Parcial de Programación 3
15 de octubre de 2022

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

Ejercicio 1 (16 puntos)

En un videojuego participan un conjunto J de jugadores que deben asignarse a un conjunto S de servidores. Cada jugador se asigna a un único servidor, y cada servidor tendrá cupo para $\frac{|J|}{|S|}$ jugadores, de forma que los mismos se distribuyan equitativamente. Para simplificar el problema suponemos que $|J|$ es múltiplo de $|S|$.

Los servidores y los jugadores se ubican distribuidos geográficamente, afectando esto al rendimiento de la conexión. Para cada par (j, s) (con $j \in J$ y $s \in S$) se conoce un valor que corresponde al tiempo estimado de viaje de un paquete de datos por la Internet entre ambos, que denotamos $ping(j, s)$. Podemos suponer que todos los pares tienen valores distintos.

Para facilitar la tarea, suponga que cada servidor s calcula previamente el tiempo de respuesta hacia cada jugador j obteniendo una lista $Ping_s$, ordenada en forma creciente por valor de $ping(j, s)$. También suponga que cada jugador j calcula previamente el tiempo de respuesta hacia cada servidor s obteniendo una lista $Ping_j$ ordenada en forma creciente por valor de $ping(j, s)$.

Una *asignación* de jugadores a servidores es *subóptima* si existen jugadores j, j' y servidores s, s' tales que:

1. j está asignado a s.
2. j' está asignado a s'.
3. $ping(j, s') < ping(j, s)$.
4. $ping(j, s') < ping(j', s')$

De lo contrario, decimos que la asignación es *óptima*.

(a) Dé un algoritmo para obtener una asignación óptima. Su algoritmo debe admitir una implementación con tiempo de ejecución $O(|J||S|)$. Asuma disponibles las listas $Ping_s$, $Ping_j$ y la función $ping(j, s)$ que se ejecuta en tiempo constante. Reescriba cualquier algoritmo visto en el curso que utilice.

(b) Demuestre que su algoritmo admite una implementación con tiempo de ejecución $O(|J||S|)$. Reescriba cualquier argumento visto en el curso que utilice.

Solución:

```
1  Algorithm Asignación
       Input: J = {1,...,|J|}, S = {1,...,|S|}
       Input: ping_J, ping_S: arreglos con los pings ordenados
       Input: ping[][], matriz de pings
       Output: J: colección de las asignaciones (j,s)
2      Definimos un arreglo de enteros C, de tamaño |S|; Inicializamos C[i] = |J|/|S| para
        |i ∈ {1..|J|}|;
3      while ∃s_i con C[i] > 0 do
4          s_i "ofrece" conexión al siguiente jugador j_h en la lista ping_S.
5          if j_h está libre then
6              asignar j_h al servidor s_i;
7              C[i] := C[i] − 1;
8          else
9              if j_h está asignado a s_k y ping[j_h, s_k] > ping[j_h, s_i] then
10                 j_h es asignado a s_i;
11                 C[i] := C[i] − 1;
12                 C[k] := C[k] + 1;
13 end
```

Figura 1: Algoritmo para obtener una asignación óptima

(a) Resolvemos el problema con una adaptación del algoritmo de Gale-Shapley, donde los servidores tienen el rol de proponente. El algoritmo se presenta el la Figura **??**.

> Este problema es muy similar al ejercicio visto en la semana 1 (Kleimberg & Tardos, Ex 1.4). En este caso el emparejamiento es exactamente "1 a n".

> Notar que, si bien la letra define conjuntos J y S abstractos, implícitamente trabajamos con índices (por ejemplo, al crear el arreglo C).

(b) La definición del arreglo C, en la línea 2 requiere tiempo $O(|S|)$.

