Parcial de Programación 3
18 de setiembre de 2023

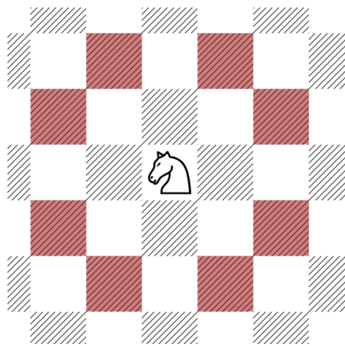
En recuadros con este formato aparecerán aclaraciones que cumplen una función explicativa pero que no eran requeridos como parte de la solución.

Ejercicio 1 (18 puntos)

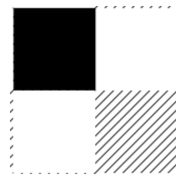
Consideramos un tablero de ajedrez de tamaño $n \times n$. Identificamos cada casilla por un par (i, j) $(1 \leq i \leq n, 1 \leq j \leq n)$ donde i y j indican fila y columna, respectivamente. Un *caballo* es una pieza de ajedrez que puede moverse de una casilla (i, j) a cualquier casilla del conjunto

$$C_{i,j} = \Big\{(i+1, j+2), (i+1, j-2), (i-1, j+2), (i-1, j-2),$$
$$(i+2, j+1), (i+2, j-1), (i-2, j+1), (i-2, j-1)\Big\} \bigcap \{1, 2, \ldots, n\}^2 .$$

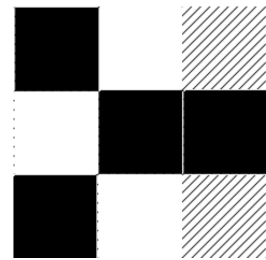
Para el caballo de la figura (a) las casillas de $C_{i,j}$ se muestran sombreadas en tono más oscuro. Notar que la intersección con $\{1, 2, \ldots, n\}^2$ en la ecuación anterior excluye casillas fuera de los límites del tablero.



(a) Movimientos del caballo



(b) Un tablero de 2×2 con un agujero



(c) Un tablero de 3×3 con cuatro agujeros

Un *tablero con agujeros* es un tablero de ajedrez en donde se eliminaron una o más casillas. El *problema del caballo* consiste en decidir si, dado un tablero con agujeros, es posible para un caballo visitar todas las demás casillas con alguna sucesión de movimientos (se puede partir de cualquier casilla y repetir casillas en el recorrido). En este caso decimos que el problema del caballo *tiene solución*.

Por ejemplo, para el tablero de la figura (b), que tiene un agujero representado en negro, el problema no tiene solución porque, cualquiera sea la casilla inicial, el caballo no puede moverse y por lo tanto no puede recorrer las demás casillas. En cambio para el tablero de la figura (c), el problema tiene solución, pues partiendo de $(1, 2)$ se puede hacer la siguiente recorrida:

$$(1, 2) \to (3, 3) \to (2, 1) \to (1, 3) \to (3, 2) .$$

Observación: La casilla de inicio es en realidad irrelevante; si el problema tiene solución partiendo de una casilla determinada entonces tiene solución partiendo desde cualquiera donde no haya agujero.

(a) Dé un algoritmo que decida si un problema del caballo tiene solución. La entrada es un tablero con agujeros representado como un entero n y una matriz booleana de $n \times n$ que indica con `true` los agujeros y con `false` las casillas disponibles. Para simplificar la gestión de los bordes, puede asumir dada una función `C(n,i,j)` que retorna en tiempo $O(1)$ la lista de casillas en $C_{i,j}$ para un tablero de tamaño $n \times n$. Por ejemplo `C(5,1,1)` retorna la lista con dos casillas `[(3,2),(2,3)]`.
Su algoritmo debe admitir una implementación que ejecute en tiempo $O(n^2)$.
Reescriba cualquier algoritmo que utilice de los estudiados en el curso.
Sugerencia: Reduzca el problema a uno de conectividad de grafos.

(b) Demuestre que su algoritmo admite una implementación con tiempo de ejecución $O(n^2)$.

Solución:

(a) Podemos construir un grafo donde los nodos son las casillas que no tienen agujero, y las aristas vienen dadas por el movimiento del caballo. El problema se reduce a decidir si el grafo resultante es conexo, lo cual se puede computar en una recorrida (por ejemplo, DFS).

