Parcial de Programación 3
26 de noviembre de 2019

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

Ejercicio 1 (14 puntos)

Consideramos un conjunto no vacío de q personas, $P = \{p_1, p_2, \ldots, p_q\}$, y un conjunto no vacío de k tareas que hay que realizar, $T = \{t_1, t_2, \ldots, t_k\}$. Dependiendo de su complejidad, cada tarea $t_j \in T$ requiere cierta cantidad mínima de personas para su realización, la cual denotamos R_j. Para cada persona $p_i \in P$, conocemos el conjunto de tareas, $T_i \subseteq T$, que es capaz de realizar la persona p_i. Además existe una cantidad máxima de tareas que pueden ser asignadas a p_i, denotada C_i, que satisface $C_i \leq k$. Para toda tarea existe al menos una persona capacitada para trabajar en ella, y toda persona está capacitada para trabajar en al menos una tarea.

Nuestro problema consiste en asignar, si es posible, tareas a personas de forma tal que se satisfagan todas las restricciones que hemos descrito:

1. Las tareas asignadas a p_i pertenecen a T_i para todo $p_i \in P$.
2. A cada persona $p_i \in P$ se le asigna una cantidad de tareas no superior a C_i.
3. Cada tarea $t_j \in T$ es asignada a por lo menos R_j personas.

(a) Modele el problema mediante una red de flujo. Dé un algoritmo que devuelve una asignación de personas a tareas que satisface todas las restricciones, si dicha asignación existe, o informa que no hay solución posible, en caso contrario. El tiempo de ejecución de su algoritmo debe ser $O(k^2q^2)$. No es necesario reescribir algoritmos contenidos en el material teórico del curso.

Sugerencia: Busque una solución en la cual cada tarea t_j es asignada a exactamente R_j personas.

(b) Demuestre que el tiempo de ejecución de su algoritmo es $O(k^2q^2)$. Puede usar sin demostración resultados contenidos en el material teórico del curso.

Solución:

(a) Creamos una red de flujo $G = (V, E)$ con un conjunto de vértices $V = \{s, t\} \cup P \cup T$, donde s y t son vértices fuente y sumidero, respectivamente. El conjunto de aristas, E, lo construimos de la siguiente manera. Para cada persona $p_i \in P$ agregamos

- una arista (s, p_i) con capacidad C_i
- un conjunto de arista $\{(p_i, t_j) : t_j \in T_i\}$, todas con capacidad unitaria.

Además, para cada tarea $t_j \in T$ agregamos una arista (t_j, t), con capacidad R_j.

Alternativamente se arriba a esencialmente la misma solución invirtiendo el sentido de las aristas e intercambiando los nodos fuente y sumidero.

Definimos R como la suma, entre todas las tareas, de las cantidades de personas requeridas por cada una, $R = \sum_{j=1}^{k} R_j$. El algoritmo para resolver el problema consta de los siguientes pasos:

1. Construir la red de flujo G
2. Ejecutar el algoritmo de Ford-Fulkerson para obtener un flujo integral de máximo valor, f.
3. Si el valor del flujo es igual a R, asignar a cada persona p_i el conjunto de tareas t_j tales que (p_i, t_j) es una arista e que transporta flujo positivo, $f(e) > 0$.
4. En caso contrario reportar que no existe una asignación que satisface las restricciones del problema.

A continuación analizamos la corrección del algoritmo propuesto.

Si el valor del flujo f es igual a R, entonces la asignación devuelta satisface todas las restricciones:

1. Las tareas asignadas a p_i pertenecen a T_i para todo $p_i \in P$, porque existe una arista de p_i a dicha tarea y, por definición de G, las aristas salientes de p_i llegan a tareas de T_i.
2. A cada persona $p_i \in P$ se le asigna una cantidad de tareas no superior a C_i, porque C_i es la capacidad de la única arista entrante a p_i y todas las aristas salientes de p_i tienen capacidad unitaria. Como el flujo f es integral, la ecuación de balance de flujo en p_i implica que no puede haber más de C_i aristas salientes de p_i que transportan flujo positivo.
3. Cada tarea $t_j \in T$ es asignada a por lo menos R_j personas. En efecto, como las únicas aristas entrantes a t provienen de tareas, y la suma de las capacidades de estas aristas es exactamente igual a R, el hecho de que se alcance un valor de flujo igual a R implica que todas estas aristas están saturadas de flujo. Por lo tanto, la única arista saliente de t_j transporta flujo R_j y, como todas las aristas entrantes a t_j provienen de personas y todas tienen capacidad unitaria, ls ecuación de balance de flujo en t_j implica que debe haber exactamente R_j personas asignadas a t_j.

