Parcial de Programación 3
24 de setiembre 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 (13 puntos)

(a) Defina el problema *emparejamiento estable* (*stable matching*)

(b) Sea $G = (V, E)$ un grafo no dirigido y sea T el árbol resultado de una recorrida DFS de G.
Sean x e y vértices de T, y sea (x, y) una arista de G que no es una arista de T.
Asuma que se cumpla la siguiente propiedad:
Para una llamada recursiva dada $DFS(u)$, todos los vértices que quedan marcados como "Explorados" entre la invocación y el final de esta llamada recursiva son descendientes de u en T.
Demuestre que o bien x es ancestro de y o bien y es ancestro de x.

Solución:

(a)

De esta respuesta se espera que:

- Se definan los conjuntos de entrada (de mismo cardinal) y sus listas de preferencia.
- Se defina emparejamiento y emparejamiento perfecto.
- Se defina el problema a resolver como encontrar un emparejamiento perfecto que no tiene inestabilidades.
- Se defina el concepto de inestabilidad.

La definición del problema se puede encontrar en la Sección 1.1 de K-T.

Consideramos un conjunto $M = \{m_1, m_2, \ldots, m_n\}$ y un conjunto $W = \{w_1, w_2, \ldots, w_n\}$. Un *emparejamiento* S es un subconjunto del producto cartesiano $M \times W$, con la propiedad de que cada miembro de M y cada miembro de W aparece en como máximo un par en S. Un emparejamiento S es *perfecto* si cada miembro de M y cada miembro de W aparece en exactamente un par de S.

Cada $m \in M$ ordena a todos los elementos de W, y decimos que m *prefiere* a w sobre w' si m ordena a w en una posición más alta que a w' en su lista de preferencias. Análogamente, cada $w \in W$ ordena a todos los elementos de M.

El problema del *emparejamiento estable* se define como encontrar un emparejamiento perfecto entre M y W que no tenga *inestabilidades*. Una *inestabilidad* se define como un par (m, w') tal que (m, w) y (m', w') se encuentran enmparejados en S, pero m prefiere a w' antes que a w, y w' prefiere a m antes que a m'.

(b)

De esta respuesta se espera una demostración rigurosa, como por ejemplo la demostración (3.7) del libro de K-T o la que que se presenta en esta solución. Una demostración, para que sea tal, debe mostrar con pasos elementales cómo la tesis se desprende de las hipótesis y la *definición* del algoritmo; no debe elaborarse a partir de presupuestos sobre el funcionamiento general del algoritmo como, por ejemplo, que la existencia de un camino entre x, y garantiza que antes de que se termina la ejecución de $DFS(x)$ se visitará a y.

La figura 1 muestra una implementación recursiva de DFS como referencia para la demostración.

Para todo vértice z que pertenece a T se cumple que debe haberse ejecutado $DFS(z)$ en algún momento, ya que después de agregar un vértice z a T, lo cual solo puede ocurrir en el paso 3 o 9, se ejecuta $DFS(z)$ en el paso inmediatamente siguiente. En consecuencia, a lo largo de la recorrida DFS debe haberse ejecutado $DFS(x)$ y $DFS(y)$ en algún momento.

Supongamos sin pérdida de generalidad que $DFS(x)$ se invoca antes que $DFS(y)$. Esto implica que y no está marcado como explorado al inicio de la ejecución de $DFS(x)$, ya que en ese momento aún no se ha ejecutado el paso 6 para el vértice y. Afirmamos que antes de que termine la ejecución de $DFS(x)$ el vértice y es marcado como explorado. En efecto, consideremos la iteración del ciclo del paso 7 en el cual se examina la arista (x, y), con x en el rol de u e y en el rol de v. Durante esa iteración, o bien la condición del paso 8 es falsa, en cuyo caso la afirmación es evidentemente cierta, o bien se ejecuta $DFS(y)$ en el paso 10, en cuyo caso se marca a y como explorado en el paso 6. En consecuencia, por la propiedad enunciada en la letra, concluimos que y es descendiente de x en T.

```
1  Marcar v como "NO explorado" para todo v ∈ V
2  Sea u ∈ V arbitrario
3  Crear un árbol T con u como raíz y sin aristas
4  Invocar a DFS(u)
5  Algoritmo DFS (u)
6      Marcar u como "explorado"
7      para cada (u, v) ∈ E incidente a u
8          si v no está marcado como "explorado" entonces
9              Agregar a v como hijo de u en T
10             Invocar recursivamente a DFS(v)
```

Figura 1: Algoritmo DFS recursivo.

