.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Ejercicios de Parcial Resueltos

Resueltos por correctores y revisados por la cátedra

Thiago Pacheco

Andrea Figueroa

1. División y Conquista

Consigna

Todos los años, la asociación de Tiro con Arco profesional realiza una preclasificación de los n jugadores que terminaron en las mejores posiciones del ranking para un evento exclusivo. En la tarjeta de invitación, se adjunta el número de posición actual y la cantidad de rivales que cada jugador logró superar en el ranking, en comparación con el año anterior.

Se cuenta con un listado que contiene el nombre del jugador y su posición en el ranking del año pasado, ordenado según el ranking actual. El objetivo es implementar un algoritmo que, dado este listado, devuelva (por ejemplo, en un diccionario) la cantidad de rivales que cada jugador logró superar. Para resolver este problema de forma eficiente, se recomienda utilizar la estrategia de ***División y Conquista****.*

Ejemplo: Dada la lista

$$\text{LISTA: } [(A,3),(B,4),(C,2),(D,8),(E,6),(F,5)],$$

se puede observar lo siguiente:

- El jugador A pasó del 3er lugar al 1er lugar, superando al jugador C.
- El jugador B llegó al 2do lugar y superó también al jugador C.
- El jugador C no superó a ningún invitado. Aunque está en la 3ra posición actual, ya tenía una mejor clasificación que el resto en el ranking anterior.

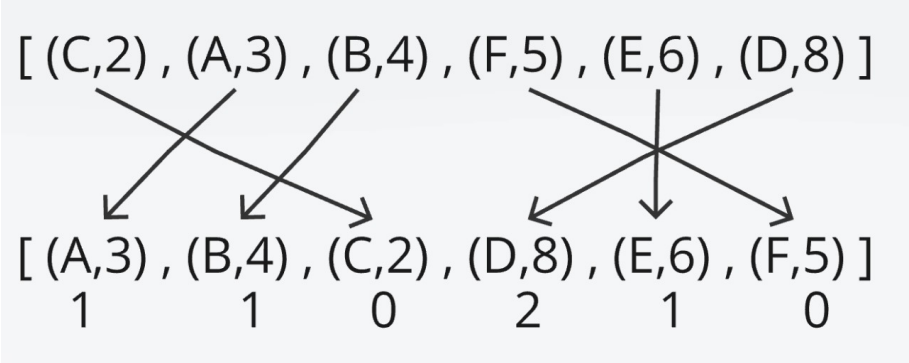


Figura 1: Ejemplo de División y Conquista

Resolución

El problema tiene similitudes con el cálculo de la cantidad de *inversiones* en un arreglo. En ese problema, se determina la cantidad de pares de elementos desordenados que necesitan intercambiarse para ordenar el arreglo. Sin embargo, en este caso, debemos calcular la cantidad de *inversiones* para cada elemento individual, lo que equivale a contar cuántos elementos cada jugador ha superado en el ranking.

Para ello, implementamos un algoritmo basado en la estructura de *Merge Sort*, añadiendo una modificación clave: cada vez que un jugador supera a otro durante la mezcla de los arreglos, contabilizamos esta superación. En otras palabras, cuando un jugador en el lado derecho del arreglo (con peor posición anterior) es colocado antes que uno en el lado izquierdo (con mejor posición anterior), actualizamos su conteo en el diccionario de adelantos.

El algoritmo tiene una complejidad temporal de $\mathcal{O}(n \log n)$, ya que mantiene la estructura eficiente de *Merge Sort*, y garantiza el cálculo correcto de adelantos para cada jugador.

