Examen de Programación 3
24 de julio de 2024

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)

Considere una generalización del problema de emparejamiento estable entre dos conjuntos, A y B, de tamaños n y m, respectivamente, donde algunas parejas no son admisibles. Cada $a \in A$ tiene una lista, denotada $pref_a$, que contiene elementos de B en orden de preferencia para a; si un elemento b de B no pertenece a $pref_a$ entonces b no es admisible para a. Análogamente, cada $b \in B$ tiene una lista, denotada $pref_b$, con los elementos de A que son admisibles para b en orden de preferencia para b. Notar que la inclusión de un elemento de A en la lista de preferencias de un elemento de B no implica que éste se encuentre la lista de preferencias del primero y viseversa.

Un emparejamiento $M \subset A \times B$ es *estable* si ningún par $(a, b) \in A \times B$ de elementos que se consideran **mutuamente admisibles** representa alguna de las siguientes inestabilidades:

1. ni a ni b están en ninguna pareja de M.
2. a no está en pareja y b prefiere a a antes que a su pareja en M.
3. b no está en pareja y a prefiere a b antes que a su pareja en M.
4. a y b se prefieren mutuamente antes que a sus respectivas parejas en M.

Se pide:

(a) Escriba un algoritmo que produzca un emparejamiento estable, según la definición de estabilidad dada en este ejercicio, en tiempo polinomial (el emparejamiento podría no ser perfecto).

(b) Demuestre la corrección de su algoritmo. Reescriba cualquier argumento que utilice de los estudiados en el curso.

Solución:

(a) El algoritmo pedido se presenta en la figura 1.

```
1  Algorithm SMI
2      Marcar e como libre para todo e ∈ A ∪ B
3      while ∃a ∈ A libre que no se le haya propuesto a toda su lista de preferencias do
4          Elegir a ∈ A arbitrario libre que no se le haya propuesto a toda su lista de preferencias
5          a se propone a b, el primer elemento de pref_a al que aún no se propuso a
6          if a ∈ pref_b then
7              if b está libre then
8                  Emparejar (a, b)
9              else
10                 Sea a' la actual pareja de b
11                 if a está antes que a' en pref_b then
12                     Separar (a', b) y emparejar (a, b)
13                 else
14                     b rechaza a
15                 end
16             end
17     end
18 end
```

Figura 1: Algoritmo para formar un emparejamiento estable.

(b) El algoritmo termina, ya que en cada iteración del ciclo del paso 3 se realiza una proposición que nunca había sido realizada antes (paso 5), y hay a lo sumo nm proposiciones distintas que pueden realizarse. Por lo tanto, en a lo sumo nm iteraciones la condición del ciclo será falsa.

Observamos que se cumple la siguientes propiedades:

(P1) Cuando un elemento b de B recibe por primera vez una propuesta de parte de un $a \in A$ admisible para b, b forma una pareja. (P2) A partir de ese momento b nunca más vuelve a estar libre, y su pareja nunca empeora con respecto a su lista de preferencias.

La propiedad (P1) es consecuencia de las condiciones en los pasos 6 y 7, que llevan al emparejamiento de b en el paso 8. La propiedad (P2) es consecuencia de que el único paso en que una pareja se separa es el paso 12, y en ese paso b no deja de estar en pareja sino que cambia por una mejor según sus preferencias.

Consideremos un par arbitrario $(a, b) \in A \times B$ de elementos que se consideran mutuamente admisibles.

Supongamos que al finalizar el algoritmo a está libre. Por la condición de parada del ciclo del paso 3, esto implica que a se propuso a b en algún momento. Esto descarta una inestabilidad de tipo 1 por (P1) y (P2). También descarta una inestabilidad de tipo 2, porque o bien b rechazó a a en primera instancia o lo reemplazó posteriormente por otro y, en cualquiera de los dos casos, (P2) garantiza que b prefiera a su pareja final antes que a a.

Supongamos ahora que al finalizar el algoritmo a está en pareja con cierto b', $b' \neq b$. Si b no está en pareja, por (P1) no recibió ninguna propuesta admisible, lo cual implica que al terminar el algoritmo a no había avanzado hasta b en su lista de preferencias y por lo tanto prefiere a b' antes que a b. Esto descarta una inestabilidad de tipo 3. Por otra parte, si b está en pareja y a prefiere a b antes que a b', entonces a debe haberse propuesto a b porque está antes en su lista de preferencia que b'. En esa propuesta, o bien b rechazó a a en primera instancia o lo reemplazó posteriormente por otro y, en cualquiera de los dos casos, (P2) implica que b prefiera a su pareja final antes que a a. Esto descarta una inestabilidad de tipo 4.

Ejercicio 2 (30 puntos)

Sea $G = (V, E)$ un grafo no dirigido y **conexo** con $|V| = n > 2$ y $|E| = m$. Se dice que $v \in V$ es un *punto de articulación* de G, si el subgrafo H que se obtiene a partir de G eliminando v y sus aristas incidentes, $H = G - \{v\}$, es disconexo (tiene más de una componente conexa).