La siguiente solución maneja el grafo implícitamente (la función `C` acompañada con la matriz de agujeros nos permite inferir las listas de adyacencias.

```
1  Algorithm Caballo(n, agujeros)
2      Hacer visitados[i][j] = false para todo i, j, 1 ≤ i, j ≤ n
3      Tomar (i, j) una casilla donde no hay agujero (si no existe, retornar true)
4      S := {(i, j)}
5      while S no vacío do
6          Retirar un par (i, j) de S
7          if not visitados[i][j] then
8              visitados[i][j] := true
9              foreach (x, y) ∈ C(i, j, n) do
10                 if not agujeros[x][y] then
11                     Agregar (x, y) a S
12                 end
13             end
14         end
15     end
16     for i = 1 to n do
17         for j = 1 to n do
18             if not agujeros[i][j] and not visitados[i][j] then
19                 return false
20             end
21         end
22     end
23     return true
24 end
```

(b) La inicialización de la matriz de visitados en el paso 2 requiere tiempo $O(n^2)$.

Para tomar una casilla sin agujero (paso 3) podemos recorrer la matriz *agujeros* hasta encontrar un valor falso. Esto lleva tiempo $O(n^2)$ en el peor caso.

Implementamos S como una lista encadenada. Con esta implementación, la asignación en el paso 4 requiere tiempo acotado por una constante.

A partir de estas observaciones deducimos que el bloque de pasos 2-4 requiere tiempo $O(n^2)$. Analizamos a continuación los pasos 5 a 15.

Notar que $C(i, j, n)$ tiene a lo sumo 8 elementos. Por lo tanto, el tiempo total de cada ejecución del paso 9 (incluyendo sus no más de 8 iteraciones) está acotado por una constante, ya que los pasos 10 y 11 requieren tiempo $O(1)$. Como el resto de las operaciones en el bucle del paso 5 (evaluación de la condición y pasos 6,7,8) requieren tiempo $O(1)$, para demostrar que el tiempo total de ejecución del paso 5 es $O(n^2)$ solo resta probar que la cantidad de iteraciones es $O(n^2)$.

Para esto, observar que los movimientos del caballo son reversibles, es decir, si $(x, y) \in C(i, j, n)$ entonces $(i, j) \in C(x, y, n)$. Por lo tanto, cada par (x, y) pertenece $C(i, j, n)$ para no más de 8 casillas (i, j) distintas. En consecuencia, cada par (x, y) se agrega a lo sumo 8 veces a S, ya que el paso 8 en combinación con la condición del paso 7 aseguran que cada par (i, j) se visita a lo sumo una vez. Esto implica que la cantidad de iteraciones del bucle del paso 5 es $O(n^2)$ como se quería probar.

Finalmente, la búsqueda de una casilla no agujereada sin explorar a partir del paso 16 se realiza en tiempo $O(n^2)$ dado que la condición del paso 18 se evalúa en tiempo $O(1)$ a partir de valores en matrices.

Hemos mostrado entonces que los bloques de pasos 2-4, 5-15, y 16-23 requieren tiempo $O(n^2)$ cada uno, de donde concluimos que el tiempo total de ejecución del algoritmo es $O(n^2)$.

Ejercicio 2 (14 puntos)

El *ciclomotín* es un juego donde se enfrentan dos jugadores. El juego es de carácter asimétrico, por lo que los jugadores tienen roles. Al jugador que inicia una partida se le denomina "mano", y al contrincante "fiador/a".

En un torneo de exhibición han de programarse n partidos emparejando $2n$ jugadores (todos deben jugar un partido). Por criterios establecidos previamente hay n jugadores que deben tomar cada rol. Consideramos M y F los conjuntos de manos y fiadores.

Cada jugador tiene asociado un *rating*, que es un entero positivo que mide su habilidad (calculado previamente según su historial de resultados). Dos jugadores con *rating* similar tienen niveles de habilidad similares.