Para identificar servidores con espacio disponible mantenemos una lista encadenada, implementando así una pila (podría también usarse una cola u otra estructura similar). Crear la pila inicialmente tiene tiempo $|O(|S|)|$. La condición para la línea 3 entonces se chequea en tiempo constante. Cuando disminuimos los cupos (líneas 7 y 11) identificamos cuándo $C[i]$ queda nulo, en ese caso no volvemos a insertar el servidor en la pila. Del mismo modo, cuando la ejecución de la línea 12 actualiza algún $C[k]$ de 0 a 1, insertamos s_k en la pila. Todas estas operaciones son de tiempo constante.

Para mantener la información de a qué servidor está asignado cada jugador, mantenemos un arreglo de tamaño $|J|$. Cada entrada tiene valores en el conjunto $S \cup \{\text{NULL}\}$. Inicialmente todas las entradas valen NULL. Iniciarlo lleva tiempo $O(|J|)$. Tanto decidir a dónde está asignado un jugador, como las (re)-asignaciones llevan entonces tiempo constante (líneas 5,9,10).

Para mantener la información sobre quién es el siguiente jugador en la lista de preferencias de cada servidor (necesario en línea 4), usamos un arreglo curr de tamaño $|S|$ de listas de jugadores. La posición s-ésima representa la siguiente posición en la lista de preferencias de s. Inicialmente, este arreglo se inicializa con valor $Ping_s$ en todas las posiciones, lo que requiere tiempo $O(|S|)$. Cuando se ejecuta la línea 4 actualizamos la entrada del arreglo en la posición s para que apunte a la cola de la lista (curr[s] := curr[s].next()).

En la línea 9, obtenemos $ping[j_h, s_i]$, y $ping[j_h, s_k]$ en tiempo constante accediendo a la matriz $ping$.

Entonces, la inicialización de las estructuras lleva tiempo $O(|J| + |S|)$ y la ejecución de una iteración del bucle tiempo constante. Vamos a probar que la cantidad de iteraciones del bucle está acotada por

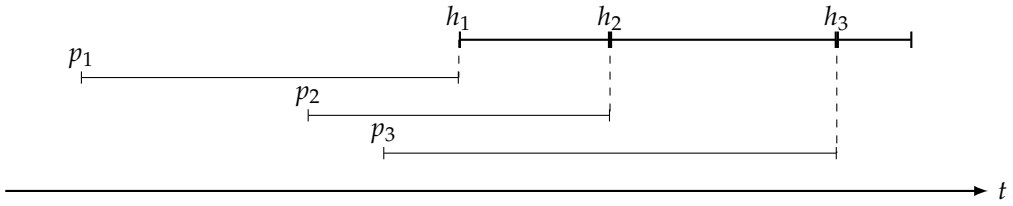
$|J||S|$. Para ello, definimos una medida de progreso. Consideramos la familia de conjuntos $\mathcal{P}_i \subseteq S \times J$, para cada iteración i del while. El conjunto $\mathcal{P}_i$ contiene las parejas (s, j) tales que s propuso conexión a j durante alguna de las primeras i iteraciones del while. Como las proposiciones no se repiten, $|P_{i+1}| = |P_i| + 1$. Como $|\mathcal{P}_\rangle|$ está acotado por $|S \times J| = |S||J|$ concluimos que a lo sumo puede haber $|S||J|$ iteraciones.

Concluimos entonces que el tiempo de ejecución del algoritmo es de orden $O(|J||S|)$.

Ejercicio 2 (18 puntos)

Una panadería planea elaborar sus panificados en el menor tiempo posible. La elaboración de cada panificado, $i \in \{1, ..., n\}$, consiste en una preparación inicial y un horneado inmediato posterior. Los tiempos de preparación y horneado son todos diferentes y denotados según p_i y h_i, respectivamente. Mientras que la preparación es independiente entre los panificados, es decir su preparación se puede realizar simultáneamente, el horneado no lo es, debido a que solo se puede hornear un panificado a la vez. Se requiere determinar en qué orden hornear los panificados de modo de minimizar el tiempo total de elaboración.