Por otra parte, si existe una asignación A que satisface todas las restricciones, entonces cada tarea $t_j \in T$ es asignada a por lo menos R_j personas. A partir de A podemos construir una asignación A' en la cual cada tarea $t_j \in T$ es asignada a exactamente R_j personas, simplemente liberando arbitrariamente algunas de las asignaciones sobrantes que pudiera haber. La asignación A' induce un flujo f' en G de la siguiente manera:

1. Para cada arista e de la forma (p_i, t_j) definimos $f'(e) = 1$ si la tarea t_j está asignada a p_i, y $f'(e) = 0$ en caso contrario. Notar que se satisfacen las restricciones de capacidad. Notar además que, por definición de G, todas las asignaciones de tareas a personas corresponden a alguna arista de esta forma y, por lo tanto, el flujo saliente de cada persona es igual a la cantidad de tareas que tiene asignada, y el flujo entrante a cada tarea t_j es igual a la cantidad de personas asignadas a dicha tarea, R_j.
2. Para cada arista e de la forma (s, p_i) definimos $f'(e)$ igual a la cantidad de tareas asignadas a p_i. Notar que esta cantidad no es mayor a C_i, por lo cual no se supera la capacidad de la arista. Además, por la observación del punto anterior, se cumple que $f'(e)$ es igual al flujo saliente de p_i, por lo cual también se satisface la ecuación de balance de flujo en p_i.
3. Para cada arista e de la forma (t_j, t) definimos $f'(e) = R_j$. Notar que no se supera la capacidad de e. Además, por la observación del punto 1, se cumple que $f'(e)$ es igual al flujo entrante a t_j, por lo cual también se satisface la ecuación de balance de flujo en t_j.

En resumen, si existe una asignación que satisface todas las restricciones entonces existe un flujo de valor R en G. Esto implica que al algoritmo de Ford-Fulkerson encuentra un flujo integral f con este valor y, en consecuencia, nuestro algoritmo devuelve una asignación correctamente.

(b) El grafo G tienen una cantidad m de aristas, que observamos que es $O(qk)$:

1. Hay q aristas de la forma (s, p_i), $p_i \in P$.
2. Cada persona está capacitada para trabajar en no más de k tareas, y hay q personas en total, por lo cual no hay más que qk aristas de la forma (p_i, t_j), $p_i \in P$, $t_j \in T$.
3. Hay k aristas de la forma (t_j, t), $t_j \in T$.

En total no hay más que $qk + q + k$ aristas, que es $O(qk)$.

Por el punto 2 en la enumeración anterior, el corte $s - t$ definido por el par $\Big(\{s\} \cup P, T \cup \{t\}\Big)$ tiene capacidad no superior a qk. Esto implica que el algoritmo de Ford-Fulkerson, ejecutado en el paso 2, termina en tiempo $O(mqk)$ que es $O(q^2k^2)$.

Alternativamente, podemos observar que como $C_i \leq k$ para todo i, $1 \leq i \leq q$, entonces el corte $s - t$ dado por $\Big(\{s\}, V \setminus \{s\}\Big)$ tiene capacidad no superior a qk.

Por otra parte, la construcción del grafo en el paso 1, así como el cálculo del valor de f y la asignación de tareas a personas del paso 3, requieren tiempo proporcional al tamaño de G, que es $O(qk)$. Finalmente, el paso 4 requiere tiempo $O(1)$. En conclusión, el algoritmo consta de 4 pasos, cada uno de los cuales requiere tiempo $O(q^2k^2)$, lo cual implica que el tiempo total de ejecución es $O(q^2k^2)$.

Ejercicio 2 (18 puntos)

Un ladrón está planificando un robo a una secuencia de casas $c_1, c_2 \ldots, c_n$ que se encuentran dispuestas a lo largo de varias cuadras. Para no llamar la atención, quiere evitar robar dos casas contiguas de la secuencia.

Para cada casa c_i se conoce el *valor* v_i en pesos de los bienes dentro de la casa, $v_i > 0$. Decimos que un subconjunto S de casas es *compatible* si no contiene casas contiguas; definimos el *valor* de S como la suma de los valores de las casas que contiene.

Queremos determinar qué casas debería robar el ladrón con el fin de maximizar la ganancia acumulada, es decir, nos interesa hallar un conjunto compatible de máximo valor.

(a) Dé un algoritmo **iterativo** eficiente que determina el máximo valor posible de un conjunto compatible. Para esto **defina y justifique** claramente una relación de recurrencia que sirva como base para un algoritmo de programación dinámica.