(a) Escriba un algoritmo que determine si G tiene un punto de articulación y en caso afirmativo devuelva alguno (si hay más de uno es suficiente devolver solo uno de ellos). En caso negativo el algoritmo debe informar que G no tiene puntos de articulación. Su algoritmo debe admitir una implementación que ejecute en tiempo $O(nm)$. Reescriba cualquier algoritmo que utilice de los estudiados en el curso.

(b) Demuestre que su algoritmo admite una implementación cuyo tiempo de ejecución es $O(nm)$. Reescriba cualquier argumento que utilice de los estudiados en el curso.

Solución:

(a) El algoritmo consistente en probar para cada nodo si el grafo resultante de eliminar dicho nodo es conexo o no.

```
1  Algorithm PtoArticulacion (G = (V,E))
2      foreach v ∈ V do
3          if not Es_conexo_sin_v(G,v) then
4              Devolver v
5              TERMINAR
6          end
7      end
8      Informar que G no tiene puntos de articulación
9  end
```

```
1  Algorithm Es_conexo_sin_v(G,v)
2      Hacer visitado[s] = false para todo s ∈ V
3      Hacer visitado[v] = true
4      Crear una cola vacía C
5      Elegir x ∈ V \ {v} arbitrario
6      Hacer visitado[x] = true
7      Agregar x a C
8      Hacer cant_nodos = 0
9      while C no está vacía do
10         Retirar el primer nodo, u, de C
11         Incrementar cant_nodos
12         foreach arista (u, w) adyacente a u do
13             if not visitado[w] then
14                 Hacer visitado[w]=true
15                 Agregar w a C
16             end
17         end
18     end
19     return (cant_nodos = n-1)
20 end
```

(b) Mostramos en primer lugar que el tiempo de ejecución de *Es_conexo_sin_v* es $O(n + m)$. Cada vez que se agrega un nodo a C, ya sea en el paso 7 o en el paso 15, este nodo se marca como visitado en el paso inmediatamente anterior y ningún otro paso posterior lo desmarca. Por lo tanto, la condición del paso 13 garantiza que ningún nodo es agregado a C más de una vez. En consecuencia, como cada ejecución del paso 10 retira un nodo de C, el ciclo while del paso 9 se ejecuta a lo sumo n veces, y además en cada iteración se retira un nodo u diferente. Por lo tanto, cada arista $(x, y) \in E$ es recorrida a lo sumo dos veces por el ciclo del paso 12 a lo largo de una ejecución de *Es_conexo_sin_v*: una vez cuando x es retirado de C y otra vez cuando y es retirado de C. Concluimos entonces que el tiempo total insumido por todas las ejecuciones de los pasos

12–16 es $O(n+m)$, ya que cada uno de los pasos $13--15$ requiere tiempo $O(1)$, usando una representación de lista de adyacencia se insume tiempo $O(1)$ para pasar de una arista adyacente a u a la siguiente, y el ciclo se inicia a lo sumo n veces. Como el resto de los pasos ejecutados en el ciclo del paso 9 requieren tiempo $O(1)$ por ejecución y se repiten a lo sumo n veces, el tiempo total de ejecución de los pasos 9–18 es $O(n+m)$. Sumando a esto el tiempo de ejecución del paso 1, que es $O(n)$ y el de los pasos 2–8 y 19, que es $O(1)$, concluimos que el tiempo de ejecución de *Es_conexo_sin_v* es $O(n+m)$.

Como G es conexo, tenemos que $n = O(m)$, por lo que el tiempo total de cada invocación a *Es_conexo_sin_v* es $O(m)$. Como se realizan a lo sumo n de estas invocaciones, el tiempo de ejecución de *PtoArticulacion* resulta ser entonces $O(nm)$.

Ejercicio 3 (35 puntos)

Dos grafos $A = (V_A, E_A)$ y $B = (V_B, E_B)$ son *isomorfos* si existe una biyección $f\colon V_A \to V_B$ tal que para todo $u \in V_A$ y $v \in V_A$, (u, v) es una arista de A si y solo si $(f(u), f(v))$ es una arista de B. Es decir, existe una biyección que preserva la relación de adyacencia entre todo par de vértices.

Considere el problema de decisión *Subgrafo Isomorfo* definido de la siguiente manera: dados dos grafos A y G representados por sus matrices de adyacencia, ¿existe un subgrafo B de G tal que A es isomorfo a B?

(a) Demuestre que *Subgrafo Isomorfo* pertenece a $\mathcal{NP}$.

(b) Demuestre que *Subgrafo Isomorfo* es $\mathcal{NP}$-Completo.
Sugerencia: el problema *Clique* es $\mathcal{NP}$-Completo: dado un grafo $G = (V, E)$ y un entero $k \leq |V|$, ¿existe $S \subset V$, con al menos k vértices, tal que para todo par de vértices $u, v \in S$ existe una arista entre ellos en G?