Implementación

A continuación, se muestra la implementación del algoritmo:

```python
def adelantos_ranking(lista):
    adelantos = {}
    for (player, _) in lista:
        adelantos[player] = 0
    merge_sort_modificado(lista, adelantos)
    return adelantos

def merge_sort_modificado(arr, adelantos):
    if len(arr) <= 1:
        return arr
    medio = len(arr) // 2
    izq = merge_sort_modificado(arr[:medio], adelantos)
    der = merge_sort_modificado(arr[medio:], adelantos)
    return merge_conteo(izq, der, adelantos)

def merge_conteo(arr1, arr2, set_adelantos):
    i = 0
    j = 0
    nuevo = []
    while i < len(arr1) and j < len(arr2):

        if arr1[i][1] <= arr2[j][1]:
            nuevo.append(arr1[i])
            i += 1

        else:
            nuevo.append(arr2[j])
            set_adelantos[arr1[i][0]] += len(arr1) - i
            j += 1
    nuevo.extend(arr1[i:])
    nuevo.extend(arr2[j:])
    return nuevo
```

En este código, la función `adelantos_ranking` inicializa el diccionario de adelantos para cada jugador. Luego, `merge_sort_modificado` realiza la división del arreglo y llama a `merge_conteo`, que es responsable de calcular los adelantos mientras mezcla los subarreglos.

Análisis de Complejidad

El algoritmo implementado se basa en la estrategia de División y Conquista, por lo que su complejidad se puede analizar utilizando el **Teorema Maestro**. La ecuación de recurrencia que describe el comportamiento del algoritmo es la siguiente:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Donde:

- $A = 2$: cantidad de llamados recursivos.
- $B = 2$: factor por el que dividimos el tamaño del problema en cada nivel.
- $C = 1$: costo del término no recursivo (partir y juntar los subproblemas).

Review del Teorema Maestro:
El Teorema Maestro establece tres casos dependiendo de la relación entre $\log_B(A)$ y C:

1. Si $\log_B(A) < C$, entonces $T(n) = O(n^C)$.
2. Si $\log_B(A) = C$, entonces $T(n) = O(n^C \log n)$.
3. Si $\log_B(A) > C$, entonces $T(n) = O\left(n^{\log_B(A)}\right)$.

De acuerdo con el **Teorema Maestro**, comparamos el valor de $\log_B(A)$ con C:

$$\log_B(A) = \log_2(2) = 1$$

En este caso, $\log_B(A) = C$. Según el teorema, cuando $\log_B(A) = C$, la complejidad temporal del algoritmo es:

$$T(n) = O(n^C \log n) = O(n \log n)$$

(La complejidad coincide con merge sort como esperabamos al realizar una variante)

2. Greedy

Consigna

Implementar un algoritmo greedy que permita obtener el Dominating Set mínimo (es decir, que contenga la menor cantidad de vértices) para el caso de un árbol (en el contexto de teoría de grafos, no un árbol binario). Indicar y justificar la complejidad del algoritmo implementado. Justificar por qué se trata de un algoritmo greedy. Indicar si el algoritmo siempre da solución óptima. Si lo es, explicar detalladamente, sino dar un contraejemplo.

Resolución

La idea es siempre pintar los padres, ya que para dominar las hojas siempre pintaremos sus padres (su único adyacente), y este cubre otras hojas y a sí mismo. Con esta implementación no es necesario considerar el caso de "eliminar mi padre si no tiene otros hijos", ya que nunca habrá padre con un solo hijo, sino que siempre será hoja y, por ende, hijo.

```python
def dominating_set_arbol(grafo):
    grados, hojas = obtener_grados_y_hojas(grafo)
    solucion = set()
    while hojas:
        hoja = hojas.pop()
        padre = obtener_padre(grafo, hoja)
        solucion.add(padre)
        for ady in grafo.adyacentes(padre):
            grados[ady] -= 1
            if grados[ady] == 1:
                hojas.add(ady)
            if grados[ady] == 0:
                hojas.discard(ady)
    return solucion

def obtener_grados_y_hojas(grafo):
    grados = {}
    hojas = set()
    for v in grafo.obtener_vertices():
        grados[v] = len(grafo.adyacentes(v))
        if grados[v] == 1:
            hojas.add(v)
    return grados, hojas
```

Complejidad

- Calcular los grados es $O(V + E)$.
- Ir pintando padres y sacando sus hijos implica ver cada vértice y arista, $O(V + E)$.
- Como es un árbol, $E = V - 1$, entonces nos queda $O(V + V - 1) \Rightarrow O(V)$.

Justificación Greedy

Un algoritmo greedy busca tomar decisiones localmente óptimas en cada paso, con el objetivo de alcanzar una solución global óptima.