A modo de ejemplo se presenta un esquema del tiempo de elaboración para una solución factible, no necesariamente óptima, de un instancia con tres panificados,



donde los tiempos de preparación y horneado se representan por intervalos etiquetados a la izquierda.

Observación: no se requiere determinar tiempos de comienzo o fin de preparaciones u horneados.

(a) Dé un algoritmo eficiente para resolver el problema.

(b) Demuestre la corrección de su algoritmo. **Sugerencia**: utilice la técnica *exchange argument*, aplicada a inversiones entre soluciones factibles.

Solución:

(a) Entrada: un conjunto de tuplas (p_i, h_i) con $i \in \{1, ..., n\}$
Salida: una secuencia $O = (o_1, ..., o_n)$, permutación de $\{1, ..., n\}$
Ordenar el horneado de los panificados según orden creciente de su tiempo de preparación; es decir, para todo par (o_i, o_{i+1}), con $i \in \{1, ..., n-1\}$, se cumple $p_{o_i} < p_{o_{i+1}}$.

(b) El tiempo de elaboración a partir del horneado del panificado o_i es la suma de sus tiempos de preparación y horneado mas el tiempo de horneado de los restantes panificados en la secuencia,

$$t_{o_i} := p_{o_i} + h_{o_i} + \sum_{j=i+1}^{n} h_{o_j}.$$

El tiempo de elaboración de los panificados en una secuencia O es el máximo tiempo de elaboración a partir del horneado de alguno de sus panificados,

$$T_O := \max_{i \in \{1,...,n\}} t_{o_i}.$$

Una secuencia óptima, $O^* = (o_1^*, ..., o_n^*)$, minimiza el tiempo máximo de elaboración entre las secuencias posibles ($\mathcal{O}$),

$$T_{O^*} := \min_{O \in \mathcal{O}} T_O.$$

Se demuestra, mediante *exchange argument*, que dada la secuencia obtenida por el algoritmo, $O^* = (o_1^*, ..., o_n^*)$, y para toda secuencia factible $O = (o_1, ..., o_n)$, se cumple que $T_{O^*} \leq T_O$.

Si O difiere de O^* entonces existe al menos una inversión entre dos posiciones contiguas de los índices en O con respecto a O^*. Es decir, en O existen panificados o_i y o_{i+1}, con $i \in \{1, ..., n-1\}$, donde o_i es horneado antes que o_{i+1} y $p_{o_i} > p_{o_{i+1}}$. Se busca transformar O revertiendo dicha inversión de forma que O transformada se asemeje a O^*, sin empeorar el tiempo de elaboración.

Sean o_i e o_{i+1} los panificados de dicha inversión, a partir de

$$O = (o_1, ..., o_{i-1}, o_i, o_{i+1}, o_{i+2}, ..., o_n)$$

se genera la secuencia transformada (O'), en base a los índices de O, donde se revierte la inversión,

$$O' = (o_1, ..., o_{i-1}, o_{i+1}, o_i, o_{i+2}, ..., o_n).$$

Se tiene que los tiempos de elaboración a partir de los horneados, t_{o_j}, de los panificados previos, $o_1, ..., o_{i-1}$, y posteriores, $o_{i+2}, ..., o_n$, a la inversión no cambian luego de revertirla.
Los tiempos de elaboración a partir del horneado de los panificados de la inversión pueden cambiar. Estos tiempos según secuencia son

