Informe de DAA

Integrantes:

- Leonardo Amaro
- Alfredo Montero
- Anthuán Montes de Oca

1 División MEX

Solución Implementada

Para la solución de este problema implementamos un algoritmo de ventana deslizante que itera por el array y encuentra los bloques MEX (definición 1.1) del mismo. Cada vez que se encuentra uno de estos bloques MEX se calculan las diferentes divisiones que contienen a este bloque MEX en conjunto con los descubiertos anteriormente. Al terminar la iteración por el array y encontrar todos los bloques MEX se suma el valor calculado para cada bloque MEX dando como resultado final todas las divisiones válidas posibles.

A continuación daremos la información necesaria para entender el funcionamiento y la correctitud de nuestro algoritmo ...

Teorema (1.1)

En un array A, una división es válida con el MEX de todos sus subarrays $= m$ si y solo si $MEX(A) = m$.

En otras palabras: todos los subarrays de una división válida tienen un MEX igual al MEX del array completo.

Demostración: .

($\Rightarrow$)

Tomemos una división $A1, A2, \ldots, An$ válida con MEX de sus subarrays m. $\forall i \in [1, n], MEX(Ai) = m$.

$\Rightarrow \forall k \in [0, m], k \in Ai$ (De lo contrario k sería el MEX de Ai) $\Rightarrow \forall k \in [0, m], k \in A$. $\Rightarrow MEX(A) \geq m$.

Si $MEX(A) > m$: $\Rightarrow m \in A \Rightarrow \exists i : m \in Ai \Rightarrow MEX(Ai) \neq m$. CONTRADICCIÓN. $\Rightarrow MEX(A) = m$.

($\Leftarrow$)

Sea $MEX(A) = m$ tomemos una división válida $A1, A2, \ldots, An$. Donde $\forall i \in [1, n], MEX(Ai) = \mu$.

Si $\mu < m$: $\forall i \in [1, n], \mu \notin Ai \Rightarrow \mu \notin A \Rightarrow MEX(A) = \mu < m$. CONTRADICCIÓN. $\mu \geq m$.

Si $\mu > m$: $\Rightarrow \forall i \in [1, n], m \in Ai \Rightarrow m \in A \Rightarrow MEX(A) \neq m$. CONTRADICCIÓN. $\mu = m$. ♠

Sabiendo esto pasemos a una definición importante para la solución del ejercicio.

Definición (1.1)

En un array A un bloque MEX es un subarray minimal de A tal que su MEX es igual a $MEX(A)$.

Teorema (1.2)

Todo subarray de una división válida de A contiene un bloque MEX.

No consideramos necesaria la demostración de este teorema pero sí nos pareció importante mencionarlo.

Teorema (1.3)

Sea A un array y Ak un subarray de A, Ak es un bloque MEX si y solo si $\forall i \in [0, MEX(A)), i \in Ak \wedge Ak[0], Ak[len-1] < MEX(A) \wedge Ak[0], Ak[len-1] \neq Ak[i] \forall i \in (0, len-1)$.

Demostración: .

($\Rightarrow$)

$MEX(Ak) = MEX(A) = m$. Si $\exists i : 0 \leq i < m; i \notin Ak \Rightarrow MEX(Ak) = i < m$. CONTRADICCIÓN. $\Rightarrow \forall i \in [0, MEX(A)), i \in Ak$.

Si $Ak[0] > m \Rightarrow$ Sea $Ak1$ subarray de Ak de $Ak[1]$ a $Ak[len-1] \Rightarrow MEX[Ak1] = m \Rightarrow Ak$ no es minimal. CONTRADICCIÓN. $\Rightarrow Ak[0] \leq m$.

Si $Ak[0] = m$. CONTRADICCIÓN TRIVIAL. $\Rightarrow Ak[0] < m$.

(La misma demostración se aplica para $Ak[len-1]$.)

Si $\exists i; Ak[i] = Ak[0] \Rightarrow$ Sea $Ak1$ subarray de Ak de $Ak[1]$ a $Ak[len-1]$ $\Rightarrow MEX[Ak1] = m \Rightarrow Ak$ no es minimal. CONTRADICCIÓN. $\Rightarrow Ak[0] \neq Ak[i] \forall i \in (0, len-1]$.

(La misma demostración se aplica para $Ak[len-1]$.)