La regla greedy en este caso está basada en la visualización de los vértices que no debemos tomar, para poder seleccionar con mayor efectividad los vértices que sí debemos incluir en el conjunto dominante.

La estrategia es la siguiente:

- Visualizar las hojas: Las hojas (vértices de grado 1) son los vértices más aislados del grafo y, en un árbol, son las que menos cobertura ofrecen y, por ende, la peor opción.
- Elegir los padres de las hojas: La decisión local óptima es seleccionar los vértices que tienen el mayor impacto de cobertura con el mínimo costo. Al seleccionar a los padres de las hojas, cubrimos no solo las hojas, sino también a los vértices vecinos y al propio vértice seleccionado.

Optimalidad del Algoritmo

El algoritmo es óptimo en este caso, porque la decisión de cubrir las hojas eligiendo sus padres garantiza una cobertura eficiente en los árboles.

Dado que se debe cubrir todo el grafo y teniendo siempre hojas (al ser un grafo árbol), tenemos dos opciones para cubrir las hojas: pintarlas a ellas o pintar a su adyacente (padre). Siendo la segunda más conveniente y la que nuestro algoritmo garantiza. Este enfoque asegura una reducción progresiva del problema: al eliminar las hojas y seleccionar sus padres, el árbol se simplifica, permitiendo que cada decisión localmente óptima contribuya al conjunto dominante más pequeño posible.

3. Backtracking

Implementar un algoritmo que (por backtracking) dado un grafo no dirigido en el que sus vértices tienen valores positivos, permita obtener el Dominating Set de suma mínima. Es decir, aquel dominating set en el cual la suma de todos los valores de los vértices sea mínima (no es importante que la cantidad de vértices del set sea mínima). Por simplicidad, considerar que el grafo es conexo.

En general, los ejercicios de `backtracking` solo requieren que se implemente el código. Sin embargo, es recomendable explicar las podas que se implementan.

```
1  def _dominating_set(grafo, vertices, actual, solucion_parcial, suma_parcial,
       solucion_actual, suma_actual):
2      if es_dominating_set(grafo, solucion_parcial):
3          if suma_actual < suma_parcial:
4             return solucion_actual, suma_actual
5          else:
6              return solucion_parcial[:], suma_parcial
7      if actual >= len(vertices) or suma_parcial >= suma_actual:
8          return solucion_actual, suma_actual
9      solucion_parcial.append(vertices[actual])
10     solucion_actual, suma_actual = _dominating_set(grafo, vertices, actual+1,
       solucion_parcial, suma_parcial + vertices[actual].valor, solucion_actual,
       suma_actual)
11     solucion_parcial.pop()
12     solucion_actual, suma_actual = _dominating_set(grafo, vertices, actual+1,
       solucion_parcial, suma_parcial + vertices[actual].valor, solucion_actual,
       suma_actual)
13     return solucion_actual, suma_actual
14
15 def es_dominating_set(grafo, solucion):
16     cubierto = set()
17     for v in solucion:
18         if v not in cubierto:
19             cubierto.add(v)
20         for w in grafo.adyacentes(v):
21             if w not in cubierto:
22                 cubierto.add(w)
23     return len(grafo) == len(cubierto)
24
25 def dominating_set(grafo):
26     return _dominating_set(grafo, grafo.obtener_vertices(), 0, [], 0, None, float('
       inf'))
```

La única poda implementada en este algoritmo es la siguiente: "si al agregar un elemento la solución supera la mejor solución encontrada hasta el momento, retrocedo", ya que no se encontrará una mejor solución en ese camino. Siempre que se implementen las podas obvias, la solución será aprobable.

Si el ejercicio permite realizar múltiples podas, o si se implementa alguna poda "difícil"que reduce considerablemente el tiempo de ejecución, y dicha poda es explicada correctamente, dependiendo del caso, Martín podría considerar un B+.