Para que el torneo sea competitivo se trata de programar los partidos de forma que cada jugador se enfrente a otro de habilidad similar. En particular decimos que una programación es *válida* si no existen $m, m' \in M$, y $f, f' \in F$ tales que se satisfacen a la vez las siguientes condiciones:

1. m se enfrenta a f.
2. m' se enfrenta a f'.
3. $|rating(m) - rating(f')| < |rating(m) - rating(f)|$.
4. $|rating(m) - rating(f')| < |rating(m') - rating(f')|$.

(a) Un organizador propone el siguiente criterio:
"Ordenar manos y fiadores por rating. Emparejar al i-ésimo mano con el i-ésimo fiador."
Muestre que el criterio propuesto no es correcto. Es decir, que no siempre se produce una programación de partidos que satisface lo pedido.

(b) Dé un algoritmo eficiente que dados M, F y una función $rating : M \cup F \to \mathbb{N}$ retorne una programación válida. Puede utilizar algoritmos vistos en el curso como rutinas auxiliares sin necesidad de reescribirlos.

(c) Demuestre que su algoritmo es correcto. Puede utilizar resultados vistos en el curso sin reescribirlos.

Solución:

(a) Basta con encontrar una instancia donde el algoritmo falle. Consideramos $n = 2$ y m_1, m_2, f_1, f_2 tales que $rating(m_1) = 500$, $rating(m_2) = rating(f_1) = 1000$, $rating(f_2) = 1500$.

(b) Reducimos el problema a un problema de emparejamiento estable.

```
1  Algorithm Emparejar(ratingM, ratingF)
2      foreach i ∈ {1...n} do
3          Ordenar los f_j, j ∈ {1...n}, crecientemente por |rating(m_i) − rafing(f_j)| en una lista L
4          Definir MPref[i,-] = L; esta es la lista de preferencia de m_i
5      end
6      foreach i ∈ {1...n} do
7          Ordenar los m_j, j ∈ {1...n}, crecientemente por |rating(f_i) − rafing(m_j)| en una lista
           L
8          Definir FPref[i,-] = L; esta es la lista de preferencia de f_i
9      end
10     emp = GS(MPref, FPref)
11     return emp
12 end
```

(c) En el algoritmo mostrado construimos una instancia del problema de emparejamiento estable entre dos conjuntos, que en este caso son los conjuntos M y F. Notar que como el algoritmo de Gale-Shapley construye un emparejamiento estable en la linea 10, nuestro algoritmo devuelve un emparejamiento de los manos con los fiadores en el cual todos los jugadores aparecen exactamente en un par. Lo que resta probar es que no aparecen pares cumpliendo la condición no deseada.

Supongamos por absurdo que en la solución retornada, `emp`, existen m, m', f, f' que cumplen las siguientes condiciones:

1. m se enfrenta a f.
2. m' se enfrenta a f'.
3. $|rating(m) - rating(f')| < |rating(m) - rating(f)|$.
4. $|rating(m) - rating(f')| < |rating(m') - rating(f')|$.

La condición 3 implica que f' aparece antes que f en la lista de preferencias de m construida en los pasos 3-4 del algoritmo. Por lo tanto, en la instancia del problema de emparejamiento estable que se resuelve en el paso 10, m prefiere a f' antes que a f.

Análogamente, la condición 4 implica que m aparece antes que m' en la lista de preferencias de f' construida en los pasos 7-8. Por lo tanto, en la instancia del problema de emparejamiento estable, f' prefiere a m antes que a m'.

Por otra parte, las condiciones 1 y 2 implican que el algoritmo de Gale-Shapley ejecutado en el paso 10 define las parejas (m, f) y (m', f'). Por lo tanto, en el emparejamiento `emp` existe una inestabilidad (m, f'), ya que m prefiere a f' antes que a f y f' prefiere a m antes que a m'. Esto contradice la corrección del algoritmo de Gale-Shapley y concluye nuestra prueba por absurdo.

Ejercicio 3 (18 puntos)

Un carpintero hace pedidos para sus clientes, que luego deben pasar a retirarlos. Cada cliente i tiene un intervalo de tiempo $[s_i, f_i]$ en el que tiene disponibilidad para ir a retirar su pedido. El carpintero quiere definir un conjunto de horas para retiro de pedidos (instantes de tiempo), $\mathcal{T}$, de forma que todo cliente i pueda retirar su pedido dentro de su horario disponible, es decir, exista $t \in \mathcal{T}$ tal que $s_i \leq t \leq f_i$.

