Proyecto DAA

Daniel Abad Fundora
Anabel Benítez González
Enzo Rojas D'Toste

Septiembre 2024

1. Problema 1: El secreto de la Isla

1.1. Especificación del problema

En el corazón del vasto océano azul, existe una isla tropical conocida como La Isla de Coba, un lugar de exuberante vegetación y antigua sabiduría. La tribu que habita en esta isla ha vivido en armonía con la naturaleza durante siglos, guiados por un consejo de ancianos que custodian el equilibrio entre los habitantes de la isla y sus recursos.

Sin embargo, una nueva generación de líderes debe ser elegida, y para ello, los ancianos han planteado un desafío ancestral. Los aspirantes al consejo deben descubrir un grupo selecto de **guardianes**, quienes, colocados en puntos estratégicos de la isla, podrían vigilar a todos los aldeanos sin dejar a ninguno sin supervisión directa o indirecta.

La isla tiene una serie de aldeas unidas entre sí por caminos. El reto es encontrar un grupo de guardianes tan pequeño como sea posible, de modo que cada aldea esté bajo la protección directa de un guardián, o al menos esté conectada a una aldea donde se haya asignado un guardián.

Los jóvenes líderes deben resolver este desafío para mostrar su valía. Con cada nueva selección de guardianes, la estabilidad de la isla se estremece. El futuro de La Isla de Coba depende de que uno de los jóvenes encuentre esta distribución óptima de guardianes y se convierta en el próximo jefe absoluto.

1.2. Modelado del problema como un grafo

Este problema puede modelarse como un grafo no dirigido, donde cada aldea es un vértice y cada camino entre aldeas es una arista. Formalmente, la tarea consiste en buscar un conjunto mínimo de nodos S tal que $|N(S)| = n$, donde $N(S)$ es la unión de las vecindades de los nodos en S, este conjunto es conocido como **conjunto dominante mínimo** en el grafo.

1.3. Demostración de que el problema es NP-Completo

Primero, demostraremos que el problema pertenece a la clase NP:

Dado un grafo y una posible solución, es fácil verificar en tiempo lineal si el conjunto de nodos dado es un conjunto dominante, simplemente recorriendo todas las aristas de cada nodo y verificando si su vecindad cubre todos los vértices.

Para demostrar que el problema es NP-completo, podemos reducir el problema conocido *Minimum Vertex Cover* (que es NP-hard) a nuestro problema.

Sea G un grafo no dirigido. Construimos un nuevo grafo G' de la siguiente manera: por cada arista en G, creamos un nuevo nodo en G' y lo conectamos a los dos nodos adyacentes de esa arista en G. Esto forma un triángulo (*clique* de tamaño 3) por cada arista de G en el grafo G'.

Ahora, demostraremos que el conjunto dominante mínimo en G' tiene el mismo tamaño que el *Minimum Vertex Cover* en G. Consideremos dos casos:

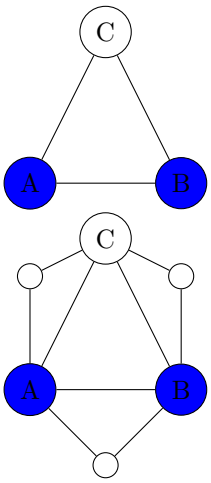
(1) Todos los nodos del conjunto dominante pertenecen a G. Como por cada arista G creamos un nodo en G', si todos esos nodos quedan cubiertos en G', entonces cada arista en G está cubierta por al menos un nodo del conjunto dominante.

(2) Al menos un nodo agregado está en el conjunto dominante. Dado que estos nodos tienen grado 2 en un *clique* de tamaño 3, su vecindad es un subconjunto de las de sus nodos adyacentes. Podemos reemplazarlos por uno de sus vecinos, obteniendo un conjunto dominante del mismo tamaño que en el caso anterior.

Nótese además que como todo nodo añadido representaba una arista del grafo original, todo vertex cover del grafo original constituye un conjunto dominante del grafo construido.

Por lo tanto, el problema es NP-completo.