Comentarios importantes:

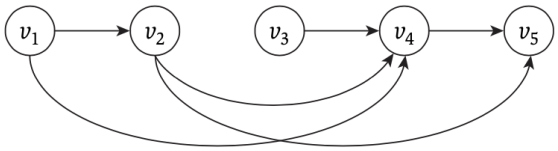
- No utilizar fuerza bruta.
- Siempre implementar auqnue sea una poda.
- No sobrescribir variables locales innecesariamente.
- No reemplazar la variable `sol_actual` por `sol_parcial` durante la exploración por error.
- No es necesario justificar la complejidad a menos que la consigna lo pida (casi nunca).
- En caso de no encontrar solución, el algoritmo DEBE devolver una solución vacía.

4. Programación dinámica

Consigna:

Definimos a un grafo ordenado como un grafo dirigido con vértices $v_1, \ldots, v_n$ en el que todos los vértices, salvo v_n tienen al menos una arista que sale del vértice, y cada arista va de un vértice de menor índice a uno de mayor índice (es decir, las aristas tienen la forma (vi, vj) con $i < j$).

Implementar un algoritmo de programación dinámica que dado un grafo ordenado (y, si les resulta útil, una lista con los vértices en orden) determine cuál es la longitud del camino más largo. Dar la ecuación de recurrencia correspondiente. Dar también el algoritmo de recontrucción de la solución. Indicar y justificar la complejidad del algoritmo implementado. Se pone a continuación un ejemplo de un grafo ordenado.



Solución:

1. **Idea:** (dispensable en la resolución que se entrega) El camino más largo para llegar al vértice v_n no deja de ser un camino para llegar hasta v_n, los posibles caminos pasan por los vertices adyacentes a v_n, es decir los que tienen una arista $v_a \to v_n$. El camino más largo para llegar a v_n será llegando del camino más largo para llegar a uno de sus directamente adyacentes (v_a) . Ya vimos que nos refiere a un problema más chico.

2. **Ecuación de recurrencia:** Sea $Op(n)$ la longitud del camino más largo para llegar hasta el vértice v_n

$$Op(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + max(Op[a]), \forall\ v_a \text{ vértice con arista entrante a } v_n \end{cases}$$

3. **Implementación:**

```
def camino_mas_largo(grafo, vertices):
    # Obtenemos vertices entrantes a  cada vertice
    v_entrantes = {v: set() for v in vertices}
    for v in vertices:
        for w in grafo.adyacentes(v):
            v_entrantes[w].add(v)
    #construccion de optimos
    previa_eleccion = {}
    previa_eleccion[vertices[0]] = null
    OPT = {v: 0 for v in vertices}
    for v in vertices:
        longitud_max = 0
        for w in v_entrantes[v]:
            if OPT[w] > longitud_max:
                longitud_max = OPT[w]
                previa_eleccion[v] = w
        OPT[v] = longitud_max + 1
        if len(v_entrantes[v]) == 0:
            OPT[v] = 0
    #reconstruccion de solucion
    camino = []
    v_actual = vertices[-1]
    anterior = previa_eleccion[-1]
    while(anterior != null)
         camino.append(anterior)
         anterior = previa_eleccion[anterior]
    return OPT, camino[::-1]
```

4. **Análisis de complejidad:** Sea v la cantidad de vertices y e la cantidad de aristas en el grafo:
   - **obtener vertices entrantes** inicialización de v_entrantes: v operaciones constantes por cada vertice $O(v)$ y accedemos a cada vertice y arista del grafo $O(v + e)$
   - **construcción de optimos** por cada vertice vemos sus adyacentes. Igual que antes vemos todo el grafo O(v + e)
   - **reconstrucción de solución**: Recorremos las elecciones previas que nos llevaron al vertice v_n con una longitud de máximo n vertices realiando operaciones constantes: $O(v)$
   - **Total:** $O(v + e)$

5. Programación lineal

Juan es ambicioso pero también algo vago. Dispone de varias ofertas de trabajo diarias, pero no quiere trabajar tres días seguidos. Se tiene la información de la ganancia del día i (G_i), para cada día. A continuación, se describe el modelo de programación lineal que maximiza el monto a ganar por Juan, sabiendo que no aceptará trabajar tres días seguidos.