($\Leftarrow$)

Al ser Ak un subarray de A, si todos los números mayores o iguales que 0 y menores que $MEX(A)$ pertenecen a $Ak \Rightarrow MEX(Ak)$ es igual a $MEX(A)$. Si los bordes de Ak contienen números únicos menores que $MEX(A) \Rightarrow$ al reducir el tamaño de alguno de los bordes el $MEX(AK)$ sería igual al valor de ese borde que es menor que $MEX(A) \Rightarrow Ak$ es minimal $\Rightarrow Ak$ es un bloque MEX. ♠

Definición (1.2)

Sea A un array y Ak un subarray de una división válida de A, se le llama *Bloque Característico* de Ak al bloque MEX contenido en Ak cuya posición inicial es más cercana a la posición inicial de Ak.

Conociendo esto podemos concretar acerca del funcionamiento de nuestro algoritmo el cual primeramente calcula el MEX del array de entrada. Con este valor calculado el algoritmo busca todos los bloques MEX del array basándose en el Teorema 1.3. Cada vez que encontramos un nuevo bloque MEX añadimos los casos resultantes de las divisiones donde solo él y los bloques previamente descubiertos son bloques característicos. De esta forma al encontrar el último bloque tenemos previamente todos los casos donde él no es bloque característico y añadimos todos donde él sí lo es, teniendo para ese momento el total de divisiones válidas del array.

Complejidad Temporal

Primeramente tenemos el cálculo del MEX, este se realiza una sola vez por ejecución, al inicio del algoritmo. Aquí tenemos un primer ciclo que itera por el grafo rellenando un diccionario de hallazgos; este primer ciclo realiza n iteraciones y en cada una sus acciones tienen tiempo constante, por lo que su complejidad es $O(n)$. Luego de este hay un segundo ciclo que itera por todos los números desde 0 hasta el máximo del algoritmo y en cada iteración busca si ese elemento está en el diccionario de hallazgos. Buscar en el diccionario se realiza en tiempo constante y aunque el ciclo se realiza por los números menores al máximo del array este ciclo termina al encontrar el primer valor que no pertenece

al array, por lo que no realizará más de n iteraciones, siendo n el tamaño del array; por lo que este ciclo también tiene complejidad de $O(n)$ al igual que el método entero.

Luego de calculado el MEX se hace una iteración por el array. Dentro de esta iteración la mayoría de las operaciones se realizan en tiempo constante, a excepción de dos ciclos independientes. El primero de estos sucede cuando se encuentra un bloque, para obtener el valor del inicio de ese bloque se recorre un array del tamaño del MEX previamente calculado; este valor es siempre menor o igual a n+1 por lo que el ciclo tiene complejidad $O(n)$. El otro ciclo ocurre justo después, donde se recorren todos los bloques ya encontrados para calcular los casos que añade este nuevo bloque; El número de bloques en el array es siempre menor o igual que n, por lo que este ciclo es también $O(n)$. Por lo tanto la complejidad final del ciclo entero sería de $O(n^2)$.

Hay un último ciclo que itera por los bloques para sacar la suma que devuelve el algoritmo como resultado final; al igual que en el análisis anterior, este ciclo es en $O(n)$.

Por lo tanto la complejidad temporal de nuestro algoritmo general será de $O(n)$. Aunque he de mencionar que tanto el MEX como el número de bloques, a pesar de tener valores máximos de $n + 1$ y n respectivamente, suelen tener valores bastante menores por lo que los tiempos del algoritmo en muchos de los casos tienen valores cercanos a los de un algoritmo lineal.

Línea de Pensamiento

Para la solución de este ejercicio primeramente analizamos las especificaciones del problema y las propiedades que podían tener las soluciones. En este análisis construimos el Teorema 1.1. Teniendo una idea general del problema escribimos un código de "fuerza bruta" para comprobar con ejemplos la posible correctitud de futuros algoritmos.

Con la intención de llegar a alguna solución "decente" intentamos buscar cotas para las soluciones válidas, con la esperanza de poder reducir los casos a probar. La única cota rescatable de entre las que encontramos fue que: el número de subarrays de una división válida debe ser menor o igual al número de veces que aparece el número menor al MEX que menos aparece en el array. Esta cota no tuvo mucha utilidad práctica pero pensamos que pudo ser la idea inicial para la futura definición de los bloques MEX.

Una vez saliendo de la pura teoría y entrando más en las técnicas para abordar el ejercicio, nuestros pensamientos estaban más encaminados en buscar una forma de "picar" el problema para trabajar con subproblemas más pequeños. Ideas como picar a la mitad, o picar por el mayor o menor elemento no dieron

ningún resultado relevante, la primera idea decente surge junto con la idea de los bloques MEX. La idea era buscar el primer y el último de estos bloques y llamar recursivo al subarray que quedara en medio de estos. Esta posible solución no fue viable ya que en el subarray podían surgir múltiples casos distintos (un solo bloque dentro, dos bloques que se interceptan, bloques que se interceptan con los extremos, etc...) y no había una forma concreta de calcular las divisiones totales de manera recursiva.