1.3.1. Ejemplo ilustrativo



Nótese que $\{A, B\}$ es un Minimun Vertex Cover del grafo original y un Conjunto Dominante Mínimo del grafo expandido

1.4. Solución 1: Solución Fuerza Bruta

Una solución exacta para este problema consiste en comprobar cada uno de los subconjuntos posibles de nodos, buscando el de menor tamaño que cumpla ser un conjunto dominante. Esta solución posee una complejidad temporal de $O(2^n n^2)$.

1.5. Solución 2: Algoritmo Aproximado utilizando estrategia Greedy

Se propone el siguiente algoritmo: Consideremos tres conjuntos de nodos, blancos, negros y grises. Inicialmente todos los nodos del grafo están pintados de blanco. Luego, iterativamente seleccionamos el nodo que tiene más aristas hacia nodos blancos, una vez seleccionado, pintamos este nodo de negro y todos sus vecinos de gris. Este proceso se repite hasta que no queden nodos blancos. Al final del proceso, los nodos que quedan pintados de negro forman un conjunto dominante.

Presentamos un pseudocódigo para este algoritmo, donde w(v) se define como la cantidad de vecinos blancos que tiene v.

```
1 S = {};
2 while exist white nodes
3     choose v where the w(v) is the maximum among all nodes
4     S = S + {v};
```

1.5.1. Analizando la precisión del algoritmo

Establezcamos una cota superior para nuestra aproximación. Sea S' una solución óptima para nuestro problema y sea S una solución hallada por nuestro algoritmo, acotaremos $|S|$ en función de $|S'|$. En primer lugar observemos que como S es un conjunto dominante del grafo, también es un conjunto dominante de cualquier subconjunto del mismo, luego para cualquier subconjunto del grafo de tamaño D, existirá un nodo con al menos $|D|/|S|$ de estos elementos en su vecindad. Sea D_i la cantidad de nodos restantes en cada iteración del algoritmo, por el razonamiento anterior tenemos que:

- $|D_0| = n$
- $|D_1| \leq n - \frac{n}{|S|}$
- $|D_2| \leq n(1 - \frac{1}{|S|})^2$

  .....
- $|D_i| \leq n(1 - \frac{1}{|S|})^i$

Este algoritmo finaliza cuando $D_i = \{\}$ o sea $|D_i| < 1$. Aplicando la desigualdad $1 + x \leq e^x \ \forall x \in R$ para $x = \frac{1}{|S|}$ tenemos que $D_i \leq n * e^{-\frac{1}{|S|}i}$.

Luego queremos hallar i tal que $n * e^{-\frac{i}{|S|}} < 1$. Multiplicando ambos miembros por $e^{\frac{i}{|S|}}$ tenemos que $n < e^{\frac{i}{|S|}}$ y aplicando ln a ambos miembros llegamos a $\ln n < \frac{i}{|S|}$, despejando i obtenemos $i > |S| \ln n$, por lo que el algoritmo termina en a lo sumo $|S| \ln n + 1$ iteraciones, como el algoritmo añade un nodo en cada iteración, esto es cota superior para el conjunto encontrado.

1.6. Hallando una cota superior más ajustada para la solución devuelta por este algoritmo

En la sección anterior demostramos que este algoritmo garantizaba una aproximación a lo sumo $\ln n + 1$ veces la solución óptima. En esta sección demostraremos algo más fuerte: La solución computada por este algoritmo es a lo sumo $\ln(\Delta + 2)$ veces la solución óptima (donde Δ es el grado máximo del grafo).

Para demostrar esto, procedamos de la siguiente manera:

Asignaremos a cada uno de los nodos un costo, de manera que el costo total sea la cantidad de elementos en nuestro conjunto dominante. La primera idea es, si $S*$ es una solución, asignarle costo 1 a cada uno de los nodos de S, sin embargo, no será lo que haremos, procedamos de otra forma equivalente, cada vez que seleccionamos un nodo, distribuimos su unidad de costo entre los nodos que entran a la vecindad de nuestro conjunto en ese momento, a este conjunto de nodos lo denominamos $d(n)$.

Sea n un nodo que pertenece a la solución óptima, por definición sabemos que $d(n) = 1$. Hallemos una cota superior para el costo de $d(n)$ en el grafo original. Nótese que por como está definido el algoritmo, el la primera vez que un nodo de $d(n)$ entra a la vecindad del conjunto dominante calculado por nuestro algoritmo, adquiere un costo menor o igual a $\frac{1}{d(n)}$, como $d(n)$ siempre será menor o igual al grado del nodo + 1, tenemos que este costo es menor o igual que $\frac{1}{g(n)+1}$. De igual forma, el segundo nodo del conjunto tendrá un costo de a lo sumo $\frac{1}{g(n)}$ y así sucesivamente, estamos en presencia de una suma armónica.

