Examen de Programación 3
13 de febrero de 2023

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

Ejercicio 1 (32 puntos)

Consideremos el problema al que se enfrenta un hospital que intenta evaluar si sus reservas de sangre donada son suficientes para atender la demanda de transfusiones.

La propia sangre de una persona tiene ciertos antígenos (A y B) presentes y una persona no puede recibir sangre con un antígeno particular si su propia sangre no tiene este antígeno presente. Esto lleva a una clasificación de la sangre en cuatro tipos: A, B, AB y O. La sangre del tipo A tiene el antígeno A, la sangre del tipo B tiene el antígeno B, la sangre del tipo AB tiene ambos y la sangre del tipo O no tiene ninguno de los dos.

Por lo tanto, los pacientes con el tipo A sólo pueden recibir los tipos de sangre A u O en una transfusión, los pacientes con el tipo B sólo pueden recibir B u O, los pacientes con el tipo O sólo pueden recibir O, y los pacientes con el tipo AB pueden recibir cualquiera de los cuatro tipos.

(a) Sean s_O, s_A, s_B y s_{AB} las cantidades disponibles de los distintos tipos de sangre en unidades enteras. Supongamos que el hospital conoce la demanda prevista de cada tipo de sangre d_O, d_A, d_B y d_{AB} para la semana siguiente. Modele este problema utilizando alguna de las técnicas vistas en el curso y **mencione** un algoritmo de tiempo polinómico que evalúe si la sangre disponible es suficiente para cubrir las necesidades previstas. Explique por qué, de esa manera, se resuelve el problema que se le plantea al hospital (*correctitud*).

(b) Considere el siguiente ejemplo. Durante la próxima semana, se espera necesitar como máximo 100 unidades de sangre. La distribución típica de los tipos de sangre es de aproximadamente un 45 % del tipo O, un 42 % del tipo A, un 10 % del tipo B y un 3 % del tipo AB. El hospital quiere saber si el suministro de sangre que tiene disponible sería suficiente si llegaran 100 pacientes con la distribución de tipos esperada. Hay un total de 105 unidades de sangre disponibles. En la tabla siguiente se indican las necesidades y las reservas disponibles.

tipo de sangre	oferta	demanda
O	50	45
A	36	42
B	11	10
AB	8	3

¿Son suficientes las 105 unidades de sangre disponibles para satisfacer las 100 unidades de demanda? Utilice un argumento basado en la capacidad de algún corte para demostrar por qué no todos los pacientes pueden recibir sangre.

Encuentre una asignación que maximice la cantidad de transfusiones.

Solución:

(a) Construímos un grafo G = (V, E) tal que el conjunto de vértices esté formado por un nodo fuente, cuatro nodos que representan el suministro de cada tipo de sangre adyacentes a la fuente, cuatro nodos que representan la demanda y un nodo sumidero que es adyacente a los nodos de demanda. Para cada nodo de suministro u y nodo de demanda v, construímos una arista (u, v) si el tipo v puede recibir sangre del tipo u y establecemos la capacidad en ∞. Construímos una arista (s, u) entre la fuente s y cada nodo de suministro u con la capacidad establecida en el suministro disponible del tipo u. Del mismo modo, para cada nodo de demanda v y el sumidero t, construímos una arista (v, t) con la capacidad establecida en la demanda del tipo v.

Luego calculamos un flujo máximo (de valor entero) en este grafo. La construcción del grafo se puede hacer en tiempo constante y el flujo máximo lo calculamos utilizando Frod-Fulkerson, que acepta

una implementación que se ejecuta en tiempo $O(mC)$, donde m es la cantidad de aristas del grafo y $C = \sum_{e \text{ salientes de } s} c_e$ (suma de las capacidades de las aristas salientes de s).

Podemos decir que hay oferta suficiente para la necesidad proyectada si y sólo si las aristas desde los nodos de demanda hasta el sumidero están todas saturadas en el flujo máximo resultante.

En efecto, si hay suministro suficiente, en el que s_{ST} unidades de tipo S se utilizan para pacientes de tipo T, entonces podemos enviar un flujo desde el nodo de suministro de tipo S al nodo de demanda de tipo T, y respetar todas las condiciones de capacidad.