En ese momento pasamos a un enfoque más iterativo, continuando con la idea de los bloques MEX. Intentamos una primera iteración para encontrar todos los bloques y luego relacionarlos entre ellos para calcular las divisiones totales, pero la fórmula al calcular la interacción entre estos era compleja y crecía con cada nuevo caso que encontrábamos, por lo que en el caso de encontrar finalmente una fórmula que abarcara todos los casos, esta sería muy compleja y la demostración de su correctitud lo sería aún más.

Fue entonces cuando decidimos utilizar el enfoque de contar los casos a medida que encontrábamos los bloques de forma tal que no contáramos ninguno de esos casos dos veces. Esta idea prometía ser, no solo más ordenada y sencilla, también era más eficiente que todas las anteriores que habíamos pensado. De esta forma, luego de afinar la manera de añadir los casos nuevos para que fuera lo más "limpio" posible, se finalizó la creación de nuestro algoritmo de ventana deslizante que da solución a nuestro problema de manera eficiente.

2 Ordenamiento de Pociones

Problema

Rodrigo es el maestro de una academia de magia y debe organizar y clasificar sus ingredientes mágicos. Son tan mágicos que, consumidos de la forma adecuada, te permiten ver criaturas fantásticas y luces de colores, dejando poca resaca (Rodrigo no los consume pues es un maestro sano). Tiene dos estantes mágicos, uno llamado "a" y otro llamado "b", ambos con n frascos de pociones, cada uno etiquetado con un número del 1 al n. Cada frasco está en un lugar aleatorio en los estantes, pero el número en cada frasco es único en cada estante.

Tu tarea es ayudar a Rodrigo a organizar ambos estantes en el orden correcto, de modo que los números en los frascos estén en orden ascendente de izquierda a derecha en ambos estantes. Sin embargo, estos estantes mágicos sólo se pueden mover siguiendo unas reglas:

1. Puedes elegir cualquier número i entre 1 y n;
2. Encuentra el frasco en el estante "a" que tiene el número i y cambia su posición con el frasco en la posición i;

3. Luego, encuentra el frasco en el estante "b" que tiene el número i y cambia su posición con el frasco en la posición i.

Tu objetivo es organizar ambos estantes con el menor número de movimientos posible, asegurándote de que todos los frascos en ambos estantes estén en el orden correcto al finalizar la tarea.

Solución

Para solucionar el problema utilizamos programación dinámica, dado que para lograr la cantidad de pasos óptimos es debido tener en cuenta ramificaciones que pueden ocurrir hasta $n-2$ pasos después de realizar un movimiento, lo cuál hace imposible lograr un algoritmo que lo resuelva sin probar todas (o casi todas) las soluciones posibles.

Intentamos utilizar un algoritmo greedy, que en cada paso analizaba todos los posibles movimientos y escogía el que más pociones ordenaba a la vez, y este funcionaba en casos aleatorios más de un 90% de las veces y en $O(n^2)$, pero cometía errores ya que podían ocurrir algunas estructuras en las que tomar una decisión que arreglaba menos pociones, permitía arreglar más en el paso próximo, o en 2 pasos más adelante, y así sucesivamente.

En este punto también notamos que en el árbol de estados de los arrays, muchos estados se repetían, en particular cuando se seleccionaba un número i y después otro j, se obtenía siempre el mismo resultado si escojías j y después i (lo cual se demostrará mas adelante para la complejidad temporal).

Entonces queda programación dinámica como última (y bastante buena) opción, la cual implementaríamos utilizando memoización. Pero hay un problema.

El código

```
def dynamic_potion_sort_aux(A: list[int], B: list[int], moves: dict
    [str, list[int]]):
    if (A, B) in moves:
        return moves[(A, B)]
```

sería una implementación típica de memoización en $O(1)$, donde los estados que ya se calcularon devuelven su resultado inmediatamente. Sin embargo 'A' y 'B' son listas, las cuales son mutables, y los objetos mutables no son hasheables, por tanto no pueden ser utilizados como llave de diccionarios (al menos no en $O(1)$), por lo que tuvimos que recurrir a

```
def dynamic_potion_sort_aux(A: list[int], B: list[int], moves: dict
    [str, list[int]]):
    list_repr = (tuple(A), tuple(B))
    if list_repr in moves:
        return moves[list_repr]
```

Que realiza $O(2n)$ operaciones en simplemente comprobar si ya se calculó ese estado. Por lo tanto encontramos una manera de representar las listas usando números binarios, aprovechandonos de que python tiene ints de tamaño arbitrario, no necesitamos utilizar arrays de bytes ni nada por el estilo.