$\frac{1}{g(n)+1} + \frac{1}{g(n)} + \frac{1}{g(n)-1} + \frac{1}{g(n)-2} + ... + \frac{1}{2} + 1$

Que como sabemos es menor que $\ln(g(n) + 2)$, por lo que por cada unidad de costo de la solución óptima, se tienen a lo sumo $\ln(\Delta(G) + 2)$ unidades de costo en la solución encontrada.

1.6.1. Análisis de la complejidad temporal

```
greedyDominatingSet(graph):
    n = length(graph)
    nodes = [0, 1, ..., n-1]
    dominatingSet = empty set
    dominatedMask = array of n False
    uncoveredNeighbors = dictionary with node keys and
    ↪  (length(graph[node]) + 1) values

    while any uncoveredNeighbors[node] > 0 for node in graph:
        node = max(nodes, key = uncoveredNeighbors[node])
        add node to dominatingSet
        uncoveredNeighbors[node] -= 1
        dominatedMask[node] = True

        for neighbor in graph[node]:
```

```
15              if not dominatedMask[neighbor]:
16                  for x in graph[neighbor]:
17                      uncoveredNeighbors[x] -= 1
18                  dominatedMask[neighbor] = True
19                  uncoveredNeighbors[neighbor] -= 1
20              uncoveredNeighbors[neighbor] -= 1
21
22      return length(dominatingSet)
```

Notemos que este algoritmo se detiene cuando todos los valores de uncoveredNeighbors sean 0, por tanto la línea 17 se ejecuta a lo sumo 2n veces, teniendo el doble for una costo amortizado de $O(n)$. El while tendría a lo sumo n interaciones, en el caso en que dominara a solamente un nodo por iteración, en cada iteración se realizan $O(n)$ operaciones por tanto la implementación es $O(n^2)$.

2. Problema 2: Distancia de árboles

2.1. Especificación del problema

Un árbol se define como un grafo no dirigido y conectado con n vértices y $n-1$ aristas. La distancia entre dos vértices en un árbol es igual al número de aristas en el camino simple entre ellos.

Se te dan dos enteros x y y. Se debe construir un árbol con las siguientes propiedades:

- El número de pares de vértices con una distancia par entre ellos es igual a x.
- El número de pares de vértices con una distancia impar entre ellos es igual a y.

Por par de vértices, nos referimos a un par ordenado de dos vértices (posiblemente el mismo o diferente).

2.2. Solución propuesta

Comencemos recordando los siguientes hechos:

1. Los árboles son grafos bipartitos ya que no tienen ciclos y, por lo tanto, no tienen ciclos impares.
2. En un bipartito conexo, todo par de nodos de biparticiones distintas está a distancia impar, mientras que todo par de nodos de la misma bipartición está a distancia par.
3. Dado un conjunto A, la cantidad de pares que se pueden formar con los elementos de este conjunto es $|A|^2$.
4. Dados dos conjuntos A y B la cantidad de pares ordenados que se pueden formar que contengan un elemento de cada conjunto es $2|A||B|$.

Teniendo esto en cuenta, procedamos a realizar un algoritmo constructivo para resolver nuestro problema.

Supongamos que existe algún árbol que cumple los requerimientos del problema. Conocemos que este árbol es un grafo bipartito, denotemos a las biparticiones A y B.

Sea $a = |A|$ y $b = |B|$. Conocemos que $x = a^2 + b^2$ y $y = 2ab$, luego $x + y = (a+b)^2$. Nótese que $x + y$ son todos los posibles pares de vértices del árbol, o sea $x + y = n^2$, luego $n = \sqrt{x+y} = a + b$.

Tenemos que $x-y = a^2-2ab+b^2 = (a-b)^2$, por lo que $a-b = \sqrt{x-y}$. Despejando las ecuaciones anteriores obtenemos $2a = \sqrt{x+y}+\sqrt{x-y}$ y $b = n-a$.

