm{OPT}(i,j-1), \mathrm{OPT}(i-1,j-1)|_{x_i=y_j}\right\}, \quad 1 \le i \le n,\ 1 \le j \le m\,,
\end{aligned}$$

donde $\mathrm{OPT}(i-1,j-1)|_{x_i=y_j}$ es igual a $\mathrm{OPT}(i-1,j-1)$ si $x_i = y_j$ y es igual a ∞ en otro caso.

Alternativamente, se puede argumentar que si $x_i = y_j$ entonces cualquier ejecución óptima en la cual x_i, y_j no se ejecutan en paralelo se puede transformar en otra, también óptima, en la cual sí lo hacen; simplemente si x_i se ejecuta antes que y_j postergamos la ejecución de x_i hasta el último ciclo y viceversa. Con esta formulación la recurrencia resulta

$$\mathrm{OPT}(0,j) = j, \quad 0 \le j \le m. \tag{1}$$

$$\mathrm{OPT}(i,0) = i, \quad 0 < i \le n. \tag{2}$$

$$\mathrm{OPT}(i,j) = \begin{cases} 1 + \mathrm{OPT}(i-1,j-1), & \text{si } x_i = y_j, \\ 1 + \text{mín}\{\mathrm{OPT}(i-1,j), \mathrm{OPT}(i,j-1)\}, & \text{si } x_i \neq y_j, \end{cases} \quad 1 \le i \le n,\ 1 \le j \le m. \tag{3}$$

Algorithm *CiclosMinimos(X, Y)*
  $\mathrm{OPT}[0,j] = j, \quad 0 \le j \le m$
  $\mathrm{OPT}[i,0] = i, \quad 0 < i \le n$
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** m **do**
      `/* Aplicamos la ecuación (3)  */`
      if $X[i] = Y[j]$ **then**
        $\mathrm{OPT}[i,j] = 1 + \mathrm{OPT}[i-1, j-1]$
      else
        $\mathrm{OPT}[i,j] = 1 + \text{mín}\left\{\mathrm{OPT}[i-1,j], \mathrm{OPT}[i,j-1]\right\}$
      end
    end
  end
  return $OPT[n,m]$
end

Figura 2: Algoritmo para calcular la cantidad mínima de ciclos para ejecutar X, Y.

(c) El algoritmo se presenta en la figura 2. El tiempo de ejecución del algoritmo es $O(nm)$, pues completamos cada entrada de la matriz OPT en tiempo $O(1)$.

Ejercicio 4 (15 puntos)

Consideramos una red de flujo *s-t* definida sobre un gafo dirigido $G = (V, E)$ con n nodos y m aristas, capacidades enteras, y tal que todo nodo está conectado a alguna arista.

Dado un flujo f de máximo valor obtenido a partir de la ejecución del algoritmo de Ford-Fulkerson y su grafo residual, G_f, ¿cómo calcularía un corte (A^*, B^*) de mínima capacidad en tiempo $O(m)$ a partir de G_f?

Solución:

Ver enunciado (7.11) del libro de Kleinberg & Tardos.