Sea la lista:

$$[5, 3, 1, 2, 4]$$

una lista válida para el problema, bien podría tomar el lugar de A o de B, ¿cómo podemos representarla en binario?

Sea $c = \lceil \log_2(n) \rceil$ (en este caso $n = 5$, $c = 3$) el exponente mas pequeño tal que $2^c > n$, también conocido como la cantidad mínima de bits con la que podemos representar cualquier número entre 1 y n, podemos representar la lista anterior de la siguiente manera (en 'big endian'):

100 010 001 011 101

Esto se puede calcular para ambas listas en $O(n)$ una sola vez en todo el algoritmo, luego solo es cuestión de mantener estos identificadores actualizados cuando se hagan swaps. Para hacer un swap utilizaremos las operaciones bit a bit de OR '|', XOR, '^', y shifting ' << ', de manera tal que sí queremos cambiar el elemento en la i-esima posición de valor r a valor k, hacemos:

```
identificador ^= r << (i * c) # Esto convierte la i-esima seccion a
    ceros
identificador |= k << (i * c) # Esto hace que el valor de la
    seccion sea k
```

Entonces hacemos eso para las dos secciones que se quieran intercambiar, y así se mantiene actualizada la representación de las listas en $O(1)$ (aunque con una constante bastante grande, más sobre esto después) cada vez que se hace un swap.

Código final

```
def dynamic_potion_sort(A: list[int], B: list[int]):
    moves = {}
    n = len(A)

    indexes_A = [-1] * n
    indexes_B = [-1] * n

    for i in range(n):
        indexes_A[A[i] - 1] = i
        indexes_B[B[i] - 1] = i

    bits_per_item = int(ceil(log2(n)))

    initial_identifier_A = A[n - 1]
    initial_identifier_B = B[n - 1]

    for i in range(n - 2, -1, -1):
        initial_identifier_A <<= bits_per_item
        initial_identifier_A |= A[i]

        initial_identifier_B <<= bits_per_item
```

```
        initial_identifier_B |= B[i]

    return dynamic_potion_sort_aux(
        A,
        B,
        moves,
        indexes_A,
        indexes_B,
        (initial_identifier_A, initial_identifier_B),
        bits_per_item,
    )

def dynamic_potion_sort_aux(
    A: list[int],
    B: list[int],
    moves: dict[tuple[int, int], list[int]],
    indexes_A: list[int],
    indexes_B: list[int],
    lists_repr: tuple[int, int],
    bits_per_item: int,
) -> list[int]:
    if lists_repr in moves:
        return moves[lists_repr]

    n = len(A)

    best_swaps_count = -1
    best_swaps = []
    best_swap_elem = -1
    for i in range(n):
        elem = i + 1

        # skip item if correct
        if A[i] == elem and B[i] == elem:
            continue

        A_item_at_i = A[i]
        B_item_at_i = B[i]

        i_index_A = indexes_A[i]
        i_index_B = indexes_B[i]

        # swap potions
        swap_items(A, i, i_index_A)
        swap_items(B, i, i_index_B)

        # update indexes array
        swap_items(indexes_A, A_item_at_i - 1, i)
        swap_items(indexes_B, B_item_at_i - 1, i)

        # update lists representation
        new_A_repr = (lists_repr[0] ^ (A_item_at_i << (
    bits_per_item * i))) ^ (
            (i + 1) << (bits_per_item * i_index_A)
        )
        new_A_repr |= (A_item_at_i << (bits_per_item * i_index_A))
    | (
```

```
            (i + 1) << (bits_per_item * i)
        )

        new_B_repr = (lists_repr[1] ^ (B_item_at_i << (
    bits_per_item * i))) ^ (
            (i + 1) << (bits_per_item * i_index_B)
        )
        new_B_repr |= (B_item_at_i << (bits_per_item * i_index_B))
    | (
            (i + 1) << (bits_per_item * i)
        )
        swaps = dynamic_potion_sort_aux(
            A,
            B,
            moves,
            indexes_A,
            indexes_B,
            (new_A_repr, new_B_repr),
            bits_per_item,
        )

        # revert swap
        swap_items(A, i, i_index_A)
        swap_items(B, i, i_index_B)

        # revert indexes array update
        swap_items(indexes_A, A_item_at_i - 1, i)
        swap_items(indexes_B, B_item_at_i - 1, i)

        if best_swaps_count == -1 or len(swaps) < best_swaps_count:
            best_swaps = swaps
            best_swaps_count = len(swaps)
            best_swap_elem = elem

    if best_swaps_count == -1:
        return []

    result = [best_swap_elem, *best_swaps]

    moves[lists_repr] = result

    return result

def swap_items(A: list[int], index1: int, index2: int):
    temp = A[index1]
    A[index1] = A[index2]
    A[index2] = temp
```