Una vez que hemos determinado las cardinalidades de las biparticiones, construimos un árbol a partir de ellas. Por ejemplo, podemos crear un camino alternando entre las biparticiones. Al llegar al último nodo de una bipartición, lo conectamos con todos los nodos restantes de la otra bipartición, es decir, aquellos que aún no han sido conectados.

Nótese que, dado que $x+y = n^2$ y $x-y = (a-b)^2$, si $x+y$ y $x-y$ no son cuadrados perfectos, no existe solución válida.

2.2.1. Analizando su complejidad temporal

```
solve(x, y)
    if not (x >= y >= 0) or not isSqrt(x + y) or not isSqrt(x -
    ↪  y)
        return "No solution exists"

    n = sqrt(x + y)
    s = sqrt(x - y)

    if (n % 2) != (s % 2)
        return "No solution exists"

    a = (n + s) // 2
    b = (n - s) // 2

    if a < 0 or b < 0 or n <= 0
        return "No solution exists"

    if n == 1
        return 1, []

    // If n > 1 at least must be one vertex in A and B
    if a == 0 or b == 0
        return "No solution exists"

    edges = []
    current_vertex = 1

    vertex_b = current_vertex
    current_vertex += 1

    if a > 1
        for i from current_vertex to current_vertex + a - 1
```

```
32              edges.push((i, vertex_b))
33
34          current_vertex += a - 1
35
36      vertex_a = current_vertex
37      current_vertex += 1
38
39      if b > 1
40          for i from current_vertex to current_vertex + b - 1
41              edges.push((i, vertex_a))
42
43      edges.push((vertex_a, vertex_b))
44
45      return n, edges
46
```

Nótese que para calcular la cantidad de nodos necesaria del grafo y las cardinalidades de las biparticiones, solo se realiza un número constante de operaciones aritméticas. Luego de este cómputo la construcción del árbol se realiza mediante un único recorrido a estos nodos, por lo que tiene complejidad temporal $O(n)$. Por tanto nuestro algoritmo tiene complejidad temporal $O(n) + O(1) = O(n)$. Como el ejercicio requiere devolver una estructura de n elementos y utilizamos un algoritmo lineal para resolverlo, podemos garantizar que no existe algoritmo asintóticamente mejor que este.

3. Problema 3: Torneo

3.1. Especificación del problema

Tan pronto como comienza el torneo de programación competitiva, tu experimentado equipo de 3 personas se da cuenta inmediatamente de que este presenta a problemas fáciles, b problemas medianos, y c problemas difíciles. Resolver un problema les tomará a cualquiera de ustedes 2, 3 o 4 unidades de tiempo, dependiendo de si el problema es fácil, mediano o difícil. Independientemente de la dificultad del problema, la última unidad de tiempo gastada en resolverlo debe gastarse utilizando la computadora compartida.

Tu equipo organiza sus esfuerzos de manera que cada uno de ustedes comienza (y termina) resolviendo problemas en unidades de tiempo enteras. Cualquier problema dado solo puede ser resuelto por un concursante; requiere una cantidad continua de tiempo (que depende de la dificultad del problema). Ninguno de los 3 puede resolver más de un problema a la vez, pero pueden comenzar a resolver un nuevo problema inmediatamente después de terminar uno. De manera similar, la computadora compartida no puede ser utilizada por más de uno de ustedes al mismo tiempo, pero cualquiera de ustedes puede comenzar a usar la computadora (para completar el problema que se está resolviendo actualmente) inmediatamente después de que alguien más deje de usarla.

Dado que el concurso dura l unidades de tiempo, encuentra el número máximo de problemas que tu equipo puede resolver. Además, encuentra una forma de resolver el número máximo de problemas.

3.2. Analizando restricciones

1. a problemas fáciles, b problemas medianos, y c problemas difíciles: La cantidad de problemas resueltos debe ser menor o igual a $a+b+c$, respetando las restricciones para cada tipo de problema.

2. Todos los problemas deben resolverse en 2, 3 o 4 unidades de tiempo continuas.

3. La última unidad de tiempo dedicada a resolver cada problema debe realizarse en la computadora compartida. La computadora compartida no puede ser utilizada por más de uno de ustedes al mismo tiempo: No puede haber dos problemas que terminen en la misma unidad de tiempo.