5.1. Definición de Variables

- Y_i: Variable binaria que indica si se trabaja el día i. Es decir:

$$Y_i = \begin{cases} 1 & \text{si se trabaja el día } i, \\ 0 & \text{en caso contrario.} \end{cases}$$

5.2. Restricciones

Para evitar que Juan trabaje tres días consecutivos, imponemos la siguiente restricción para todo $i > 2$:

$$Y_i + Y_{i-1} + Y_{i-2} \leq 2$$

Nota sobre la restricción: No necesitamos utilizar el "truco de la gran M en este caso porque:

- Si $Y_i = 0$, la restricción se cumple automáticamente (es una tautología).
- Si $Y_i = 1$, la restricción obliga a que al menos una de las variables Y_{i-1} o Y_{i-2} sea 0 (o ambas).

5.3. Función Objetivo

La función objetivo consiste en maximizar la utilidad total de trabajar ciertos días. Esto se expresa como:

$$\text{Max:} \quad \sum_i Y_i \cdot G_i$$

donde G_i representa la ganancia asociada con trabajar el día i.

5.4. Modelo Completo

El modelo de programación lineal se puede resumir como:

$$\text{Maximizar:} \quad \sum_i Y_i \cdot G_i$$

$$\text{Sujeto a:} \quad Y_i + Y_{i-1} + Y_{i-2} \leq 2 \quad \forall i > 2$$

$$Y_i \in \{0, 1\} \quad \forall i$$

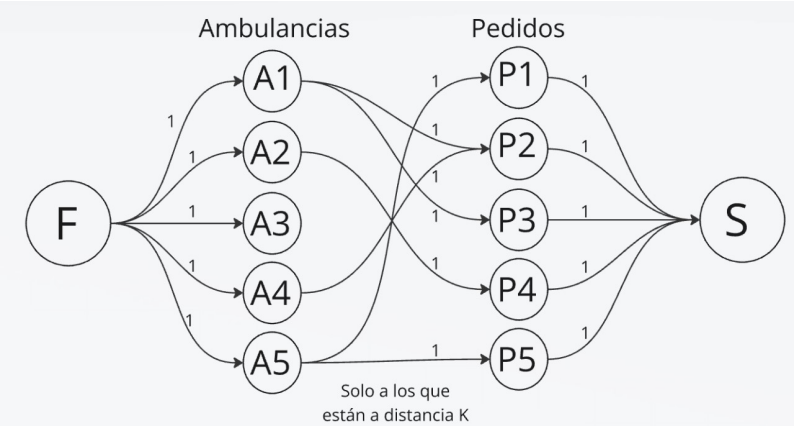
6. Redes

La Cruz Roja cuenta con n ambulancias, de las cuáles conoce la ubicación de cada una. En un momento dado llegan p pedidos de ambulancias para socorrer personas. Debido a diferentes reglas que tienen, una ambulancia no debe trasladarse más de k kilómetros. Se quiere saber si se puede hacer una asignación de ambulancias a los pedidos, asignando cada una a como máximo 1 pedido. Implementar un algoritmo que resuelva este problema, utilizando redes de flujo. Indicar y justificar la complejidad del algoritmo implementado para el problema planteado.

Modelado del grafo

Modelamos el problema como un grafo dirigido para maximizar la cantidad de pedidos que pueden ser atendidos por las ambulancias, respetando las restricciones de distancia máxima k entre una ambulancia y un pedido. El grafo tiene la siguiente estructura:

- Se define un nodo fuente s y un nodo sumidero t.
- Cada ambulancia se representa como un vértice.
- Cada pedido se representa como un vértice.
- La fuente s se conecta a cada vértice de ambulancia con una arista de capacidad máxima 1.
- Cada ambulancia se conecta a los pedidos que están a una distancia $\leq k$ con una arista de capacidad máxima 1.
- Cada pedido se conecta al sumidero t con una arista de capacidad máxima 1.