Ejercicio 2 (13 puntos)

Una empresa de electrodomésticos ha desarrollado una aspiradora inteligente marca INCUMBA que se desplaza de manera autónoma para limpiar el hogar. El ambiente del hogar puede verse como una cuadrícula que cuenta con celdas libres, por donde la aspiradora puede transitar, y celdas ocupadas, que representan obstáculos como muebles. Esta información se tiene disponible como una matriz con n entradas que representan el estado de cada celda ($\{libre, ocupado\}$).

Las celdas son cuadradas de lado 1; la aspiradora INCUMBA se posiciona en el centro de la celda de la cuadrícula y se mueve de una celda a otra, pudiendo elegir hacerlo hacia cualquier celda vecina libre. Al moverse entre una celda y su vecina en diagonal, la aspiradora recorre una distancia de $\sqrt{2}$.

Asuma que existe un camino transitable entre cualquier par de celdas libres, y que la INCUMBA necesita volver a su dock de carga por el camino más corto posible para ahorrar la mayor cantidad de batería.

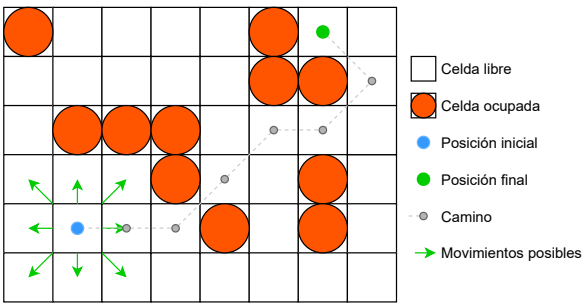


Figura: Cuadrícula de ocupación del entorno del hogar de tamaño $n = 6 \times 8$.

(a) Escriba un algoritmo que permita que la aspiradora INCUMBA encuentre uno de los caminos transitables más cortos desde su posición actual hasta su dock de carga.

El algoritmo recibe una matriz como la mencionada anteriormente, las dimensiones de la matriz (filas, columnas), la posición inicial de la aspiradora (fila, columna) y la posición del dock de carga (fila, columna). Reescriba cualquier algoritmo que utilice de los presentados en el curso, adaptándolo para que sea aplicable a la INCUMBA.

Puede asumir dada una función `V(f,c)` que retorna en tiempo $O(1)$ la lista de celdas vecinas de la celda `(f,c)` , $1 \leq f \leq$ filas, $1 \leq c \leq$ columnas. Por ejemplo `V(1,1)` retorna la lista con tres celdas `[(1,2),(2,1),(2,2)]`.

(b) Demuestre que su algoritmo admite una implementación con tiempo de ejecución $O(n \log n)$, siendo n la cantidad de entradas de la matriz. Enuncie los resultados teóricos del libro de referencia que utilice.

Solución:

El problema se modela con un grafo dirigido $G = (V, E)$ con un vértice por cada celda libre y una arista por cada par de celdas libres adyacentes. El costo de la arista es $\sqrt{2}$ o 1 según las celdas estén en diagonal o no. En el grafo se aplica el algoritmo de Dijkstra para encontrar el camino más corto desde el vértice que corresponde a la celda inicial hacia cada uno de los otros vértices. Para cada vértice se mantiene la distancia del camino conocido más corto (cualqueira de ellos si hay más de uno) desde el vértice de inicio, y el vértice inmediatamente anterior en ese camino, el 'padre'. Al terminar el algortimo de Dijkstra se construye el camino recorriendo la secuencia de padres desde el vértice fin hasta el inicio.