Ejemplo: Si tenemos tres clientes que pueden retirar sus pedidos de 8:00 a 9:00, de 9:00 a 12:00 y de 14:00 a 16:00, entonces el carpintero podría fijar dos horas para retiro de pedido: una a las 9:00 y otra a las 14:00 (o en cualquier otro momento entre las 14:00 y las 16:00 inclusive).

Cada vez que el carpintero entrega un pedido tiene que interrumpir su trabajo para atender a los clientes. Su objetivo es minimizar el número de veces que tiene que interrumpir su trabajo haciendo que $|\mathcal{T}|$ sea lo más chico posible.

(a) Diseñe un algoritmo eficiente que, dados el conjunto de intervalos de tiempo de disponibilidad $\mathcal{I} = \{[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]\}$ de n clientes, defina un conjunto de horas de entrega $\mathcal{T}$ con la menor cantidad de horas posible.

(b) Demuestre la corrección de su algoritmo.

Solución:

(a) El algoritmo de la figura 2 resuelve el problema.

```
1 T = {}
2 while I ≠ {} do
3     t = mín{f : [s, f] ∈ I}
4     agregar t a T
5     eliminar de I los intervalos [s, f] tales que s ≤ t ≤ f
6 end
7 return T
```

Figura 2: Algoritmo *greedy* para resolver el probelma

(b) En primer lugar notamos que nuestro algoritmo termina, porque en cada iteración se elimina al menos un intervalo de $\mathcal{I}$, y hay una cantidad finita de ellos. Más aún, solo se eliminan intervalos cubiertos por algún elemento de $\mathcal{T}$, de modo que cuando termina todo cliente i con disponibilidad en el intervalo $[s_i, f_i]$ tiene un tiempo de entrega $t \in \mathcal{T}$, tal que $s_i \leq t \leq f_i$. Solo resta probar que la cardinalidad de $\mathcal{T}$ es mínima.

Sea $\mathcal{T} = \{t_1, \ldots, t_k\}$ el conjunto devuelto por nuestro algoritmo para una determinada instancia del problema, y sea $\mathcal{O} = \{o_1, \ldots, o_m\}$ un conjunto solución para esa instancia. Ambos conjuntos están enumerados de menor a mayor: $t_i < t_{i+1}$, $1 \leq i < k$, y $o_j < o_{j+1}$, $1 \leq j < m$.

Notar que el algoritmo construye el conjunto $\mathcal{T} = \{t_1, \ldots, t_k\}$ incorporando los tiempos $t_1, \ldots, t_k$ en ese orden, ya que cada ejecución del paso 4 elige siempre el mínimo tiempo de finalización entre los intervalos que no han sido eliminados de $\mathcal{I}$ en iteraciones anteriores, y esta sucesión de mínimos no puede decrecer.

Mostramos a continuación que nuestro algoritmo está "por delante" de $\mathcal{O}$ en el siguiente sentido.

Proposición 1.
Para todo i, $1 \leq i \leq \text{mín}\{m, k\}$, se cumple que $o_i \leq t_i$.

Demostración.
Probaremos el enunciado por inducción en i, para $1 \leq i \leq \text{mín}\{m, k\}$.

Veamos que el enunciado se cumple para $i = 1$. Nuestro algoritmo comienza eligiendo como primer horario de entrega, t_1, el final más temprano entre todos los intervalos de disponibilidad; sea $[s_j, f_j]$ un intervalo con tal tiempo de finalización, es decir $f_j = t_1$. Como $\mathcal{O}$ es una solución, debemos tener $o_1 \leq f_j$, porque de lo contrario el cliente j no estaría disponible para retirar su entrega en el tiempo o_1 y tampoco en ninguno de los otros tiempos de entrega en $\mathcal{O}$, que son mayores que o_1. Esto muestra que $o_1 \leq t_1$.

Asumamos ahora que el enunciado se cumple para cierto valor de i, $1 \leq i < \text{mín}\{m, k\}$, y veamos que se cumple para $i+1$, es decir, $o_{i+1} \leq t_{i+1}$.