Sugerencia: Considere una función de la forma $OPT(i)$.

(b) Dé un algoritmo eficiente que determina un conjunto compatible de máximo valor.

Solución:

(a) Definimos $OPT(i)$ como la ganancia máxima posible que puede obtener el ladrón robando casas en el conjunto $\{c_1, c_2, \ldots, c_i\}$, $0 \leq i \leq n$. Notar que el valor que debe calcular el algoritmo pedido en esta parte es $OPT(n)$.

Para $i = 0$, no hay casas disponibles para robar y, por lo tanto, la ganancia máxima posible es 0. Por otra parte, para $i = 1$, la única opción posible con ganancia positiva para el ladrón es robar la única casa disponible, obteniendo una ganancia v_1. A partir de estas consideraciones concluimos que OPT satisface

$$OPT(0) = 0\,, \tag{1}$$
$$OPT(1) = v_1\,. \tag{2}$$

Para $i > 1$ el ladrón tiene la disyuntiva entre robar o no la casa c_i. En el primer caso, obtiene una ganancia v_i en la casa c_i y, como no puede robar en dos casas contiguas, la mayor ganancia posible que puede obtener con el resto de las casas está dada por $OPT(i-2)$. En el segundo caso no gana nada en la casa c_i y puede elegir cualquier conjunto compatible de las restantes, lo cual tiene como máximo un valor de $OPT(i-1)$. En consecuencia, se satisface la siguiente ecuación

$$OPT(i) = \text{máx}\{OPT(i-1), OPT(i-2) + v_i\}\,, \quad i > 1\,, \tag{3}$$

que junto con (1) y (2) definen completamente OPT.

De forma más rigurosa, para $i > 1$ afirmamos que se cumple (3).

En efecto, sea i arbitrario tal que $1 < i \leq n$, y sea $S \subseteq \{c_1, c_2, \ldots, c_i\}$ un conjunto compatible de valor $v(S) = OPT(i)$. Sean también $S_1 \subseteq \{c_1, c_2, \ldots, c_{i-1}\}$ y $S_2 \subseteq \{c_1, c_2, \ldots, c_{i-2}\}$ conjuntos compatibles de valor $OPT(i-1)$ y $OPT(i-2)$, respectivamente. Tanto S_1 como $S_2 \cup \{c_i\}$ son conjuntos compatibles contenidos en $\{c_1, c_2, \ldots, c_i\}$, con valores $OPT(i-1)$ y $OPT(i-2) + v_i$, respectivamente, por lo cual la definición de $OPT(i)$ implica que

$$OPT(i) \geq \text{máx}\{OPT(i-1), OPT(i-2) + v_i\}\,. \tag{4}$$

Veamos que también se cumple

$$OPT(i) \leq \text{máx}\{OPT(i-1), OPT(i-2) + v_i\}\,. \tag{5}$$

Si $c_i \notin S$, entonces, como $S \subseteq \{c_1, c_2, \ldots, c_{i-1}\}$, la definición de $OPT(i-1)$ implica que

$$v(S) \leq OPT(i-1)\,, \quad c_i \notin S\,. \tag{6}$$

De forma similar, si $c_i \in S$, el hecho de que S sea compatible implica que $S \setminus \{c_i\}$ es un subconjunto compatible de $\{c_1, c_2, \ldots, c_{i-2}\}$ con valor $v(S) - v_i$, de modo que la definición de $OPT(i-2)$ implica que

$$v(S) - v_i \leq OPT(i-2)\,, \quad c_i \in S\,. \tag{7}$$

Las desigualdades (6) y (7) junto con el hecho de que $v(S) = OPT(i)$ muestran que se cumple (5), lo cual, junto con (4), prueba (3).

El algoritmo de la figura 1 utiliza la relación de recurrencia (1)–(3) para resolver el problema.

(b) El algoritmo de la figura 2 utiliza el arreglo OPT que calcula el algoritmo anterior para construir un conjunto compatible de máximo valor.

```
1  Algorithm CalcOPT
2      Hacer OPT[0] = 0 y OPT[1] = g_1
3      for i = 2 to n do
4          Hacer OPT[i] = máx{OPT[i − 1], OPT[i − 2] + g_i}
5      end
6      Devolver OPT[n]
7  end
```

Figura 1: Algoritmo para calcular la ganancia máxima posible.

```
1  Algorithm ConstruirSolución
2      Hacer i = n y S = {}
3      while i ≥ 1 do
4          if OPT[i] = OPT[i − 1] then
5              Hacer i = i − 1
6          else
7              Agregar i a S
8              Hacer i = i − 2
9          end
10     end
11     Devolver S
12 end
```

Figura 2: Algoritmo para construir un conjunto óptimo de casas.