(a)

```
1  Algoritmo CaminoMasCorto (M,filas,columnas,inicio,fin)
2      Sea G el grafo con que se modela el problema
3      para cada (f,c) en ({1...filas},{1...columnas}) tal que M[(f,c)] es libre
4          agregar a G un vértice etiquetado (f,c)
5      para cada vértice v etiquetado (f,c) de G
6          para cada (f',c') en la lista de vecinos V[(f,c)] tal que M[(f',c')] es libre
7              Sea w el vértice de G etiquetado (f',c')
8              agregar w a la lista de adyacentes de v con costo l_vw igual a √2 o 1 según las celdas
                estén en diagonal o no
9      d[inicio] ← 0, d[v] ← ∞ para cada v ≠ inicio
10     S ← {inicio}
11     mientras S ≠ V
12         (u,w) ← argmin_{(u',w')∈E,u'∈S,w'∈V\S} {d[u'] + l_u'w'}
13         S ← S ∪ {w}
14         d[w] ← d[u] + l_uv
15         padre[w] ← u
16     p ← fin
17     camino ← (p)
18     mientras p ≠ inicio
19         p ← padre[p]
20         camino ← p . camino
21     Devolver camimo
```

Figura 2: Algoritmo para encontrar el camino de largo mínimo.

(b) Se mantiene las estructuras de listas de adyacencia, distancias y padres mediante arreglos bidimensionales por lo que se puede acceder en $O(1)$ a la información de cada vértice.

El ciclo de la línea 3 es $O(n)$ creando las listas de adyacentes vacías.

El ciclo de la línea 5 es $O(m)$. Cada vértice tiene a lo sumo 8 adyacentes, por lo que su grado es $O(1)$. Entonces el costo del ciclo es $O(n)$.

La línea 9 es $O(n)$.

El ciclo de la línea 11 es el algoritmo de Dijkstra visto en el libro de referencia, del que se conoce que es $O(m \log n)$, al que se le agregó la asignación del padre de w. Esta asignación tiene costo $O(1)$ por iteración y como hay $O(n)$ iteraciones el costo que se agregó es $O(n)$. Entonces el costo es $O(m \log n + n)$, que como $m = O(n)$ es $O(n \log n)$.

Finalmente para la construcción del camino se accede en cada iteración del ciclo de la línea 18 una vez a *padre* y se inserta en *Camino* lo cual tiene costo $\Theta(1)$. Este ciclo tiene $O(n)$ iteraciones por lo que su costo total también es $O(n)$.

El resto de los pasos tiene costo $O(1)$.

Entonces, como hay una cantidad de pasos independiente del tamaño de la entrada, el costo es el del paso de mayor costo, $O(n \log n)$.

Solución alternativa

(a) Se crea y mantiene una cola de prioridad, Candidatos. Sus elementos son triplas (v, d, p), donde v es una celda, d es la distancia del camino más corto conocido desde la posición inicial hasta v, y p es la celda inmediatamente anterior a v en ese camino. Si p está indefinido se representa con $\perp$. Las operaciones de la cola de prioridad son:

- **Crear** que la crea sin elementos,
- **Agregar** que inserta una tripla,
- **Minimo** que devuelve una tripla cuya distancia tenga valor mínimo,
- **EliminarMinimo** que remueve la tripla devuelta por **Minimo**,

- **Actualizar** que, dada una tripla (v, d', p'), si v está en la cola de prioridad en una tripla (v, d, p) y se cumple $d' < d$, entonces (v, d', p') sustituye a (v, d, p).

Se crea y mantiene un conjunto S con las celdas para los que se encontró el camino más corto desde la posición inicial. Sus elementos son pares (v, p), donde, tal como en la cola de prioridad, p es la celda inmediatamente anterior a v en ese camino. Esta celda p también puede considerarse el *padre* de v en un árbol obtenido por el algoritmo. El conjunto tiene operaciones para crear, incluir, determinar si una celda está en él (o sea, si la celda es el primer componente de algún par del conjunto), y obtener el padre de una celda v que está en el conjunto (o sea, el segundo componente del par (v, p)).