A la inversa, si hay un flujo que satura todas las aristas desde los nodos de demanda al sumidero, entonces hay un flujo entero con esta propiedad; si envía f_{ST} unidades de flujo desde el nodo de suministro del tipo S al nodo de demanda del tipo T, entonces podemos utilizar f_{ST} unidades del tipo S para pacientes del tipo T.

(b) Considere un corte que contenga la fuente s y los nodos de suministro y demanda del tipo B y AB. La capacidad de este corte es $50 + 36 + 10 + 3 = 99$, por lo que no se pueden satisfacer las 100 unidades de demanda.

Ejercicio 2 (32 puntos)

Considere el algoritmo de Gale-Shapley para formar un emparejamiento estable entre dos conjuntos, $M = \{m_1, m_2, \ldots, m_n\}$ y $W = \{w_1, w_2, \ldots, w_n\}$. En la versión presentada en la Figura 1, las preferencias de los elementos de M están determinadas por listas de preferencias que constituyen parte de la entrada del algoritmo (igual que en el libro de referencia), pero las preferencias de los elementos de W se resuelven en tiempo de ejecución mediante invocaciones a una función de booleana, $pref$. Para $w \in W$, $m \in M$ y $m' \in M$, el resultado de $pref(w, m, m')$ es **true** si w prefiere a m antes que a m' y **false** en caso contrario.

```
1  Algorithm Gale-Shapley
2      Inicialmente p está libre para todo p ∈ M ∪ W
3      while existe m ∈ M libre que no se ha propuesto a todo w ∈ W do
4          Sea w el elemento de W de mayor preferencia para m al cual m no se ha propuesto
5          if w está libre then
6              Emparejar m con w
7          else
8              Sea m' la actual pareja de w
9              if pref(w, m, m') then
10                 Separar a w de m' y emparejar m con w
11             else
12                 w rechaza a m
```

Figura 1: Algoritmo para formar un emparejamiento estable.

(a) Muestre que el algoritmo termina. Repita cualquier argumento que utilice de los estudiados en el curso.

(b) Supongamos que $pref(w, m, m')$ requiere tiempo $O(n^3)$ si w está al final de la lista de preferencias de m y requiere tiempo $O(1)$ en otro caso. Muestre que este algoritmo admite una implementación cuyo tiempo de ejecución es $O(n^2)$. Puede citar sin reescribir resultados presentados en el libro del curso.

Solución:

(a) El algoritmo tiene un único ciclo, y cada una de sus iteraciones se puede interpretar como una propuesta. Cada propuesta corresponde a un par (m, w) diferente porque m no se había propuesto a w. Por lo tanto en cada iteración aumenta de manera estricta (se incrementa en uno) la cantidad de propuestas, por lo que esta cantidad sirve como `medida de avance`. Como la cantidad de pares (m, w) es n^2, este valor es una cota superior a la cantidad de propuestas y por lo tanto de iteraciones.

Por lo tanto si el algortimo no hubiera terminado antes por no quedar ningún m libre, terminaría por haberse realizado todas las propuestas.

Entonces, el algoritmo termina en $O(n^2)$ iteraciones.

> Se debe notar que para demostrar que el algoritmo termina no hace falta demostrar que termina con un emparejamiento perfecto, ni que ningún m queda libre.

(b) En el curso se vio que el algoritmo de emparejamiento estable admite una implementación $O(n^2)$. El algoritmo propuesto puede tomar esa misma implementación, excepto por tener que hacer a lo sumo una invocación a `pref` en cada iteración. Como ya se vio que la cantidad de iteraciones es $O(n^2)$ si se demuestra que cada una de esas invocaciones es $O(1)$ se habrá demostrado lo pedido.

Notemos que (y fue visto en el curso) una vez que w recibe una propuesta nunca deja de estar en pareja. Además, se puede ver que cada w que está en pareja lo está con un solo m, por lo que la cantidad de ms en pareja es igual a la cantidad de ws en pareja.

Supongamos que en cierta iteración durante la ejecución del algoritmo ocurre que cierto $m \in M$ hace una propuesta a $w^* \in W$, que es el último en su lista de preferencia. Como m propone en orden de preferencia, esto implica que todo $w \in W \setminus \{w^*\}$ ha recibido una propuesta y, por lo tanto, está en

pareja. Como m está libre y ningún $m' \in M$ está en pareja con más de un $w \in W$, necesariamente w^* está libre. Por lo tanto, la condición del paso 5 se satisface y en consecuencia el paso 9 no se ejecuta. Concluimos que para ninguna invocación a $pref(w, m, m')$ se cumple que w está al final de la lista de preferencias de m, y en consecuencia todas las invocaciones requieren tiempo $O(1)$.