Cuando se elige t_{i+1} en el paso 3, existe un intervalo, $[s_j, f_j]$, con $f_j = t_{i+1}$, que no fue eliminado de $\mathcal{I}$ en iteraciones anteriores. Esto implica que se cumple $s_j > t_i$, porque en caso contrario tendríamos $s_j \leq t_i < t_{i+1} = f_j$, y $[s_j, f_j]$ hubiese sido eliminado a más tardar al agregar t_i a $\mathcal{T}$. A partir de este hecho, combinado con la hipótesis de inducción que establece que $t_i \geq o_i$, obtenemos $s_j > o_i$, de modo que $[s_j, f_j]$ no es cubierto por o_i ni ninguno de los tiempos de entrega anteriores en $\mathcal{O}$. Concluimos entonces que debemos tener $o_{i+1} \leq f_j$, porque de lo contrario $[s_j, f_j]$ tampoco sería cubierto por o_{i+1}, ni ninguno de los tiempos de entrega posteriores en $\mathcal{O}$. Recordando que $t_{i+1} = f_j$, esto muestra que $o_{i+1} \leq t_{i+1}$, lo cual concluye la demostración de la proposición.

□

Veremos que la proposición que acabamos de demostrar implica que se cumple $k \leq m$, de donde, como $\mathcal{O}$ es arbitraria, se concluye que nuestro algoritmo construye una solución de cardinalidad mínima. En efecto, supongamos por absurdo que $k > m$. Cuando nuestro algoritmo elige t_{m+1} en el paso 3, existe un intervalo, $[s_j, f_j]$, con $f_j = t_{m+1}$, que no fue eliminado de $\mathcal{I}$ en iteraciones anteriores. Siguiendo el mismo razonamiento que en el paso inductivo en la demostración de la proposición anterior, esto implica que se cumple $s_j > t_m$, porque de lo contrario $[s_j, f_j]$ hubiese sido eliminado a más tardar al agregar t_m a $\mathcal{T}$. Pero por la proposición anterior tenemos $o_m \leq t_m$, por lo cual se cumple $s_j > o_m$ y en consecuencia $[s_j, f_j]$ no es cubierto por $\mathcal{O}$. Llegamos a un absurdo, de donde concluimos que se debe cumplir $k \leq m$.

Versión basada en propiedad estructural Decimos que un conjunto de instantes de tiempo $\mathcal{T}$ es una *solución válida* para el conjunto de intervalos $\mathcal{I}$ si cada intervalo de $\mathcal{I}$ contiene algún instante de $\mathcal{T}$. Un conjunto de instantes de tiempo $\mathcal{T}$ es una *solución óptima* para el conjunto de intervalos $\mathcal{I}$ si es una solución válida de mínima cardinalidad.

Denotando con $\mathcal{V}$ al conjunto de soluciones válidas, tenemos

$$\mathcal{V} = \{\mathcal{T}, \text{conjunto de reales : Para cada } [s, f] \in \mathcal{I} \text{ existe } t \in \mathcal{T} \text{ tal que } s \leq t \leq f\},$$

y el conjunto de soluciones óptimas es $\text{argmin}_{\mathcal{T} \in \mathcal{V}}\{|\mathcal{T}|\}$.

Definimos un *conjunto disjunto de intervalos* como un conjunto de intervalos en el que cada par de intervalos es disjunto, no se solapan. Si S es un subconjunto disjunto de $\mathcal{I}$ cualquier solución válida de $\mathcal{I}$ debe tener cardinalidad mayor o igual a $|S|$ porque cada instante de $\mathcal{T}$ está contenido en a lo sumo un intervalo de S. Entonces, si encontramos un conjunto disjunto S de cardinalidad k y una solución válida $\mathcal{T}$ que también tiene cardinalidad k sabemos que $\mathcal{T}$ es una solución óptima.

Supongamos que ordenamos el conjunto disjunto S de manera creciente según sus tiempos de finalización. Queremos determinar qué condiciones se deben cumplir para que la cardinalidad de las soluciones óptimas sea $|S|$, y que pueda encontarse una de ellas mediante S.

Sabemos que una solución válida, $\mathcal{T}$, debe tener al menos un instante contenido en cada intervalo de S, y para que sea óptima debe tener exactamente uno. Sea t_j el instante contenido en el j-ésimo intervalo de S.