4. Tu equipo organiza sus esfuerzos de manera que cada uno de ustedes comienza y termina resolviendo problemas en unidades de tiempo enteras.

5. Un problema solo puede ser resuelto por un concursante.

6. Ninguno de los tres puede resolver más de un problema a la vez.

7. Todos los problemas deben ser completados en un torneo de l unidades de tiempo

3.3. Solución 1: Fuerza Bruta

El algoritmo consiste en un **backtracking** que explora todas las posibles combinaciones de problemas que pueden ser resueltos, quedándose con la que mayor cantidad de problemas contenga.

La idea es simular todos los torneos posibles de la siguiente manera: cuando un participante termina el problema que tenía asignado, se intenta asignarle cada categoría de problema disponible en cada momento posible a partir del tiempo en que terminó el problema anterior. Este proceso se realiza de forma recursiva para probar todas las combinaciones posibles. Antes de asignar un nuevo problema, se verifica que se cumplan todas las restricciones enumeradas anteriormente, ya que de lo contrario se saldría del conjunto de soluciones factibles.

A continuación un pseudocódigo de este algoritmo donde al final quedará la mejor asignación de problemas en la lista "maxSolvedProblems", esta lista guarda por cada problema el participante que lo resuelve, el nivel de dificultad del problema y el instante de tiempo en que terminó de resolverse.

```
1  getOptimalDistribution(time, remainingProblems,
   ↪  participantsEndTime, solvedProblems)
2      if length(solvedProblems) > length(maxSolvedProblems)
3          maxSolvedProblems = solvedProblems
4
5      (nextAvailableParticipant, nextAvailableTime) =
       ↪  min(participantsEndTime);
6
7       for startTime from nextAvailableTime to time
8          for each difficulty in [2, 3, 4]
9              if remainingProblems[difficulty] is 0
10                 continue
11
12             endTime = difficulty + startTime
13
14             if endTime > time or contains(participantsEndTime,
               ↪  endTime)
15                 continue
16
17             participantsEndTime[nextAvailableParticipant] =
               ↪  endTime
18             remainingProblems[difficulty]--
19             solvedProblems.push((difficulty,
               ↪  nextAvailableParticipant, endTime))
20
21             GetBestDistribution(time, remainingProblems,
               ↪  participantsEndTime, solvedProblems)
22
23             participantsEndTime[nextAvailableParticipant] =
               ↪  nextAvailableTime
```

```
            remainingProblems[difficulty]++
            solvedProblems.pop()
```

Este algoritmo es completo porque explora todas las posibles combinaciones de problemas que pueden ser resueltos dentro del tiempo límite, asegurando que se encontrará la solución óptima. Sin embargo, su complejidad es exponencial debido a la gran cantidad de combinaciones.

3.4. Solución 2: Estrategia Greedy

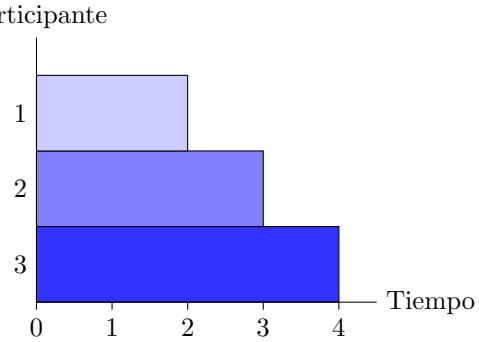
Ahora vamos a simular el torneo de la siguiente manera: primero, identificamos al participante del equipo que termina su problema asignado más pronto. A este participante le asignamos un nuevo problema siguiendo esta estrategia: intentamos que el tiempo de inicio de su próxima tarea sea lo más cercano posible al tiempo de finalización de su tarea anterior, asignándole la tarea más pequeña que quepa, de esta manera dejamos huecos lo más pequeño posibles en cada decisión. Si no es posible, incrementamos en 1 el tiempo ideal de inicio de la próxima tarea hasta que encontremos una asignación válida. Este proceso de asignación de problemas se repite hasta que no se encuentre ninguna asignación válida ya sea porque se agotó el tiempo, se acabaron los problemas o por cualquiera de las otras restricciones.