Solución:

(a) Definimos un certificado C como una lista de pares $(a, v) \in V_A \times V_G$, donde V_A y V_G son los conjuntos de vértices de $A = (V_A, E_A)$ y $G = (V_G, E_G)$, respectivamente. Cada elemento (a, v) de la lista C indica que $f(a) = v$ en una presunta biyección $f\colon V_A \to V_B$ que preserva la relación de adyacencia entre vértices de A y los de un subgrafo $B = (V_B, E_B)$ de G.

```
1  Algorithm Certificador (A = (V_A, E_A),G = (V_G, E_G),C = (a_1, v_1), (a_2, v_2) , ... (a_n, v_n))
2      if largo de C ≠ |V_A| then
3          return false
4      end
5      if a_i = a_j para algún i ≠ j, 1 ≤ i, j ≤ n then
6          return false
7      end
8      if v_i = v_j para algún i ≠ j, 1 ≤ i, j ≤ n then
9          return false
10     end
11     Sean M_A y M_G las matrices de adyacencia de A y G, respectivamente.
12     foreach (a, a') ∈ V_A × V_A do
13         Buscar en C el vértice v ∈ V_G tal que (a, v) ∈ C
14         Buscar en C el vértice v' ∈ V_G tal que (a', v') ∈ C
15         if M_A[a, a'] ≠ M_G[v, v'] then
16             return false
17         end
18     end
19     return true
20 end
```

Figura 2: Algoritmo certificador para *Subgrafo Isomorfo*.

Demostramos a continuación que el algoritmo de la figura 2 es un certificador eficiente para el problema *Subgrafo Isomorfo*.

Comenzamos mostrando que es de tiempo polinomial. Los pasos 2,5, 8, 13 y 14 requieren claramente tiempo polinomial en el largo de C. El ciclo del paso 12 se repite $|V_A|^2$ veces y el resto de los pasos requieren tiempo $O(1)$, por lo cual concluimos que el tiempo de ejecución del algoritmo completo es polinomial en el tamaño de la entradas A y C.

Supongamos que (A, G) es una instancia SÍ del problema. Entonces existe una biyección $f\colon V_A \to V_B$ que preserva la relación de adyacencia entre vértices de A y los de un subgrafo $B = (V_B, E_B)$ de G. Definiendo C como la lista que contiene los $|V_A|$ pares de la forma $(a, f(a))$, $a \in V_A$, se cumple que:

- El tamaño de C es polinomial en el tamaño de la entrada A por definición.
- Las condiciones de los pasos 2 y 5 no se cumplen por definición de C
- La condición del paso 8 no se cumple porque f es sobreyectiva.

- La condición del paso 15 no se cumple para ningún $(a, a') \in V_A \times V_A$ porque f preserva la relación de adyacencia.

Vemos entonces que existe un certificado de tamaño polinomial en las entradas del problema que hace que el certificador responde true.

Supongamos ahora que para ciertas entradas A, G, C el algoritmo responde true. Las condiciones de los pasos 2 y 5 garantizan que para cada $a \in V_A$ existe una única entrada (a, v) en C. Podemos definir entonces una función $f : V_A \to V_G$ donde $f(a)$ es el elemento v de esa única entrada. Más aún, esa función es inyectiva por la condición del paso 8. Sea V_B la imagen de f y B el subgrafo de G con conjunto de vértices V_B. La restricción de f al codominio V_B es efectivamente una biyección, y además preserva la relación de adyacencia entre vértices de A y los de B porque en caso contrario la condición del paso 15 habría evaluado en true para algún par (a, a'). Esto demuestra que (A, G) es una instancia SÍ del problema.

(b) Como *Clique* es $\mathcal{NP}$-Completo y *Subgrafo Isomorfo* pertenece a $\mathcal{NP}$, alcanza con probar que *Clique* $\leq_P$ *Subgrafo isomorfo*, para lo cual definimos a continuación una reducción de tiempo polinomial. El algoritmo de la figura 3 resuelve *Clique* usando la función auxiliar *ResolverSubgrafoIsomorfo* que resuelve el problema *Subgrafo Isomorfo*.

La reducción se realiza en tiempo polinomial en el tamaño de la entrada ya que, como $k \leq |V|$, la construcción de K requiere tiempo $O(|V|^2)$.

```
1 Algorithm ResolverClique (G = (V, E),k)
2     Construir un grafo completo K de k vértices
3     return ResolverSubgrafoIsomorfo (K, G)
4 end
```

Figura 3: Algoritmo para resolver *Clique* usando como rutina auxiliar un algoritmo que resuelve *Subgrafo Isomorfo*.

Si G contiene un *Clique* de tamaño k, entonces ese subgrafo es por definición un grafo completo de k vértices y por lo tanto forma un isomorfismo con K. En consecuencia el algoritmo devuelve true. Recíprocamente, si el algoritmo devuelve true, entonces G contiene un subgrafo B isomorfo con K, y ese subgrafo B es un Clique de tamaño k de G. Concluimos que este algoritmo devuelve true si y solo si $(G = (V, E), k)$ es una instancia SÍ de *Clique*.