Se debe cumplir que todos los intervalos de $\mathcal{I}$ cuyo tiempo de inicio es menor o igual a t_j contengan uno de los instantes $t_1, \ldots, t_j$. Esto es porque si no contuvieran ninguno entonces su tiempo de finalización sería menor que t_j, y como los intervalos están ordenados de manera creciente y son disjuntos ningún t_h, con $h > j$, podría estar contenido en el intervalo, y como consecuencia $\mathcal{T}$ no sería un conjunto válido para $\mathcal{I}$.

Veamos que cada t_j no puede elegirse de manera arbitraria en el intervalo correspondiente. Supongamos que t_j no es el tiempo de finalización del intervalo en que está contenido. Siendo así, sin alterar la maximalidad de S, entre t_j y el tiempo de finalización del intervalo podría haber otro intervalo de $\mathcal{I}$ (no perteneciente a S) que no podría contener ningún instante de $\mathcal{T}$. Por lo tanto elegimos que cada instante de $\mathcal{T}$ coincida con el tiempo de finalización del intervalo en que está contenido.

Entonces todos los intervalos que empiecen después de t_j y no después de t_{j+1} deben contener a t_{j+1}, tiempo de finalización del $j+1$-ésimo intervalo S, por lo que ese intervalo debe ser el que termina primero entre los que no contenían ningún elemento de $t_1, \ldots, t_j$. Esto implica que si t_j es el último elemento de $\mathcal{T}$ ningún intervalo de $\mathcal{I}$ empieza después de t_j.

De estas consideraciones se obtiene el algoritmo. En cada paso se elige el intervalo que termina antes entre todos los que no incluyen ningún instante de $\mathcal{T}$, y se incluye en $\mathcal{T}$ el tiempo de finalización de ese intervalo.

De manera simétrica, y teniendo en cuenta que todos los intervalos cuyo tiempo de finalización es mayor o igual que t_j deben contener algún instante en $t_j, \ldots, t_{|S|}$, recorriendo los intervalos en orden decreciente de sus tiempos de inicio en cada paso se elige el intervalo que termina después entre todos los que no incluyen elementos de $\mathcal{T}$, y se incluye en $\mathcal{T}$ su tiempo de inicio.

(a) Con el algoritmo de la figura 3 se obtiene una solución óptima.

```
1 Ordenar I de manera creciente según el tiempo de finalización de sus intervalos /* En
     nuestra notación los renombramos en ese orden: f_1 ≤ f_2 ≤ ... ≤ f_n        */
2
3 Crear T con f_1 como único elemento
4 for i = 2 to n do
5     if s_i es mayor que el último elemento de T, t_|T|, then
6         Incluir f_i al final de T
7     end
8 end
9 return T
```

Figura 3: Conjunto óptimo de instantes

(b) Claramente el algoritmo termina porque el único ciclo que contiene itera exactamente $n-1$ veces.

Hay que probar que todo intervalo de $\mathcal{I}$ contiene algún instante de $\mathcal{T}$ para probar que $\mathcal{T}$ es una solución válida. Recordando que $\mathcal{I}$ está ordenado de manera creciente se cumple que en la línea 5 el tiempo de finalización del intervalo considerado es mayor que $t_{|\mathcal{T}|}$, ya que es el tiempo de finalización de un intervalo anterior. Si su tiempo de inicio es menor o igual a $t_{|\mathcal{T}|}$ entonces contiene a uno de los instantes de $\mathcal{T}$. En otro caso el intervalo contiene al nuevo instante que se agrega a $\mathcal{T}$.

Veamos ahora que la solución es óptima. Definimos S como el conjunto de todos los intervalos $[s_i, f_i]$ para los cuales f_i se agrega a $\mathcal{T}$ ejecutando el paso 6. Sabemos que S es disjunto por construcción, ya que, por la condición del paso 5, el tiempo de inicio de cada intervalo de S es mayor que el tiempo de finalización del anterior intervalo de S, y de manera trivial el primero de esos

intervalos no se solapa con ninguno anterior. Entonces no puede haber una solución válida con menos elementos que $\mathcal{T}$, por lo que $\mathcal{T}$ es una solución óptima.