El resultado que se devuelve es **Camino**, que es una lista de celdas. Se inserta al inicio de la lista.

```
1  Algoritmo CaminoMasCorto (M,filas,columnas,inicio,fin)
2      Crear S
3      Crear Candidatos
4      para cada v tal que M[v] = libre, v ≠ inicio
5          Agregar en Candidatos (v, ∞, ⊥)
6      (v, d, p) ← (inicio, 0, ⊥)
7      mientras v ≠ fin
8          Incluir (v, p) en S
9          para cada w en V[v],
10             si M[w] = libre y w no está en S entonces
11                 Actualizar (w, d + c, v) en Candidatos, donde c es √2 o 1 según v y w están en
                    diagonal o no
                   /* Notar que, según la definición de Actualizar, si d + c no es
                      menor que la distancia que ya se conocía para w entonces no se
                      actualiza. Y si se actualiza también se modifica el vértice
                      desde el que se llega, que es v.                              */
12         (v, d, p) ← Minimo (Candidatos), EliminarMinimo (Candidatos)
13     Crear Camino
14     p ← fin
15     Insertar p en Camino
16     mientras p ≠ inicio
17         p ← padre de p en S
18         Insertar p en Camino
19     devolver Camino
```

Figura 3: Algoritmo para encontrar el camino de largo mínimio.

(b) La cantidad de celdas libres es $O(n)$.

El conjunto S se puede representar mediante una matriz de las mismas dimensiones de M, en cuyas entradas se mantiene el padre del vértice correspondiente, con valor incial $\perp$. De esta forma las operaciones en S son $\Theta(1)$. La creación de S es entonces $O(n)$.

La cola de prioridad se implementa con un heap binario. El ciclo de la línea 4 es $O(n)$, porque simplemente se asigna cada elemento del heap en $O(1)$.

En cada iteración del ciclo de la línea 7 se incluye un elemento en S y se lo remueve de `Candidatos`, donde no se vuelve a insertar. Por lo tanto la canitdad de iteraciones es $O(n)$. En cada iteración se incluye un elemento en S en $O(1)$, se obtiene el mínimo de un heap en $O(1)$, se elimina el mínimo de un heap en $O(\log n)$, y se ejecuta el ciclo de la línea 9. En este ciclo se consulta si w está en S en $O(1)$ y se hacen actualizaciones en un heap, cada una de las cuales es $O(\log n)$. Pero la cantidad de actualizaciones no es mayor a 8, $O(1)$, por lo que el costo total de todas las actualizaciones es $O(\log n)$, y del ciclo interno es $O(\log n)$. Entonces todo el ciclo de la línea 7 es $O(n \log n)$.

Finalmente para la construcción del camino se accede en cada iteración del ciclo de la línea 16 una vez a S y se inserta en *Camino* lo cual tiene costo $\Theta(1)$. Este ciclo tiene $O(n)$ iteraciones por lo que su costo total también es $O(n)$.

Entonces, como hay una cantidad de pasos independiente del tamaño de la entrada, el costo es el del paso de mayor costo, $O(n \log n)$.