A continuación un pseudocódigo de este algoritmo donde al final quedará la mejor asignación de problemas en la lista "maxSolvedProblems", esta lista guarda por cada problema el participante que lo resuelve, el nivel de dificultad del problema y el instante de tiempo en que terminó de resolverse.

```
getOptimalDistribution(time, remainingProblems)
    participantsEndTime = [0, 0, 0]
    do
        nextAvailableParticipant =
        ↪ indexOfMin(participantEndTimes)
    while
        assignProblem(nextAvailableParticipant, time,
        ↪ remainingProblems, participantsEndTime)

assignProblem(participant, time, remainingProblems,
↪ participantsEndTime)
    participantEndTime = participantsEndTime[participant]

    for startTime from participantEndTime to time
        for each difficulty in [2, 3, 4]
            if remainingProblems[difficulty] is 0
                continue

            endTime = difficulty + startTime

```

```
            if endTime > time or contains(participantsEndTime,
            ↪  endTime)
                continue

            maxSolvedProblems.push((difficulty, participant,
            ↪  endTime))
            participantsEndTime[participant] = endTime
            remainingProblems[difficulty] -= 1
            return true

    return false
```

Veamos un ejemplo de la decisión tomada por nuestro algoritmo dado el torneo con l=7, a=5, b=5, c=5, en un instante de tiempo $i = 2$:



Dado este estado en el instante de tiempo $i = 2$ cuando termina el primer participante tiene varias opciones:

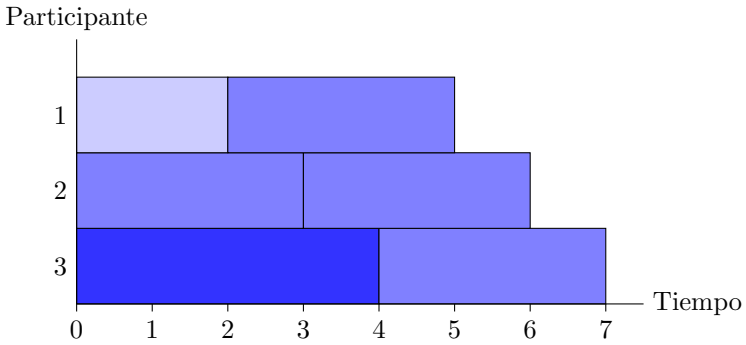
1. Empezar un problema medio en el instante 3 (inmediatamente después que termina el anterior) que terminaría en el instante 5 (primer instante de finalización disponible).

2. Empezar un problema fácil en el instante 4 que terminaría en el instante 5 (primer instante de finalización disponible). Veamos que en este caso se gastaría un bloque fácil innecesariamente, pues hace la función de uno medio.

3. Empezar un problema difícil en el instante 3 que terminaría en el instante 6, dejando un instante de finalización sin utilizar (por el momento).

4. Demás asignaciones de pares (tiempo, dificultad) válidas que son menos interesantes.

Nuestro algoritmo elegirá la primera opción, iniciar el problema que dura 3 unidades de tiempo en el instante 3. Notemos que si corremos nuestro algoritmo hasta el final sobre el problema, devolvería la siguiente asignación:



Podemos observar que esta asignación es óptima, ya que en cada uno de los instantes 2, 3, 4, 5, 6 y 7, que son todos los disponibles bajo las restricciones, se completa un problema.

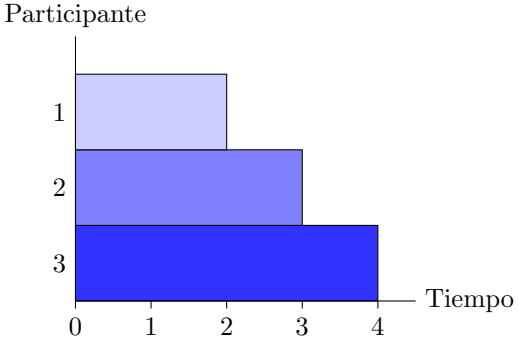
Esta estrategia es intuitivamente buena porque siempre busca minimizar el tiempo para un participante entre el final de una tarea y el inicio de la siguiente. Además, utiliza problemas de mayor dificultad primero para reservar los más sencillos, buscando aprovechar todos los instantes de finalización disponibles.