$$t_{o_i}(O) := p_{o_i} + h_{o_i} + h_{o_{i+1}} + \sum_{j=i+2}^{n} h_{o_j}$$

$$t_{o_{i+1}}(O) := p_{o_{i+1}} + h_{o_{i+1}} + \sum_{j=i+2}^{n} h_{o_j}$$

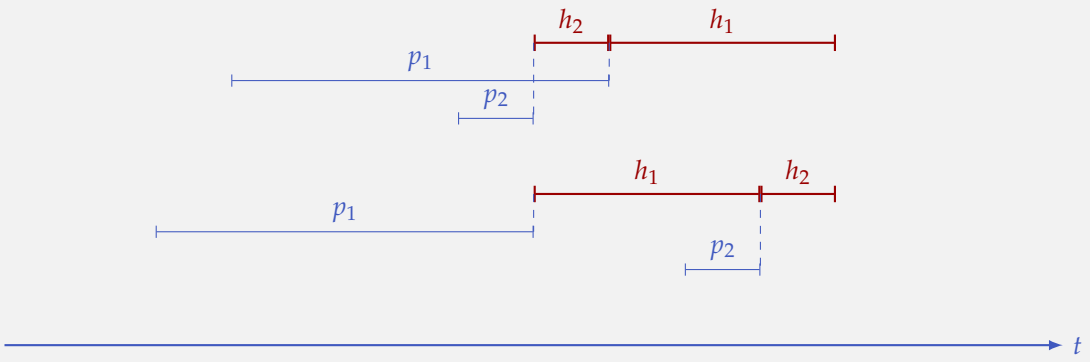
$$t_{o_{i+1}}(O') := p_{o_{i+1}} + h_{o_{i+1}} + h_{o_i} + \sum_{j=i+2}^{n} h_{o_j}$$

$$t_{o_i}(O') := p_{o_i} + h_{o_i} + \sum_{j=i+2}^{n} h_{o_j}$$

Se tiene que $t_{o_i}(O')$ es menor o igual que $t_{o_i}(O)$.
Para que $t_{o_{i+1}}(O')$ sea menor o igual que $t_{o_i}(O)$ se requiere que $p_{o_{i+1}} \leq p_{o_i}$; lo cual es el criterio del algoritmo para generar su secuencia solución. Usando dicho criterio se tiene que $T_{O'} \leq T_O$.

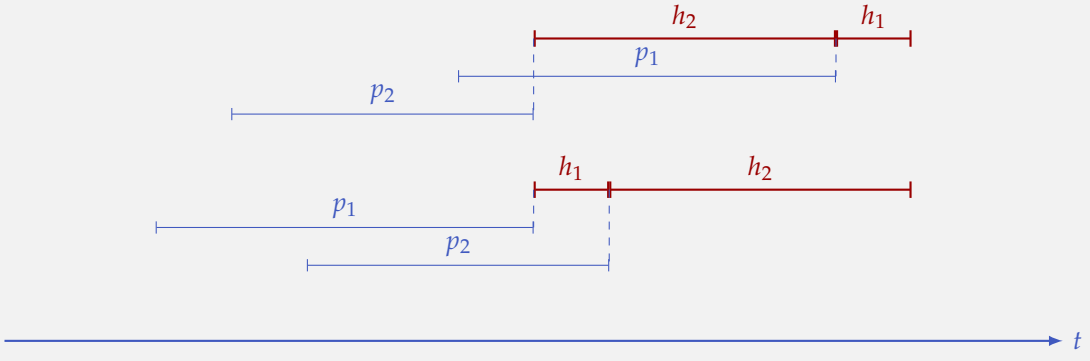
La eliminación reiterada de las inversiones de cualquier secuencia O con respecto a O^* mediante el criterio del algoritmo no aumenta el tiempo total de elaboración en O^*. Por lo que el algoritmo determina una secuencia solución que es al menos tan buena como cualquier otra secuencia.

En la siguiente figura la solución óptima se obtiene horneando primero el panificado 2 y luego el panificado 1. Se puede ver que ordenando de manera decreciente según p_i, o de manera decreciente según $p_i + h_i$, o de manera decreciente según h_i no se obtiene la solución óptima.