Correctitud

La idea del algoritmo es particularmente simple, comienza con los arrays de pociones ‘A‘ y ‘B‘, y prueba recursivamente todas las acciones posibles. Tras probar cada una devuelve la que llegó a la solución en menos pasos. No creemos que el backtracking requiera demostración formal alguna.

Complejidad temporal

Para realizar el intercambio de pociones, según la descripción del problema, es necesario encontrar en qué posición se encuentra la poción i, para poder intercambiar su lugar con la poción que está en la posición i del array. Para acelerar estas búsquedas se crean dos arrays llamados '$indexesA$' e '$indexesB$', donde el elemento i del array contiene la posición en que se encuentra la poción i en A o B correspondientemente, lo cual hace que la operación de swappeo sea $O(1)$ a costo de un paso que se ejecuta una sola vez en $O(n)$, y de mantenerlas actualizadas para cada llamada del array, lo cual se hace con un swappeo en $O(1)$.

```
def swap_items(A: list[int], index1: int, index2: int):
    temp = A[index1]
    A[index1] = A[index2]
    A[index2] = temp
```

Al caso, la función '$swap_items$' intercambia dos elementos en un array en un evidente $O(1)$.

Entonces el método de preparación:

```
def dynamic_potion_sort(A: list[int], B: list[int]):
    moves = {}
    n = len(A)

    indexes_A = [-1] * n
    indexes_B = [-1] * n

    for i in range(n):
        indexes_A[A[i] - 1] = i
        indexes_B[B[i] - 1] = i

    bits_per_item = int(ceil(log2(n)))

    initial_identifier_A = A[n - 1]
    initial_identifier_B = B[n - 1]
    for i in range(n - 2, -1, -1):
        initial_identifier_A <<= bits_per_item
        initial_identifier_A |= A[i]

        initial_identifier_B <<= bits_per_item
        initial_identifier_B |= B[i]

    return dynamic_potion_sort_aux(
        A,
        B,
        moves,
        indexes_A,
        indexes_B,
        (initial_identifier_A, initial_identifier_B),
        bits_per_item,
    )
```

En 5 y 6 se crean dos arrays en $O(n)$ cada uno, y el bucle de 8-10 los inicializa con las posiciones de los elementos de A y B en $O(n)$ también. La creación de

los identificadores de las listas en 12-22 también se realiza en $O(n)$. Por tanto la complejidad temporal del método de inicialización es $O(n + n + n) = O(n)$.

En el método principal, la complejidad temporal de un algoritmo de programación dinámica es:

O(cantidad de estados únicos $\times$ complejidad temporal de cada uno)

Analicemos entonces cada estado, para después calcular el total. Tomémoslo por secciones:

```
1  def dynamic_potion_sort_aux(
2      A: list[int],
3      B: list[int],
4      moves: dict[tuple[int, int], list[int]],
5      indexes_A: list[int],
6      indexes_B: list[int],
7      lists_repr: tuple[int, int],
8      bits_per_item: int,
9  ) -> list[int]:
10     if lists_repr in moves:
11         return moves[lists_repr]
12
13     n = len(A)
14
15     best_swaps_count = -1
16     best_swaps = []
17     best_swap_elem = -1
```

En 10 se busca si este estado ya se ha calculado antes, lo cual es una operación de $O(1)$ típicamente, ya que 'moves' es un diccionario.

Entre 13 y 17 se realizan otras inicializaciones misceláneas también en $O(1)$.