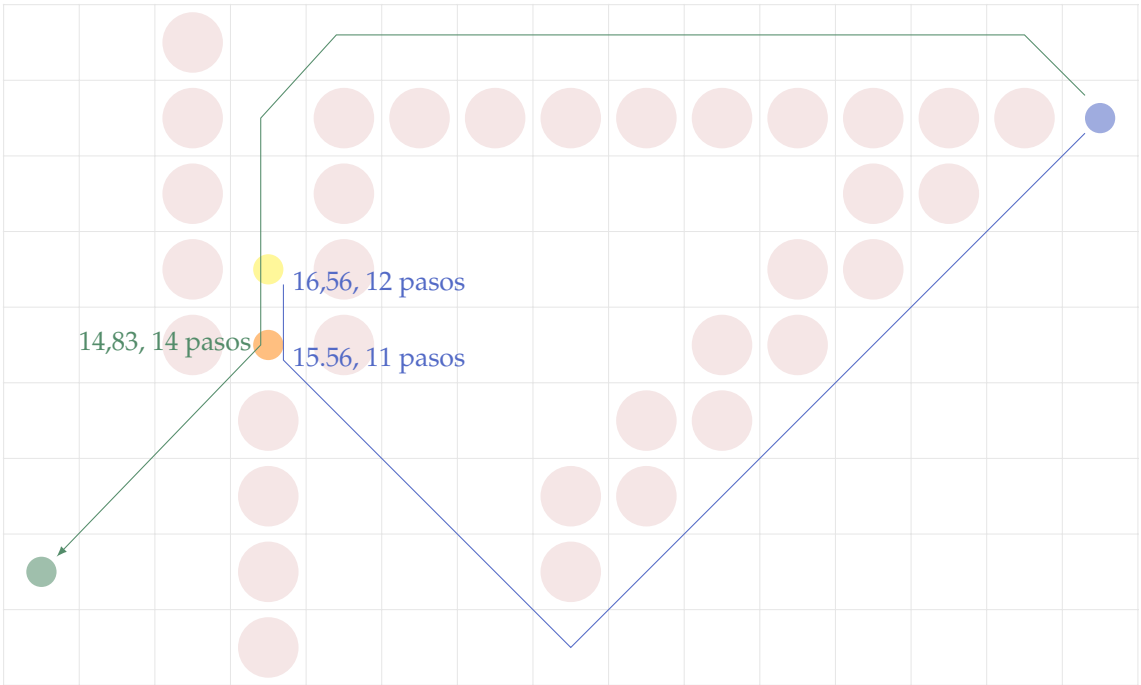
Ejemplo de estrategias alternativas que no resuelven el problema.

En el ejemplo se ve que la solución debe usar el camino indicado con la línea verde.

Se debe notar que en el primer paso, a pesar de poder acercarse al objetivo según ambas coordenadas, la solución se aleja según la coordenada vertical, por lo que se debe descartar una posible estrategia de intentar acercarse todo lo que sea posible en cada paso.

La solución requiere 14 pasos para llegar a la celda marcada con naranja, a pesar de que hay un camino, el indicado con azul, con 11. Esto muestra que no son aplicables estrategias basadas en `BFS`.

Tampoco sirve una modificación de `BFS` que actualice si se llega a una celda ya visitada por un camino más corto que aquel por el cual se había llegado antes. Se debe notar que el camino indicado en azul también requiere menos pasos para llegar a la celda que está marcada con amarillo. En la alternativa aquí sugerida al seguir el camino verde se podría modificar la distancia y el camino para llegar a la celda amarilla cuando se procesa la celda inmediatamente superior, pero la recorrida termina, por lo que no se llega a modificar la celda naranja. Si la modificación del algoritmo incluyera volver a encolar el vértice correspondiente la celda modificada, entonces dejaría de cumplirse los análisis de tiempo de ejecución porque se volverían a recorrer aristas. En este ejemplo eso pasaría al menos con todos lo que están a la izquierda de la celda naranja.



Ejercicio 3 (14 puntos)

En la sección 4.1 del libro de referencia se describe el problema de *Planificación de intervalos (Interval Scheduling)* y se diseña un algoritmo que se demuestra que es correcto usando la técnica que denomina *Stays Ahead*.

El problema consiste en, dado un conjunto I de intervalos, seleccionar un subconjunto de máxima cardinalidad A de I, tal que ningún par de intervalos de A se solapan.

El algoritmo es:

```
1 Algoritmo Planificación de intervalos (conjunto de intervalos I)
2     Inicializar el conjunto resultado A vacío
3     mientras I no es vacío
4         Sea e el intervalo de I que finaliza primero.
5         Agregar e al resultado A.
6         Remover de I el intervalo e y todos los intervalos que se solapan con e.
7     Devolver A.
```

Figura 4: Algoritmo para seleccionar un subconjunto de máxima cardinalidad de intervalos que no se solapan.

Demuestre la corrección del alogoritmo mediante la técnica *exchange-argument*.