De esta manera:

- Una ambulancia puede atender como máximo un pedido, dado que su entrada y salida tienen capacidad de flujo igual a 1.
- Un pedido solo puede ser atendido por una ambulancia debido a las mismas restricciones de capacidad.
- Las aristas entre ambulancias y pedidos garantizan que una ambulancia solo pueda atender pedidos dentro de la distancia máxima k.

```python
def modelar_grafo(ambulancias, pedidos, distancias, k):
    grafo = Grafo(es_dirigido=True)
    grafo.agregar_vertice("s")
    grafo.agregar_vertice("t")

    for pedido in pedidos:
        grafo.agregar_vertice(pedido)
        grafo.agregar_arista(pedido, "t", 1)

    for ambulancia in ambulancias:
        grafo.agregar_vertice(ambulancia)
        grafo.agregar_arista("t", ambulancia, 1)
        for pedido in pedidos:
            if distancias[(ambulancia, pedido)] <= k:
                grafo.agregar_arista(ambulancia, pedido, 1)
    return grafo
```

Resolución del Problema

Una vez modelado el grafo y calculado el flujo máximo con el algoritmo de Ford-Fulkerson, evaluamos qué pedidos son cubiertos analizando las aristas con flujo asignado entre ambulancias y pedidos. Si todas las aristas ambulancia $\rightarrow$ pedido tienen flujo 1, entonces todos los pedidos son atendidos.

```python
def curz_roja(ambulancias, pedidos, distancias, k):
    grafo = modelar_grafo(ambulancias, pedidos, distancias, k)
    flujo = flujo(grafo, "s", "t")
    res = []
    for a in ambulancias:
        for p in pedidos:
            if grafo.estan_unidos(a, p) and flujo[(a, p)] == 1:
                res.append((a, p))
    if len(res) == len(pedidos):
        return True, res
    return False, []
```

Complejidad del Algoritmo

Sea:

- $A = \text{len(ambulancias)}$: número de ambulancias.
- $P = \text{len(pedidos)}$: número de pedidos.

Modelado del Grafo

El modelado recorre cada ambulancia y cada pedido para crear los vertices y recorre todos los pedidos por cada ambulancia para conectar las aristas necesarias:

$$O(A \cdot P)$$

Cálculo del Flujo Máximo

El cálculo del flujo máximo se realiza utilizando el algoritmo de Ford-Fulkerson. En teoría, este algoritmo tiene una complejidad de $O(V \cdot E^2)$, ya que en cada iteración puede ser necesario recorrer todas las aristas y los vértices. Sin embargo, dado que todas las aristas tienen una capacidad de 1, el comportamiento del algoritmo se optimiza.

En este caso, el algoritmo no necesita volver a procesar las mismas aristas ni recorrer caminos repetidos, ya que una vez que un camino es encontrado y el flujo es asignado, ese camino ya no se considera en futuras iteraciones. Al ser todas las aristas de capacidad 1, el algoritmo simplemente va descubriendo un camino posible en cada iteración, asignando flujo a ese camino sin recalcular sobre aristas ya procesadas ni repetir el mismo camino.

Este enfoque reduce la complejidad de las iteraciones, ya que cada camino encontrado es único y no se reevalúa, lo que permite que el algoritmo recorra el grafo de manera más eficiente. Como resultado, la complejidad se reduce a $O(V + E)$ ya que recorre todo el grafo.

$$O(V + E) = O((A + P) + (A \cdot P)) = O(A \cdot P)$$

Complejidad Total

La complejidad final es dominada por el cálculo del flujo, quedando:

$$O(A \cdot P)$$

Aclaración

Por casualidad, en el ejercicio tocó un cálculo de complejidad un poco más complicado. En el parcial, no es necesario una explicación tan detallada, pero el cálculo debe ser correcto y acompañado de una breve explicación.

7. Reducciones- Problemas NP completos

Consigna:

El Hitting Set Problem es: Dado un conjunto A de n elementos, m subconjuntos $B_1, B_2, \ldots, B_m$ de A ($Bi \subseteq A \forall i$), y un número k, existe un subconjunto $C \subseteq A$ con $|C| \leq k$ tal que C tenga al menos un elemento de cada B_i (es decir, $C \cap B_i \neq \varnothing$) ?

Dominating Set es un problema NP-Completo. Demostrar que Hitting Set Problem es un problema NP-Completo, utilizando Dominating-Set para esto.

Un set dominante (Dominating Set) de un grafo G es un subconjunto D de vértices de G, tal que todo vértice de G pertenece a D o es adyacente a un vértice en D. El problema de decisión del set dominante implica, dado un grafo G y un número k, determinar si existe un set dominante de a lo sumo tamaño k.