Esta sección se llevó a cabo en $O(1)$.

```
1  for i in range(n):
2      elem = i + 1
3
4      # skip item if correct
5      if A[i] == elem and B[i] == elem:
6          continue
7
8      A_item_at_i = A[i]
9      B_item_at_i = B[i]
10
11     i_index_A = indexes_A[i]
12     i_index_B = indexes_B[i]
13
14     # swap potions
15     swap_items(A, i, i_index_A)
16     swap_items(B, i, i_index_B)
17
18     # update indexes array
19     swap_items(indexes_A, A_item_at_i - 1, i)
20     swap_items(indexes_B, B_item_at_i - 1, i)
```

La línea 5 comprueba si en el elemento i no es necesario hacer ningún swap, saltándose los índices que ya están ordenados. Esto es en $O(1)$.

En 8-12 se inicializan variables con las posiciones del elemento i en A y B, así como el valor del elemento que se encuentra en la posición i en A y B. Todo

en $O(1)$.

En 15-20 se realizan las operaciones de swappeo, mostrado anteriormente que es en $O(1)$.

Esta sección también se ejecutó en $O(1)$.

```
# update lists representation
new_A_repr = (lists_repr[0] ^ (A_item_at_i << (bits_per_item * i))) ^ (
    (i + 1) << (bits_per_item * i_index_A)
)
new_A_repr |= (A_item_at_i << (bits_per_item * i_index_A)) | (
    (i + 1) << (bits_per_item * i)
)

new_B_repr = (lists_repr[1] ^ (B_item_at_i << (bits_per_item * i))) ^ (
    (i + 1) << (bits_per_item * i_index_B)
)
new_B_repr |= (B_item_at_i << (bits_per_item * i_index_B)) | (
    (i + 1) << (bits_per_item * i)
)
```

Un montón de operaciones de bits, pero todo en $O(1)$, aunque introduce una constante bastante grande, cuyas consecuencias veremos más tardes.

```
swaps = dynamic_potion_sort_aux(
    A,
    B,
    moves,
    indexes_A,
    indexes_B,
    (new_A_repr, new_B_repr),
    bits_per_item,
)

# revert swap
swap_items(A, i, i_index_A)
swap_items(B, i, i_index_B)

# revert indexes array update
swap_items(indexes_A, A_item_at_i - 1, i)
swap_items(indexes_B, B_item_at_i - 1, i)

if best_swaps_count == -1 or len(swaps) < best_swaps_count:
    best_swaps = swaps
    best_swaps_count = len(swaps)
    best_swap_elem = elem
```

En 1 está la llamada recursiva, el efecto de esta se verá al calcular la cantidad de estados únicos, ya que como vimos antes, si un estado ya fue calculado el método devuelve instantáneamente.

En 12-17 se revierten los swaps hechos antes de la llamada recursiva, todo nuevamente en $O(1)$.

En 19-22 se comprueba si encontramos un nuevo mejor swap, o si es el primero. $O(1)$.

El bucle entero corre n veces y sus operaciones interiores se realizan en $O(1)$, por tanto la ejecución del bucle es $O(n)$.

```
if best_swaps_count == -1:
    return []

result = [best_swap_elem, *best_swaps]

moves[lists_repr] = result

return result
```

El paso de finalización, en 1 si no se realizó ningún swap, se devuelve el array vacío. $O(1)$.

En otro caso se construye un array con mejor swap, y los swaps definidos recursivamente, en $O(n)$ por la deconstrucción de la lista '$best_swaps$'.

En 6 se guarda el resultado de este estado para la memoización. $O(1)$ típicamente.

Entonces en general, la ejecución de cada estado del problema es $O(n+n) = O(n)$.

Ahora, determinemos la cantidad máxima de estados que puede tener el problema.

No se ...

Asumamos peor caso $O(n!^2)$ estados (que no es, son muchos menos, pero esta parte de la demostración la dejamos a medias por *skill issue*), la complejidad temporal sería $O(n \cdot n!^2)$.

Tesoro pirata

En una isla desierta, un grupo de piratas ha trazado un mapa lleno de senderos secretos. Cada sendero tiene un tesoro escondido en alguna parte de su trayecto. Los piratas de la tripulación del infame capitán Jack Amarow quieren esconderse en las intersecciones de los senderos para vigilar sus tesoros. En una intersección se pueden cruzar dos senderos o más. Todas las intersecciones tienen unas palmeras a las que los piratas se pueden encaramar. Mientras más alta la palmera, más difícil es subirse y por tanto más costo tiene.

Tu misión es ayudar a los piratas a encontrar el conjunto mínimo de intersecciones donde deben esconderse para poder vigilar todos los senderos que llevan a sus tesoros y que el costo de subirse en las palmas sea mínimo.