Sugerencia: considere como medida de similutud entre soluciones el índice de la menor posición en la que son diferentes.

Solución:

La respuesta debe incluir.

- Demostración que el algoritmo termina dando como resultado una secuencia de intervalos A.
- Demostración de que A es válida, o sea, que está compuesta por intervalos pertenecientes a I sin solapamientos.
- La secuencia A se va a comparar con otra secuencia, O, que también es una solución válida. Se puede postular que es óptima.
- La secuencia O se obtuvo con otro algoritmo por lo que, para que sean válidos los argumentos usdos en la demostración, se la reordena con el mismo criterio que A, o sea, de manera creciente según los tiempos de finalización de los intervalos.
- Se considera la primera posición, i, en que las dos secuencias son diferentes.
- La diferencia puede consistir en que una de ellas tiene largo $i-1$, pero primero se supone que la cardinalidad de ambas es mayor o igual a i, y a_i es el intervalo de A, y o_i es el intervalo de O.
- Se considera otra secuencia, O', idéntica a O excepto porque se incluye a_i en lugar de o_i.
- Se hace notar que a_i no se solapa con ninguno de los intervalos anteriores de O' porque son los mismos que los de A, que es una solución válida.
- Se demuestra que a_i no se solapa con ninguno de los intervalos posteriores de O'. Esto se debe hacer de manera rigurosa, para lo cual debe usarse el criterio de selección del algoritmo.
- Se hace notar que los otros intervalos de O' no se solapan porque también son intervalos de O.
- Se concluye que O' es válida.
- Se hace notar que la cardinalidad de O' es igual a la de O.
- Se hace notar que, en comparación con O, O' se acerca a A, ya que es igual por lo menos hasta la posición i.
- Si O' sigue siendo distinta a A se repite el cambio. Como las longitudes de las secuencias son finitas se va a llegar a que ambas sean iguales hasta que al menos una de las dos termine.
- Si la diferencia se debe a que la longitud de O' es menor, entonces O no es óptima, lo que sería una contradicción si se había supuesto que lo es. En todo caso se demuestra que O no es mejor que A.
- Se demuestra que la diferencia no puede ser que la longitud de A sea menor. Para esto se hace ver que o_i es compatible con A, porque cada elemento de A es igual a uno de O, que es solución válida. Siendo así, el algoritmo no lo habría removido de I y no habría terminado hasta incluirlo.

Decimos que un conjunto de intervalos es *compatible* si ningún par de sus intervalos se superponen. Suponemos que el conjunto I tiene n elementos.

Veamos que el algoritmo termina y devuelve un conjunto compatible. Hay un único bucle controlado por la cardinalidad de I, y como en cada iteración se remueve al menos un elemento, en a lo sumo n iteraciones el algoritmo termina. El conjunto resultado A es compatible porque en cada iteración se remueven de I todos los intervalos que no son compatibles con A.

Vamos a demostrar que la cardinalidad de A es máxima entre todos los subconjuntos compatibles de I mediante el método *exchange argument*.

Con este método la solución del algoritmo se compara con otra genérica, O, que es compatible (que se puede suponer que es óptima) para demostrar que el valor objetivo de A (mayor cardinalidad en este problema) no es peor que el de O. Si O es diferente a A se modifica O para obtener otra solución, digamos O', que cumpla

- tiene menos diferencias con A de las que tenía O,
- también es una solución compatible,
- no empeora su valor objetivo con respecto al que tenía O.

Además se debe cumplir que la cantidad de diferencias entre una solución genérica y A sea finita. Por lo tanto, si O' sigue siendo diferente a A se podría repetir el procedimiento una cantidad finita de veces hasta transformar O en A, demostrando que el valor de A es mejor o igual que el de O. Como O es genérica y por lo tanto puede ser óptima, también A es óptima.

Sean $A = a_1, \ldots, a_k$ los intervalos elegidos por el algoritmo y $O = o_1, \ldots, o_m$ una solución genérica. Ambas secuencias corresponden a intervalos que no se solapan y están ordenadas de manera creciente según los tiempos de finalización de los intervalos.