Ejercicio 3 (18 puntos)

Considere el siguiente juego de mesa donde hay un tablero de $n \times m$ celdas y fichas de tipo **X** y de tipo **O**. Inicialmente cada celda está vacía o tiene una sola ficha (de cualquiera de los dos tipos). El juego consiste en encontrar un subconjunto de fichas tal que, al quitar las demás, las fichas que quedan en cada fila (si las hay) son del mismo tipo, y en cada columna hay al menos una ficha. El jugador gana si logra este objetivo, lo cual será posible o no dependiendo del estado inicial del tablero. Se define el problema de decisión *SOLITAIRE* de la siguiente manera: dado un estado inicial del tablero, T, ¿es posible ganar el juego?

Considere la siguiente transformación de instancias de *3-SAT* a instancias de *SOLITAIRE*. Sea (X, C) una instancia de *3-SAT*, donde $X = \{x_1, \ldots, x_n\}$ es el conjunto de variables y $C = \{C_1, \ldots, C_m\}$ el conjunto de cláusulas. Recuerde que en ninguna cláusula aparece una misma variable más de una vez. Se define la instancia T de *SOLITAIRE* de la siguiente forma:

1. Se crea un tablero vacío de tamaño $n \times m$, siendo n la cantidad de variables y m la cantidad de cláusulas.
2. Por cada término de la forma x_i o $\overline{x_i}$ que aparece en una cláusula C_j, se coloca una ficha de tipo **X** o de tipo **O**, respectivamente, en la celda (i, j).

Ejemplo: Considere una instancia (X, C) de *3-SAT*, donde $X = \{x_1, x_2, x_3, x_4\}$ y $C = \{x_1 \vee x_2 \vee x_3,\ \overline{x_1} \vee \overline{x_3} \vee x_4,\ \overline{x_2} \vee x_3 \vee x_4\}$. La figura 3 ilustra el tablero que se obtiene como resultado de aplicar la transformación. La figura 4 muestra una posible forma de ganar el juego para ese tablero.

	C_1	C_2	C_3
x_1	X	O	
x_2	X		O
x_3	X	O	X
x_4		X	X

Figura 3: Instancia de *SOLITARE* generada a partir de la instancia *3-SAT* del ejemplo.

	C_1	C_2	C_3
x_1	X		
x_2			
x_3		O	
x_4		X	X

Figura 4: Posible configuración ganadora para el tablero de la figura 3.

(a) Sea (X, C) una instancia de *3-SAT* y sea T el tablero obtenido luego de aplicarle la transformación. Demuestre que si T es una instancia SÍ de *SOLITARE* entonces (X, C) es una instancia SÍ de *3-SAT*. En otras palabras, si es posible ganar el juego dado por T, entonces el conjunto C se puede satisfacer.
Sugerencia: Considere una configuración ganadora del tablero T y basado en esto asigne valores de verdad a las variables de X.

(b) El recíproco de la parte anterior también es cierto, es decir, si (X, C) es una instancia SÍ de *3-SAT* entonces T es una instancia SÍ de *SOLITARE*. Asumiendo que esto es cierto (**no es necesario probarlo**), muestre que *3-SAT* $\leq_P$ *SOLITARE*.

(c) Demuestre que *SOLITAIRE* es NP-completo. Puede usar sin demostración resultados contenidos en el material teórico del curso.

Solución:

(a) Si T es una instancia SÍ de *SOLITARE*, entonces existe un subconjunto de fichas que determinan un estado de tablero ganador. A partir de este tablero ganador podemos construir una asignación de verdad, $v : X \to \{0, 1\}$, que satisface las cláuslas de C de la siguiente manera: si en la fila x_i hay fichas de tipo **X** definimos $v(x_i) = 1$; si en cambio hay fichas de tipo **O** definimos $v(x_i) = 0$. Si no hay fichas de ningún tipo definimos $v(x_i)$ con un valor arbitrario, por ejemplo 0. Observar que en un tablero ganador las fichas de cada fila son todas del mismo tipo, por lo cual no hay ninguna ambigüedad en esta definición de v.

> En el ejemplo de la figura 4 obtenemos: $v(x_1) = 1$, $v(x_2) = 0$, $v(x_3) = 0$ y $v(4) = 1$. Observar que con esta asignación se satisfacen todas las cláusulas del ejemplo.