Ejercicio 3 (36 puntos)

Sea $G = (V, E)$ un grafo dirigido acíclico. Sus n vértices se identifican con enteros 1 a n y se asume que la secuencia $(1, \ldots, n)$ es un ordenamiento topológico de G. Se quiere calcular uno de los caminos más largos de G. A modo de ejemplo, en el grafo de la Figura 2, la secuencia $(1, 2, 3, 4, 5, 6)$ es un ordenamiento topológico y el único camino más largo es $(2, 3, 5, 6)$.

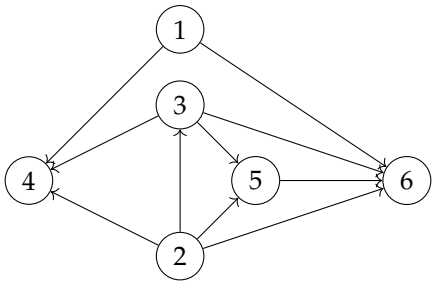

Figura 2: Ejemplo de grafo.

(a) Defina una relación de recurrencia que exprese para cada vértice v la longitud del camino más largo que empieza en v. Justifique.

(b) De un algoritmo iterativo que, mediante la relación definida en la parte (a), calcule la longitud del camino más largo de G. El tiempo de ejecución del algoritmo debe ser $O(|V| + |E|)$. Explique por qué se cumple ese tiempo.

(c) De el algoritmo que devuelva el camino más largo. Puede hacerlo indicando qué modificaciones se debe hacer al algoritmo de la parte (b).

Solución:

(a) Si un vértice v no tiene aristas salientes el camino más largo que empieza en v consiste en ese vértice, por lo que su longitud es 0. En otro caso, para cada camino que empieza en v el segundo vértice debe ser un w tal que (v, w) es arista del grafo. En particular el camino más largo que empieza en v se obtiene agregando v al principio del camino P, el más largo de los caminos que empiezan en cada uno de esos vértices w. La longitud del nuevo camino es uno más que la longitud de P.

$$L(v) = \begin{cases} 0 \text{ si no hay aristas salientes de } v, \\ 1 + \text{máx}_{(v,w)\in E}\{L(w)\} \end{cases}$$

(b) Se calcula para cada vértice la longitud del camino más largo que empieza en él. El más largo de esos caminos es el más largo del grafo.

$$\text{CML} = \max_{v\in\{1\ldots n\}} \{L_v\}.$$

Cada L_v se calcula después de haber calculado los L_w para $w > v$.

```
1  Algorithm CaminoMasLargo (G = (V,E))
       Precondición: (1, ..., n) es un ordenamiento topológico de G
2      CML ← 0
3      for v ← n to 1 do
4          L[v] ← 0
5          foreach (v, w) ∈ E do
6              if L[w] ≥ L[v] then
7                  L[v] ← 1 + L[w]
8          if L[v] ≥ CML then
9              CML ← L[v]
10     return CML
11 end
```

Figura 3: Algoritmo para encontrar la longitud del camino más largo.

El ciclo de la línea 3 se ejecuta exactamente una vez para cada vértice. El ciclo de la línea 5 se ejecuta exactamente una vez para cada arista, cada una de las cuales se pueden encontrar en $O(1)$ manteniendo la representación del grafo con listas de adyacencia. Cada una del resto de las operaciones es $O(1)$. Por lo tanto el costo del algortimo es $O(|V| + |E|)$.

(c) Se debe calcular el vértice en que inicia el camino más largo, y para cada vértice cuál es el siguiente en el camino más largo que empieza en él.

Si se cumple la condición de la línea 8 se asigna v como valor de `inicio` (esta variable puede inicializarse por ejemplo con n, aunque de todas formas se va a modificar).

Para cada vértice v, al empezar cada iteración de la línea 3 se inicializa `sig[v]` con un valor que indique que no está definido, por ejemplo 0. Si se cumple la condición de la línea 6 se asigna w como valor de `sig[v]`.

Al final, la línea 10 se sustituye con lo siguiente

```
10     camino ← (inicio)
11     actual ← inicio
12     while está definido sig[actual] do
13         actual ← sig[actual]
14         agregar actual al final de camino
15     end
16     return camino
```