3.4.1. Analizando la correctitud del algoritmo

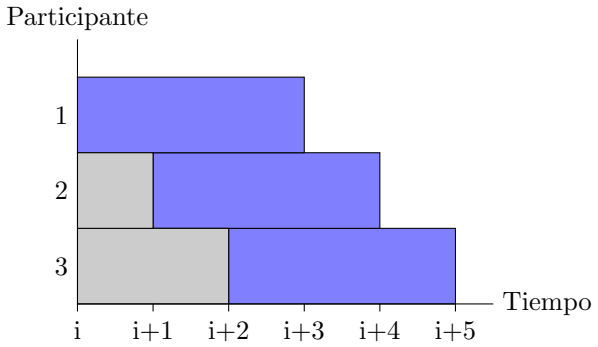
En problemas de este estilo donde se debe hallar la mayor cantidad de elementos que se pueden seleccionar tal que se cumplan ciertas restricciones, se debe demostrar que la solución hallada por el algoritmo es factible y óptima.

Es trivial que nuestro algoritmo brinda una solución factible, ya que en cada iteración de nuestro algoritmo, verificamos que se preserve la factibilidad, sin embargo no es trivial que retorna una solución óptima, a continuación procederemos a demostrarlo.

Iniciemos analizando las soluciones generadas por nuestro algoritmo, supongamos el torneo donde $l = t$, $a = \infty$, $b = \infty$, $c = \infty$. Notemos que en este caso nuestro algoritmo iniciará con un bloque de esta forma:

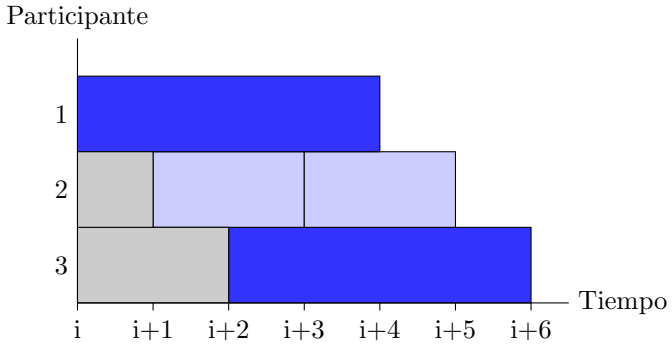


Luego $\forall i \equiv 2$ mód (3) llenará el espacio con bloques de esta forma:



Podemos observar que utilizando 1 problema fácil, 1 problema difícil y $l-3$ problemas medios, podemos lograr que en todo momento $x \geq 2$ termine un problema, maximizando de esta manera el número de problemas resueltos.

Ahora, veamos qué sucede en el torneo donde $l = t$, $a = \infty$, $b = 1$ y $c = \infty$. El primer bloque sería el mismo. Luego, $\forall i \equiv 2$ mód (4) llenaríamos el espacio de la siguiente forma:

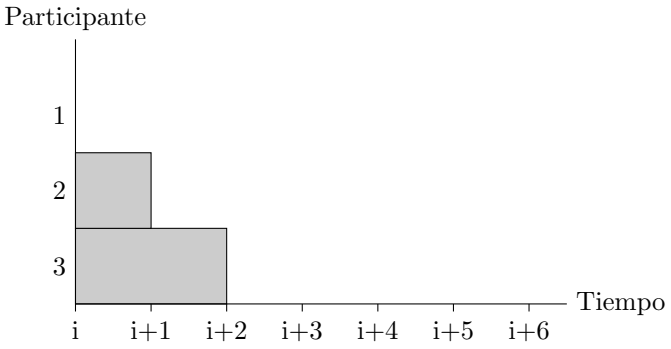


Observemos que de esta manera, utilizando $(l-2)/2$ problemas fáciles, $(l-2)/2$ problemas difíciles y 1 problema medio, también logramos que en todo momento $x \geq 2$ termine un problema, maximizando de esta manera el número de problemas resueltos.