Sea C_j una cláusula arbitraria de la instancia (X, C). En la columna correspondiente a C_j hay por lo menos una ficha en el estado ganador del tablero. Si es de tipo **X**, esta ficha corresponde a un término de la forma x_i, al cual v le asigna valor verdadero. Si por el contrario es de tipo **O**, corresponde a un término de la forma $\overline{x_i}$, al cual v le asigna valor verdadero porque $v(x_i) = 0$. En cualquier caso vemos que C_j se satisface y, como es arbitraria, concluimos que todas las cláusulas de C son satisfechas.

(b) Por la solución de la parte a y el enunciado de la parte b, sabemos que la transformación definida en la letra del problema cumple que T es una instancia SÍ de *SOLITARE* si y solo si (X, C) es una instancia SÍ de *3-SAT*, donde T es la instancia que se obtiene con la transformación a partir de (X, C). Por lo tanto, es posible resolver una instancia de *3-SAT* realizando esta transformación e invocando a un algoritmo que resuelve *SOLITARE* una única vez (que es obviamente una cantidad polinomial de invocaciones).

Para probar que *3-SAT* $\leq_P$ *SOLITARE* solo resta demostrar que la transformación se puede hacer en tiempo polinomial en el tamaño de la representación de (X, C). Efectivamente, esto se puede hacer recorriendo las cláusulas de C y, para cada una de ellas, llenando una columna de T de acuerdo a los tres términos que figuran en la cláusula.

> A modo de ejemplo, podemos representar (X, C) mediante un arreglo con m ternas, una por cláusula, donde cada elemento de la terna representa un término de la cláusula. Para representar un término sobre una variable, digamos x_i, usamos $O(\log n)$ bits para representar el índice i y un bit adicional para indicar la negación o no de la variable. Por otra parte, un tablero T se puede representar mediante un par de enteros, n, m, que determinan sus dimensiones, junto con una lista de ternas (fila, columna, tipo), que determinan la posición y tipo de cada una de las fichas del tablero. Con estas representaciones, la construcción de T se hace en tiempo lineal en el tamaño de la representación de (X, C).

(c) Como *3-SAT* es NP-completo y, por la parte anterior, *3-SAT* $\leq_P$ *SOLITARE*, solo resta mostrar que *SOLITARE* está en $\mathcal{NP}$. Un certificado para una instancia T de *SOLITARE* que contiene R fichas podría ser, por ejemplo, una lista L de hasta R posiciones de tablero que forman (tentativamente) una configuración ganadora. Un algoritmo certificador realizaría los siguientes pasos:

1. Recorrer L y, para cada elemento (i, j), verificar que en el tablero de T existe una ficha en la posición (i, j); si no es así responder NO. El tiempo requerido para este paso es $O(1 + RQ)$, donde Q denota el tiempo máximo de acceso a una posición específica del tablero de T. El tiempo Q depende de la representación que se use para T, pero claramente puede implementarse de forma que este tiempo sea $O(|T|)$, donde $|T|$ denota el tamaño de la representación de T, de modo que este paso en total requiere tiempo polinomial en $|T|$.
2. Si $R < m$ responder NO. En caso contrario, para cada j, $1 \leq j \leq m$, buscar en L un elemento (i, j) que hace referencia a la columna j. Si no se encuentra tal elemento, responder NO. Este paso, que requiere tiempo $O(1 + R^2)$ (polinomial en $|T|$), asegura que el certificado incluye al menos una ficha en cada columna del tablero.
3. Recorrer L y, para cada elemento (i, j), buscar entre los elementos anteriores en L otro elemento, (i, j'), que hace referencia a la misma fila i, tal que el tipo de ficha en (i, j') es diferente del tipo de ficha en (i, j). Si se encuentra tal elemento responder NO. Este paso lleva tiempo $O(R^2 Q)$, que es polinomial en $|T|$, y asegura que las fichas de cada fila sean todas del mismo tipo.
4. Si en ninguno de los pasos anteriores se dio una condición para responder NO, responder SÍ.

El tamaño del certificado es claramente polinomial en $|T|$ y el tiempo total de ejecución del certificador es también polinomial en $|T|$, lo cual demuestra que *SOLITARE* está en $\mathcal{NP}$.

> Alternativamente, si el tablero se representa mediante una lista F de ternas (fila, columna, tipo), el certificado podría ser una lista L de posiciones dentro de F. Con esta representación, el paso 1 es innecesario porque todas las fichas del certificado son, por construcción, fichas del tablero original.