Si se toma como hipótesis que las cantidades de intervalos de A y O son iguales, por ejemplo $A = a_1, \ldots, a_m$ y $O = o_1, \ldots, o_m$, entonces no habría más nada que demostrar porque la optimalidad de A consiste en tener máxima cantidad de intervalos.

La secuencia A está en el orden en que el algoritmo selecciona los elementos. La secuencia O es posiblemente un reordenamiento de la forma en que se obtuvo la otra solución. Este reordenamiento no afecta la validez ni la cardinalidad de esa solución.

Como medida de similitud entre soluciones consideramos la longitud del prefijo en que son iguales.

Supongamos que ambas secuencias son iguales hasta la posición $i - 1$.

El ejemplo con el que se presenta el método exchange argument en el libro de referencia, minimizar la máxima tardanza, incluye conceptos como inversiones, tiempo ocioso y tiempo total. Esos conceptos no son esenciales al método, son solo un ejemplo.

En el problema planteado en el libro no se deben elegir algunos intervalos, sino que hay que reordenar todos ellos, por lo que en dos soluciones diferentes necearimente hay inversiones. En el actual problema en cambio, si O se ordena según el mismo criterio que A es imposible que haya una inversión.

En el problema de minimizar la máxima tardanza el inicio y la finalización de cada intervalo no son parte de los datos, sino que son el resultado esperado. En ese contexto tiene sentido intentar eliminar tiempos ociosos y demostrar que la solución óptima no los tiene. En el problema actual es imposible demostrar que la solución no tiene tiempos ociosos porque los tiempos de inicio y fin son parte de los datos y no pueden modificarse. En cualquier solución óptima es posible que el tiempo de fin de algún intervalo sea menor estricto que el de inicio del siguiente, por lo que necesariamente hay tiempos ociosos.

Vamos a crear una solución O' más cercana a A de lo que es O y que tenga la misma cantidad de elementos que O.

Para eso hacemos que $o'_j = a_j = o_j$ para $j \in \{1 \ldots i-1\}$, $o'_i = a_i$, $o'_j = o_j$ para $j > i$. O sea, O' es igual a O, excepto en la posición i, en donde es igual a A.

Veamos que O' consiste en intervalos que no se solapan, y por lo tanto es compatible. Excepto por o'_i son los mismos intervalos de O, que no se solapan entre sí. Veamos que o'_i no se solapa con ninguno de los otros intervalos de O'. Como o'_i es un intervalo de A no se solapa con los intervalos o'_j para $j \in \{1 \ldots i-1\}$ porque también son intervalos de A.

Como las secuencias están ordenadas de manera creciente según su tiempo de finalización, y como los itervalos de O no se solapan, la finalización de o_i es menor o igual al inicio de los intevalos o_j, para

$j > i$. Por el criterio de decisión del algoritmo, en la iteración $i-$ésima el tiempo de finalización de a_i es el mínimo entre los que no se solapan con los intervalos que ya están en A. Como o_i es uno de esos intervalos, su tiempo de finalización es mayor o igual que el de a_i, que es igual a o'_i, por lo que o'_i tampoco se solapa con los intervalos de O posteriores a o_i, que, por construcción, son los mismos que los de O'.

> Este punto debe demostrarse de manera rigurosa, haciendo valer el criterio con el cual el algoritmo selecciona los intervalos.

Entonces O' se acerca a A, es compatible y tiene la misma cantidad de elementos que O. En a lo sumo mín$\{k, m\}$ transformaciones se obtiene una solución que es igual a A desde 1 hasta mín$\{k, m\}$. Si m es menor que k entonces la solución O no era óptima. Veamos que m no puede ser mayor que k. Si m fuera mayor que k entonces o'_{k+1} sería compatible con los intervalos de A, ya que estos están contenidos en O'. Se llega a una contradicción porque en ese caso el algoritmo no lo habría removido en la iteración k ni en ninguna de las anteriores, y por lo tanto no habría terminado y lo habría incluido en la siguiente iteración.