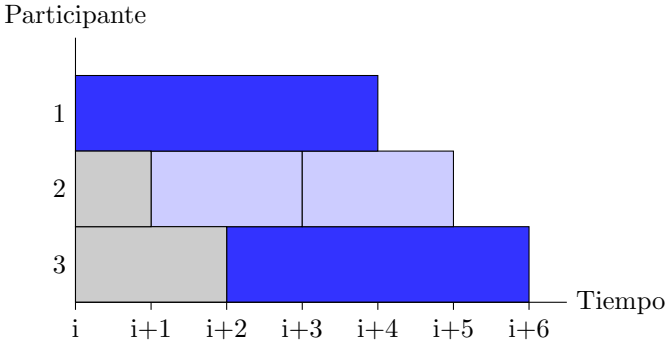
Notemos que siempre podemos sustituir un problema medio por uno fácil, ya que el participante solo necesita esperar un instante más para comenzar el problema y lo termina en el mismo momento.

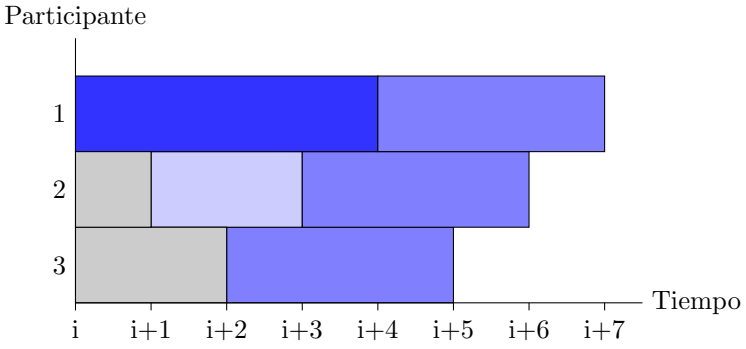
Minimización de instantes en que no termina un problema

Dado el siguiente estado, tocaría asignar un problema al participante 1.

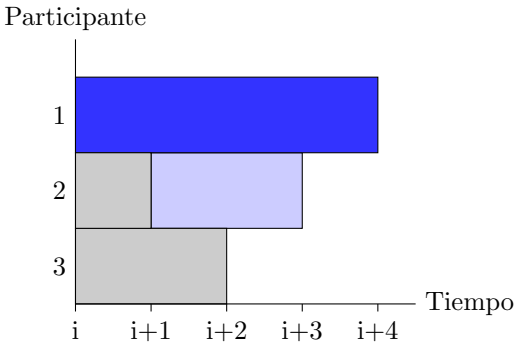


Analicemos qué ocurre al asignarle un problema difícil. Las maneras en que podemos llenar los espacios para aprovechar todos los instantes de finalización son las siguientes:





Podemos notar que ambas tienen en común la siguiente asignación.



Teorema 1. *Si se introduce un problema de 4, es necesario introducir uno de 2, de lo contrario se desperdicia un instante de finalización.*

Demostración: Supongamos lo contrario, fue introducido un problema difícil y no se desperdició un instante de finalización en sus 3 primeros instantes, luego en esos 3 instantes hay 3 finalizaciones de problema producidas por los otros dos competidores. Por principio del Palomar hay un jugador que finalizó dos problemas en este intervalo, como no hay problemas de tamaño menor que 2, tuvo que terminar un problema en el primer instante y luego poner un problema de tamaño 2.

Dado el teorema vemos que los problemas de 2 evitan que los problemas de 4 desperdicien un instante de finalización. Ahora, consideremos los bloques de tamaño 3, estos no desperdician instantes de finalización y tampoco evitan que los bloques de tamaño 4 desperdicien instantes de finalización. Esto significa que la acción de nuestro algoritmo de introducir todos los bloques de tamaño 3 no afecta el consumo de instantes de finalización.

Finalmente, solo gastamos bloques de tamaño 2 para acompañar a los bloques de tamaño 4. No tiene sentido gastar bloques de tamaño 2 por separado, ya que los bloques de tamaño 3 cumplen la misma función de no consumir instantes de finalización y no evitan que los bloques de tamaño 4 consuman. Dado que solo usamos bloques de tamaño 2 con los bloques de tamaño 4, tenemos que minimizamos la cantidad de instantes de finalización desperdiciados, lo que es igual a maximizar la cantidad de problemas.

3.4.2. Analizando la complejidad temporal

Este algoritmo realiza $O(1)$ operaciones en cada intento de asignación en un instante i de tiempo, como siempre $i <= l$, siendo l el tiempo que dura el torneo, luego nuestro algoritmo tiene una complejidad $O(l)$.