Resumen

Se tiene un grafo $G(V, E)$ no dirigido donde cada vértice tiene un costo de seleccionarlo (la altura de la palmera). Se necesita seleccionar un conjunto $V' \subseteq V$ donde se cumple que $G'(V', E')$ donde $E' = \{< v, e > \in E | v, e \in V'\}$ de forma que se cumpla que G' es el cubrimiento menor tal que $E' = E$ y que el costo de seleccionar V' sea mínimo. Llamaremos a este problema Cobertura Mínima de Aristas con Costo Mínimo (CMACM).

Solución fuerza bruta

El algoritmo usa un acercamiento recursivo para explorar todos los cubrimientos de vértices posibles, apuntando a encontrar el que tiene el menor costo. Mantiene la mejor cobertura encontrada hasta el momento comparando la cantidad de nodos y el costo de cada cubrimiento potencial. Este es un método de fuerza bruta y puede que no sea eficiente con grafos grandes, pero garantiza que encuentra la solución óptima.

Correctitud

Probar la correctitud de este algoritmo es bastante sencillo, ya que genera todas las combinaciones posibles y devuelve la mejor.

Complejidad temporal

La complejidad temporal del código dado se puede analizar examinando la función recursiva `find_min_cover_aux`:

- **Llamadas Recursivas:** La función realiza una llamada recursiva para cada vértice que no está en la cobertura actual. En el peor de los casos, esto significa explorar todos los subconjuntos de vértices.
- **Exploración de Subconjuntos:** Para un grafo con n vértices, hay 2^n subconjuntos posibles. La función explora cada subconjunto para encontrar la cobertura mínima.
- **Cálculo de Costos:** Para cada subconjunto, la función calcula el costo y verifica si el subconjunto cubre el grafo. Esto implica verificar la cobertura y actualizar la mejor cobertura y el costo.
- **Caso Base:** El caso base verifica si la cobertura actual cubre el grafo, lo cual toma $O(n)$ tiempo en el peor de los casos (asumiendo que el método `is_covered` verifica todos los vértices).

Combinando estos factores, la complejidad temporal está dominada por el número de subconjuntos explorados, que es $O(2^n)$. Por lo tanto, la complejidad temporal general del código es:

$$O(2^n \cdot n)$$

NP-Completitud

Por la descripción del problema se puede pensar que el problema tiene características donde coincide con el problema Vertex-Cover.

Primero debemos convertir nuestro problema a un problema de decisión, para luego demostrar que este problema de decisión asociado a nuestro problema, es NP-Completo.

Problema de decisión

Dada la misma entrada del problema original, junto a 2 valores enteros k y n, devuelve verdadero en caso de que exista un subconjunto de vértices de tamaño n con costo k del grafo G, falso en otro caso. Nombremos este problema Cobertura Limitada de Aristas con Costo Limitado de Decisión ($CLACD$).

1. Probar que $CLACD \in NP$

   Para esto vamos a partir del concepto de cuando un problema es NP. Un problema P es NP si para cada instancia x del problema que devuelve true, existe un certificado y con $|y| = O(|x|^c)$ para alguna c constante y existe un algoritmo A que toma a x y y como entrada y cumple que $A(x, y) = 1$.

   El certificado que vamos a utilizar es un conjunto de vértices, que al ser verificados, cumplen con las condiciones.

   $|y| = k$, es el máximo tamaño que puede tener un conjunto de vértices pertenecientes a un grafo G, con $V \in G$. La entrada del problema es $(G(V, E), n, k)$ con un coste en cada vértice, con lo cual $|x| = O(|V|+|E|)$.

   $|y| = O(|x|^c) \rightarrow k = O((|V| + |E|)^c)$ para $c = 1$ se cumple que $k = O(|V| + |E|)$.

2. Seleccionar un problema de NPC conocido:

   El problema que vamos a utilizar es el problema **Vertex Cover**.

3. Describir un algoritmo que compute una función f la cual debe mapear para cada instancia x de **Vertex Cover** a una instancia $f(x)$ de $CLACD$.