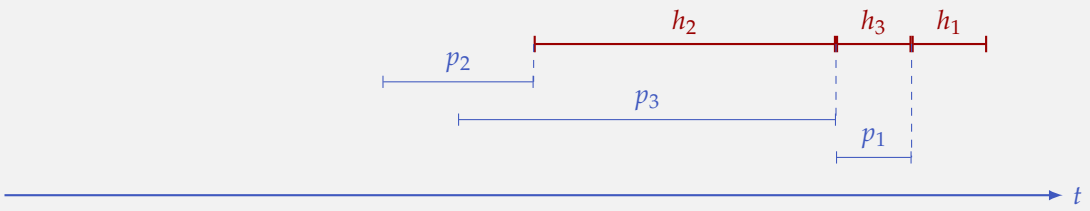


Además se observa que el orden en que se hornean no tiene por qué coinidir con el orden en que se empiezan a preparar. Se debe recordar que el horneado empieza inmediatamente terminada la preparación.

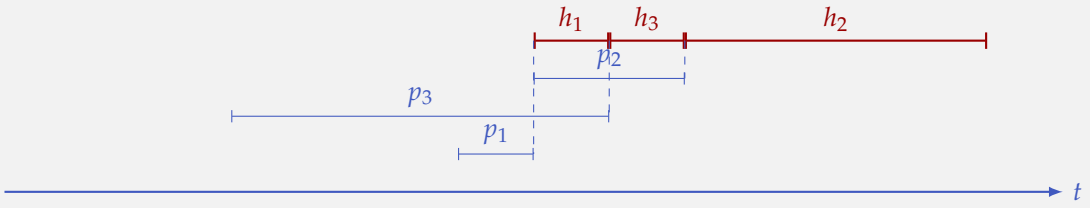
En la siguiente figura la solución óptima también se obtiene horneando primero el panificado 2 y luego el panificado 1. Se puede ver que tampoco se obtiene la solución óptima ordenando de manera creciente según $p_i + h_i$, o de manera creciente según h_i.



En la siguiente ejemplo se puede ver que la eliminación de una inversión que no ocurre en posiciones consecutivas no necesariamente mejora la solución.



Hay dos inversiones, $(2,1)$ y $(3,1)$. La inversión $(2,1)$ no ocurre en posiciones consecutivas. Si se elimina también se elimina la inversión $(3,1)$, pero se crea la nueva inversión $(3,2)$.



En este ejemlo, al eliminar la inversión, como no se da en posiciones consecutivas, aumentó el tiempo total de elaboración.

Ejercicio 3 (16 puntos)

Sea $G = (V, E)$, donde $|V| = n$ y $|E| = m$, un grafo dirigido y acíclico (DAG) con costos positivos en las aristas, $c(u, v)$ para todo (u, v) en E. Sea s un nodo de G, tal que existe un camino de s a v para todo nodo v en V.

(a) Dé un algoritmo *CostoCaminosDeMenorCosto(G, s)* que, dados el grafo G y el nodo s, calcule los costos de los caminos de menor costo de s a todo otro nodo en G. Su algoritmo debe admitir una implementación con tiempo de ejecución $O(m + n)$. Reescriba cualquier algoritmo visto en el curso que utilice.

Sugerencia: Comience por calcular un orden topológico de G.

(b) Demuestre que su algoritmo admite una implementación con tiempo de ejecución O($m + n$). Enuncie cualquier resultado visto en el curso que utilice.

Solución:

(a) El algoritmo de la Figura **??** soluciona el problema. El algoritmo se diseña teniendo en cuenta la siguiente recurrencia:

$$d[v] = \begin{cases} 0 & \text{si } v = s \\ \min_{(u,v)\in E}\{d[u] + c(u, v)\} & \text{en otro caso.} \end{cases} \quad (1)$$