Solución:

1. **Demostración que HS pertenece a NP** Debemos plantear el validador eficiente de una supuesta solución de HS en código.

```
def verificar_hitting_set(A, subconjuntos, k, C):
    for subconjunto in subconjuntos:
        if not subconjunto.issubset(A):
            return False
    if not C.issubset(A):
        return False
    if len(C) > k:
        return False
    for subconjunto in subconjuntos:
        if not any(elemento in C for elemento in subconjunto):
            return False
    return True
```

Análisis de complejidad:

- Verificar para cada $B_i \subseteq A$ resulta $O(m \mid A \mid)$ pues cada B_i máximo tiene $m \mid A \mid$ elementos.
- Verificar que todos los elementos de C estén en A resulta $O(\mid C \mid)$ que es máximo tiene k elementos que es menor que el tamaño de A: $O(\mid A \mid)$
- Revisar que $|C| \leq k$ es $O(1)$
- Verificar que $C \cap B_i \neq \varnothing \; \forall i$ recorriendo todos los subconjuntos B_i y verificando al menos una intersección, lo cual tiene costo $O(m \mid A \mid)$.

Con lo cual el costo temporal resulta $O(m \mid A \mid$ con lo cual el costo temporal es polinimial entonces $HS \in NP$

2. **Demostración que HS es NP-completo mediante reducción** $DS \leq_p HS$

Veamos las definiciones de los problemas

Definición de DS

- **Entrada:** un grafo G y un numero k
  Pregunta:¿Existe un conjunto $D \subseteq V$ (siendo V los vertices de G) tal que $\mid D \mid \leq k$ y cada elemento de V es *dominado* por al menos un vértice en D?

Definición de HS

- **Entrada:** Un conjunto A de n elementos, m subconjuntos $B_1, B_2, \ldots, B_m$ de A ($Bi \subseteq A \forall i$), y un número k.
- **Pregunta:** ¿Existe un subconjunto $C \subseteq A$ con $|C| \leq k'$ tal que C tenga al menos un elemento de cada B_i (es decir, $C \cap B_i \neq \varnothing$) ?

Construcción de la reducción:

La idea es hacer que cada vértice y sus adyacentes sean un subconjunto B_i (siendo entonces $m =\mid V \mid$ la cantidad de subconjuntos) siendo estos subconjuntos del conjunto $A = V$, luego el número de elementos que le pediremos el HS será la cantidad de elementos que queremos en nuestro dominating set: $k' = k$.

Planteada la idea debemos escribir cuales serían los pasos para la transformación:

- Tomamos el conjunto V de vértices del grafo como el conjunto A para el llamado a HS.
- Por cada vértice $v_i \in V$ armamos un subconjutno de V: B_i al que le agregamos este vértice y sus adyacentes (los vértices dominados por vi). Costo: $O(v + e)$
- Establecemos $k' = k$, donde k' es el tamaño máximo permitido del conjunto C en HS.

Correspondencia entre soluciones

Planteamos el si y solo si:

- Si **existe conjunto** $D \subseteq V$ con máximo k elementos que **es un set dominante para el grafo** G:

  Entonces el conjunto D tendrá un elemento de cada subconjunto B_i (como se definió previamente) pues si D es un dominating set entonces todo vertice está en D o tiene un adyacente suyo en D, es decir, D tiene al menos un elemento de cada B_i, además, si sabemos que $\mid D \mid \leq k$ entonces tenemos un HS del conjunto $A = V$, $B_i = v_i \cup$adyacentes de v_i , k =k.
- si **existe un subconjunto** $C \subseteq A = V$ con $\mid C \mid \leq k$ que es un Hitting set para $A = V$, $B_i = v_i \cup$adyacentes de v_i

  Entonces como el conjunto C tiene al menos un elemento de cada B_i que por definición harían que C domine a todos los vertices.

Complejidad de reducción

- Construir $A = V$ toma $O(v)$
- Construir los subconjuntos B_i recorre cada vértice y sus adyacentes, lo que toma $O(V + E)$.

Con esto ya establecimos la reducción polinomial.