   Necesitamos demostrar la existencia de un algoritmo que convierta la entrada del problema **Vertex Cover** a la entrada de nuestro problema y uno que transforme la salida de nuestro problema en la salida del **Vertex Cover**, y que esto ocurra en tiempo polinomial, pudiéramos representar este proceso de la siguiente manera:

   $$\text{Input(Vertex Cover)} \rightarrow \text{Input(CLACD)} \Longrightarrow S \Longrightarrow \text{Output(CLACD)} \rightarrow \text{Output(Vertex Cover)}$$

   Donde S es un algoritmo que resuelve $CLACD$. La entrada del algoritmo **Vertex Cover** es un grafo $G(V, E)$ y un entero positivo k que representa el tamaño máximo del conjunto de cobertura de vértices que se busca. Un algoritmo que mapee el grafo G en otro $G'(V', E)$ donde $V' = \{v \in V\}$ y a cada nodo se le adiciona un costo, el cual es 0; además el número de costo k es 0. Resta entonces convertir la salida de nuestro algoritmo a la salida del **Vertex Cover**. En caso que $CLACD$ de verdadero la salida del **Vertex Cover** es verdadera, de igual forma si da falso.

4. Probar que este algoritmo que computa f se resuelve en tiempo polinomial.

   Complejidad temporal:

   $$\text{Input(Vertex Cover)} \rightarrow \text{Input(CLACD)}$$

   Sea la entrada $G(V, E)$, por cada $v \in V$ se crea v' igual a v pero con la adición de un valor de coste igual a 0, esto se realiza $|V|$ veces. Luego por cada par $< v, w > \in E$ se crea el par correspondiente $< v', w' >$ de forma que $< v', w' > \in E'$, esto se realiza $|E|$ veces. Por tanto la complejidad temporal de la conversión es $O(|V| + |E|)$.

   $$\text{Output(CLACD)} \rightarrow \text{Output(Vertex Cover)}$$

   Ocurre homologamente en sentido contrario, siendo evidente que la complejidad temporal es $O(|V| + |E|)$.

   Luego el algoritmo que computa f se hace en tiempo polinomial.

2.1 NP-hard

Vamos a demostrar ahora que el problema $CMACM \in$ **NP-hard**. Para esto nos basaremos en que $CLACD \in NPC$, se puede reducir en $CMACM$. Para esto, al igual que en el caso anterior, vamos a hacer una reducción. Ya seleccionamos un problema que pertenece a NPC. Debemos describir un algoritmo que compute f, convertir las entradas de $CLACD$ a $CMACM$, es muy sencillo pues ambos algoritmos reciben un grafo, la diferencia es que $CMACM$ irá probando diferentes valores de k y n hasta encontrar los mínimos de ambos valores mientras que $CLACD$ solo recibirá el grafo. Para el caso de convertir la salida de $CMACM$ en la salida de $CLACD$, si la salida del primero es un conjunto vacío de vértices, entonces la salida es -1; sino es vacío, se verifica si la cardinalidad de la salida es menor o igual a n y que el costo es menor o igual a k.

Complejidad temporal:

Convertir las entradas de los algoritmos es $O(1)$ pues estos algoritmos utilizan el mismo grafo de entrada pero con enteros de parámetros que son desechados. Para el caso de las salidas se computa en $O(|V|)$ ya que revisa cada vértice devuelto, siendo la cantidad máxima posible $|V|$, y a la misma vez que se cuenta la cantidad de vértices, se suma el costo total, siendo $O(1)$ para cada vértice.

Hemos demostrado que nuestro problema es **NP-hard**, por tanto es tan difícil como cualquier problema **NP-completo**, para los cuales hasta la fecha no existe solución en tiempo polinomial, por eso brindamos ahora nuestra solución para resolverlo de forma aproximada.

2.2 Solución

La función `find_min_cover_approx` es un algoritmo greedy de aproximación para encontrar el menor cubrimiento de vértices en un grafo.

- **Emparejamiento Maximal:** El algoritmo comienza encontrando un emparejamiento maximal en el grafo. Un emparejamiento es un conjunto de aristas tal que no hay dos aristas que compartan un vértice común. Un emparejamiento maximal es un emparejamiento que no puede ser extendido añadiendo otra arista.

- **Cobertura de vértices a partir de un emparejamiento:** Una vez que se encuentra un emparejamiento máximo, el conjunto de todos los extremos de las aristas en este emparejamiento forma una cobertura de vértices. Esto se debe a que cada arista en el emparejamiento está cubierta por sus extremos, y dado que el emparejamiento es máximo, cualquier arista que no esté en el emparejamiento debe compartir un vértice con una arista en el emparejamiento.

- **Ratio de Aproximación:** Este enfoque garantiza una aproximación de 2. Esto significa que el tamaño de la cobertura de vértices encontrada es como máximo el doble del tamaño de la cobertura de vértices mínima. Esto se debe a que el tamaño de la cobertura de vértices mínima es al menos el tamaño del emparejamiento máximo, y la cobertura de vértices encontrada por este método es como máximo el doble del tamaño del emparejamiento máximo.