Por lo tanto, para poder determinar cada $d[v]$ es necesario conocer los $d[u]$ correspondientes a los vértices origen de sus aristas entrantes.

```
1  Algorithm CostoCaminosDeMenorCosto(G, s)
2      Inicializar conjunto vacío S
3      foreach u ∈ V do
4          Hacer indeg[u] igual al grado de entrada de u
5          Agregar u a S si indeg[u] es cero
6      end
7      Crear una lista vacía OT
8      while S no es vacío do
9          Retirar un vértice u de S y agregarlo al final de OT
10         foreach arista (u, v) saliente de u do
11             Decrementar indeg[v]
12             if indeg[v]=0 then
13                 Agregar v a S
14             end
15         end
16     end
17     Sea v_1, v_2, ..., v_n el orden de los nodos en OT
18     Hacer d[v_1] = 0 y d[v_i] = ∞ para todo 2 ≤ i ≤ n
19     for i = 2 a i = n do
20         foreach arista (u, v_i) incidente a v_i do
21             if d[u] + c(u, v_i) < d[v_i] then
22                 d[v_i] = d[u] + c(u, v_i)
23             end
24         end
25     end
26     return d
27 end
```

Figura 2: Algoritmo que calcula los largos de los caminos más cortos desde el nodo s a todo otro nodo del grafo.

(b) Representamos el grafo G mediante listas de adyacencia e identificamos los vértices con los enteros de 1 hasta n. Puede asumirse que G viene dado mediante esa representación, o en caso alternativo la construcción de G puede realizarse en tiempo $O(m+n)$ recorriendo una única vez cada arista de E y agregándola a la representación.

Para justificar el tiempo de ejecución de calcular un orden topológico (OT) de G basta con enunciar el resultado teórico del libro que muestra que el algoritmo presentado admite una implementación cuyo tiempo de ejecución es $O(n+m)$ utilizando una representacón de G mediante listas de adyacencia. Una justificación detallada se presenta a continuación.

Representamos el conjunto S mediante una pila, tal que las inserciones y supresiones de vértices de S requieren tiempo $O(1)$. La iteración del paso **??** se ejecuta una vez por cada vértice, y calcular el grado de entrada de un vértice u equivale a contar su cantidad de aristas incidentes, que lleva un tiempo $O(n_u)$ recorriendo su lista de adyacentes. Como $\sum_{u \in V} n_u = m$ (para grafos dirigidos), el tiempo total de ejecución para la inicialización de el arreglo $indeg$ y el conjunto S es $O(m+n)$.

Por otro lado, el ciclo del paso **??** se ejecuta un total de n veces, ya que cada vértice u de G es retirado a lo sumo una vez de S en el paso **??** y una vez retirado no se lo vuelve a agregar. El ciclo del paso **??** se ejecuta n_u veces en cada iteración, dando un total de $\sum_{u \in V} n_u$ iteraciones. Como cada operación en los ciclos es de tiempo constante, el tiempo total que insume la ejecución del ciclo del paso **??** es $O(n+m)$.
Entonces, utilizando la regla de la suma, concluimos que el tiempo de ejecución de el algoritmo para calcular el OT es $O(m+n)$.

Por otra parte, en el ciclo del paso **??** se itera una vez por cada nodo v distinto de s, y en el cuerpo del ciclo se itera una vez por cada arista incidente a v, realizando dos operaciones de tiempo constante en cada iteración. Como ya explicamos anteriormente, el tiempo de ejecución de un ciclo de estas características es $O(m+n)$.

Por último, concluímos por relga de la suma que el tiempo de ejecución total del algoritmo es $O(m+n)$.

Solución alternativa sin obtener el orden topológico de manera explícita

```
1  Algorithm CostoCaminosDeMenorCosto(G, s)
2      Inicializar conjunto vacío S
3      foreach u ∈ V do
4          Hacer indeg[u] igual al grado de entrada de u
5          Agregar u a S si indeg[u] es cero
6      d[u] = ∞ para cada vértice u ∈ V
7      d[s] = 0
8      while S ≠ ∅ do
9          u = obtener y remover un elemento de S
10         foreach (u, v) ∈ E do
11             d[v] = mín{d[v], d[u] + c(u, v)}
12             indeg[v] = indeg[v] − 1
13             if indeg[v] == 0 then S = S ∪ {v}
```

Figura 3: Versión sin obtener OT de manera explicita.